We have come a long way! Given an arbitrary grammar, we now know how to identify it to be an LL(1) grammar. We must now see how to parse a given string (according to the given grammar). In order to do so, to make our life easier, the parsing technique will involve construction of a parsing table. This table will aid us in the parsing process. Let us now see how to make such a table.  
  
 We will look at the process of making such a table taking an example of the following LL(1) grammar:  
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} and the set of terminals is: {+, \*, (, ), i}  
  
 Each row in our parsing table corresponds to a non-terminal of the given LL(1) grammar. Thus in our table, we first write down all the non-terminals of the grammar in each row of the leftmost column of the table as shown below:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |
| E |  |  |  |  |  |  |
| E’ |  |  |  |  |  |  |
| T |  |  |  |  |  |  |
| T’ |  |  |  |  |  |  |
| F |  |  |  |  |  |  |

Each column of the table corresponds to a terminal symbol. We also add an extra column indicating the $ symbol. The table thus is:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | i | + | \* | ( | ) | $ |
| E |  |  |  |  |  |  |
| E’ |  |  |  |  |  |  |
| T |  |  |  |  |  |  |
| T’ |  |  |  |  |  |  |
| F |  |  |  |  |  |  |

Next, we look at each of the production rules in the grammar one by one. Assume the production is of the form A -> (alpha). Then, we find out FIRST(alpha), and for each symbol in FIRST(alpha), we add A -> (alpha) in the row corresponding to A and the column corresponding to that particular symbol.  
 For example, let us take the first production rule: E -> TE’. Find out FIRST(TE’). This is: {(, i}. Hence, we add this production E -> TE’ into the row for [E, (] and [E, i]. Thus, the table now becomes:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | i | + | \* | ( | ) | $ |
| E | E->TE’ |  |  | E->TE’ |  |  |
| E’ |  |  |  |  |  |  |
| T |  |  |  |  |  |  |
| T’ |  |  |  |  |  |  |
| F |  |  |  |  |  |  |

Next, let us consider the production: E’ -> +TE’. Here, FIRST(+TE’) = {+}. Therefore, we add this production rule into the table for [E’, +].

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | i | + | \* | ( | ) | $ |
| E | E->TE’ |  |  | E->TE’ |  |  |
| E’ |  | E’->+TE’ |  |  |  |  |
| T |  |  |  |  |  |  |
| T’ |  |  |  |  |  |  |
| F |  |  |  |  |  |  |

There is another important rule: If lambda is present in FIRST(alpha) where the production is of the form A -> (alpha), then, we find FOLLOW(A), and for each symbol in this FOLLOW(A), we add the rule A -> (alpha) to the entry corresponding to row A and the column { FOLLOW(A) }.   
 For example, we now have the rule E’ -> lambda. FIRST(lambda) = {lambda}. Thus, we now find FOLLOW(E’). This is: {), $}. Thus, we now add the production rule E’ -> lambda into the table corresponding to [E’, )] and [E’, $]. The table thus becomes:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | i | + | \* | ( | ) | $ |
| E | E->TE’ |  |  | E->TE’ |  |  |
| E’ |  | E’->+TE’ |  |  | E’->lambda | E’->lambda |
| T |  |  |  |  |  |  |
| T’ |  |  |  |  |  |  |
| F |  |  |  |  |  |  |

We notice that it is unnecessary to write the LHS of each production rule in each cell in the table, because it is rather obvious. For a given row A, the LHS will always be A. Thus, henceforth, we will skip writing the entire production rule in the table cell. Instead, we will only write the RHS of the production.  
 Next let us take the rule: T -> FT’. We have FIRST(FT’) = {(, i}. Hence, we add this production rule into [T, (] and [T, i] of our table:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | i | + | \* | ( | ) | $ |
| E | TE’ |  |  | TE’ |  |  |
| E’ |  | +TE’ |  |  | lambda | lambda |
| T | FT’ |  |  | FT’ |  |  |
| T’ |  |  |  |  |  |  |
| F |  |  |  |  |  |  |

Take the rule: T’ -> \*FT’. FIRST(\*FT’) = {\*}. Thus, we add this production into [T’, \*] of the table:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | i | + | \* | ( | ) | $ |
| E | TE’ |  |  | TE’ |  |  |
| E’ |  | +TE’ |  |  | lambda | lambda |
| T | FT’ |  |  | FT’ |  |  |
| T’ |  |  | \*FT’ |  |  |  |
| F |  |  |  |  |  |  |

Next consider T’ -> lambda. Since FIRST(lambda) = {lambda} contains lambda in the first set, therefore, we find FOLLOW(T’) = {+, ), $}. Thus, we add T’ -> lambda into [T’, +], [T’, )] and [T’, $].

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | i | + | \* | ( | ) | $ |
| E | TE’ |  |  | TE’ |  |  |
| E’ |  | +TE’ |  |  | lambda | lambda |
| T | FT’ |  |  | FT’ |  |  |
| T’ |  | lambda | \*FT’ |  | lambda | lambda |
| F |  |  |  |  |  |  |

Take the rule: F -> (E). FIRST((E)) = {(}. Thus, we add this rule into [F, (]:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | i | + | \* | ( | ) | $ |
| E | TE’ |  |  | TE’ |  |  |
| E’ |  | +TE’ |  |  | lambda | lambda |
| T | FT’ |  |  | FT’ |  |  |
| T’ |  | lambda | \*FT’ |  | lambda | lambda |
| F |  |  |  | (E) |  |  |

Finally, we have F -> i. FIRST(i) = {i}. Therefore, we add this production into [F, i]. Thus, our final table looks as follows:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | i | + | \* | ( | ) | $ |
| E | TE’ |  |  | TE’ |  |  |
| E’ |  | +TE’ |  |  | lambda | lambda |
| T | FT’ |  |  | FT’ |  |  |
| T’ |  | lambda | \*FT’ |  | lambda | lambda |
| F | i |  |  | (E) |  |  |

In this parsing table, each entry uniquely determines what production rule must be applied at a given step. For example, suppose we have the current input symbol as + and we are supposed to expand non-terminal E’. Clearly, we have to choose either of the productions to expand E’: either choose E’ -> +TE’ or choose E’ -> lambda. The above table will help us making this decision. All we need to do is look into the table corresponding to the entry [E’, +]. We notice that we have +TE’. This clearly indicates that at this point, we must expand non-terminal E’ by choosing the production rule: E’ -> +TE’ and NOT E’ -> lambda!  
 We notice that some of the entries in the table are empty. What does this mean? These entries denote an error. For example, suppose we have current input symbol as + and we have the non-terminal E. If we look at the table for the entry [E, +], we observe that it is empty. This indicates that at this point, we can halt and announce that this string cannot be parsed!  
 To sum up what we have done, the following is a rough sketch of the algorithm to construct our LL(1) parsing table (let us denote the table by M) for a given grammar G:

For each production rule A -> (alpha) of the grammar, do the following:

1. For each terminal ‘a’ in FIRST(alpha), add A -> (alpha) to M[A, ‘a’]. Alternatively, we could instead add just (alpha) to M[A, ‘a’] since it is obvious that the LHS is A.

2. If lambda is in FIRST(alpha), then for each symbol ‘b’ in FOLLOW(A), add A -> (alpha) to M[A, ‘b’]. Again, we can instead add just (alpha) to M[A, ‘b’] since it is obvious that the LHS is A.

At the end of performing the above steps for all production rules in the grammar, all the empty entries in the table denote error!

Now, what would happen if we applied the above algorithm to a grammar that is not LL(1)? Let us try performing this, since it leads to an interesting result.

We know that no left recursive grammar can be LL(1). Thus, the following grammar cannot be LL(1):

S -> Sa | b

Let us try to construct the LL(1) parsing table for this simple grammar. Our parsing table will have only one row, since there is only one non-terminal, S. It will have three columns, one for $ and the other two for each of the terminals a and b.

|  |  |  |  |
| --- | --- | --- | --- |
|  | a | b | $ |
| S |  |  |  |

Now, let us consider each production one by one. First let us take S -> Sa. FIRST(Sa) = {b}. Thus, we add S -> Sa in the table for [S, b]:

|  |  |  |  |
| --- | --- | --- | --- |
|  | a | b | $ |
| S |  | S -> Sa |  |

Next let us consider S -> b. FIRST(b) = {b}. Therefore, we must add this production rule into our table at the entry for [S, b]. Thus our table becomes:

|  |  |  |  |
| --- | --- | --- | --- |
|  | a | b | $ |
| S |  | S -> Sa  S -> b |  |

Notice what has happened!!! In the entry for M[S, b], we have got two production rules! This means that if our current input symbol was ‘b’ and we had to expand the non-terminal S, then, at this point, WE CANNOT DO SO!!! This is because we have got two choices of productions! This clearly indicates that the given grammar cannot be parsed by an LL(1) parser. In other words, this grammar is NOT LL(1).

So, what does this indicate? This shows a very useful result: If you remember, in the previous document, you were asked to determine if a grammar was LL(1) or not, and to do so, you had to apply those 4 rules, and see if the results were desirable. **Instead of doing all that (which may be tedious), you can simply construct the LL(1) parsing table for the grammar.** If at any point, you get multiple entries for any cell in the table, you can halt and tell that the given grammar is not LL(1). If you manage to complete filling the entire table without getting multiple entries in any of the cells, then, you can conclude that this grammar is an LL(1) grammar!!!

**PROBLEMS:**

For each of the following grammars, construct the LL(1) parsing table. As a result, determine if these grammars are LL(1) grammars or not (DON’T apply the methods of the previous document to determine if these grammars are LL(1). Use the table to determine this!):

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.

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}.

4.

S -> iEtSU | a

U -> eS | lambda

E -> b

5.

S -> aABb

A -> aAc | lambda

B -> bB | c

6.

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

7.

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

8.

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

9.

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

10.

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

**FOR THE ASSIGNMENT:**

Write a function that takes a grammar G as its parameter and returns the LL(1) parsing table for that grammar G. If the grammar is not LL(1), it returns something else to indicate error. Choose a suitable data structure for the parsing table.