There are basically two broad categories of parsing algorithms (and hence correspondingly parsers): Top-Down Parsing Algorithms (the parsers that use these are called Top-Down Parsers), and Bottom-Up Parsing Algorithms (the parsers that use these are called Bottom-Up Parsers). The First Parsing Algorithm that we studied, CYK Parsing, was a Bottom-Up Parsing Algorithm. In case of a top-down parsing algorithm, we start with the start symbol of the grammar and try to work our way downwards to see if the given string can eventually be reached or not. In Bottom-Up Parsing Algorithms, it is the reverse. We start with the given string and see if we can reduce this string to S (or the start symbol of the given grammar). What we will look at for the purposes of this assignment is a Top-Down Parsing Algorithm known as LL(k) Parsing.  
 In fact, the whole topic of Parsing is a very complicated one (be it Top-Down or Bottom-Up). Therefore, I do not hope to cover the entire intricacies of LL(k) parsing in one document. It shall be spread out over the next few documents.  
 To understand the LL(k) Parsing Algorithm, it is MOST CRUCIAL to understand two essential concepts. In fact, if I were to choose the most toughest and also important concepts of this course, I would say that it has to be these: the FIRST set and FOLLOW set.  
 I shall first deal with the FIRST set in this document, leaving the concept of FOLLOW for later, since I wish to ensure that these essential concepts are firmly understood before we proceed. Ok, enough story! Onto FIRST!  
  
 Consider a grammar such as follows:  
S -> cAd  
A -> ab | c  
  
 We define FIRST as follows (very crudely): The FIRST set of a given sequence of symbols is the set of all the first terminals that this sequence can generate. Importantly, note that this sequence can consist of terminal symbols as well as non-terminal symbols.  
 In case you have got confused by this definition already, some examples may help clarify. What this essentially states is that given some sequence of symbols, such as XYZ, the FIRST set of this sequence, that is, FIRST(XYZ) is the set of all first terminal symbols that we can generate from XYZ.  
 For example, let us find FIRST(abcd). To do so, ask yourself the question, "What is the very first terminal symbol that can be generated from this sequence?" All the answers that you get will constitute the FIRST set! In this case, what is the very first terminal symbol that can be generated from abcd? Obviously, the answer is a. (Here, we shall always assume that lowercase symbols represent the terminals while uppercase symbols represent non-terminals). The very first terminal symbol is a. Therefore, FIRST(abcd) = {a}.  
 How about: FIRST(S)? Again, you ask, "What is the very first terminal symbol that can be generated by S?" Now, we look at the grammar rules for S. Observe that we have a rule: S -> cAd. Thus, the very first terminal symbol that can possibly be generated by S is c. Therefore, FIRST(S) = {c}.  
 How about: FIRST(aS)? Remember what I told earlier? We can find the first of any sequence of symbols, be it terminals or non-terminals, or a mix of both. Thus, we ask, "What is the very first terminal symbol that can be generated from aS?" Obviously, the answer is a. This is because a is a terminal, and because it appears as the very first symbol in the sequence, therefore, the first terminal that this sequence (aS) derives will be a. Therefore, FIRST(aS) = {a}. In general, therefore, we can conclude the following: FIRST(kXYZ...) = {k}, where k is a terminal symbol and XYZ, ... can be either terminals or non-terminals. In other words, if our given sequence of symbols starts with a terminal symbol, then, regardless of whatever else follows in that sequence, we can blindly say that the FIRST set of that sequence is that terminal itself!  
  
 What about: FIRST(A)? We have to find out, "What is the first terminal symbol that A can derive?" Looking at the productions for A, we observe that we can have either the derivation: A => ab, OR we can have: A => c. Thus, we notice that A can derive two strings, one of which begins with the terminal 'a', while the other begins with the terminal 'c'. Hence, FIRST(A) = {a,c}. Remember that FIRST is a SET!!!! It represents the SET of all possible first terminals derived from the sequence.  
  
 What about: FIRST(ASAA)? The trick in finding out the FIRST set is to effectively consider all the possible derivations that can be made from the very first symbol in the sequence. Thus, in this case, we have to find out what is the very first terminal symbol derivable from ASAA. We have ASAA => abSAA OR ASAA => cSAA. What we have done here is basically replace the leftmost symbol of the given sequence (ASAA) with the right hand sides of its production. When we do this, clearly, we observe that the very first terminal symbols derived are either 'a' or 'c'. Hence, FIRST(ASAA) = {a,c}. What did you notice? This is the same as FIRST(A)!

Hence, in this case, FIRST(XYZ) = FIRST(X). In other words, to find the FIRST of a sequence of symbols, it is enough to find the FIRST set of the first symbol in the sequence. But will this always hold? No! Complications arise when we have lambda!

Before we move on to that, let us consider:

S -> cAd | B | Ab

A -> ab | c

B -> db

Now, what is FIRST(S)? We observe that the possible derivations for S are:

S => cAd OR

S => B OR

S => Ab

Hence, FIRST(S) = FIRST(c) U FIRST(B) U FIRST(A)

(This is because the very first terminal symbol that can be generated can be the first terminal generated by choosing B or A or it can be the terminal c itself).

Now, FIRST(c) = {c}

FIRST(B) = {d}

FIRST(A) = {a,c}

Therefore, FIRST(S) = {c,d,a}

Now, let us consider the following:

S -> cAd | B | Ab | lambda

A -> ab | c | lambda

B -> db

Now, what is FIRST(A)? What is the very first terminal symbol that can be derived from A? Here, it is the same answer as earlier, except that lambda is also now included in the set! This is because lambda can also now be derived as the very first symbol (we consider it to be a terminal) from A. Thus, FIRST(A) = {a, c, lambda}

What is FIRST(Abd)? What is the very first terminal symbol that can be generated from the sequence Abd? Well, like earlier, you can say it is same as FIRST(A). But, there is a small change. Because A can derive lambda, therefore, FIRST(Abd) will also contain b! Why? Because, consider the following:

Abd => lambda bd => bd

Notice that because FIRST(A) was lambda, the non-terminal A derived lambda as the first symbol. And we know that when we concatenate lambda with any other symbol X, we get X itself (lambda is the identity operator for concatenation). Hence, we observe that A has disappeared and hence the very first terminal symbol that has been generated in this case is ‘b’. Thus, FIRST(Abd) = {a, c, b}. Here, we do not include lambda in the first set, since the sequence Abd by itself does not derive lambda as the first terminal.

Similarly, what is FIRST(S)? As earlier, we have:

S => cAd OR

S => B OR

S => Ab OR

S => lambda

Hence, FIRST(S) = FIRST(c) U FIRST(B) U FIRST(Ab) U FIRST(lambda)

FIRST(c) = {c}

FIRST(B) = {d}

FIRST(Ab) = {a, c, b}. This is because A derives lambda. Hence we also have to consider ‘b’

FIRST(lambda) = {lambda}

Thus, FIRST(S) = {a, b, c, d, lambda}

In general, therefore, we can give the following algorithm for computing FIRST(X), where X is any grammar symbol (terminals, non-terminals):

1. If X is a terminal, then FIRST(X) = {X}

2. If X is a non-terminal and X -> Y1Y2Y3...Yk is a production for some k>=1, then, place ‘a’ in FIRST(X) if for some i, a is in FIRST(Yi), and lambda is in all of FIRST(Y1), FIRST(Y2), … FIRST(Yi-1). In other words, if the first i-1 non-terminals all derive lambda, then you can include FIRST of Yi into the set FIRST(X). If lambda is in all of FIRST(Yi), for 1<=i<=k then, include lambda in FIRST(X). For example, everything in FIRST(Y1) is surely included in FIRST(X). If Y1 does not derive lambda, then we add nothing more to FIRST(X), but if Y1 derives lambda as its first terminal symbol, i.e., FIRST(Y1) contains lambda, then we add FIRST(Y2) to FIRST(X), and so on.

3. If X -> lambda is a production, then we add lambda to FIRST(X).

Now, we can compute FIRST for any string (or sequence of terminals, non-terminals) X1X2X3...Xn as follows: Add to FIRST(X1X2X3...Xn) all non-lambda symbols of FIRST(X1). Also add the non-lambda symbols of FIRST(X2), if lambda is in FIRST(X1); the non-lambda symbols of FIRST(X3), if lambda is in FIRST(X1) and FIRST(X2); and so on. Finally, add lambda to FIRST(X1X2X3...Xn) if for all i, lambda is in FIRST(Xi).

**PROBLEMS:**

1. Consider the following grammar:  
S -> aSc | T | lambda  
T -> bTc | lambda  
Compute:

(a) FIRST(S)

(b) FIRST(T)

(c) FIRST(ST)

(d) FIRST(TSb)

(e) FIRST(ScT)

2. Consider the following grammar:

E -> TE’

E’ -> +TE’ | lambda

T -> FT’

T’ -> \*FT’ | lambda

F -> (E) | i

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

Compute FIRST sets for each of the non-terminals.

3. Consider the grammar:

A -> BAa | lambda

B -> bBc | AA

Compute FIRST(A) and FIRST(B).

4. Compute the FIRST sets for each of the non-terminals of the following:

S -> aAa | bS

A -> BB | C

B -> bC

C -> B | c | lambda

5. Compute the FIRST sets for each of the non-terminals of the following:

S -> LB

B -> aSaL | bL

E -> c | L

J -> dEJ | f

L -> eEJ

The set of terminals are: {a, b, c, d, e, f} while non-terminals are: {S, B, E, J, L}

**FOR THE ASSIGNMENT:**

Write a function named FIRST which takes two parameters: a grammar G and a sequence of symbols X. It should return the FIRST set of X for the corresponding grammar G.