Creating a Syntax Tree

There are several ways to make a syntax tree. The procedure outlined in this handout is an adaptation of the bottom-up parsing scheme you have already seen. Starting with a token stream (the output of scanning), we can create a syntax tree and even prepare intermediate code in the following way.

1. From the grammar of the language, create a bottom-up parse table.

2. Next, we turn the language's CFG into an attribute grammar. Note that for each of the productions in the grammar, there will be an item with a cursor at the end. This will give rise to a "reduce" action. For each reduce action, we will annotate the grammar with a semantic action to create some leaf or subtree for the syntax tree. (Actually, since a leaf is already a subtree with one node, we could simply say we are creating a subtree.)

3. For some specified input, trace it according to the bottom-up parse table. Whenever you come to a reduce action, also perform the semantic action. At this point you will be creating a subtree. To keep track of this subtree for later use, push it onto a stack. So in addition to the stack of previously visited states, we'll have a stack of subtrees.

4. When finished parsing, you should have arrived in an accept (happy) state. There will be one final reduce action here. Then, empty the stack of subtrees into a linked list or a syntax tree. An example of what a syntax tree looks like is on page 202 of the textbook.

Let's illustrate the procedure with a couple examples.

Example #1 – a grammar for infix expressions, given on page 194 of the book.

E → E + T | E – T | T	follow(E) = $, +, –,)
T → T * F | T / F | F	follow(T) = $, +, –, *, /,)
F → – F | (E) | const	follow(F) = $, +, –, *, /,)

I_0	E → ● E + T	1
	E → ● E – T	1
	E → ● T	2
	T → ● T * F	2
	T → ● T / F	2
	T → ● F	3
	F → ● – F	4
	F → ● (E)	5
	F → ● const	6
I_1	E → E ● + T	7
	E → E ● – T	8
I_2	E → T ●	(–1, E) on $,+,–,)
	T → T ● * F	9
	T → T ● / F	10
I_3	T → F ●	(–1,T) on $,+,–,*,/,)
I_4	F → – ● F	11
	F → ● – F	4
	F → ● (E)	5
	F → ● const	6
I_5	F → (● E)	12
	E → ● E + T	12
	E → ● E – T	12
	E → ● T	2
	T → ● T * F	2
	T → ● T / F	2
	T → ● F	3
	F → ● – F	4
	F → ● (E)	5
	F → ● const	6
I_6	F → const ●	(–1, F) on $,+,–,*,/,)
I_7	E → E + ● T	13
	T → ● T * F	13
	T → ● T / F	13
	T → ● F	3
	F → ● – F	4
	F → ● (E)	5
	F → ● const	6
I_8	E → E – ● T	14
	T → ● T * F	14
	T → ● T / F	14
	T → ● F	3
	F → ● – F	4
	F → ● (E)	5
	F → ● const	6
I_9	T → T * ● F	15
	F → ● – F	4
	F → ● (E)	5
	F → ● const	6
I_{10}	T → T / ● F	16
	F → ● – F	4
	F → ● (E)	5
	F → ● const	6
I_{11}	F → – F ●	(–2, F) on $,+,–,*,/,)
I_{12}	F → (E ●)	17
	E → E ● + T	7
	E → E ● – T	8

I_{13}	E → E + T ●	(–3, E) on $,+,–,)
	T → T ● * F	9
	T → T ● / F	10
I_{14}	E → E – T ●	(–3, E) on $,+,–,)
	T → T ● * F	9
	T → T ● / F	10
I_{15}	T → T * F ●	(–3, T) on $,+,–,*,/,)
I_{16}	T → T / F ●	(–3, T) on $,+,–,*,/,)
I_{17}	F → (E) ●	(–3, F) on $,+,–,*,/,)

So here is the parse table:

State	+	–	*	/	(	)	const	$	F	T	E
0		4			5		6		3	2	1
1	7	8									
2	–1,E	–1,E				–1,E		☺			
3	–1,T	–1,T	–1,T	–1,T		–1,T		–1,T			
4		4			5		6		11		
5		4			5		6		3	2	12
6	–1,F	–1,F	–1,F	–1,F		–1,F		–1,F			
7		4			5		6		3	13	
8		4			5		6		3	14	
9		4			5		6		15		
10		4			5		6		16		
11	–2,F	–2,F	–2,F	–2,F		–2,F		–2,F			
12	7	8				17					
13	–3,E	–3,E	9	10		–3,E		☺			
14	–3,E	–3,E	9	10		–3,E		☺			
15	–3,T	–3,T	–3,T	–3,T		–3,T		–3,T			
16	–3,T	–3,T	–3,T	–3,T		–3,T		–3,T			
17	–3,F	–3,F	–3,F	–3,F		–3,F		–3,F			

On page 194 in the book you can find the 9 productions of the grammar along with their corresponding semantic actions. I've summarized them here.

State	Reducing what	Semantic action	Real change to tree?
13	E_1 → E_2 + T ●	make_subtree(+, E_2, T)	Yes
14	E_1 → E_2 – T ●	make_subtree(–, E_2, T)	Yes
2	E → T ●	copy: E = T	No
15	T_1 → T_2 * F ●	make_subtree(*, T_2, F)	Yes
16	T_1 → T_2 – F ●	make_subtree(/, T_2, F)	Yes
3	T → F ●	copy: T = F	No
11	F_1 → – F_2 ●	make_subtree(±, F_2)	Yes
17	F → (E) ●	copy: F = E	No
6	F → const ●	create_leaf(const)	Yes

Now, let's create a syntax tree for the expression – (7 + 8). We trace according to the parse table, and each time we have a reduce operation, we will also perform the corresponding semantic action to create a subtree. But in some cases, the semantic action won't really change the tree. In the input column, I didn't draw the cursor. Instead I've shown the input as being "consumed" so that the cursor is at the beginning of the remaining input expression you see.

state stack	input	goto	subtree stack
0	– (7 + 8)	4	
0 4	(7 + 8)	5	
0 4 5	7 + 8)	6	
0 4 5 6	+ 8)	–1, F	Look up rule 6: create leaf for "7"; push this "7" node
0 4 5	F + 8)	3	
0 4 5 3	+ 8)	–1, T	Look up rule 3: no change

0 4 5	T + 8)	2	
0 4 5 2	+ 8)	–1, E	Look up rule 2: no change
0 4 5	E + 8)	12	
0 4 5 12	+ 8)	7	
0 4 5 12 7	8)	6	
0 4 5 12 7 6	)	–1, F	Rule 6: create leaf for "8"; push this "8" node
0 4 5 12 7	F)	3	
0 4 5 12 7 3	)	–1, T	Rule 3: no change
0 4 5 12 7	T)	13	
0 4 5 12 7 13	)	–3, E	Rule 2: no change
0 4 5	E)	12	
0 4 5 12	)	17	
0 4 5 12 17	$	–3, F	Rule 17: pop; pop; make_subtree(+, 7, 8); push
0 4	F $	11	
0 4 11	$	–2, F	Rule 11: pop; make subtree(±, (+, 7, 8)); push
0	F $	3	
0 3	$	–1, T	Rule 3: no change
0	T $	2	
0 2	$	☺	Rule 2: no change

Don't forget that a happy state is also a reduce, and we could be happy in states 2, 13 or 14 because of reducing the start symbol. In this example, when we are done parsing, there is just one tree on the stack: we pop it and that is our syntax tree.

If we can handle infix expressions, we can also do prefix or postfix expressions. In fact, a prefix or postfix expression would be simpler because they don't need parentheses. Fewer tokens, fewer states. We may do something along these lines in lab.

Example #2 – to illustrate a list of things, rather than just one expression. This time, we need to be able to store different subtrees in some sort of list. (This is to facilitate the left-to-right or top-down propagation of symbol table information, should we want to check variable types.) Let's return to the grammar of variable declarations that we saw in labs 1 and 2.

D → int L ;
L → V , L | V
V → id = num

We have already written the sets of items and the parse table when we did lab #2. And we've had some practice parsing it, so let's skip to the part where we create the various subtrees at the reduction operations.

The semantic actions can be defined as follows:

D → int L ; • Just copy the L subtree into D, which means the subtree stack is essentially unchanged.

L → V , L • Pop the V subtree and L subtree, and create a sequence (linked list) with the V subtree followed by the L subtree. This will look like the sequence of consecutive statement nodes in the figure on page 202 in the book.

L → V • Just copy V's subtree into L, which leaves the subtree stack essentially unchanged.

V → id = num • Create a subtree with = as the root and the id and num leaves as child nodes.

Let's suppose the input is "int a = 5 , b = 6 ;". During the parse, here are the reductions we encounter:

1. Upon reading a = 5, we are using item V → id = num • . Create a subtree with = as the root. Push.
2. Upon reading b = 6, so we create another subtree with = as root. Push.
3. When we reduce L → V • the subtrees don't change.
4. When we reduce L → V , L ; • we pop the 2 subtrees and create a sequence of declarations. Push.
5. Finally, when we are done parsing, we just need to copy L's subtree into D, the start symbol. This is the syntax tree.