**Name** – Bhargav Shamuvel Gurav **PRN** – 2041009

**Class** – L.Y. B-Tech (Computer) **Batch** – B1

**Course Code** – CO406U **Course Name** - CDL

**Practical no. B (1)**

**Aim:** Write a C program to implement operator precedence parsing.

**Theory :**

Operator Precedence Parsing is also a type of Bottom-Up Parsing that can be used to a class of Grammars known as Operator Grammar.

A Grammar G is Operator Grammar if it has the following properties −

* Production should not contain ϵ on its right side.
* There should not be two adjacent non-terminals at the right side of production.

Example1 − Verify whether the following Grammar is operator Grammar or not.

E → E A E |(E)|id

A → +| − | \*

Solution

No, it is not an operator Grammar as it does not satisfy property 2 of operator Grammar.

As it contains two adjacent Non-terminals on R.H.S of production E → E A E.

We can convert it into the operator Grammar by substituting the value of A in E → E A E.

E → E + E |E − E |E \* E |(E) | id.

Operator Precedence Relations

Three precedence relations exist between the pair of terminals.

| Relation | Meaning |
| --- | --- |
| p <. q | p has less precedence than q. |
| p >. q | p has more precedence than q. |
| p =. q | p has equal precedence than q. |

Depending upon these precedence Relations, we can decide which operations will be executed or parsed first.

Association and Precedence Rules

* If operators have different precedence

Since \* has higher precedence than +

Example−

In a statement a + b \* c

∴ + <. \*

In statement a \* b + c

∴ ∗ . > +

* If operators have Equal precedence, then use Association rules.

(a) Example minus; In statement a + b + c here + operators are having equal precedence.

As '+' is left Associative in a + b + c

∴ (a + b) will be computed first, and then it will be added to c.

i.e., (a + b) + c

+ .> +

Similarly, '\*' is left Associative in a \* b \* c

(b) Example − In a statement a ↑ b ↑ c here, ↑ is the Right Associative operator

∴ It will become a ↑ (b ↑ c)

∴ (b ↑ c) will be computed first.

∴ ↑<. ↑

* Identifier has more precedence then all operators and symbols.

∴ θ <. id           $ <. id

id . > θ            id . > $

id . >)

(<. id.

* $ has less precedence than all other operators and symbols.

$ <.           ( id . > $

$ <. +         ). > $

$ <.\*

Example2 − Construct the Precedence Relation table for the Grammar.

E → E + E | E ∗ E/id

Solution

Operator-Precedence Relations

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Id | + | \* | $ |
| Id |  | .> | .> | .> |
| + | <. | .> | <. | .> |
| \* | <. | .> | .> | .> |
| $ | <. | <. | <. |  |

Advantages of Operator Precedence Parsing

* It is accessible to execute.

Disadvantages of Operator Precedence Parsing

* Operator Like minus can be unary or binary. So, this operator can have different precedence’s in different statements.
* Operator Precedence Parsing applies to only a small class of Grammars.

**Program Code:**

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

void main()

{

char stack[20],ip[20],opt[10][10][1],ter[10];

int i,j,k,n,top=0,col,row;

for(i=0;i<10;i++)

{

stack[i]=NULL;

ip[i]=NULL;

for(j=0;j<10;j++)

{

opt[i][j][1]=NULL;

}

}

printf("Enter the no.of terminals:\n");

scanf("%d",&n);

printf("\nEnter the terminals:\n");

for(i=0;i<n;i++)

{

scanf("%s",&ter[i]);

}

printf("\nEnter the table values:\n");

for(i=0;i<n;i++)

{

for(j=0;j<n;j++)

{

printf("Enter the value for %c %c: ",ter[i],ter[j]);

scanf("%s",opt[i][j]);

}

}

printf("\n\*\* OPERATOR PRECEDENCE TABLE \*\*\n");

for(i=0;i<n;i++)

{

printf("\t%c",ter[i]);

}

printf("\n");

for(i=0;i<n;i++)

{

printf("\n%c",ter[i]);

for(j=0;j<n;j++)

{

printf("\t%c",opt[i][j][0]);

}

}

stack[top]='$';

printf("\nEnter the input string: ");

scanf("%s",ip);

i=0;

printf("\nSTACK\t\t\tINPUT STRING\t\t\tACTION\n");

printf("\n%s\t\t\t%s\t\t\t",stack,ip);

while(i<=strlen(ip))

{

for(k=0;k<n;k++)

{

if(stack[top]==ter[k])

col=k;

if(ip[i]==ter[k])

row=k;

}

if((stack[top]=='$')&&(ip[i]=='$'))

{

printf("\nString is accepted\n");

break;

}

else if((opt[col][row][0]=='<') ||(opt[col][row][0]=='='))

{

stack[++top]=opt[col][row][0];

stack[++top]=ip[i];

printf("Shift %c",ip[i]);

i++;

}

else

{

if(opt[col][row][0]=='>')

{

while(stack[top]!='<'){--top;}

top=top-1;

printf("Reduce");

}

else

{

printf("\nString is not accepted");

break;

}

}

printf("\n");

for(k=0;k<=top;k++)

{

printf("%c",stack[k]);

}

printf("\t\t\t");

for(k=i;k<strlen(ip);k++)

{

printf("%c",ip[k]);

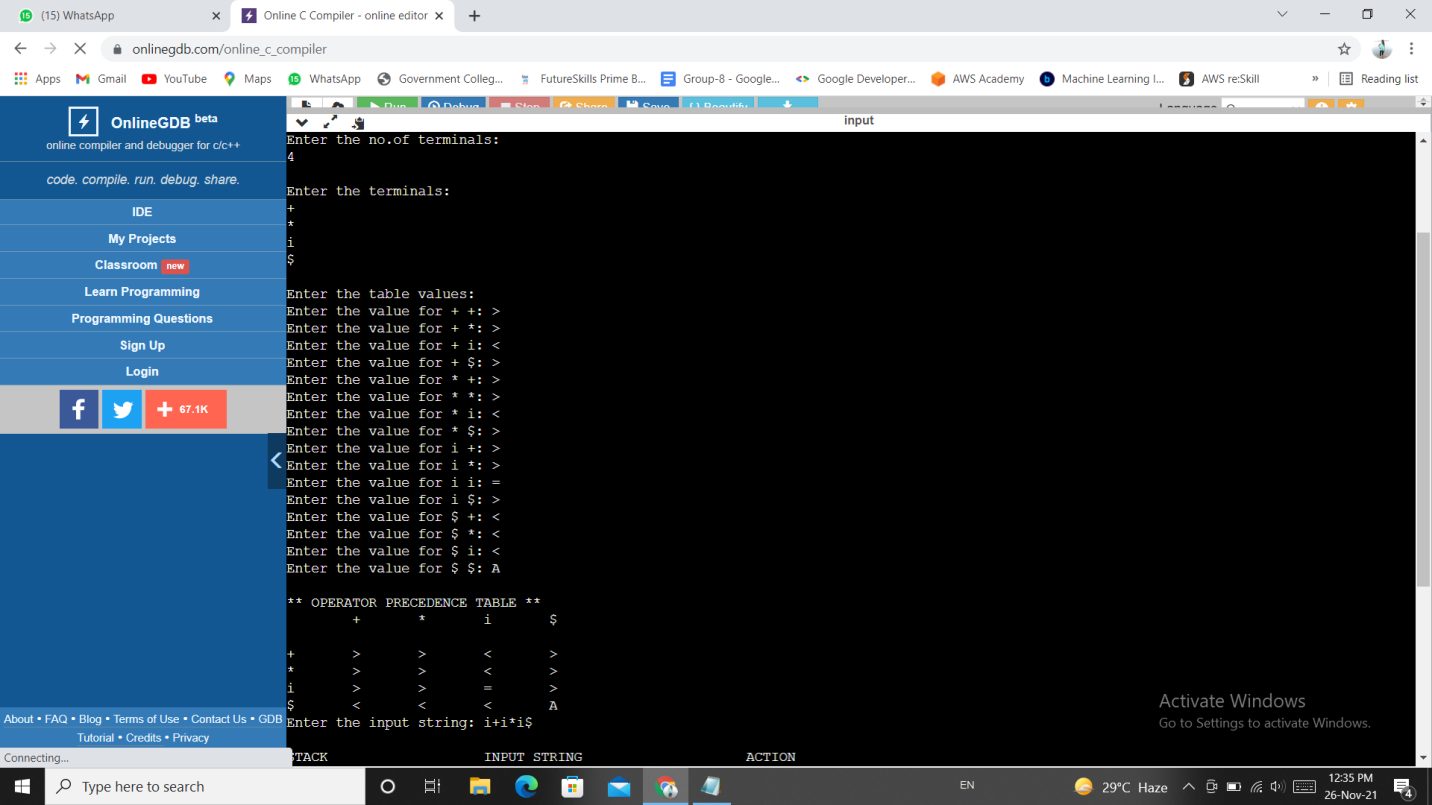
}

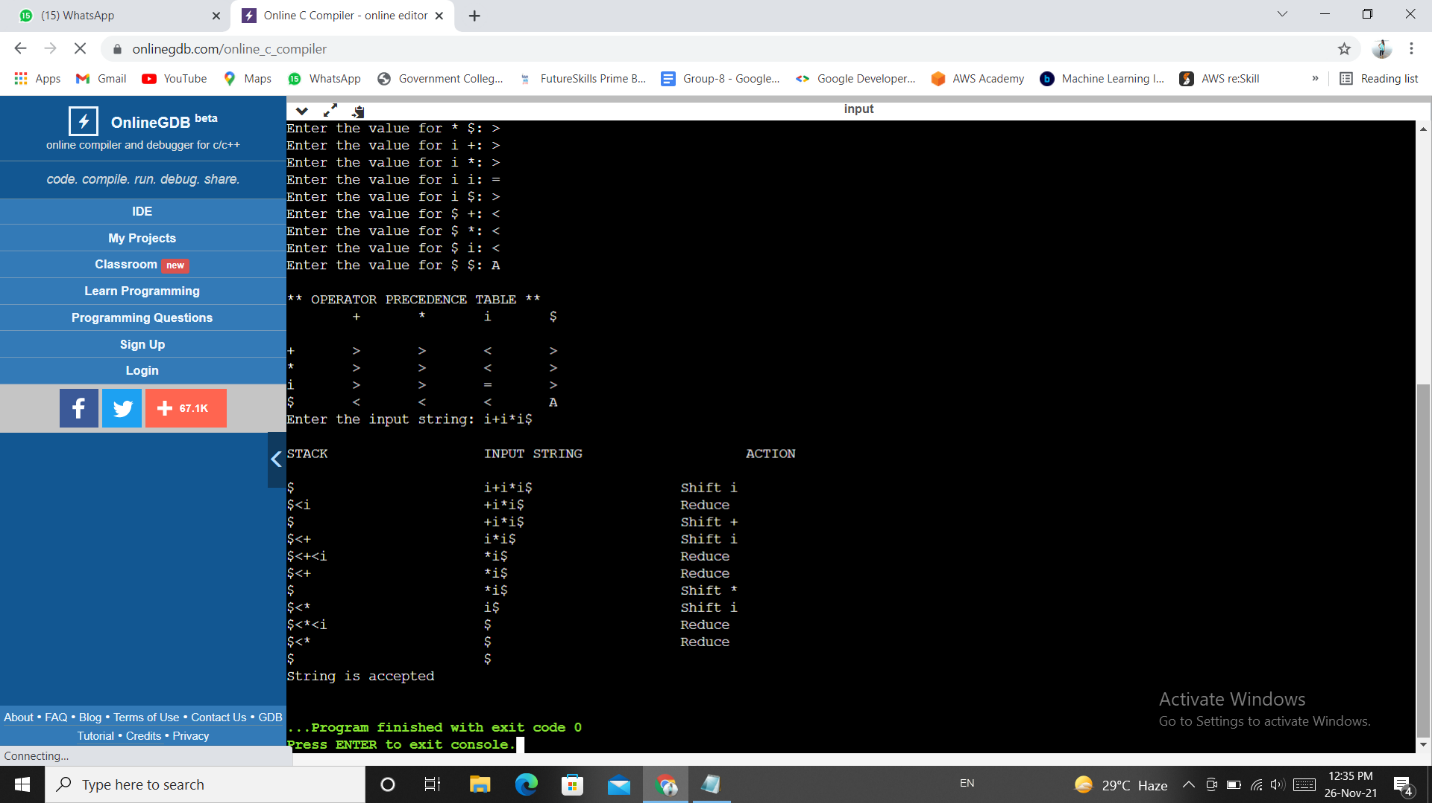
printf("\t\t\t");

}

}

**Output:**

****

****

**Conclusion :** In this practical we learnt how operator precedence parser works for parsing operator precedence grammar.