I hope we have not lost sight of the big picture in the act of concentrating on the smaller details! Our goal has not yet been reached. We intend to make an LL(1) parser that will tell whether a given string w can be generated by a grammar G. We have not yet made the parser! We have so far concentrated on the little details required to make the parser. Now that all the details have been taken care of, we are in the final leg of our LL(1) parser journey. It only remains now to see how the parser works.

Let us understand the working of the LL(1) parser with the help of an example. Consider the following grammar:

S -> F

S -> (S+F)

F -> a

Here, the set of non-terminals is: {S, F} and the set of terminals is: {a, +, (, )}.

Let us assume we are given the string w = (a+a) to parse.

An LL(1) parser consists of:

- An input buffer, which holds the input string w, that is to be parsed. The input string is generally followed by the end marker symbol $. Thus, in this case, our input buffer will have: (a+a)$.

- A stack on which to store a sequence of grammar symbols during the process of parsing.

- A parsing table which tells what grammar rule to apply given the symbol currently on top of the stack and the symbol currently being pointed to in the input buffer.

Now, for our above grammar, we can construct the LL(1) parsing table as follows:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | ( | ) | a | + | $ |
| S | (S+F) |  | F |  |  |
| F |  |  | a |  |  |

When the parser starts, the stack contains two symbols: S and $, where $ is the end marker symbol to indicate the bottom of the stack and also the end of input, and S is the start symbol of the grammar. Thus, initially, the stack is as follows:

[S, $], where the left end indicates the top end of stack.

In each step, the parser reads the next available symbol from the input buffer, and the topmost symbol from the stack. If the input symbol and the stack symbol match, the parser discards them both, leaving only the unmatched symbols in the input buffer and on the stack.

Thus, in the first step, the parser reads the input symbol ‘(’ and the top of stack symbol S. Now it looks up the parsing table for [S, (]. This entry says that it needs to use the production rule S -> (S+F). Thus, the parser now applies this rule, by rewriting the S on the top of stack with (S+F). (In other words, it first pops the S then pushes (S+F) onto the stack in reverse order). So, the contents of stack are:

[ (, S, +, F, ), $ ].

Since the ‘(’ from the input buffer did not match the topmost symbol S from the stack, it was not removed and so it remains in the input buffer as the current symbol.

Now, the parser looks at the current symbol in the input buffer, which happens to be ‘(’ and compares it with the top of stack, which also happens to be ‘(’. Since they match, therefore, the parser discards both. So, the stack now becomes:

[ S, +, F, ), $ ]

And the input buffer is: a+a)$.

Now the parser has ‘a’ as the current symbol in the input stream and S as the top of stack. So, it looks at the parsing table for the entry in [S, a]. The parsing table instructs it to apply the rule S -> F. Thus, it replaces the top of stack (in other words, it pops S from the stack and pushes F onto it), so that the stack now becomes:

[ F, +, F, ), $].

Since the current input symbol did not match the top of stack, therefore, it is not removed and so the input buffer remains: a+a)$

Now, the parser has ‘a’ as its current symbol and F as the top of stack symbol. So it looks at the parsing table for the entry [F, a]. The table instructs it to apply the rule F -> a. Thus, it pops F from the top of stack and pushes a onto it. Thus, the stack becomes:

[ a, +, F, ), $ ]

Again, input buffer remains: a+a)$ since the current symbol did not match the top of stack. Now, it compares the current symbol ‘a’ with top of stack symbol, which is also ‘a’. Since they match, therefore, both are discarded. Hence the stack now becomes:

[ +, F, ), $]

And the input buffer becomes: +a)$. Now, it compares the top of stack symbol ‘+’ with the current input symbol. Since they match, they are both discarded. Hence, the stack is now:

[ F, ), $ ].

And the input buffer becomes: a)$. Now the current input symbol is a, while the top of stack symbol is F. Since they don’t match, the input symbol is not discarded. Instead it looks up the parsing table at [F, a]. The table instructs it to apply the rule: F -> a. Hence it pops F off the stack, and replaces it with a. Thus, the stack becomes:

[ a, ), $ ].

The input buffer is: a)$. At this point it compares the top of stack with current input. They match and hence both are discarded. Thus, input buffer becomes: )$ while the stack becomes:

[ ), $ ].

Now, the current input symbol is compared with the top of stack symbol. They match and hence are discarded. Thus, input buffer becomes: $ and the stack becomes:

[ $ ]

Now, when the parser sees that the current input symbol is $, and also the top of stack is $, it concludes that it has reached the end of the input at the same time that the stack has become empty, and therefore, it halts and announces successful parsing of the string.

If at all, at any point in the parsing process, the parser had got a terminal symbol on the top of stack that was different from the current symbol of the input buffer, or if it had looked up the table and the table had no entry, or if it reaches the $ while the top of stack is not a $, then, in all of these cases, the parser would halt and announce failure to parse the string.

The following is a rough sketch of the algorithm of the parsing process:

Input: A string w to be parsed and a parsing table M for grammar G

Let a be the first symbol of w.

Let X be the top of stack symbol.

While (X != $) //while stack is not empty

{

If (X == a)

Pop the stack and let a be the next symbol of w.

Else if (X is some other terminal)

ERROR

Else if (M[X, a] is an error entry, i.e., empty cell in table)

ERROR

Else if (M[X, a] = X -> Y1Y2...Yn)

{

Pop the stack.

Push YnYn-1...Y2Y1 onto the stack with Y1 on the top;

}

Let X be the top of stack symbol

}

**PROBLEMS:**

For each of the following grammars, and the strings given, apply the LL(1) parsing algorithm to parse the strings and indicate whether the strings belong to the language of the grammar or not:

1.

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}. Parse the following:

(a) i+i\*i

(b) i\*i+i

(c) i+i+i\*i

(d) i\*(+i)

(e) i+(i+(+i))

(f) i+(i\*i

(g) ((i)+(i\*i)

2.

S -> aABb

A -> aAc | lambda

B -> bB | c

Parse the following:

(a) aacbcb

(b) abbcb

(c) abcacc

(d) accccb

(e) acb

3.

S -> LB  
B -> aSaL | bL  
E -> c | L  
J -> dEJ | f  
L -> eEJ  
Parse the following:

(a) ecfbecf

(b) eecdcffbecf

(c) ecfaecf

(d) eecfbecf