LR Parsing

1 Introduction

The LL Parsing that is provided in JFLAP is what is formally referred to as LL(1) parsing. Grammars that can be parsed using this algorithm are called LL grammars and they form a subset of the grammars that can be represented using Deterministic Pushdown Automata.

The key properties of the LL parsing algorithm are

- The first L means the input string is processed from left to right.
- The second L means the derivation will be a leftmost derivation (the leftmost variable is replaced at each step).
- 1 means that one symbol in the input string is used to help guide the parse.
- It is a top-down parser which means you start your parsing by using the start symbol and then considering the possible production rules that can be used to build up the string in a top-down manner. Contrast this to the SLR parser which works in a bottom-up fashion.

2 Building the first and follow sets

The example we will do is to parse a simple arithmetic expression (note that this is the same example used in our LR parsing tutorial). 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 LLParsing_Parens.jff.

$$S \rightarrow (S)S$$
$$S \rightarrow \varepsilon$$

The LL 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).

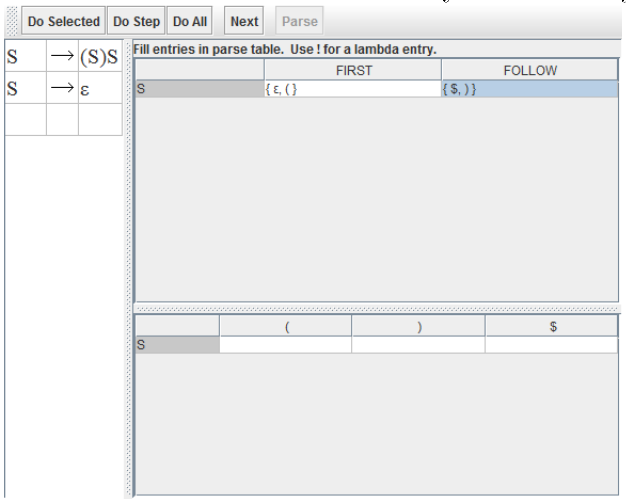
The FIRST set of a variable is just the first terminal that can possibly appear in a derivation that starts from that variable. The follow sets consist of the terminals that could be immediately after the variable in question. In JFLAP when you load up the grammar and hit LL parse you have to compute FIRST and FOLLOW sets first up.

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 since S is the only non-terminal, it is easy to see that the first set for S will be $\{(,!\}$.

As far as the follow set goes, since a closing) can come immediately after an S, that is the only element in the follow set $\{)\}$.

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



S	→	(S)S
S	→	ε

Fill entries in parse table. Use ! for a lambda entry.

	FIRST	FOLLOW
S	{ ε, (}	{ $,) }

	(	)	$
S			

The next step is building the all important parse table.

There are two main rules for filling in the table. To understand those rules it is important to define FIRST(w), where w is a string. FIRST(w) for any string $w = X_1X_2 \dots X_n$ (a string of terminals and nonterminals) as follows:

1. FIRST(w) contains $FIRST(X_1)\{\varepsilon\}$.

2. For each $i = 2, \dots, n$ if FIRST(X_k) contains ε for all k = 1, ..., i 1, then FIRST(w) contains FIRST(X_i)$\{\varepsilon\}$.

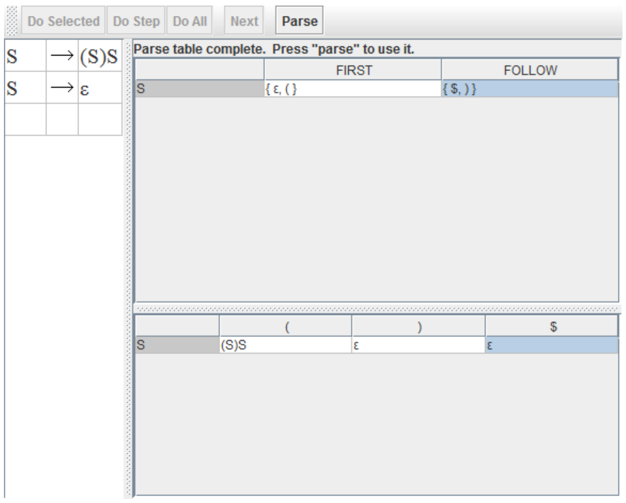
3. If every single X_i has ε in their FIRST set, then ε will be in FIRST(w).

Now given that definition of FIRST(w) for any string w, we can build up the parse table as follows.

1. For production $A \to w$, for each terminal in FIRST(w), put w as an entry in the A row under the column for that terminal.

2. For production $A \to w$, if the empty string is in FIRST(w), put w as an entry in the A row under the columns for all symbols in the FOLLOW(A).

Applying these 2 rules to our simple grammar we obtain the following parse table



S	→	(S)S
S	→	ε

Parse table complete. Press "parse" to use it.

	FIRST	FOLLOW
S	{ ε, (}	{ $,) }

	(	)	$
S	(S)S	ε	ε

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 The actual parsing!

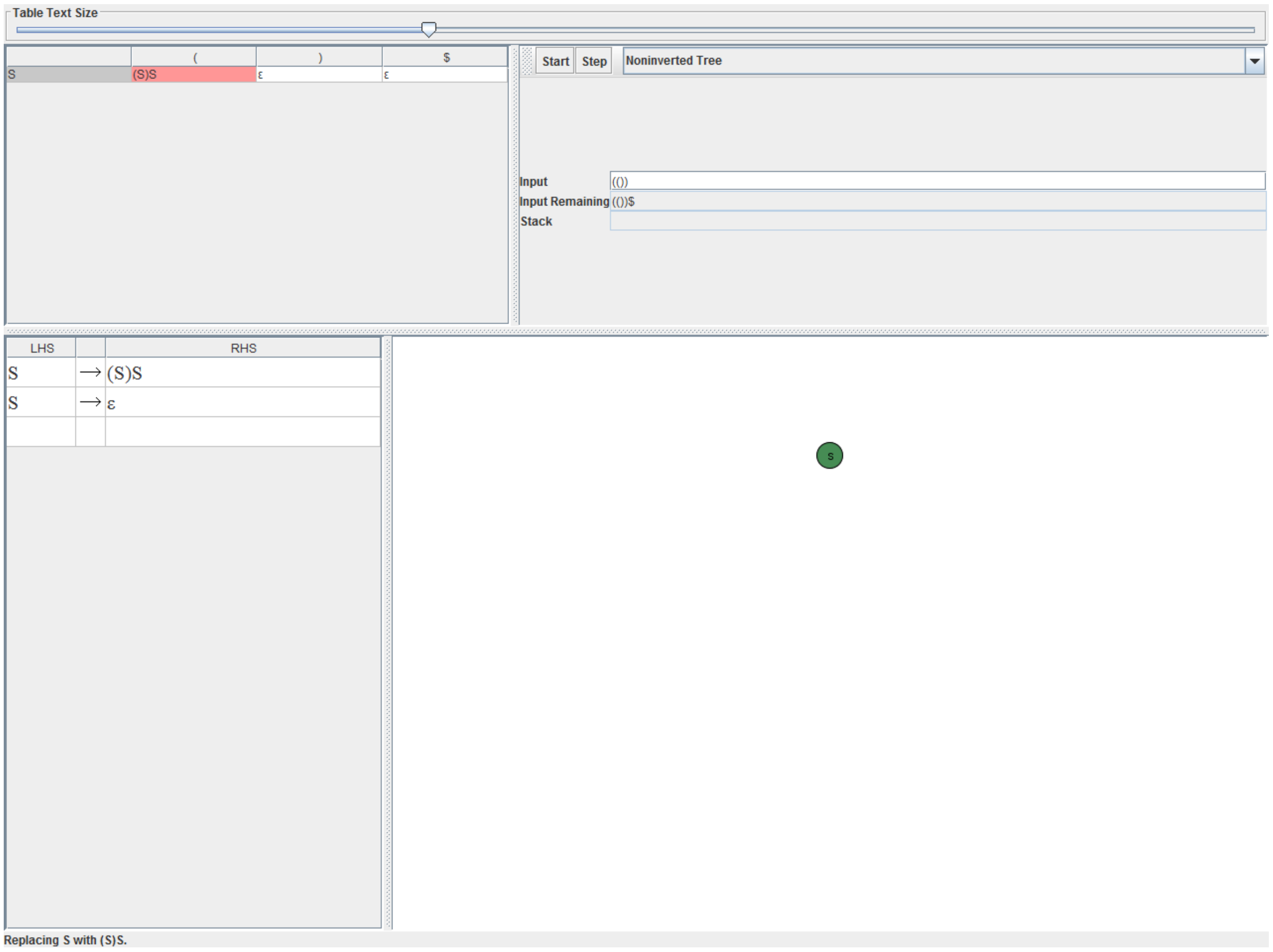
Click on parse and let us try and parse the simple set of matched parentheses - (()).

In the first step, we just mark the termination of the string with a $.

The parsing is done in a top-down fashion, so the first step would be to begin with the start symbol S.

Also, the start symbol S would be on the top of the stack.

Now the next step will be to look at the next character in the input, which is (and then use the parse table to see which rule gets applied.



	(	)	$
S	(S)S	ε	ε

Input (())
Input Remaining (())$
Stack

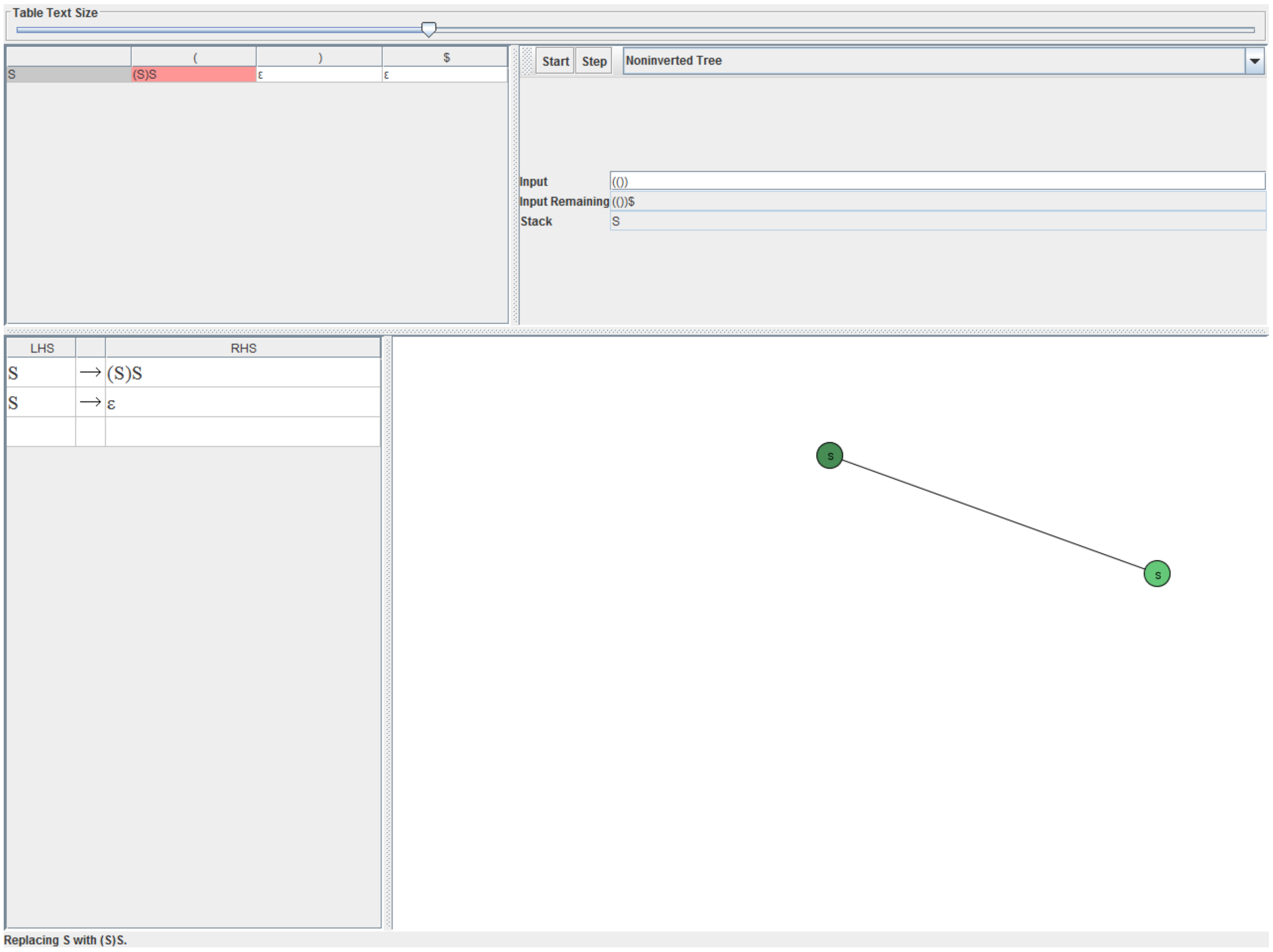
LHS		RHS
S	→	(S)S
S	→	ε

Replacing S with (S)S.

The rule application means that the variable on the stack gets popped off

A stack stores variables and terminals from right sides of productions. Variables are popped off the stack and replaced by the right side of a production, and terminals are popped off and matched in the order they appear in the string.

In our case the right side of our production rule is the symbol S, so that gets put on the stack. That also means the derivation tree changes to add a node corresponding to S.



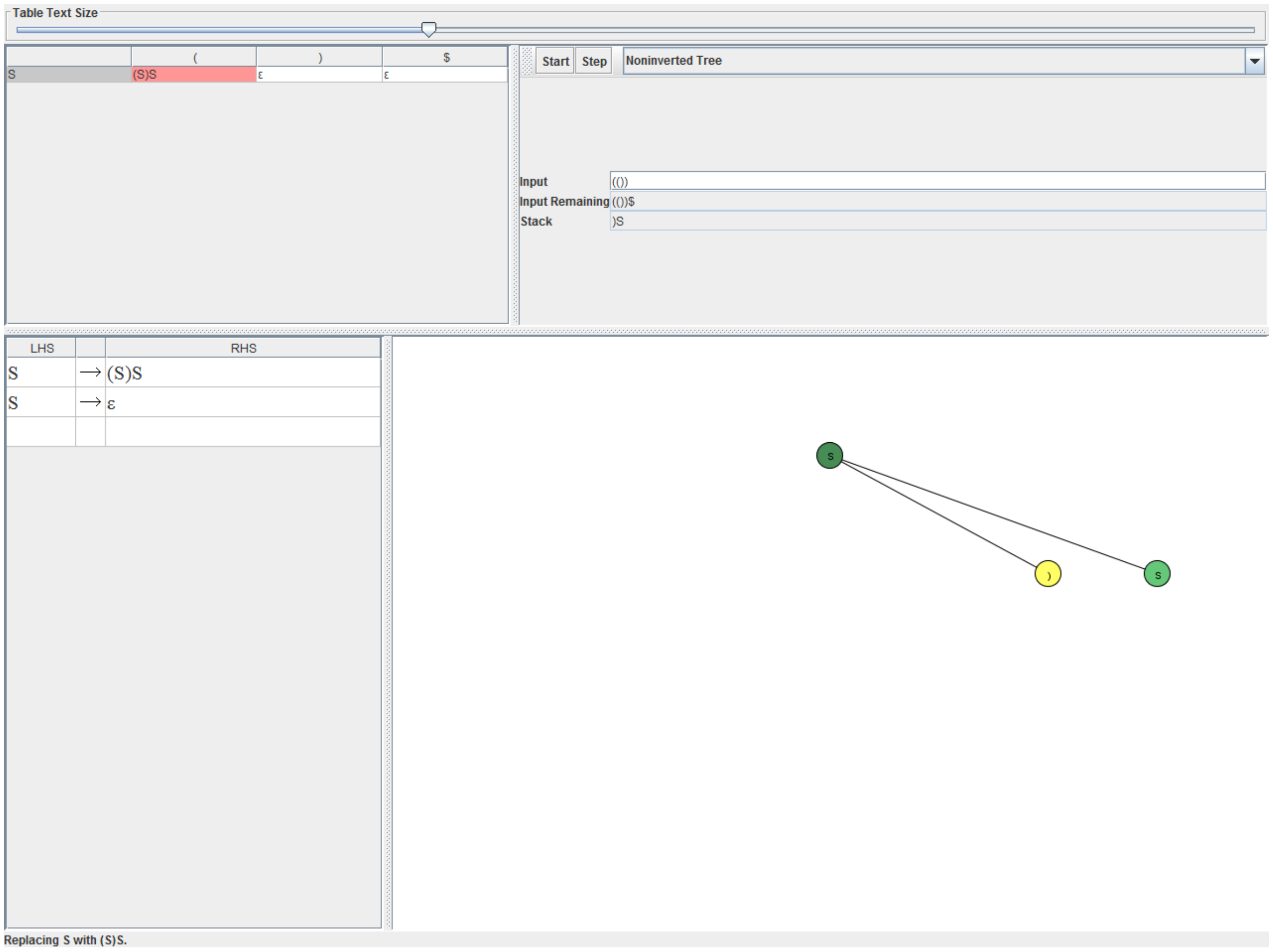
	(	)	$
S	(S)S	ε	ε

Input (())
Input Remaining (())$
Stack S

LHS		RHS
S	→	(S)S
S	→	ε

Replacing S with (S)S.

The next step adds more of the right side of the production to the stack



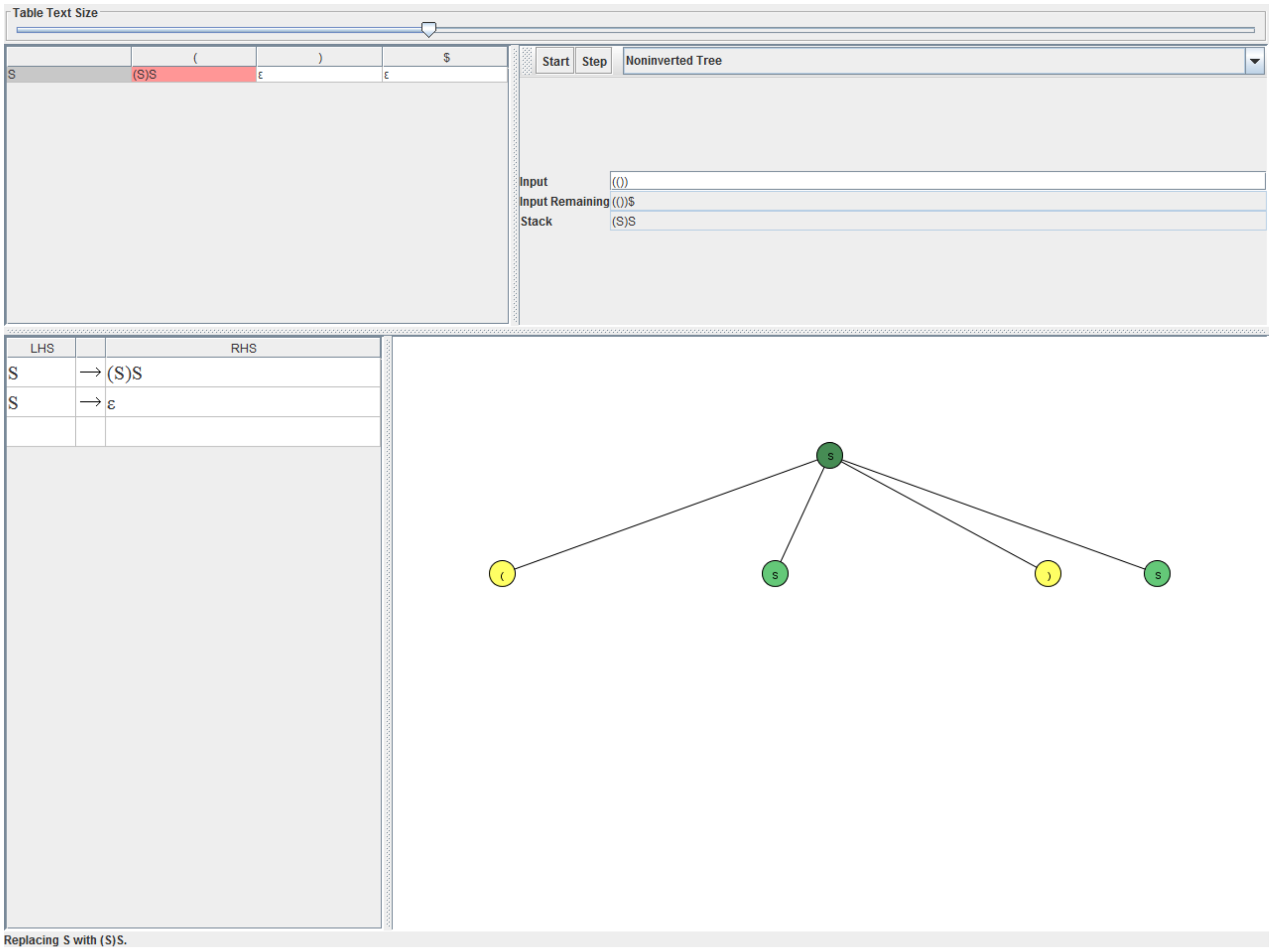
	(	)	$
S	(S)S	ε	ε

Input (())
Input Remaining ())$
Stack)S

LHS		RHS
S	→	(S)S
S	→	ε

Replacing S with (S)S.

In two more steps, the stack has now accomodated the entire right side of the rule and the derivation tree has made its first level.



	(	)	$
S	(S)S	ε	ε

Input (())
Input Remaining (())$
Stack (S)S

LHS		RHS
S	→	(S)S
S	→	ε

Replacing S with (S)S.

The next step is to observe the top of the stack and match the (with the (that is the start of the remaining string and removes both of them (we have effectively derived the start of the string).

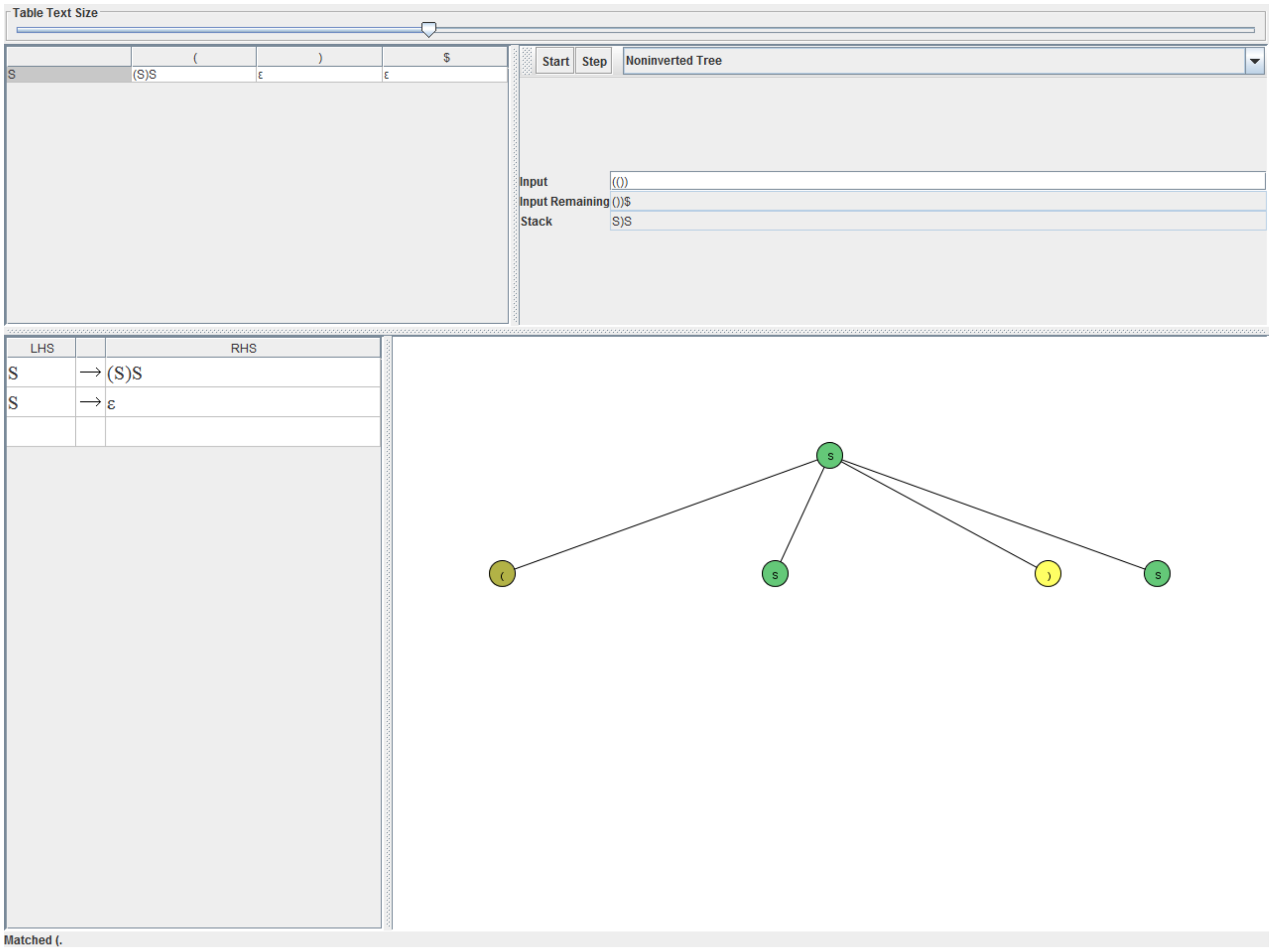


Table Text Size

	(	)	$
S	(S)S	ε	ε

Start Step Noninverted Tree

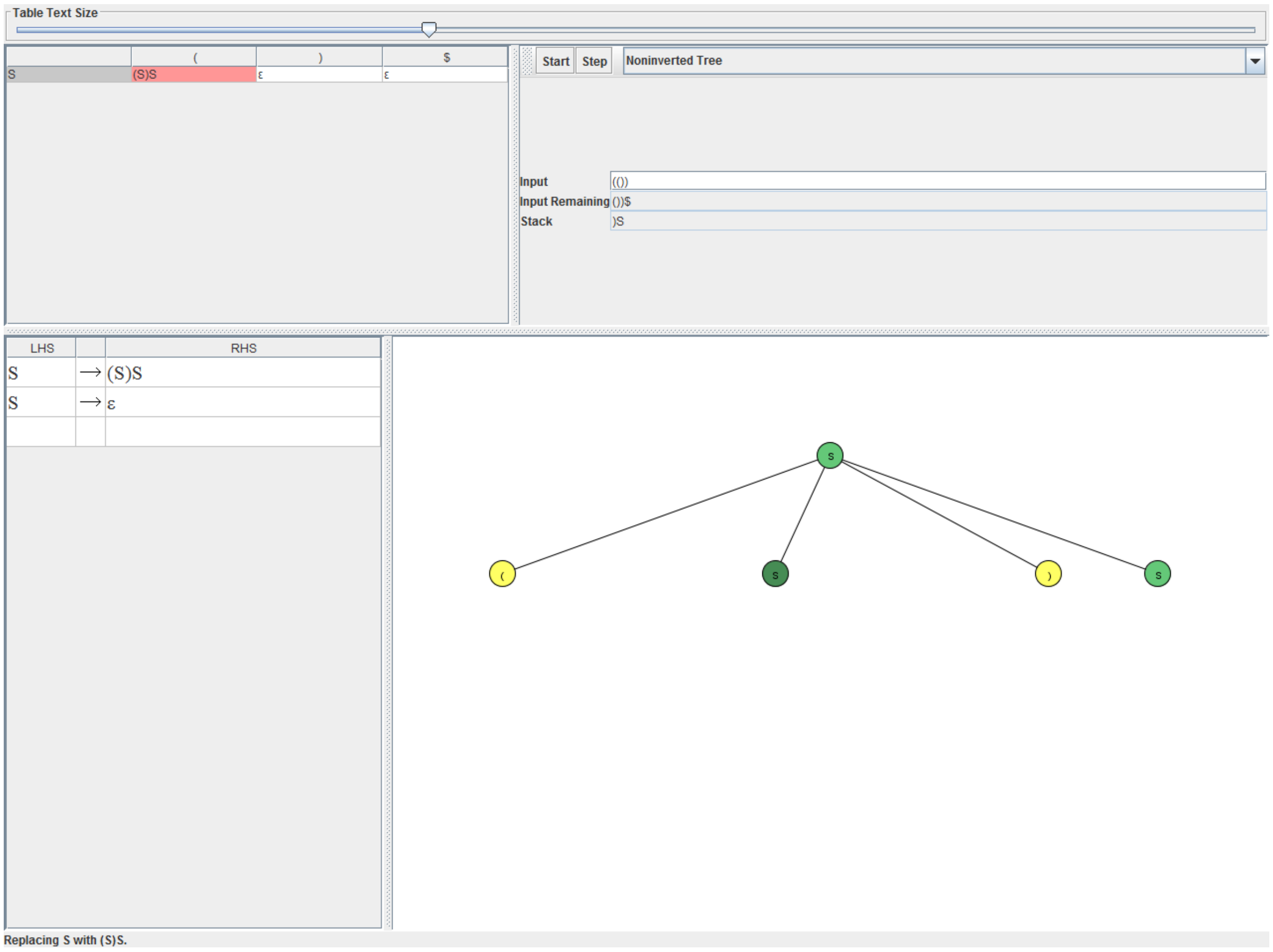
Input: (())
Input Remaining: ())$
Stack: S)S

LHS		RHS
S	→	(S)S
S	→	ε

Matched (.

JFLAP also puts a message at the bottom corner telling you that the (was matched.

Next step, the S is on the top of the stack and the next symbol in the input is the (. That means the highlighted rule will be applied and the S will be popped



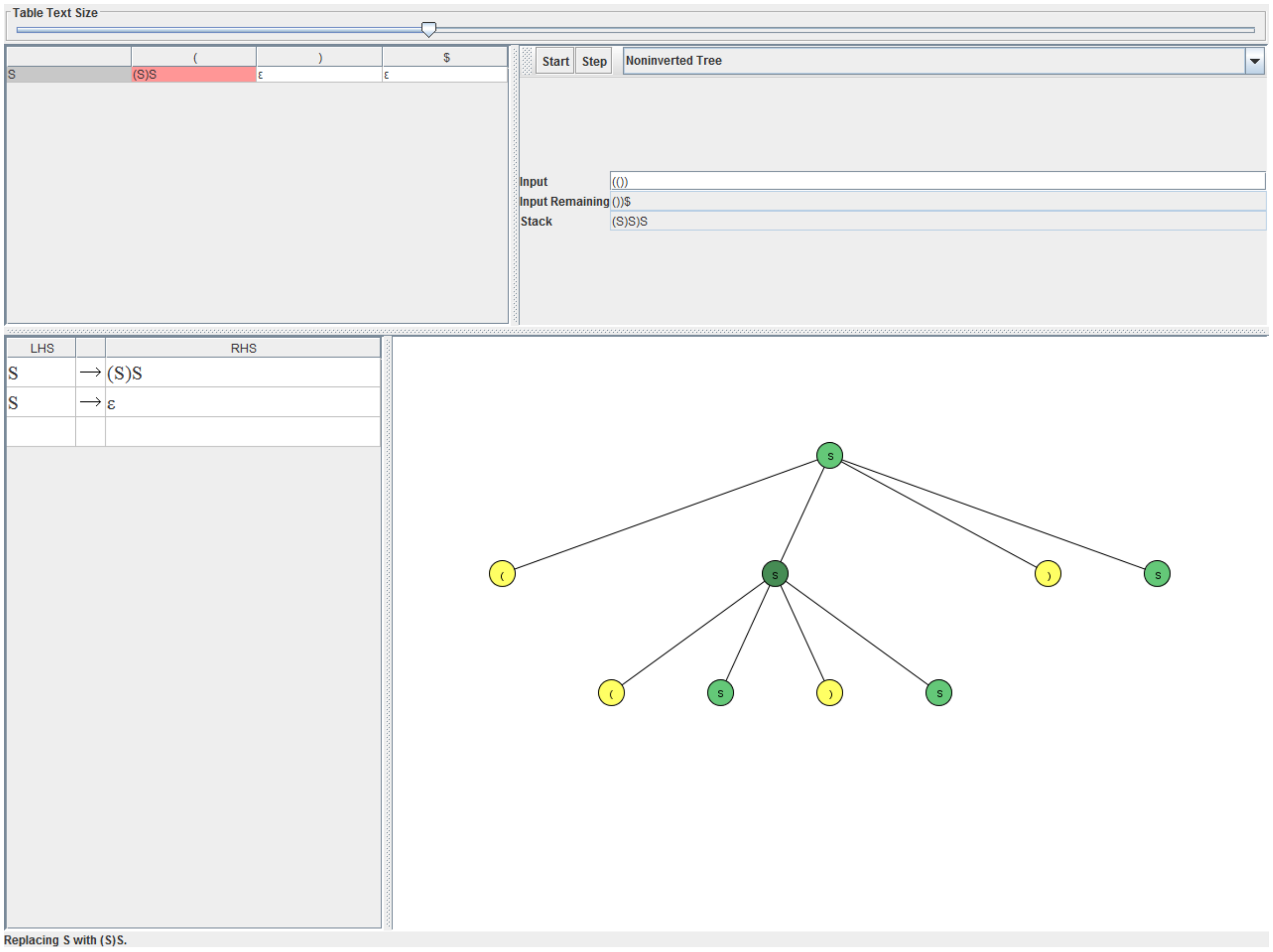
	(	)	$
S	(S)S	ε	ε

Input (())
Input Remaining ())$
Stack)S

LHS		RHS
S	→	(S)S
S	→	ε

Replacing S with (S)S.

As before, the right side of the production is pushed into the stack and the derivation tree is modified as well.



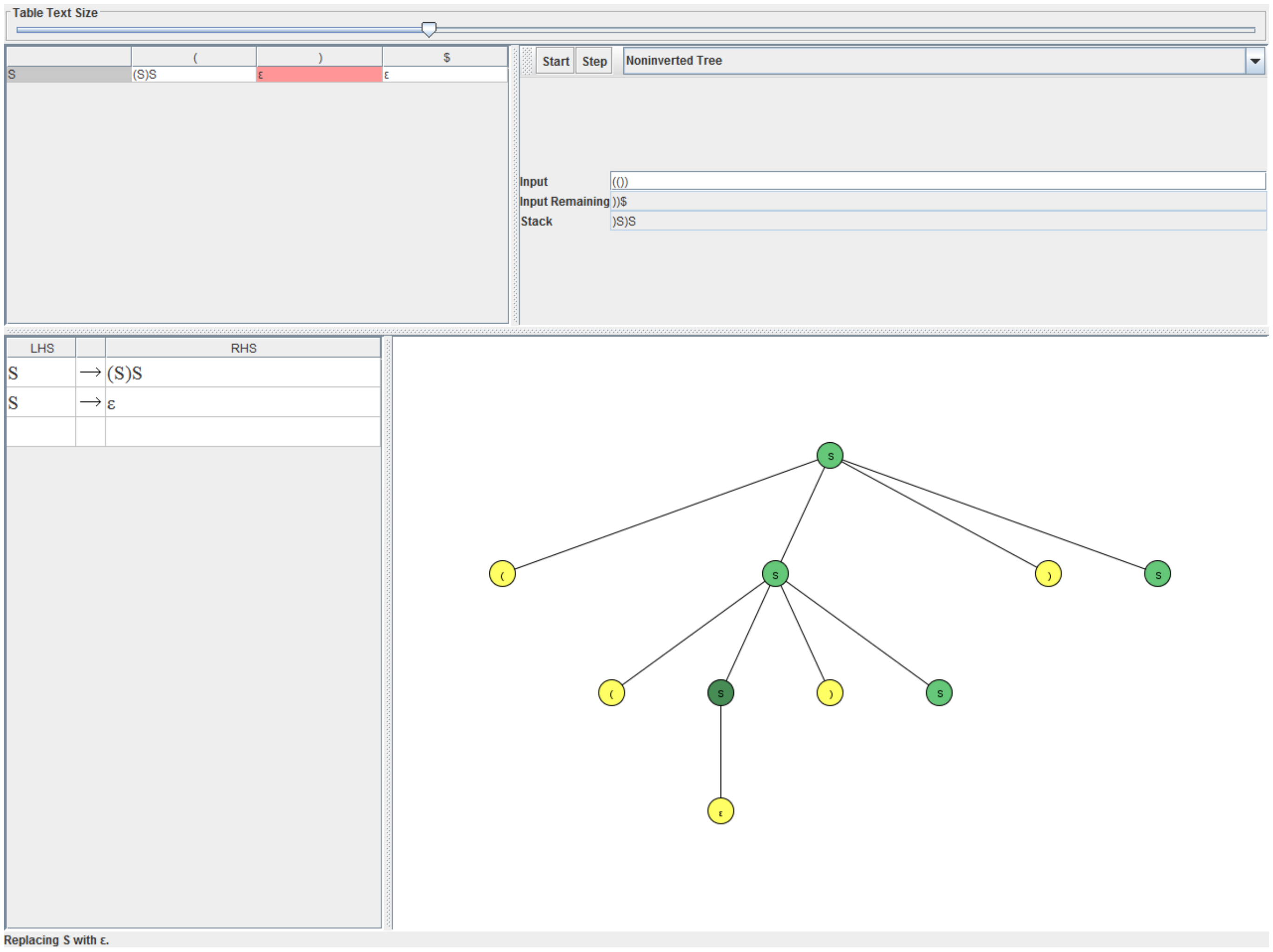
	(	)	$
S	(S)S	ε	ε

Input (())
Input Remaining ())$
Stack (S)S)S

LHS		RHS
S	→	(S)S
S	→	ε

Replacing S with (S)S.

Now after matching the) we are in a situation where there is an S on the top of the stack and the first symbol at the start of the input is a). That means we will have to use the $S \to \varepsilon$ rule as shown below.



	(	)	$
S	(S)S	ε	ε

Input (())
Input Remaining))$
Stack)S)S

LHS		RHS
S	$\longrightarrow$	(S)S
S	$\longrightarrow$	ε

Replacing S with ε.

Again the) are matched.
Then we have one more step of using the $S \rightarrow \varepsilon$ shown below.

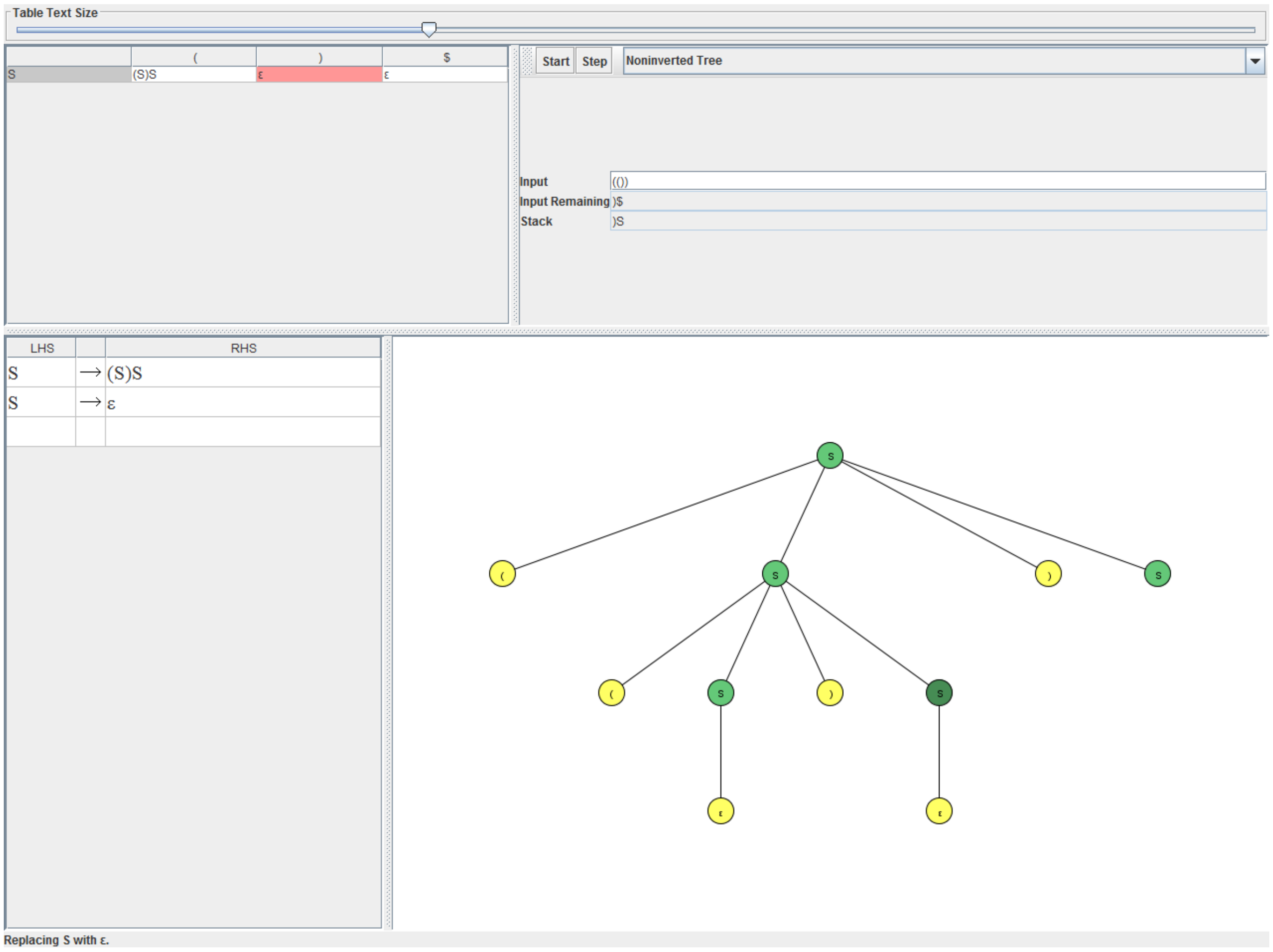


Table Text Size

	(	)	$
S	(S)S	ε	ε

Input (())
Input Remaining)$
Stack)S

LHS		RHS
S	$\longrightarrow$	(S)S
S	$\longrightarrow$	ε

Replacing S with ε.

Finally you end up with an S on the stack and the end of input symbol (the \$). That means the following highlighted rule is applied as shown below

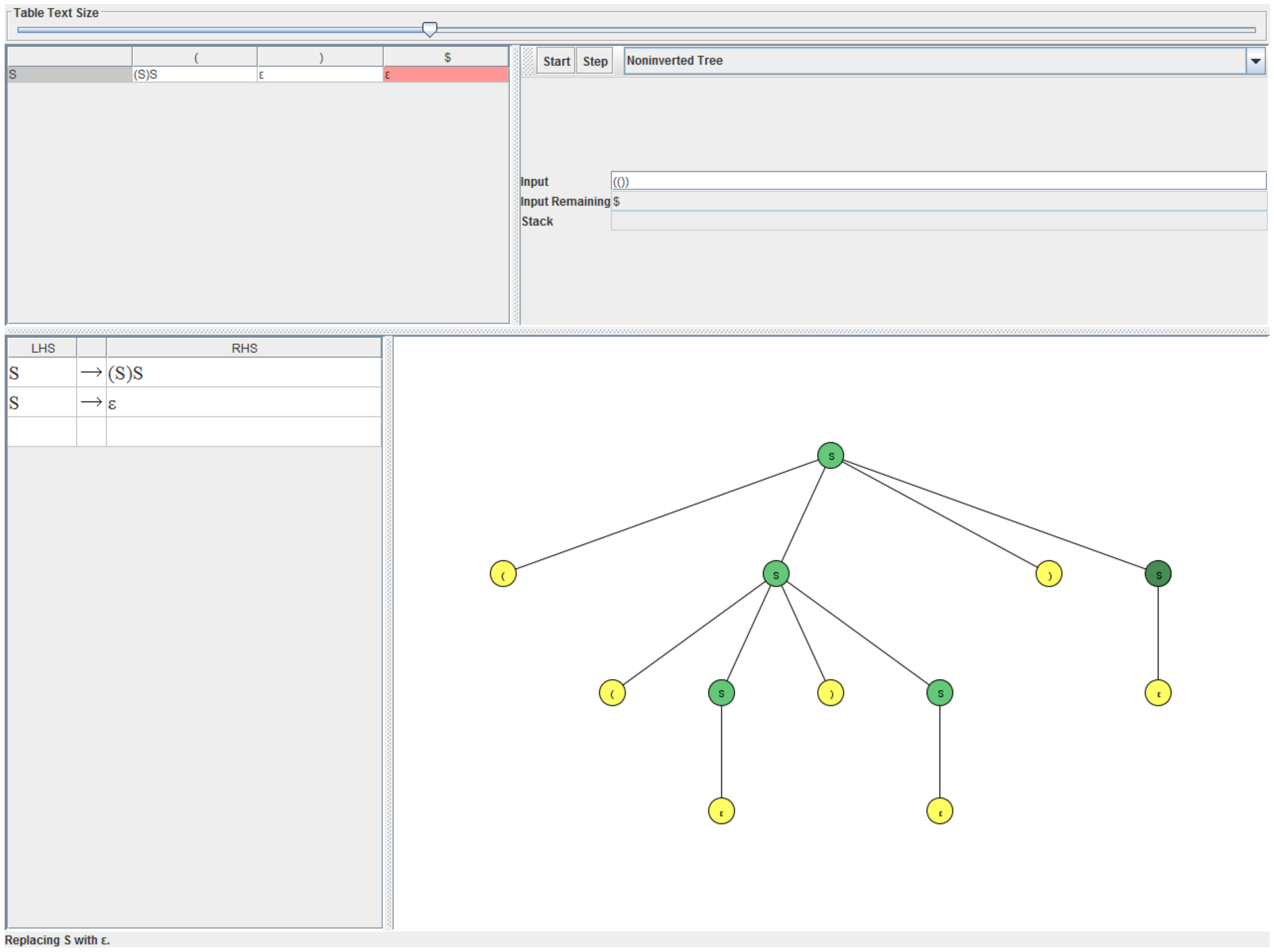


Table Text Size

	(	)	$
S	(S)S	ε	ε

Input (())
Input Remaining $
Stack

LHS		RHS
S	→	(S)S
S	→	ε

Replacing S with ε.

and that completes the derivation.

A more complicated example

Now that the basics are clear, we can try a more complicated example. Either enter these rules into JFLAP or download the file called LLParsingabd.jff.

$$
\begin{aligned}
S &\to Aa \\
A &\to BD \\
B &\to b \\
B &\to \varepsilon \\
D &\to d \\
D &\to \varepsilon
\end{aligned}
$$

Step 1 - the First sets for each of the non-terminals

For a quick example, we can see that A for instance has the possibility of generating BD. Since both B and D have an $\varepsilon rule$, the first set for A will have to include ε. Since B can produce a b and D can produce a d, this also means that b and d show up in the first set.

Here are the first sets for the remaining

$$
\begin{aligned}
B &- \{b, \varepsilon\} \\
D &- \{d, \varepsilon\} \\
S &- \{a, b, d\}
\end{aligned}
$$

Now the follow sets need to be determined. Again, just for a quick example let us look at the follow set for A. Clearly a needs to be in it thanks to the first rule. And that is the only thing that can immediately follow an A. Hence we are done with A. Fill out the others.

$$
\begin{aligned}
B &- \{d, a\} \\
D &- \{a\} \\
S &- \{\$\}
\end{aligned}
$$

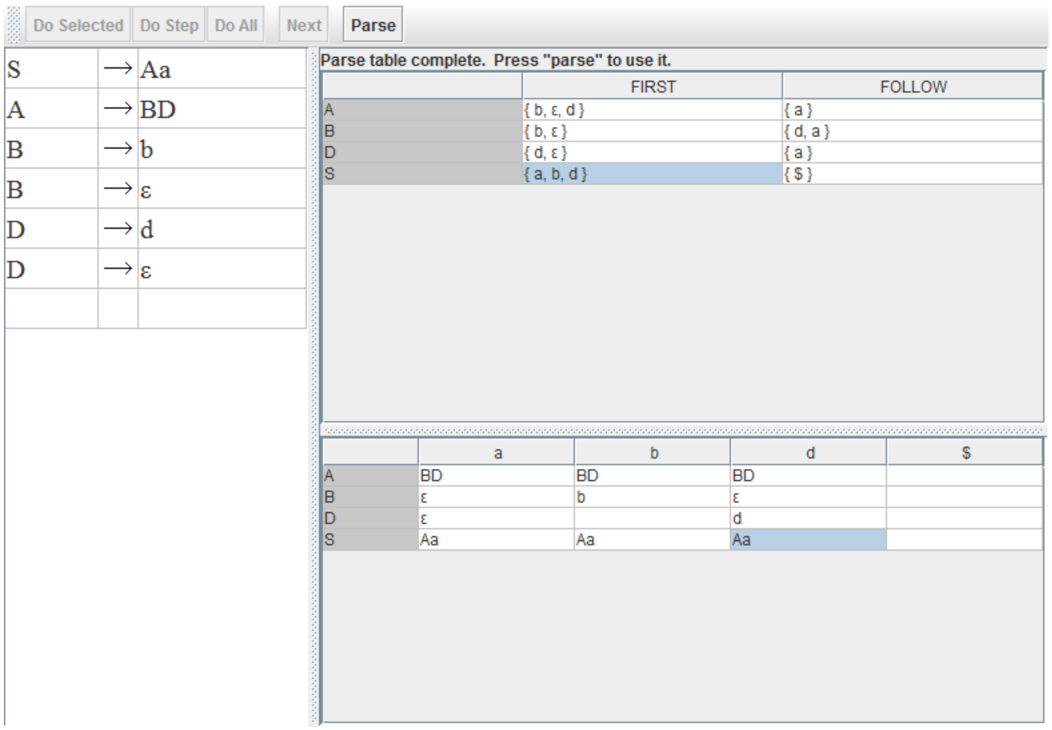
Remember that the $ sign indicates the end of input.

Now we have to build our parse table. The same idea is used. We can do it on a per production rule basis.

The first rule is $S \to Aa$. The FIRST for Aa is going to correspond to the FIRST set of A which we have computed to be $\{\underline{,}d, \varepsilon\}$. This means we will fill in Aa corresponding to the columns for both b and d. Also since A does contain an ε in its FIRST set, a will need to be added to the FIRST set of Aa. Therefore, corresponding to columns for b, d and a, we will put down Aa.

And for one rule that has an ε in its first set, if we pick the rule corresponding to $B \to \varepsilon$, we have to put ε in the columns corresponding to the terminals we find in the FOLLOW set for B. This means that ε will be put down in the columns corresponding to a and d.

Try it : - fill the rest of the parse table yourself.



S	→	Aa
A	→	BD
B	→	b
B	→	ε
D	→	d
D	→	ε

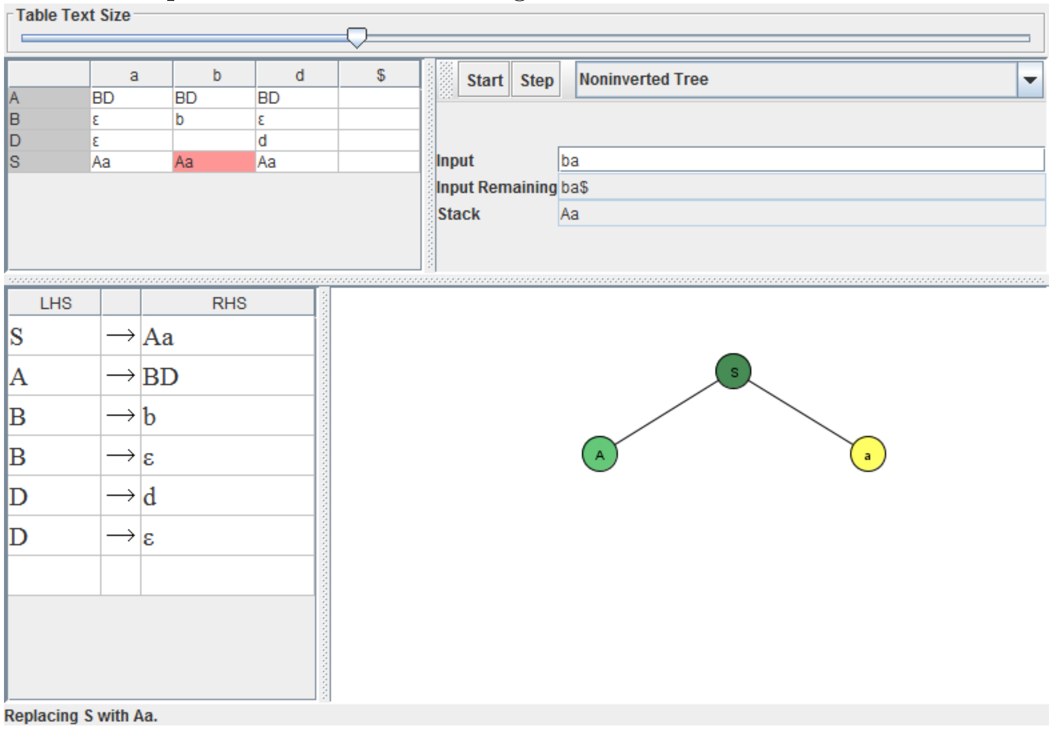
Parse table complete. Press "parse" to use it.

	FIRST	FOLLOW
A	{ b, ε, d }	{ a }
B	{ b, ε }	{ d, a }
D	{ d, ε }	{ a }
S	{ a, b, d }	{ $ }

	a	b	d	$
A	BD	BD	BD	
B	ε	b	ε	
D	ε		d	
S	Aa	Aa	Aa	

Now to try and parse a string using the rules of this grammar. Let us try to do ba.

The S upon seeing a b as the first character of the input remaining, gets popped and the derivation tree produces the Aa of the right side.



	a	b	d	$
A	BD	BD	BD	
B	ε	b	ε	
D	ε		d	
S	Aa	Aa	Aa	

Input ba

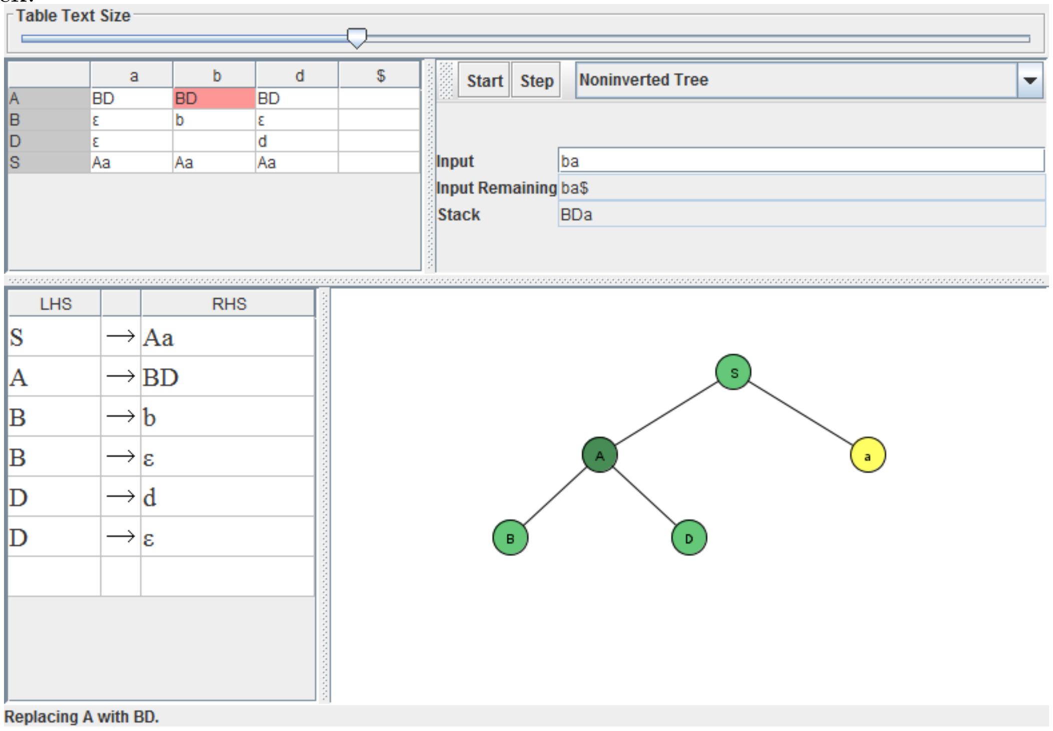
Input Remaining ba$

Stack Aa

LHS		RHS
S	→	Aa
A	→	BD
B	→	b
B	→	ε
D	→	d
D	→	ε

Replacing S with Aa.

Now the A is on top of the stack and the entry corresponding to the b column is BD. So to make that step we use the rule as follows. Note how both B and D are pushed into the stack.



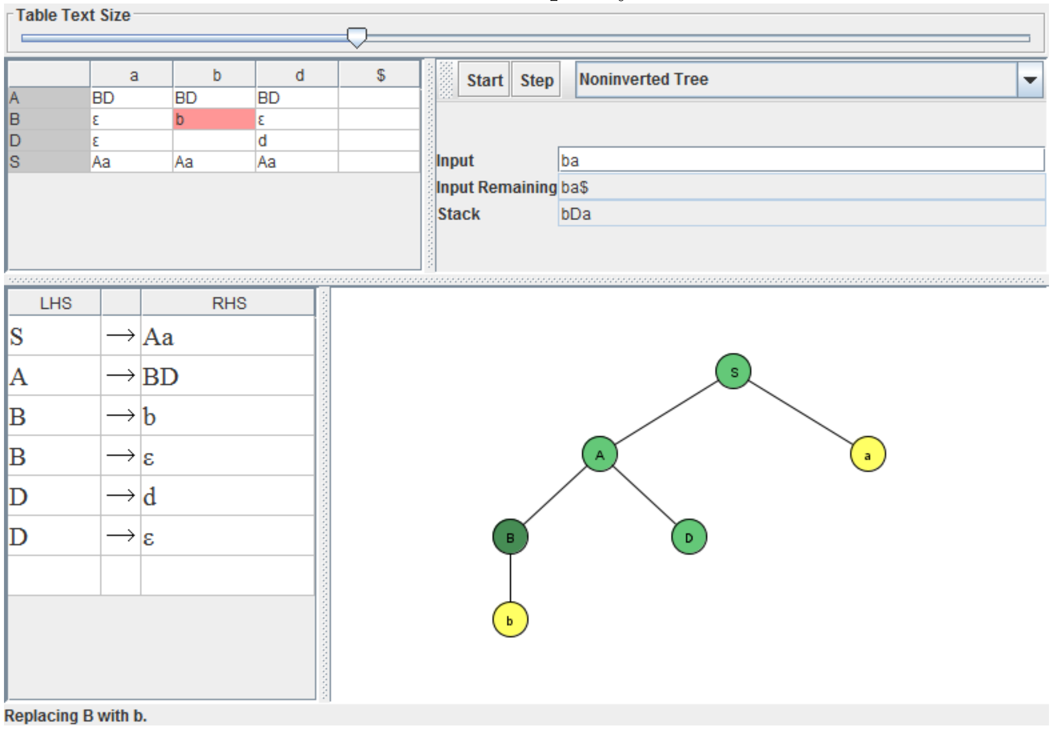
	a	b	d	$
A	BD	BD	BD	
B	ε	b	ε	
D	ε		d	
S	Aa	Aa	Aa	

Input	ba
Input Remaining	ba$
Stack	BDa

LHS		RHS
S	$\longrightarrow$	Aa
A	$\longrightarrow$	BD
B	$\longrightarrow$	b
B	$\longrightarrow$	ε
D	$\longrightarrow$	d
D	$\longrightarrow$	ε

Replacing A with BD.

The B on the stack with b as the next input symbol results in the $B \rightarrow b$ rule being used.



	a	b	d	$
A	BD	BD	BD	
B	ε	b	ε	
D	ε		d	
S	Aa	Aa	Aa	

Input	ba
Input Remaining	ba$
Stack	bDa

LHS		RHS
S	$\longrightarrow$	Aa
A	$\longrightarrow$	BD
B	$\longrightarrow$	b
B	$\longrightarrow$	ε
D	$\longrightarrow$	d
D	$\longrightarrow$	ε

Replacing B with b.

The b on the stack gets matched with the b at the start of the input which then leaves

D on the stack with an a at the beginning of the input.

When we look at the D row and the column corresponding to a, we see the $D \to \varepsilon$ rule. Apply the rule and get the following

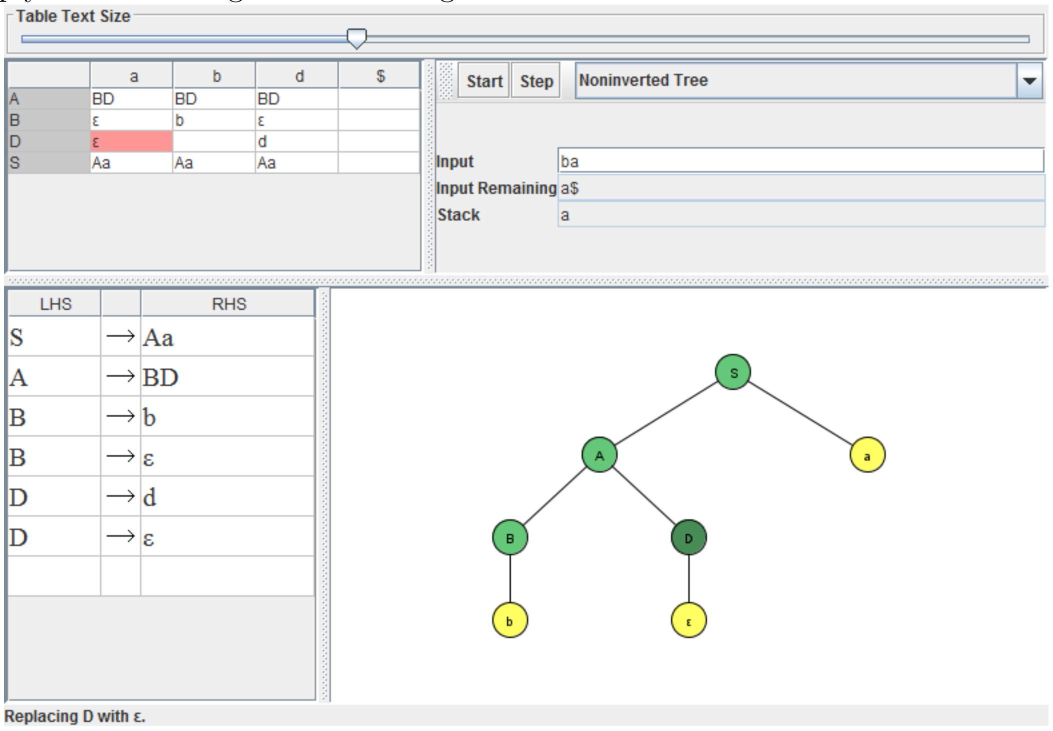


Table Text Size

	a	b	d	$
A	BD	BD	BD	
B	ε	b	ε	
D	ε		d	
S	Aa	Aa	Aa	

Start Step Noninverted Tree

Input: ba

Input Remaining: a$

Stack: a

LHS		RHS
S	→	Aa
A	→	BD
B	→	b
B	→	ε
D	→	d
D	→	ε

Replacing D with ε.

And of course, finally, the a gets matched with an a and the string is successfully parsed.

4 Questions

- Try parsing the string bda using the same grammar.
- What is wrong with trying to use LL parsing on the following grammar.

$$S \to 0S1S$$
$$S \to 1S0S$$
$$S \to \varepsilon$$