Summary: SLR(1) Parsing

We have previously looked at LR(0) parsing, where the action to take is conditioned totally by the symbol on the top of the stack (0 lookahead symbols are used, i.e., the next input symbol is not examined to determine which action to apply).

Now we begin to look at LR parsers using 'lookahead': the action to take is conditioned by looking at the next token on the input. The action column of the LR(0) parser is thus expanded to a table with one column for each possible terminal (as only terminals appear on the input queue).

In practice, the Action Table and the Goto Table are merged into one table (see below).

1 Basic Concepts

1.1 Starters

The starter set for a nonterminal is the set of terminals which could potentially be the first token on the left of the nonterminals structure. The starter set of a nonterminal N can be found as follows:

1) For each production where nonterminal N is the LHS,
   a. If the leftmost symbol of the RHS of the production is a terminal, add that to the starter set of N
   b. If the leftmost symbol of the RHS of the production is a nonterminal, append the starter set of that nonterminal to the starter set of N.

Example: Given the following grammar, the starter set of A is \{a, y\}. The starter set of S is also \{x, a, y\}.

\[
\begin{align*}
S & \rightarrow A \mid xb \\
A & \rightarrow aAb \mid B \\
B & \rightarrow y
\end{align*}
\]

1.2 Followers

The follower set of a given nonterminal N is derived as follows:

In each rule, where N occurs on the RHS,

a) if N is followed by a terminal, add to followers
b) If N is followed by a nonterm, X, add the STARTER set of X
   c) If N is at the end of the rule, append the FOLLOW set of the rule’s LHS

Example: Given the grammar:

\[
\begin{align*}
1. & E' \rightarrow E \$
2. & E \rightarrow T \\
3. & E \rightarrow E + T \\
3. & T \rightarrow \text{id} \\
4. & T \rightarrow ( E )
\end{align*}
\]

For this grammar, T only occurs in context (c), at the end of a rule. Its follow set is thus that for E

- E in rule 1 is followed by $
• E in rule 3 is followed by +
• E in rule 5 is followed by )
• Follow set of T is thus: \{ $, +, ) \}

1.3 Building the DFA
Exactly as for LR(0).

1.4 Constructing the Parse Table
The actions are placed in the Goto table.
1. For each transition arc between \( S_i \) and \( S_j \), with label X:
   a. Enter \( j \) in the cell \((S_i, X)\) (i.e. enter the end state in the row of the start state under the arc label)
   b. If the label is a terminal, precede with a ‘s’
2. For each state with double circle (a reduce state):
   a. Find the completed rules in the state’s closure (those with the dot after the last symbol)
   b. For each of these, find the Follow set of the LHS symbol.
   c. For each terminal in the Follow set of the LHS, place an entry ‘r’ followed by the rule number (e.g., r5 if it is rule 5).
3. In place of the action reduce for the topmost rule (here \( E' \ := \ E \ $\)), put the action ‘accept’.

1.5 Using the Parse Table (for SLR(1), LR(1), LALR(1))
Initially we have:
1. The input vector: a vector of the input tokens, terminated by token $
2. Next token pointer: pointer to the next input token to be processed, initially pointing at the first token.
3. The Stack: initially we place \( S_0 \) on the stack.

Recursively:
1. Apply action: Apply the action given in the cell indexed by current state and next input token.
   • If shift, move the next token onto the top of the stack, and move the pointer to the next token.
   • If reduce, look up the rule given in the action, and remove \( n \times 2 \) items from the stack (where \( n \) is the number of symbols on the RHS of the rule). Then place the LHS of the production on the stack.
   • If accept, then finish parsing, we have a finished analysis.
2. Determine the Goto State: The top of the stack now contains a state and a symbol (terminal or nonterminal). In the parse table, look up the row for that state, and the column for the symbol.
   • If there just a number there, take this as the new state number.
   • If there is “s” followed by a number, take the number as the new state number.
   • If there is “r” followed by a number, check your parse table, you made an error in drawing it.
   • If the cell is empty, check your parse table, you made an error in drawing it.
3. **Change the State:**
   - put the new state number on the top of the stack

**REMEMBER:**
- **Action** is conditioned by next input token.
- **Goto** is conditioned by top of stack.
- If the action was a *shift*, they will be the same token.

### 1.6 Grammar Limitations

A grammar is SLR(0) if:

1. there are no shift-reduce conflicts, i.e., it is never the case that in constructing the parse table, both a shift action and a reduce action are placed in the same cell (note that because the next input symbol conditions input for SLR(1), some cases of s-r conflict in an LR(0) are no problem for an SLR(1) grammar).

2. there are no reduce-reduce conflicts: there are no cases where the closure of a state contains two or more dot final items, and where the follow set of the LHSs of the dot-final items intersect (the intersection represents a conflict). Again, SLR(1) grammars can handle some of the reduce-reduce conflicts of LR(0) grammars, if the next input symbol distinguishes which production to reduce by.

### 2 SLR(1) Parsing with Lambda productions

A lambda production has the form:

\[ P \rightarrow \lambda \]

...where \( \lambda \) corresponds to 0 tokens. This type of production allows for a nonterminal to be recognised without consuming any input tokens.

For example, given the grammar:

\[
\begin{align*}
A & \rightarrow aB \\
B & \rightarrow b \mid \lambda
\end{align*}
\]

...we could recognise strings: “ab” or “a” (the second if the lambda production is used).

A grammar with lambda productions can be rewritten without them, by replacing all instances of the LHS of the lambda with its possible expansions, e.g., the above grammar is equivalent to:

\[
\begin{align*}
A & \rightarrow ab \mid a
\end{align*}
\]

However, for some compiling tasks it may be important to keep the lambda’s nonterminal in the grammar (e.g., it might be important for semantic interpretation). We thus address here how lambda productions can be handled in the LR(1) family of parsers (including SLR, LR and LALR).
2.1 Calculating STARTERS with Lambda

The calculation of STARTERS without lambda

For each production where nonterminal N is the LHS,
   a. If the leftmost symbol of the RHS of the production is a terminal, add that to the starter set of N
   b. If the leftmost symbol of the RHS of the production is a nonterminal, append the starter set of that nonterminal to the starter set of N.

The calculation of STARTERS with lambda

For each production where nonterminal N is the LHS,
   a. If the leftmost symbol of the RHS of the production is a terminal, add that to the starter set of N
   b. If the leftmost symbol of the RHS of the production is a nonterminal, append the starter set of that nonterminal to the starter set of N.
   c. Where there is more than one symbol on the RHS and the starter set generated by cases (a), (b) includes lambda, delete lambda and append the starter set for the RHS with the first symbol eliminated.

Example: Given the following grammar, the starter set of S is \{x, y\}. The starter set of A is \{x, \lambda\}.

\[
\begin{align*}
S &::= A \ B \ C \\
A &::= x \\
A &::= \lambda \\
B &::= y
\end{align*}
\]

2.2 Calculating FOLLOWS with Lambda

The calculation of Follow without lambda

In each rule, where N occurs on the RHS,
   a. if N is followed by a terminal t, add t to followers
   b. If N is followed by a nonterm X, add the STARTER set of X
   c. If N is at the end of the rule, append the FOLLOW set of the rule’s LHS

The calculation of Follow with lambda

As before, but also:
   d. if the starters of X (in rule b) included lambda:
      i. delete lambda from the FOLLOW set
      ii. repeat (a-d) with X deleted from the RHS

2.3 Building the DFA

One builds the DFA as normal except note that since \lambda matches zero tokens, and thus the following two items are equivalent:

\[
Y ::= \lambda \quad = \quad Y ::= \lambda
\]
Basically, where an item has a dot in front of a lambda, automatically move the dot to the position AFTER the lambda.
Thus, when an item for a lambda production appears in a state’s closure, it is always automatically a reduce state.

2.4 Building the Parse Table
This process works exactly the same as in the normal case. When one is processing a lambda production, it is always a reduce action (never ‘shift’). For SLR(1), one proceeds as usual, finding the potential followers of the LHS of the lambda production, and placing a reduce action under each of these.
Note that if another item in the same state would allow a “shift” or “reduce” operation for any of these terminals, then a shift-reduce or reduce-reduce conflict would occur, but this is no different as for productions which are not lambda.

2.5 Using the Parse Table
The parsing procedure is exactly as for cases without lambda productions. Note the rule given previously:

If reduce, look up the rule given in the action, and remove n∗2 items from the stack (where n is the number of symbols on the RHS of the rule). Then place the LHS of the production on the stack.

In the case of a lambda production, the λ actually accounts for 0 (zero) symbols, so we remove 2 * 0 = 0 items from the stack. In other words, the stack is left unchanged. We then place the LHS of the lambda production on the stack.
For instance, assuming we had the following stack at the point of applying a reduce by Y :- λ.

```
Stack: 0 a 2 B 3
```

We then apply the reduce, deleting zero of the items on the stack, and then add Y:
```
Stack: 0 a 2 B 3 Y
```

From this point, we proceed as always to the Goto step, and thus look up the goto state in the Y column of State 3.