We have understood the two cornerstone concepts required to understand Parsing - the First set and the Follow set. Many concepts hinge on these fundamentals.

We shall now try to understand one of the most important top-down parsing algorithm: LL(1) Parsing. The first L indicates the reading direction (**L**eft to right), the second L indicates the derivation order (**L**eftmost derivation) and the 1 indicates that there is a one-symbol look ahead. A grammar that can be parsed using LL(1) parsing technique is called an LL(1) grammar.

The idea here is to be able to choose the correct production rule at every step of the top-down production, WITHOUT BACKTRACKING!! LL(1) parsing is a predictive parsing technique that aims to parse a string without requiring to backtrack, and hence it does parsing in linear time. However, no left-recursive or ambiguous grammar can be LL(1). This means that LL(1) parsers are not all-powerful; they cannot parse absolutely any given grammar. They are only able to parse certain restricted subset of the Context-Free Grammars.

Because this is a top-down parsing algorithm, we start with the start symbol of the given grammar, and at each step, we must choose the correct production rule to be applied so that we ultimately get the desired string.

Now just imagine yourself to be the compiler (or at least the parser). Let us say you were given a grammar such as:

S -> aA | bB

A -> bc

B -> cd

And you were given the string w = abc to parse. Now what would you do? You will first of all start off with S. Then, at this point, you must be able to determine which of the two productions to choose: S -> aA or S -> bB. How do you do this? You look at what is your first symbol in w. It is a. Now, it is logical that you cannot choose the production S -> bB, because if you chose this production, then, the first terminal symbol that you would derive would be b and not a. Hence, the only option left for you is to choose S -> aA. Hence, you choose this rule. Thereafter, things are easy. You have S => aA => abc . Success!

Let us consider another grammar instead:

S -> aA | Bb

A -> bc

B -> ac

Now, you start off with S. At this point you have a choice to make: either choose S -> aA or choose S -> Bb. Well, now you know that if you chose S -> aA, then, the first symbol that you will derive will be a, which matches the first symbol of w. But, what about the production Bb? You are now supposed to determine whether or not it is a valid choice to choose this production. How do you do this? Obviously it would help making your decision if you knew what was the first symbol derived from this production, so that you could see if this symbol happened to be the same as the first input symbol. If the terminal a is not a possible first symbol derivable from S -> Bb, then you know for sure that choosing this production would be a wrong move! Therefore, at this point, you have to find out FIRST(Bb)! What do you get? Clearly, FIRST(Bb) = {a}. What just happened? You will notice that the very first symbol derivable by choosing this production is a. Also, at the same time, the very first symbol derivable by choosing the production S -> aA is also a. So, at this point, you cannot deterministically tell whether to choose the production S -> aA or S -> Bb. THIS LEADS US TO ONE IMPORTANT RULE:

**For a grammar to be LL(1), it MUST be the case that whenever we have productions like X -> (alpha) | (beta) | (gamma) | … then, the FIRST sets of (alpha), (beta), (gamma), etc. must all be disjoint!!!!!! In other words, there cannot be common elements in the first sets of each of the RHS. (Here, alpha, beta, gamma, etc. denote arbitrary sequence of grammar symbols, be it terminals or non-terminals).**

Hence, the above grammar is not an LL(1) grammar, and therefore, we cannot make an LL(1) parser for it!!!

Let us now consider another grammar:

S -> Abc

A -> B | C

B -> bd

C -> lambda

Let us assume we are given the string w = bc. Now, as usual, we start with S. Here, we choose the only production: S -> Abc. Now, we need to expand A. But, here, we have a choice: Either choose A -> B or A -> C. Well, if you were a parser, what would you do? You would try to see which of the two rules will help you match the first symbol. Therefore, you find FIRST(B) and FIRST(C). Clearly, FIRST(B) = {b} and FIRST(C) = {lambda}. Well, at this point, what do you do? Can you say that because FIRST(B) has got the terminal ‘b’ in it (the one we are looking for), therefore, we must choose A -> B as our next choice? Well, if we did, then, what would happen? We would get: S => Abc => Bbc. Now, B only has got one rule: B -> bd. If we applied this: Bbc => bdbc. Clearly, this was the wrong choice we made!!! Thus, just because we saw a desired symbol in FIRST(B), we should not take that production. Because the other production had lambda! If the other production did not have lambda, then, it would have been fine to choose A -> B. But, because FIRST(C) = {lambda}, therefore, it could be possible that we make this choice, thereby making C vanish. In this case, what matters is the symbol that follows A. (By the way, the correct way to obtain the desired string is as follows: S => Abc => Cbc => bc). But, how can we be sure that choosing A -> C is the right thing to do? This is where the concept of look ahead comes!! At this point, we are in a dilemma, whether to choose A -> B or A -> C in the sentential form: Abc. We know that FIRST(B) = {b} and FIRST(C) = {lambda}. We do not know whether we should choose A -> B (this looks very promising because the first symbol that B derives matches the first symbol of w), or we could choose A -> C (though this does not appear very promising, it could still be possible that by making C vanish into lambda, the next symbol that appears could potentially be the first symbol that we hope to match). Thus, there is only one way to find out! We need to find out what appears in the FOLLOW(A). This is where we are doing the look ahead. We look one symbol ahead to see what we can get. Clearly, FOLLOW(A) = {b}. Now, we are in deep soup!! If FOLLOW(A) did not contain the terminal ‘b’, then, we would have known for sure that we cannot choose A -> C, because by choosing A -> C, there would have been no way to derive terminal ‘b’, which was the first expected symbol that we hope to match. If FOLLOW(A) did not contain b, then we would have very deterministically told that we must choose A -> B only because this is our only hope of getting a ‘b’ as the first symbol. But, what has happened here? We observe that FOLLOW(A) contains ‘b’. What does this tell us? We now know that if we were to choose the production A -> B, then we would get ‘b’ as the first terminal symbol. We also now know that if we were to choose A -> C, then also we would get ‘b’ as the first terminal symbol. Clearly, at this point, we cannot make a deterministic choice of which production to take! THIS LEADS US TO THE NEXT IMPORTANT RULE:

**For a given grammar to be LL(1), then, for any production of the form A -> (alpha) | (beta) | … if FIRST(beta) contains lambda, then, (alpha) should not derive any string beginning with a symbol in FOLLOW(A). In other words, if FIRST(beta) contains lambda, then, FIRST(alpha) and FOLLOW(A) MUST be disjoint sets. Similarly, if FIRST(alpha) contains lambda, then, FIRST(beta) and FOLLOW(A) must be disjoint sets.**

In other words, the above grammar is NOT LL(1) grammar!!!

Note that if we slightly modified the above grammar so that:

S -> Abc

A -> B | C

B -> cd

C -> lambda

Now, if we are asked to parse w = bc, then, we first choose S -> Abc. Now, at this point we have a choice to make: either choose A -> B or choose A -> C. We have FIRST(B) = {c} and FIRST(C) = {lambda}. Now, because C derives lambda, we need to look one symbol ahead to see what symbol comes after A. So, FOLLOW(A) = {b}. Now, we can make a very deterministic choice! We know for sure that if we choose A -> B, then, we are surely not going to get b. Also, we know for sure that if we choose A -> C, then, we can make the C vanish (by choosing C -> lambda), and eventually, the symbol that appears next is a ‘b’ - the symbol we are looking for. Thus, we choose A -> C. Notice that our earlier problem is now eliminated ALL because FOLLOW(A) does not have any common elements with FIRST(B)!

At this point, it is important to mention that we have seen a few ways to conclude that a given grammar is not LL(1):

1. If the grammar is left recursive.

2. If the grammar is ambiguous.

3. If the first sets of the RHS of any production rule are not disjoint. (also called First-First Conflict)

4. The First-Follow Conflict (the last case we discussed above).

**PROBLEMS:**

Which of the following are LL(1) grammars and which are not? Justify.

1.

S -> 0S1 | 01

2.

S -> SS+ | SS\* | a

Here, {+, \*, a} is the set of terminals, while {S} is the set of non-terminals.

3.

E -> TE’

E’ -> +TE’ | lambda

T -> FT’

T’ -> \*FT’ | lambda

F -> (E) | i

Here, the set of non-terminals is: {E, E’, T, T’, F}, while the set of terminals is: {+, \*, (, ), i}

4.

S -> iEtS | iEtSeS | a

E -> b

Here, the set of non-terminals is: {S, E}, while the set of terminals is: {i, t, e, a, b}.

5.

S -> iEtSU | a

U -> eS | lambda

E -> b

6.

S -> aABb

A -> aAc | lambda

B -> bB | c

7.

S -> aSc | T | lambda  
T -> bTc | lambda

8.

A -> BAa | lambda  
B -> bBc | AA

9.

S -> aAa | bS  
A -> BB | C   
B -> bC  
C -> B | c | lambda

10.

S -> LB  
B -> aSaL | bL  
E -> c | L  
J -> dEJ | f  
L -> eEJ

11.

T -> R | aTc  
R -> lambda | RbR