Experiment-5

AIM: Construct a LL(1) parser for an expression

DESCRIPTION:

LL(1) Parsing: Here the 1st L represents that the scanning of the Input will be done from the Left to Right manner and the second L shows that in this parsing technique, we are going to use the Left most Derivation Tree. And finally, the 1 represents the number of look-ahead, which means how many symbols are you going to see when you want to make a decision.

Essential conditions to check first are as follows:

- The grammar is free from left recursion.
- The grammar should not be ambiguous.
- The grammar has to be left factored in so that the grammar is deterministic grammar.

Steps to Construct an LL(1) Parser for Expressions

1. Define the Grammar
2. Compute FIRST and FOLLOW Sets
3. Construct the Predictive Parsing Table Using the FIRST and FOLLOW sets
4. Parsing Algorithm

An LL(1) parser for expressions involves:

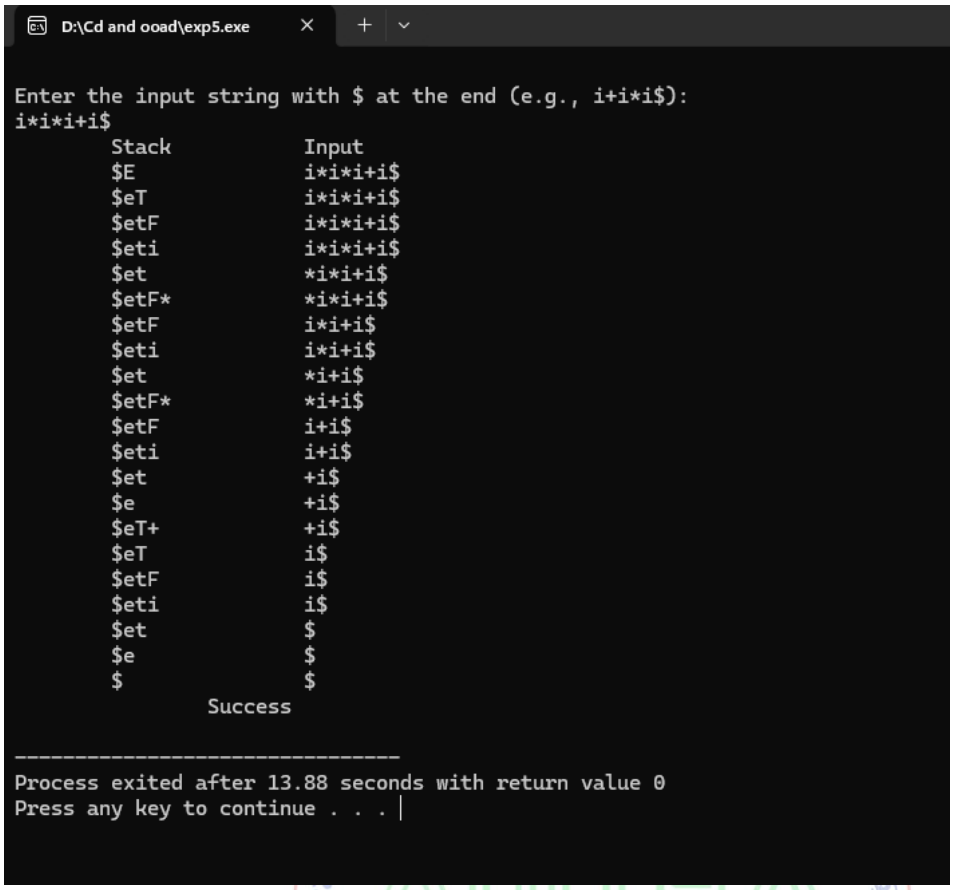
- Left-factored grammar
- Computing FIRST and FOLLOW sets
- Building a predictive parsing table
- Using a stack-driven parsing method

Program:-

```
#include<stdio.h>
#include<string.h>
int stack[20], top = -1;
 void push(int item) {
    if (top >= 20) {
printf("Stack Overflow\n");
        return;
    }
    stack[++top] = item;
}
int pop() {
    if (top <= -1) {
printf("Stack Underflow\n");
        return -1; // Return -1 to indicate underflow
    }
    return stack[top--];
}
char convert(int item) {
    switch (item) {
        case 0: return 'E';
        case 1: return 'e';
        case 2: return 'T';
        case 3: return 't';
        case 4: return 'F';
        case 5: return 'i';
        case 6: return '+';
        case 7: return '*';
        case 8: return '(';
        case 9: return ')';
        case 10: return '$';
        default: return '?'; // Error case
    }
}
int main() {
    int m[10][10], i, j, k;
    char ips[20];
    int ip[10], a, b, t;
    // Fill in the parsing table
m[0][0] = m[0][3] = 21;
m[1][1] = 621;
m[1][4] = m[1][5] = -2;
m[2][0] = m[2][3] = 43;
```

```
m[3][1] = m[3][4] = m[3][5] = -2;
m[3][2] = 743;
m[4][0] = 5;
m[4][3] = 809;
printf("\nEnter the input string with $ at the end (e.g., i+i*i$):\n");
scanf("%s", ips);
   for (i = 0; i<strlen(ips); i++) {
      switch (ips[i]) {
         case 'E': k = 0; break;
         case 'e': k = 1; break;
         case 'T': k = 2; break;
         case 't': k = 3; break;
         case 'F': k = 4; break;
         case 'i': k = 5; break;
         case '+': k = 6; break;
         case '*': k = 7; break;
         case '(': k = 8; break;
         case ')': k = 9; break;
         case '$': k = 10; break;
         default: printf("Invalid input\n"); return 1; // Exit if input is invalid
      }
ip[i] = k;
   }
ip[i] = -1;
push(10); // Push $ onto stack
push(0); // Push 0 (start symbol) onto stack
i = 0;
printf("\tStack\t\tInput\n");
   while (1) {
printf("\t");
      for (j = 0; j <= top; j++)
printf("%c", convert(stack[j]));
printf("\t\t");
        for (k = i; ip[k] != -1; k++)
printf("%c", convert(ip[k]));
printf("\n");
        if (stack[top] == ip[i]) {
            if (ip[i] == 10) {
printf("\t\tSuccess\n");
                return 0;
            } else {
                top--;
i++;
            }
        } else if (stack[top] <= 4 && stack[top] >= 0) {
            a = stack[top];
            b = ip[i] - 5;
            t = m[a][b];
            top--;
            while (t > 0) {
push(t % 10);
                t = t / 10;
            }
        } else {
printf("Error\n");
            return 1;
        }
    }
    return 0;
}
```

OUTPUT:



```
Enter the input string with $ at the end (e.g., i+i*i$):
i*i*i+i$
        Stack           Input
        $E              i*i*i+i$
        $eT             i*i*i+i$
        $etF            i*i*i+i$
        $eti            i*i*i+i$
        $et             *i*i+i$
        $etF*           *i*i+i$
        $etF            i*i+i$
        $eti            i*i+i$
        $et             *i+i$
        $etF*           *i+i$
        $etF            i+i$
        $eti            i+i$
        $et             +i$
        $e              +i$
        $eT+            +i$
        $eT             i$
        $etF            i$
        $eti            i$
        $et             $
        $e              $
        $               $
                Success

--------------------------------
Process exited after 13.88 seconds with return value 0
Press any key to continue . . .
```

Experiment-6

6.1 AIM: Write a program to perform loop unrolling.

DESCRIPTION:

Loop unrolling is a loop transformation technique that aims at improving the performance of a program at the expense of code size. When a loop is unrolled, instructions that control the loop are eliminated (full unroll) or reduced (by unroll factor) and replaced with multiple copies of the loop body, thereby reducing the computational overhead but increasing the size of the code.

- Increases program efficiency.
- Reduces loop overhead.
- If statements in loop are not dependent on each other, they can be executed in parallel.

Advantages of Loop Unrolling:

1. **Reduces Loop Overhead**
   - Fewer condition checks and counter increments improve performance.
2. **Improves Instruction-Level Parallelism (ILP)**
   - Allows the CPU to execute multiple instructions simultaneously.
3. **Optimizes Memory Access**
   - Reduces cache misses by prefetching data in advance.
4. **Enhances CPU Pipelining**
   - Minimizes branch mispredictions and improves branch prediction efficiency.

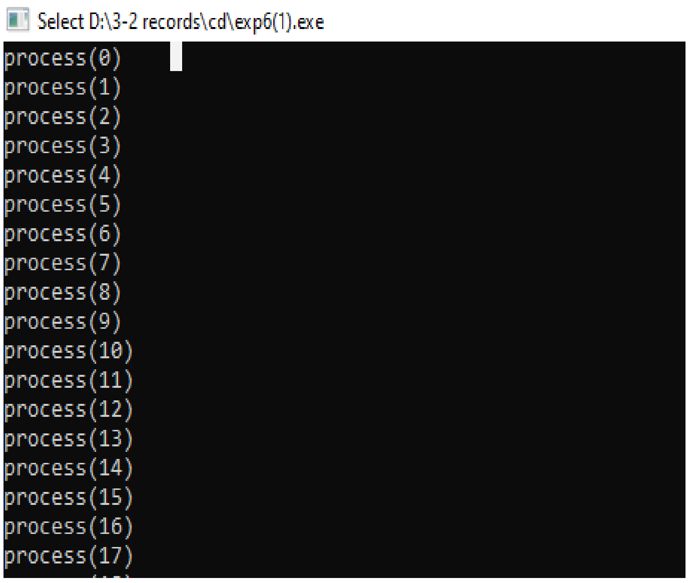
When to Use Loop Unrolling?

- When performance is critical (e.g., matrix computations, image processing).
- When loop overhead is significant compared to computation time.
- When the loop body is simple and does not lead to excessive code bloat.
- When compilers do not automatically optimize the loop effectively.

Program:-

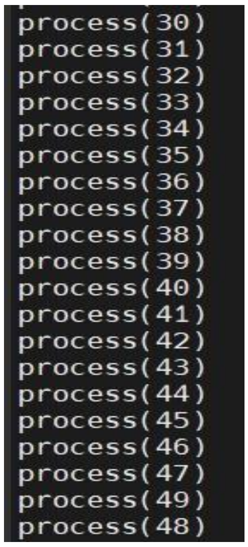
```
#include<stdio.h>
#define TOGETHER (8)
int main(void)
{
int i = 0;
int entries = 50;
int repeat;
int left = 0;
repeat = (entries / TOGETHER);
left = (entries % TOGETHER);
while (repeat--)
 {
 printf("process(%d)\n", i );
 printf("process(%d)\n", i + 1);
 printf("process(%d)\n", i + 2);
 printf("process(%d)\n", i + 3);
 printf("process(%d)\n", i + 4);
 printf("process(%d)\n", i + 5);
 printf("process(%d)\n", i + 6);
 printf("process(%d)\n", i + 7);
 i += TOGETHER;
 }
switch (left)
 {
 case 7 : printf("process(%d)\n", i + 6);
 case 6 : printf("process(%d)\n", i + 5);
 case 5 : printf("process(%d)\n", i + 4);
 case 4 : printf("process(%d)\n", i + 3);
 case 3 : printf("process(%d)\n", i + 2);
 case 2 : printf("process(%d)\n", i + 1);
 case 1 : printf("process(%d)\n", i);
 case 0 : ;
 } }
```

Output:-



Select D:\3-2 records\cd\exp6(1).exe

```
process(0)
process(1)
process(2)
process(3)
process(4)
process(5)
process(6)
process(7)
process(8)
process(9)
process(10)
process(11)
process(12)
process(13)
process(14)
process(15)
process(16)
process(17)
```



```
process(30)
process(31)
process(32)
process(33)
process(34)
process(35)
process(36)
process(37)
process(38)
process(39)
process(40)
process(41)
process(42)
process(43)
process(44)
process(45)
process(46)
process(47)
process(49)
process(48)
```

6.2 AIM: Write a program for constant propagation

DESCRIPTION:

- ➢ Constant Propagation is one of the local code optimization technique in Compiler Design. It can be defined as the process of replacing the constant value of variables in the expression.
- ➢ In simpler words, we can say that if some value is assigned a known constant, than we can simply replace the that value by constant.
- ➢ Constants assigned to a variable can be propagated through the flow graph and can be replaced when the variable is used.
- ➢ Constant propagation is executed using reaching definition analysis results in compilers, which means that if reaching definition of all variables have same assignment which assigns a same constant to the variable, then the variable has a constant value and can be substituted with the constant.

Types of Constant Propagation

1. **Local Constant Propagation**
   - Applied within a **single basic block** in the program.
2. **Global Constant Propagation**
   - Applied **across multiple basic blocks**, requiring data flow analysis to track constants through branches.

Advantages of Constant Propagation

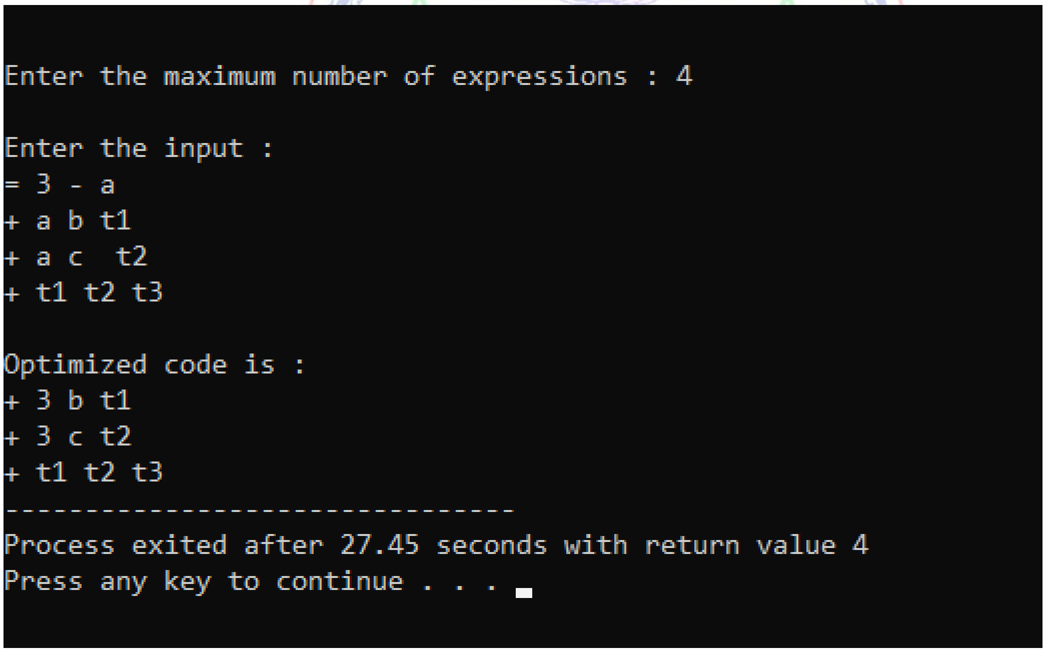
1. **Eliminates Redundant Computations**
   - Reduces runtime arithmetic operations.
2. **Improves Code Performance**
   - Enhances execution speed by simplifying expressions.
3. **Reduces Memory Usage**
   - Eliminates unnecessary variables, reducing stack/register usage.
4. **Enables Further Optimizations**
   - Simplified expressions allow **constant folding**, **dead code elimination**, and **loop optimizations**.

Program:-

```
#include<stdio.h>

#include<string.h>

#include<ctype.h>

void input();
void output();
void change(int p,char *res);
void constant();
struct expr{
  char op[2],op1[5],op2[5],res[5];
  int flag;
}arr[10];
int n;
void main(){
  input();
  constant();
  output();
}
void input(){
  int i;
  printf("\n\nEnter the maximum number of expressions : ");
  scanf("%d",&n);
  printf("\nEnter the input : \n");
  for(i=0;i<n;i++){
        scanf("%s",arr[i].op);
        scanf("%s",arr[i].op1);
        scanf("%s",arr[i].op2);
        scanf("%s",arr[i].res);
        arr[i].flag=0;
  }
}
void constant(){
  int i;
  int op1,op2,res;
  char op,res1[5];
  for(i=0;i<n;i++){
        if(isdigit(arr[i].op1[0]) && isdigit(arr[i].op2[0]) || strcmp(arr[i].op,"=")==0){
        /*if both digits, store them in variables*/
                op1=atoi(arr[i].op1);
                op2=atoi(arr[i].op2);
                op=arr[i].op[0];
                switch(op){
                        case '+':
                                res=op1+op2;
                                break;
                        case '-':
                                res=op1-op2;
                                break;
                        case '*':
                                res=op1*op2;
                                break;
                        case '/':
                                res=op1/op2;
                                break;
                        case '=':
                                res=op1;
                                break;
                }
                sprintf(res1,"%d",res);
                arr[i].flag=1;
                change(i,res1);
        }
  }
}
void output(){
  int i=0;
  printf("\nOptimized code is : ");
```

```
    for(i=0;i<n;i++){
          if(!arr[i].flag)
                    printf("\n%s %s %s %s",arr[i].op,arr[i].op1,arr[i].op2,arr[i].res);
    }
  }
  void change(int p,char *res){
    int i;
    for(i=p+1;i<n;i++){
          if(strcmp(arr[p].res,arr[i].op1)==0)
                    strcpy(arr[i].op1,res);
          else if(strcmp(arr[p].res,arr[i].op2)==0)
                    strcpy(arr[i].op2,res);
    }
  }
```

Output:-



```

Enter the maximum number of expressions : 4

Enter the input :
= 3 - a
+ a b t1
+ a c  t2
+ t1 t2 t3

Optimized code is :
+ 3 b t1
+ 3 c t2
+ t1 t2 t3
--------------------------------
Process exited after 27.45 seconds with return value 4
Press any key to continue . . .
```