Name: Isha Aggarwal

Roll No: S20180010067

Desk Calculator

Postfix SDT

0. L -> E n {print(E.val)}
1. E -> E_1+ T { E.val = E_1.val + T.val}
2. E -> T {E.val = T.val }
3. T -> T_1 * F {T.val = T_1.val * F.val }
4. T-> F {T.val = F.val}
5. F -> F_1 ^ A { F.val = F_1.val^ A.val}
6. F -> A {F.val = A.val}
7. A -> - B { A.val = -B.val}
8. A -> B { A.val = B.val}
9. B -> (E) {B.val = E.val }
10. B -> num {B.val = num.lexval }

LR(1) automaton and CLR parse table

Created these using an online tool since drawing by hand was almost impossible due to a large number of states.

LR(1) Automata

This is created by using the functions CLOSURE(I) and GOTO(I,X). We start with the state 0 having [L' ->. L, $] and close this set. Further, GOTO function is used and other items sets are produced. Closure and goto is applied repeatedly to create the LR(1) automaton.

CLR Parse Table

Once the automaton is obtained, it is quite easy to build the parse table.

Implementation:

The Postfix SDT and the CLR parse table is stored in the memory. LR parsing is done to check if the input string is in the language of the Grammar. Translation occurs as a side effect of parsing, whenever reduction happens, the semantic action corresponding to that production is executed.

Implemented in C++11(parsing and translation, but the parse table is created manually and stored in memory directly) and lex(for generating lexical analysis). Parsing is implemented using stack, and the attribute values are stored in a separate array which is manipulated as and when required.

The semantic actions are converted to the following form. Here top is the top pointer of the stack containing grammar symbols. A separate array is used to store the attribute values corresponding to the grammar symbols such that both have the same positions in the arrays.

Productions and Semantic actions

0. L -> E n { print(array[top -1] }
1. E -> E_1+ T { array[top – 2] = array[top -2] + array[top] }
2. E -> T
3. T -> T_1 * F { array[top – 2] = array[top -2] * array[top] }
4. T-> F {
5. F -> F_1 ^ A { array[top – 2] = array[top -2] ^ array[top] }
6. F -> A
7. A -> - B { array[top – 1] = -1*array[top] }
8. A -> B
9. B -> (E) { array[top – 2] = array[top - 1] }
10. B -> num

A screenshot of successful run is also there in the folder.

Test_Case
