LR Parsing Example
Martha Kosa

You have seen several parsing techniques so far, ranging from the simple but inefficient brute force algorithm to the efficient LL(1) parsing algorithm. Now you will practice with another one: SLR(1) parsing. It is efficient like the LL(1) parsing algorithm, but it produces a rightmost derivation of a valid string in reverse order instead of a leftmost derivation. In addition to a stack and a parse table, it uses a DFA, where the input alphabet for the DFA is the set of all possible grammar symbols, both terminals and nonterminals. It performs shift and reduce operations.

Our example is the language $\{a^{m+1}b^mc^n \mid m, n \geq 0\}$. It is the concatenation of the two languages $\{a^{m+1}b^m \mid m \geq 0\}$ and $\{c^n \mid n \geq 0\}$. For your convenience, this grammar has been created for you.

Try It!

1. Open the grammar file **amplus1bmcn.CFG.flap**.
2. Select ***Input > Build SLR(1) Parse Table***. You should see a window similar to the following.

1. Fill in the values for the FIRST sets for each of the nonterminal symbols. If the nonterminal symbol appears as the lefthand side of a lambda rule, lambda will be in its first set. Do not type any commas! You can check your work as you go along by clicking the **Next** button after you fill in a set. If you have already practiced a lot with FIRST sets, you can click the **Do Step** button.
2. Fill in the values for the FOLLOW sets for each of the nonterminal symbols. Do not type any commas! Don't forget about the $! You can check your work in the same way as you did previously.
3. After your complete the FOLLOW sets, you will see the following dialog.

1. This gets you to start the DFA that will help build the SLR(1) parse table. Click on the **Yes** button.
2. You should now see the following dialog.

1. Click on the first rule. Why? You should see the following.

Select the first entry in the mini dialog. Why?

1. You need to select three more rules, one at a time, and three more entries in the

corresponding mini dialogs. Which entries should you select, and why? Remember that you are in the **initial** state of the DFA. Your dialog should look similar to the following.

1. Click on the **OK** button.
2. Click on the **Do Step** button. Zoom and resize as necessary to see the complete DFA. Your complete DFA should look similar to the following. What are the relationships among the labels on the DFA states, the transitions, and the production rules of the grammar? What is the significance of the final states?

1. Now we fill in the entries of the parse table. Notice that the rows of the parse table correspond to states in the DFA and the columns correspond to symbols in the grammar and the special end of input symbol $. We start with row 0, corresponding to the initial state. We look at the transitions coming out of that state and use the label on the transition as the column. The destination state of the transition will be placed in the table entry at the corresponding row and column. If the label is a terminal symbol, we will shift (denoted by **s**) before going to the destination state. The table entry will contain **s** followed by the destination state. How many entries do you need to complete in row 0? Why? You can click the **Next** button to check your work.
2. Complete the entries in row 1. Look carefully at the label on state 1. Where is the dot, which signifies where we are in attempting to apply a production rule? What kind of state is state 1? What column should we use? Why? We fill in the entry in column $ of row 1. The label is **S' ▪ S ▪.** This means we have completed work with our original start symbol. This means our string will be accepted. Type **acc** in the entry.
3. Complete the entries in row 2. Look carefully at the label on state 2. It contains **Y ▪ ▪.** This corresponds to a lambda rule. Since state 2 is not a final state, we need to find a neighboring state with **Y ▪ ▪** in its label. This is state 5. We will reduce (denoted by **r**) before going to the neighboring state. The table entry will contain **r** followed by the destination state. How many entries do you need to complete in row 2? Why? As before, you can click the **Next** button to check your work.
4. Complete the entries in row 3. Look carefully at the label on state 3. It contains X ▪ **a** ▪**.** This corresponds to the full application of a production rule. This means that when we encounter any other terminal symbol (including $), we need to reduce and remain in state 3. Type **r3** in the applicable entries. As before, you can click the **Next** button to check your work.
5. Complete the entries in the remaining rows. As usual, you can click the **Next** button to check your work. Remember that the $ column will need to be used for states that are final or contain labels corresponding to completed rules. You will need to be careful when workin with states containing labels corresponding to completed production rules. You will need to reduce and possibly move to

another state. Where will you move corresponds to a state with a label with the **lefthand side** of the production rule followed by ☒ because we are working our way back toward the start symbol in a parse. When you finish, your parse table should look like the following.

1. Use the table to parse the string **aabcc**. Enter **aabcc** in the input box and click the **Start** button. Your window should look similar to the following (after possible resizing).

1. Click the **Step** button until the string is successfully parsed. Observe what entries in the table are consulted, what is done to the stack, and how the parse tree is built. Note the actions described at the bottom of the window after each step. What information is stored on the stack? How many trees are present at early stages in the parse? How many trees are present at the end of the parse?
2. Choose **Derivation Table** in the combo box. Observe that the derivation proceeds backward from the string to the start symbol and that reversing the derivation results in the rightmost nonterminal symbol being expanded at each step.

Now let's see what happens when we attempt to parse an invalid string. In any valid string, the a's occur before the b's, and the b's occur before the c's, and the number of a's is exactly one more than the number of b's. What happens if we have too many b's?

Try It!

1. If it is not already open, open the file **amplus1bmcn.CFG.flap.**
2. If the SLR(1) parse table is not already built, build it as before. You can click the **Do All!** button and then the **Parse** button to save time.
3. Enter the string **aabbcc** in the input box, and then click the **Start** button.
4. Click the **Step** button until the string is rejected. Observe what entries in the table are consulted, what is done to the stack, and how the parse tree is built. Note the actions described at the bottom of the window after each step. How many times did you click the **Step** button? Why can't the parse continue?

The parser stops when the first error is detected. For more practice, attempt to parse some other invalid strings, such as strings with too many a's or strings where the a's, b's, and c's are not in the proper order.