Lecture 8 addendum
CS 421, Spring 2010

This note discusses two points related to lecture 8. First, it gives a more elaborate example of FIRST and FOLLOW sets than what we did in class. Then it gives an explanation of the term "LL(1);" this is solely for your edification and will not be covered on the exam.

1 FIRST and FOLLOW sets

This example is from http://www.jambe.co.nz/UNI/FirstAndFollowSets.html. It is a standard stratified grammar, left-factored to make it LL(1) (it is a slightly expanded version of grammar G_F from lecture 8).

The grammar is:

E → T E'
E' → + T E' | ϵ
T → F T'
T' → * F T' | ϵ
F → (E) | id

As in class, we will construct FIRST and FOLLOW sets by inspection, rather than by following a precise algorithm. (The webpage cited above, as well as many others and any compiler textbook, shows how to calculate these sets precisely.)

For FIRST sets, we can make some immediate observations:

FIRST(F) = { (, id }
FIRST(E') = { +, • }
FIRST(T') = { *, • }

FIRST(T) will include any tokens in FIRST(F). Furthermore, since FIRST(F) does not contain •, and since T has just one production and it starts with F, it is clear that FIRST(T) is equal to FIRST(F). Similarly, FIRST(E) is the same as FIRST(T). Thus,

FIRST(E) = { (, id }
FIRST(T) = { (, id }

FOLLOW sets are more difficult to calculate by inspection, but if we write down enough derivations, we can get them. (You will not be responsible to calculate FOLLOW sets as complicated as this one; the point of this example is so that you understand what FIRST and FOLLOW sets are and how they are used in constructing top-down parsers.)

When a non-terminal can occur as the very last symbol in a sentential form, we say that "eof" is in its FOLLOW set; thus, eof is always in the FOLLOW set of the start symbol (E).

Here are two (partial) derivations in this grammar:

E ⇒ TE' ⇒ T ⇒ FT' ⇒ (E)T' ⇒ (TE')T' ⇒ (T)T' ⇒ ...
E ⇒ TE' ⇒ T+TE' ⇒ FT'+TE' ⇒ F+TE' ⇒ F+FT'E' ⇒ F+F*FT'E' ⇒ ...

Remember, the rule is simple: a token t is in FOLLOW(A) if it occurs immediately after A in any sentential form; "eof" is in FOLLOW(A) if it occurs at the end of any sentential form. Thus, from these two partial derivations, we can read off the following:

FOLLOW(E) $\supseteq$ { eof,) }
FOLLOW(E') $\supseteq$ { eof,) }
FOLLOW(T) $\supseteq$ { eof, +,) }
FOLLOW(T') $\supseteq$ { eof, + }
FOLLOW(F) $\supseteq$ { +, * }

In fact, here are the correct FOLLOW sets:

FOLLOW(E) = { eof,) }
FOLLOW(E') = { eof,) }
FOLLOW(T) = { eof, +,) }
FOLLOW(T') = { eof, +,) }
FOLLOW(F) = { +, *,), eof }

I'll leave it as an exercise to find derivations that show ')' and eof are in FOLLOW(F), and ')' is in FOLLOW(T').

Now that we've got our FIRST and FOLLOW sets, we can verify that this grammar is LL(1). We can pretty easily see that it is not left-recursive. We need to verify that we can always decide which production to use for any non-terminal, given just one token of lookahead. For E, T, and F, this is obvious: E and T each have only one production, and the two productions for F start with distinct tokens. For E' and T', we first have to verify that there is just one nullable production from each; this is clear, because in both cases the non-epsilon production begins with a token (so is obviously not nullable).

We then have to check just one fact for each of E' and T'. Consider E'. The question is this: If we see a "+" as the lookahead symbol, which production should we use? The first is clearly applicable, but the second (the epsilon production) might also be applicable, *if "+" were in FOLLOW(E').* Thus, we need to verify that "+" is not in FOLLOW(E'); it is not. Similarly, for T', we need to verify that "*" is not in FOLLOW(T'), and it is not.

So the grammar is LL(1). For each parsing function, the choice of right-hand side is clear: for E and T, there is just one choice; for F, check for "(" or id; for E', a "+" indicates the first production, and anything else — or, technically, anything in FOLLOW(E') — indicates the epsilon production; for T', similarly, but for "*" instead of "+".

2 Why LL(1)?

First, the easy parts: The first L refers to the fact that parsing occurs during a left-to-right scan of the input; the "1" means that parsing decisions are based only on the next input token; if these decisions were based on the next *two* input tokens, and then we would call these grammars LL(2).

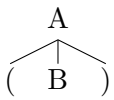
Now the hard part: the second L. This stands for "leftmost derivation." Pretend that as a recursive-descent parse proceeds, it constructs the parse tree behind the scenes, in the following way: There is a global parse tree, which starts with a single node labeled with the grammar's start symbol. At any point in the parse, when we call `parseA`, for any non-terminal A, there is a particular occurrence of A in the parse tree, and we are looking to expand the tree by adding children to that node. *As soon as we decide which production from A to use*, we add that right-hand side as children of that node. We then handle each of its symbols in turn, in the same way. Each time we make a decision about what production to use for a given non-terminal, we add it to the tree.

For example, consider the grammar

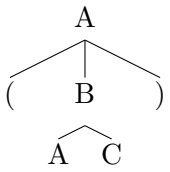
A → id | (B)
B → A C
C → + A C | epsilon

and input ((x)).

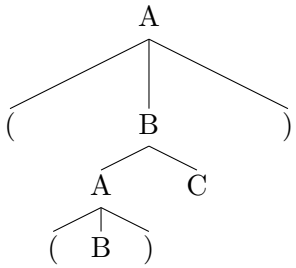
We start with tree A and immediately realize that we need to use the second production for A, so we add (, B, and) as children of A:



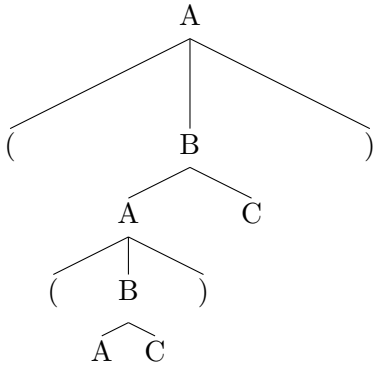
We match the first (, then call parseB. parseB realizes — it has no choices — that it has to use production B → A C, so it adds that to the tree:



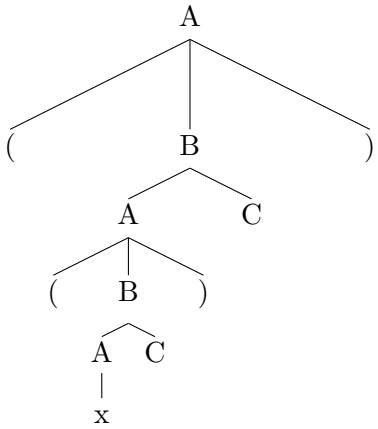
and then calls parseA. parseA again decides on the production A → (B), and adds this to the tree:



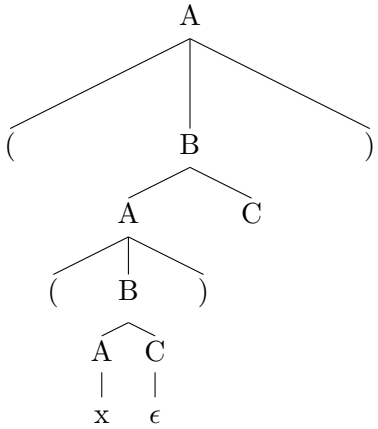
It again matches the (and calls parseB, which again expands the tree



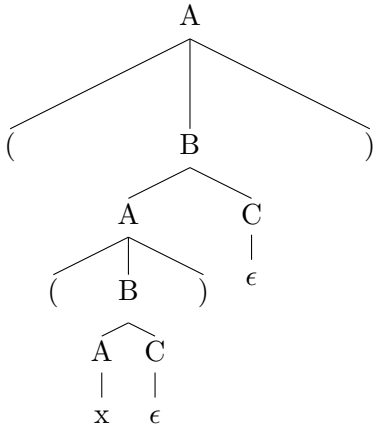
and calls parseA. This time parseA matches the id and adds that:



parseA returns to parseB, which calls parseC. C doesn't see a +, so it uses the epsilon production:



We continue, eventually calling parseC again, which fills in the final part of the parse tree:



We're done. Now let's write down the *frontiers* of these parse trees at each step:

A ⇒ (B) ⇒ (A C) ⇒ ((B) C) ⇒ ((A C) C) ⇒ ((x C) C) ⇒ ((x) C) ⇒ ((x)).

This is a derivation sequence, as defined in class. What is notable about this derivation sequence is that, at each point, the *leftmost* non-terminal is the one that is replaced by a right-hand side of a production. It is characteristic of top-down parsing that the order in which it decides on what productions to use mimics a derivation with this property — a so-called leftmost derivation. The second "L" in "LL(1)" stands for "leftmost derivation."