LR Parsing

1 Introduction

Before reading this quick JFLAP tutorial on parsing please make sure to look at a reference on LL parsing to get an understanding of how the First and Follow sets are defined. It also makes sense to get a good understanding of shift-reduce parsing.

LR Parsing does parsing for deterministic context free grammars. It reads input from left to right and produces a bottom-up parse. To avoid backtracking or guessing, the LR parser is allowed to peek ahead at k lookahead input symbols before deciding how to parse earlier symbols. JFLAP does the LR parsing with k=1. Specifically JFLAP implements SLR(1) parsing.

An LR parser scans and parses the input text in one forward pass over the text. The parser builds up the parse tree incrementally, bottom up, and left to right, without guessing or backtracking

It is a specific type of a general category of parsers called shift-reduce parsers.

2 Building the first and follow sets

The example we will do is to parse a simple arithmetic expression. Our grammar only uses 1 as an integer. But you can imagine adding more rules to get the same for all possible integers.

Enter the following grammar into JFLAP or load the file LRParsing_SimpleExpression.jff.

$$\begin{aligned} S &\to E \\ E &\to T + E \\ E &\to T \\ T &\to 1 * T \\ T &\to 1 \\ T &\to (E) \end{aligned}$$

The LR parsing works by first creating these sets called FIRST and FOLLOW for each of the non-terminals (aside from the special start variable S' that is added in).

The FIRST set of a variable is just the first terminal that can possibly appear in a derivation that starts from that variable. In JFLAP when you load up the grammar and hit SLR parse you get the following as a first step.

If you are incorrect in figuring out the FIRST sets for any variable JFLAP highlights the rows where there is an error. Also remember that the exclamation point (!) represents the empty string in this setting.

In our example let us figure out the First set for E. Since the 2 rules for E are $E \rightarrow T + E$ and $E \rightarrow T$, it makes sense to concentrate on what T can give you. It is clear that T can produce a 1 and a (as the first character. 1 and (therefore become elements of the first set. Enter those two elements in (remember to not put a comma between the elements since JFLAP does that for you).

Finish the remaining first sets yourself. JFLAP will be nice and correct you if you make any mistakes.

At the end this should be what you have for your first sets for this example

Define FOLLOW sets. $ is the end of string character.

	FIRST	FOLLOW
E	{ 1, (}	{ }
S	{ 1, (}	{ }
T	{ 1, (}	{ }

After determining the first sets, click next to get to stage where you have to determine follow sets for those same variables.

The follow sets consist of the terminals that could be immediately after the variable in question.

Again, let us take the the example of E and see what its follow set will be. Clearly) is one possibility because of the $T \rightarrow (E)$ rule. The other possibility is something just ending with E which means that you have reached the end of the string. $ is used for the end of string character in this setting so the follow set for E then becomes $\{), \$\}$.

Do the rest of the follow sets yourself. Again, JFLAP is there to correct you. If you feel stuck at any point, you can click on the 'do selected' button and get the answer to that particular follow set.

At the end here is what things should look like after defining the first and the follow sets.

Define FOLLOW sets. $ is the end of string character.

	FIRST	FOLLOW
E	{ 1, (}	{), $ }
S	{ 1, (}	{ $ }
T	{ 1, (}	{ $, +,) }

At this point you are ready to go to the next step of building the parser which is making the DFA. So click next. If you have done everything correctly, JFLAP will display a cool congratulatory message.

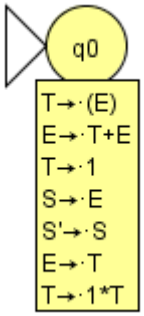
3 Building the DFA

The first step in building the DFA for parsing is to define the initial state. The way to think about the DFA is to begin with the start of the input is marked with some symbol. In JFLAP is a '.'. As we step from state to state the '.' gets moved around to indicate the amount that has been processed.

So create the first state by using the start symbol and for each subsequent symbol adding the rule that has a '.' as the first thing on the right side.

While defining the state, you have to click each rule that you can get to and pick one of

the possible positions of the '.' as part of your label. Since this is just the start state, we will have the '.' as the first thing on the right side in every one of the transitions and the initial state should look like the one shown below.



Once the initial state is defined, the subsequent states of the DFA are made by creating transitions using whatever terminal or variable comes right after the current position of the '.'. For instance, in this initial state we need a transition on a (, a transition on a T, a transition on an S, a transition on an E and finally a transition on a 1.

To make these transitions in JFLAP click the 'goto set on symbol' button, draw out a transition arrow and put down a terminal and JFLAP will ask you to pick the transitions that should label the next state.

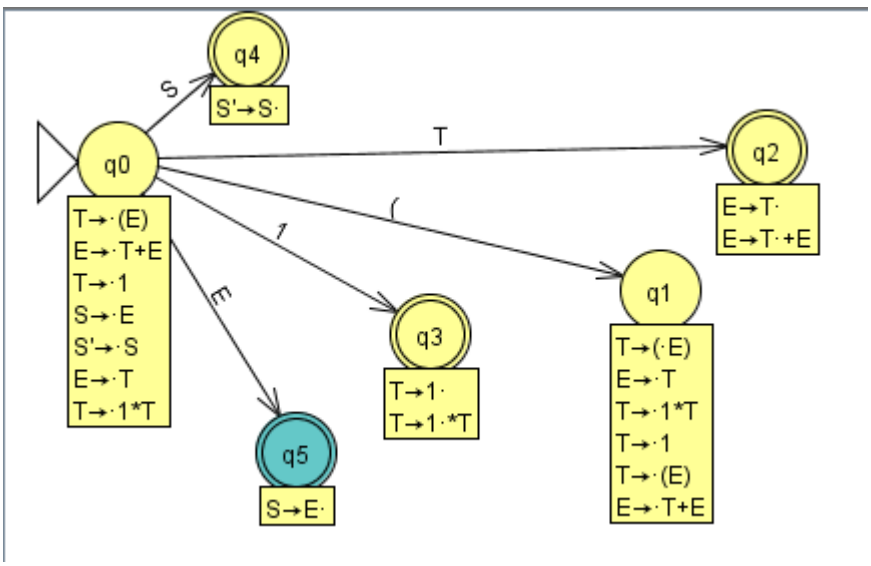
Let us walk through 2 cases. Transition on an S. In this case, the $S' \to S$ rule is the one that will used and all that happens is we advance the '.' one step further. In this case the '.' has now advanced all the way to the end which means that this state also needs to be denoted as a final state in the DFA. As usual, use the attribute editor button in order to do that.

For the transition on a (, we first have to add the label of $T \to (.E)$. As a result of the '.' being right in front of a variable though, all productions that have E on the left side now have to be dealt with. That means that $E \to .T + E$ has to be added and $E \to .T$. Again, since we have the '.' in front of T, the rules corresponding to T have to dealt with in a similar manner. So you end up with the label for the (transition basically being

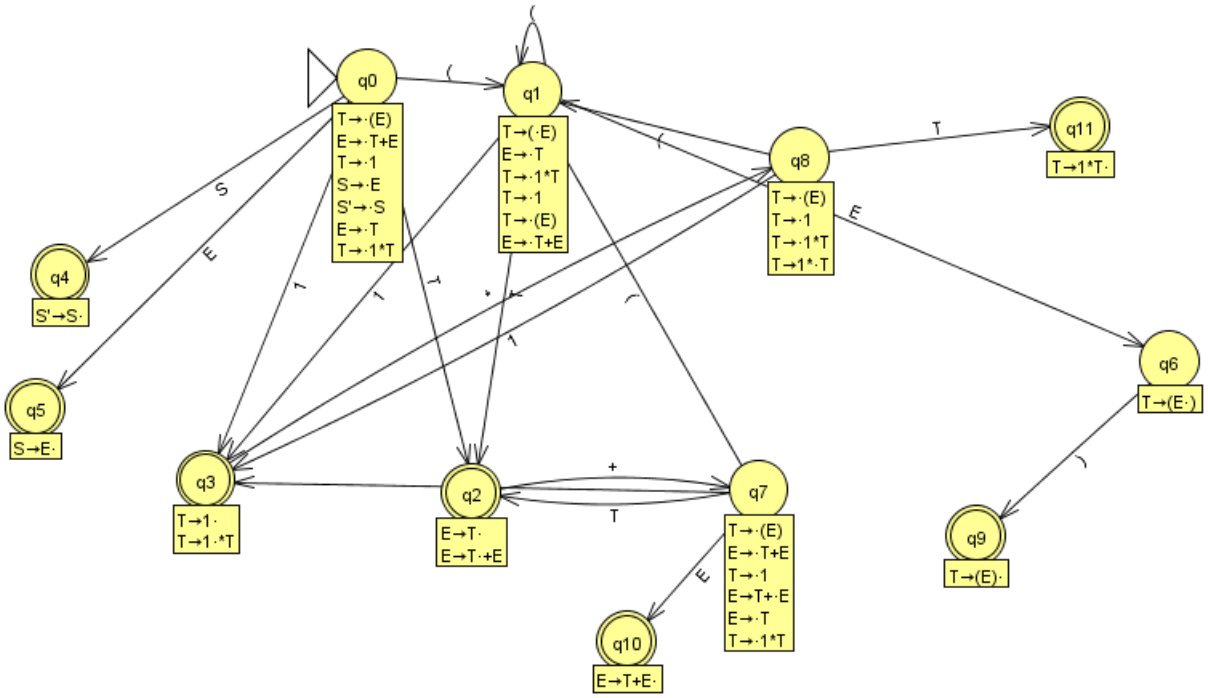
$$\begin{aligned} T &\to (.E) \\ E &\to .T \\ E &\to .T + E \\ T &\to .1 * T \\ T &\to .1 \\ T &\to .(E) \end{aligned}$$

Define the transitions and the new states they give rise to for T, E and 1. At the end of this step you should have the following partial DFA.

Note that if a state's label contains any transitions that have a '.' at the end, that state needs to be marked as a final state.



Now that you have seen how to make states and transitions, go ahead and complete the DFA. The final DFA is included here for completeness but try and do it yourself.



4 Building the parse table

The heart and soul of a parser's working is a parse table. This involves filling up a table that contains all the terminals and the variables along the columns and the states in the rows. The states in JFLAP just show up as the numbers so for this particular case please note that the terminal 1 and state 1 are two different things.

We being, as expected with the first row. The row that corresponds to state 0. There are a few different possibilities for each entry

- shift - If the transition has a terminal on it then that terminal along with the destination state are put into the parsing stack. To represent this in the parse table we write 's' followed by the number associated with the destination state.
- goto - If the transition is on a variable then we just need to put the destination state into the parse table.
- reduce - If you are dealing with a final state, there has to be a rule that has the '.' as the rightmost character on the RHS. This rule will be used to pop items from the parsing stack so in the parse table we have to reference that rule. In JFLAP the rules are numbered for you and if you hover over any rule you get the reference number associated with it. If the state is q and say the rule that has the '.' as the rightmost character has the variable A on the LHS. Then for every element in the follow set of A we put down 'rn' where n is the rule number.
- accept - If you get to a state where the start symbol is being popped out, just put an 'acc' in the column corresponding to the end character the $.

In state 0 when you see a (, you move to state 1 and you also have to shift (on top of the parse stack. This shift operation is entered into the parse table as s1. When you see a 1, the transition is to state 3 hence s3 in the table.

In state 0 and S takes you to $q4$ so you put a 4 into the corresponding column. The same reasoning fills up the columns corresponding to E and T.

JFLAP can check your work for you as you go along. You can click on any cell in the parse table (or highlight many of them) and click do selected.

The only place you get an accept is state $q4$ so in row 4 we put the 'acc'.

For an example of what reduce looks like consider state 3 which is a final state because $T \to 1.$. First we get the rule number for $T \to 1$ which is 5. Now for this row, we need to find everything that is in the follow set for T, so that is $\{), +, \$\}$ and in every column we put corresponding to these symbols we put down $r5$.

Now that you have an example of every possible entry in the parse table, go ahead and try and complete it. The finished parse table looks like this.

5 Example of the parser in action

Now that we have a parse table completely filled out we can parse any string that is provided as input. Let us take the example of parsing a simple expression like $1 + 1$ which can be generated via this grammar and show how that is done using the grammar. Before diving into the LR parser in action convince yourself that this string is actually something that the grammar is capable of producing.

	(	)	*	+	1	$	E	S	T
0	s1				s3		5	4	2
1	s1				s3		6		2
2		r3		s7		r3			
3		r5	s8	r5		r5			
4						acc			
5						r1			
6		s9							
7	s1				s3		10		2
8	s1				s3				11
9		r6		r6		r6			
10		r2				r2			
11		r4		r4		r4			

In JFLAP go ahead and click parse which will create another tab and allow you to enter an input string for parsing in the input textbox.

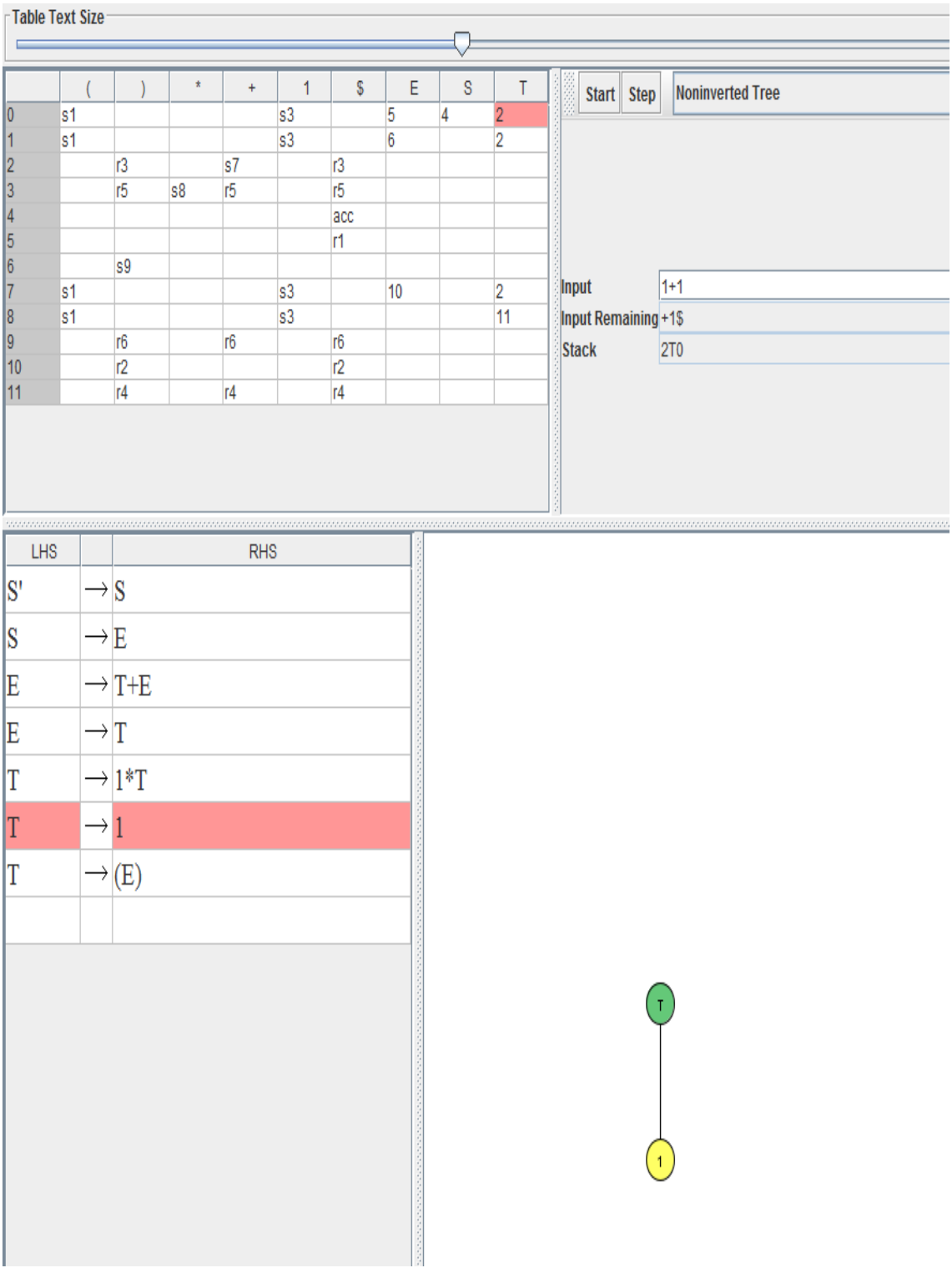
Enter 1 + 1 in there and click start. The input remaining at that point becomes the entire string and the stack will contain the start state which is 0.

Now the first symbol is a 1 and when we step through (click step in JFLAP), we look at the entry corresponding to the terminal 1 and row 0 and we see it is s3 which means that we will shift the pair of 1 (the terminal) and 3 (the state) on the stack.

In the next step we look at state 3 and see what happens when the input is +. We notice there is a reduce rule in row 3 and the column corresponding to a +. A reduce works in the following manner - If the entry is reduce r and rule r has m symbols in its RHS, then pop m pairs off the parse stack. Let s' be the state now revealed on top of the parse stack and N be the LHS nonterminal for rule r. Then consult the parse table and push the state given by row s' and column N onto the stack. The input remaining is not changed by this step.

As per that logic, since the rule is $T \rightarrow 1$ here are the steps taken

1. pop 1 pair off the stack. That means the stack will now just have state 0
2. 0 is on top of the stack and the LHS non-terminal in this rule is T. The parse table corresponding to T is 2.
3. 2 and T get pushed into the stack.
4. In the derivation tree that is being constructed we create a node for T and show it deriving the 1 as shown below.

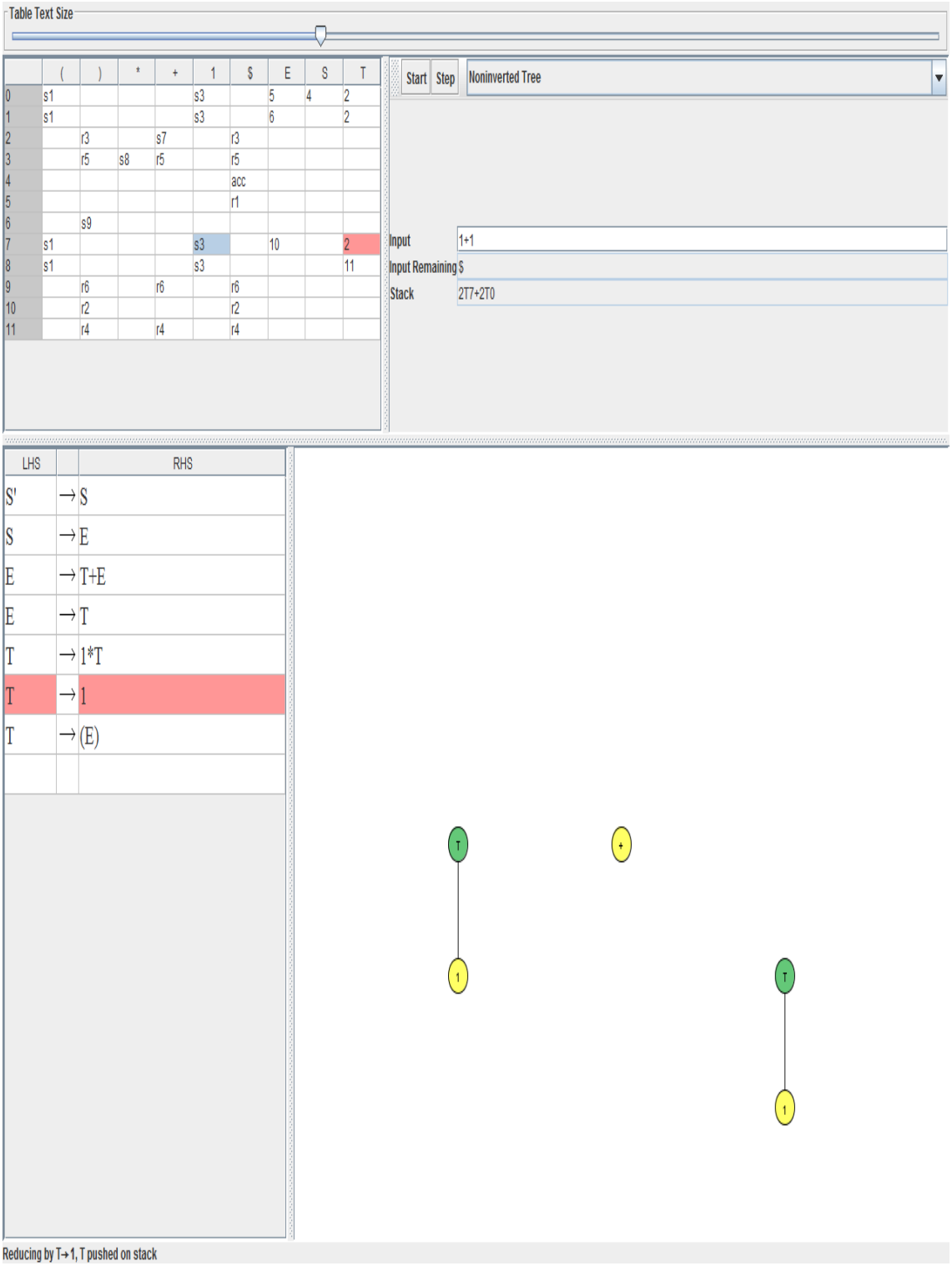


Now we are in the stage where we have a 2 on the stack and the lookahead symbol is a +. 2 in the row and + in the columns gives us s7 which means + and 7 are pushed onto the stack and we have consumed the + from the input.

Now we have to see what happens when we are state 7 and the lookahead has a 1. That is s3 so the 1 and 3 gets pushed onto the stack.

We are at state 3 and the lookahead now is the end of string character, the $. The entry

is r5 which means we have to look at the $T \rightarrow 1$ rule and do our reduce step. Since 7 is the state revealed on the stack after the pop we look up the entry corresponding to 7 and T which gives you 2 on the stack and the derivation tree gets modified thusly.



The subsequent steps are described below and to follow along using JFLAP just press step repeatedly.

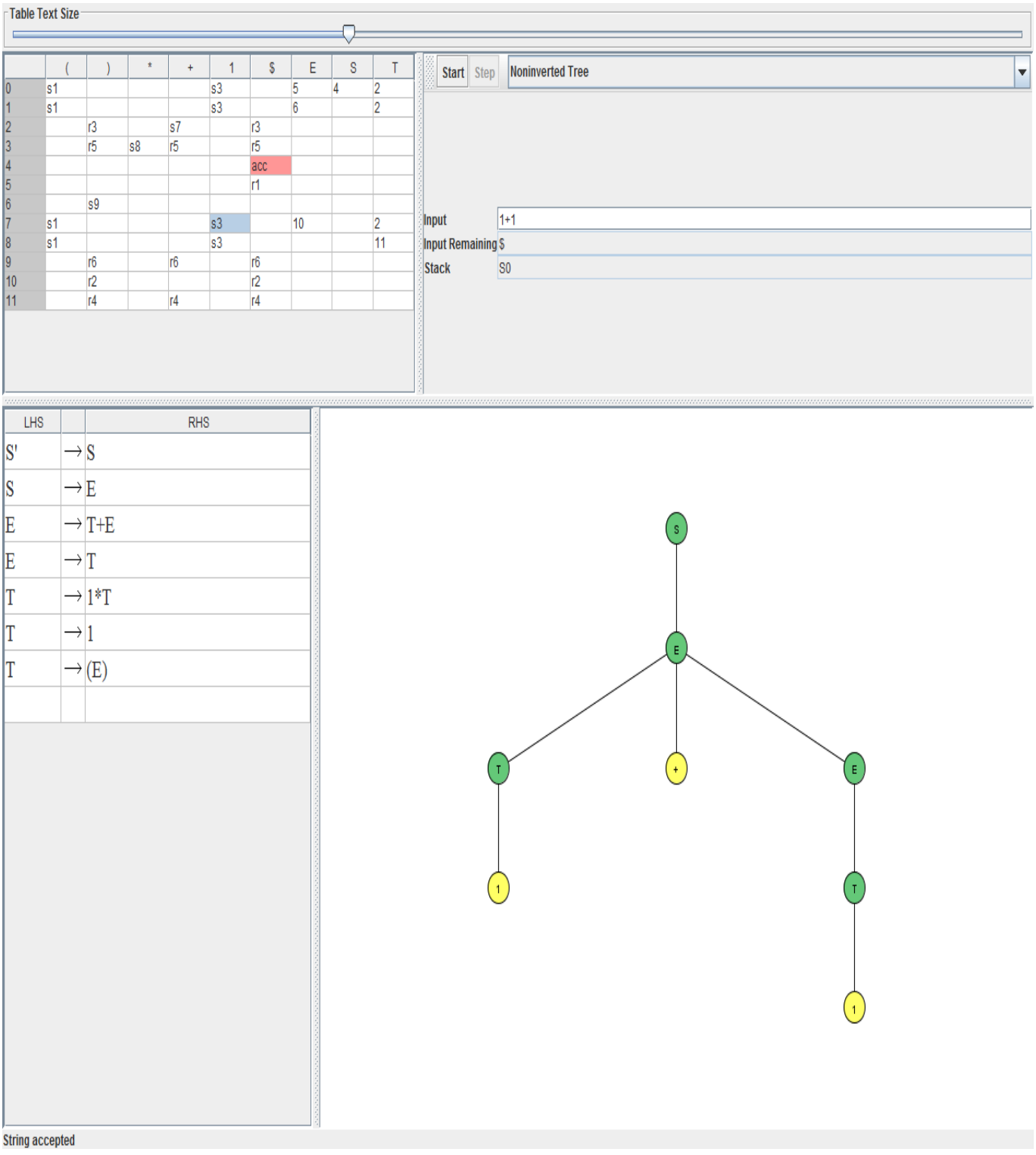
The 2 with a $ combination gives you r3 which means rule 3 needs to be used. rule 3 is $E \rightarrow T$. The state on the stack is 7 and the entry in the parse table corresponding to E for 7 is 10. so 10 gets put on the stack and the derivation tree now has the addition of the $E \rightarrow T$ rule.

The 10 with a $ combination has the entry r2. This one is interesting because we now have a rule which has 3 symbols on the right. So 3 pairs get kicked off the stack when we use it and the state that is revealed on the stack is a 0. So we need to look up the parse table for column E corresponding to 0. That turns to be 5 and therefore we get the following

Now we have state 5 and the $ which again has a reduce entry. This time it is r1 which is $S \rightarrow E$. Upon doing the pop, 0 is the state that is revealed. The row corresponding to 0 and the column corresponding to S gives us a go to 4.

Finally with 4 being the state on top of the stack and $ being the lookahead symbol we get the 'acc' entry which corresponds to this string being accepted.

The final derivation tree is shown below



Notice how the tree has been built up in this bottom-up fashion which is a feature of these LR parsers.

6 Questions