```
// #include "/home/student/Desktop/KaustavLABS3/CD LAB/LAB
04/lab04_q1_symbol_table_header.h"
#include "/home/kaustav/Desktop/KaustavLABS3/CD LAB/LAB
04/lab04_q1_symbol_table_header.h"

int curr = 0;
// char str[100];
static char str[700000000];

// FILE *fp = fopen("lab04_q1_input.c", "r");
FILE *fp;
struct token *currentToken;
/////////////////////////////////////////////////////////////////
///////////////////////////////////////////////

// LAB 07
void Program();
void declarations();
void data_type();
void identifier_list();
void identifier_list_factors();
void identifier_list_factors_array();
void assign_stat();
void assign_stat_factors();

// LAB 08
void statement_list();
void statement();
void expn();
void eprime();
void simple_expn();
void seprime();
void term();
void tprime();
void factor();
void relop();
void addop();
void mulop();

// LAB 09
void decision_statement();
void dprime();
void looping_statement();
```

```
//////////////////////////////////////////////////////////////////
////////////////////////////////////////////////
void success()
{
   printf("SUCCESS\n");
   exit(0);
}

void invalid()
{
   printf("Error at Row %d : Column %d ::", currentToken->row,
currentToken->column);
   exit(0);
}

void tokenDebug()
{
   printf("Token Scanned < %s , %s > \n ", currentToken->lexeme,
currentToken->type);
   // insert_into_local_symbol_table_helper(currentToken);
}

//////////////////////////////////////////////////////////////////
////////////////////////////////////////////////

void Program()
{
   if (strcmp(currentToken->lexeme, "main") == 0)
   {
       currentToken = getNextToken(fp), tokenDebug();
       if (strcmp(currentToken->lexeme, "(") == 0)
       {
           currentToken = getNextToken(fp), tokenDebug();
           if (strcmp(currentToken->lexeme, ")") == 0)
           {
               currentToken = getNextToken(fp), tokenDebug();
               if (strcmp(currentToken->lexeme, "{") == 0)
               {
                   currentToken = getNextToken(fp), tokenDebug();

                   declarations();
                   statement_list();

                   if (strcmp(currentToken->lexeme, "}") == 0)
```

```
                        return;
                    else
                    {
                        printf("} expected \n");
                        invalid();
                    }
                }
                else
                {
                    printf("{ expected \n");
                    invalid();
                }
            }
            else
            {
                printf(") expected \n");
                invalid();
            }
        }
        else
        {
            printf("( expected \n");
            invalid();
        }
    }
    else
    {
        printf("main expected \n");
        invalid();
    }
}

void declarations()
{
    char first_of_declarations[2][10] = {"int", "char"};
    int flag = 0;
    for (int i = 0; i < sizeof(first_of_declarations) /
sizeof(first_of_declarations[0]); ++i)
    {
        if (strcmp(currentToken->lexeme, first_of_declarations[i])
== 0)
            flag++;
    }

    if (flag)
```

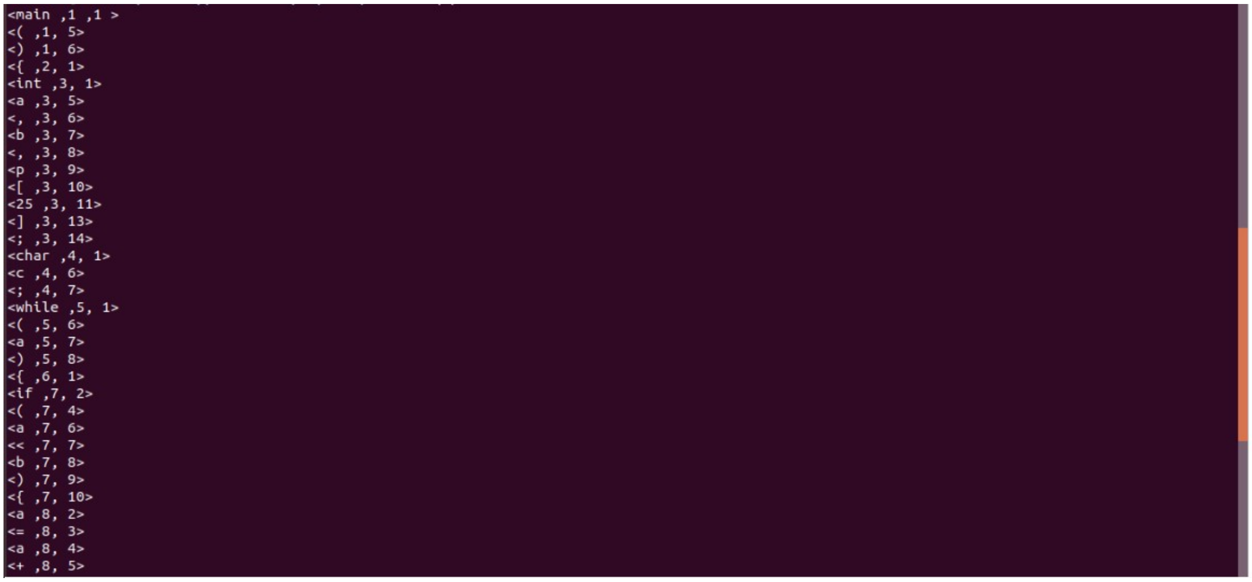
```
   {
       data_type();
       identifier_list();
       if (strcmp(currentToken->lexeme, ";") == 0)
       {
           currentToken = getNextToken(fp), tokenDebug();
           declarations();
       }
       else
       {
           printf("here ; expected \n");
           invalid();
       }
   }
}

void data_type()
{
   if ((strcmp(currentToken->lexeme, "int") == 0 ||
strcmp(currentToken->lexeme, "char") == 0))
   {
       currentToken = getNextToken(fp), tokenDebug();
       return;
   }
}

void identifier_list()
{
   if (strcmp(currentToken->type, "identifier") == 0)
   {
       currentToken = getNextToken(fp), tokenDebug();
       identifier_list_factors();
   }
   else
   {
       printf("identifier expected\n");
       invalid();
   }
}

void identifier_list_factors()
{
   if (strcmp(currentToken->lexeme, ",") == 0)
   {
       currentToken = getNextToken(fp), tokenDebug();
       identifier_list();
   }
   else if (strcmp(currentToken->lexeme, "[") == 0)
   {
       currentToken = getNextToken(fp), tokenDebug();
       if (strcmp(currentToken->type, "constant") == 0)
       {
           currentToken = getNextToken(fp), tokenDebug();
           if (strcmp(currentToken->lexeme, "]") == 0)
           {
               currentToken = getNextToken(fp), tokenDebug();
               identifier_list_factors_array();
           }
           else
           {
               printf("] expected \n");
               invalid();
           }
       }
       else
       {
           printf("constant expected \n");
           invalid();
       }
   }
   // else
   // {
   //     printf(", or [ expected \n");
   //     invalid();
   // }
}

void identifier_list_factors_array()
{
   if (strcmp(currentToken->lexeme, ",") == 0)
   {
       currentToken = getNextToken(fp), tokenDebug();
       identifier_list();
   }
}

void assign_stat()
{
   if (strcmp(currentToken->type, "identifier") == 0)
   {
       currentToken = getNextToken(fp), tokenDebug();
       if (strcmp(currentToken->lexeme, "=") == 0)
       {
           currentToken = getNextToken(fp), tokenDebug();
           // assign_stat_factors();
           expn();
       }
       else
       {
           printf("= expected\n");
           invalid();
       }
   }
   else
   {
       printf("identifier expected\\n");
       invalid();
   }
}

void statement_list()
{
   if (strcmp(currentToken->type, "identifier") == 0)
   {
       statement();
       statement_list();
   }
}
void statement()
{
   char first_of_statement[][10] = {"identifier", "if", "while",
"for"};
   int flag = 0;

   for (int i = 0; i < sizeof(first_of_statement) /
sizeof(first_of_statement[0]); ++i)
   {
       if (i == 0)
       {
           flag++;
           if (strcmp(currentToken->type, "identifier") == 0)
           {
               currentToken = getNextToken(fp), tokenDebug();
               assign_stat();
               if (strcmp(currentToken->lexeme, ";") == 0)
               {
                   currentToken = getNextToken(fp), tokenDebug();
                   return;
               }
               else
               {
                   printf("; expected \n");
                   invalid();
               }
           }
       }
       else if (i == 1)
       {
           flag++;
           if (strcmp(currentToken->lexeme, first_of_statement[i])
== 0)
           {
               currentToken = getNextToken(fp), tokenDebug();
               decision_statement();
               return;
           }
       }
       else if (i == 2 || i == 3)
       {
           flag++;
           if (strcmp(currentToken->lexeme, first_of_statement[i])
== 0)
           {
               currentToken = getNextToken(fp), tokenDebug();
               looping_statement();
               return;
           }
       }
   }
   if (!flag)
   {
       printf("identifier , decision_statement or
looping_statement expected \n");
       invalid();
   }
}
void expn()
{
   simple_expn();
   eprime();
```

```
}
void eprime()
{
   if (strcmp(currentToken->type, "relational_operators") == 0)
   {
       currentToken = getNextToken(fp), tokenDebug();
       simple_expn();
   }
}
void simple_expn()
{
   term();
   seprime();
}

void seprime()
{
   char first_of_seprime[2][2] = {"+", "-"};
   int flag = 0;

   for (int i = 0; i < sizeof(first_of_seprime) /
sizeof(first_of_seprime[0]); ++i)
   {
       if (strcmp(currentToken->lexeme, first_of_seprime[i]) == 0)
           flag++;
   }

   if (flag)
   {
       addop();
       term();
       seprime();
   }
}

void term()
{
   factor();
   tprime();
}

void tprime()
{
   char first_of_tprime[3][3] = {"*", "/", "%"};
   int flag = 0;
   for (int i = 0; i < sizeof(first_of_tprime) /
sizeof(first_of_tprime[0]); ++i)
   {
       if (strcmp(currentToken->lexeme, first_of_tprime[i]) == 0)
           flag++;
   }

   if (flag)
   {
       mulop();
       factor();
       tprime();
   }
}
void factor()
{
   if (strcmp(currentToken->type, "identifier") == 0 ||
strcmp(currentToken->type, "constant") == 0)
   {
       currentToken = getNextToken(fp), tokenDebug();
       return;
   }
   else
   {
       printf("identifier or constant expected \n");
       invalid();
   }
}
void relop()
{
   if (strcmp(currentToken->type, "relational_operators") == 0)
   {
       currentToken = getNextToken(fp), tokenDebug();
       return;
   }
   else
   {
       printf("relational_operators expected \n");
       invalid();
   }
}

void addop()
{
   if (strcmp(currentToken->lexeme, "+") == 0 ||
strcmp(currentToken->lexeme, "-") == 0)
   {
       currentToken = getNextToken(fp), tokenDebug();
       return;
   }
   else
   {
       printf("+ or - expected \n");
       invalid();
   }
}
void mulop()
{
   if (strcmp(currentToken->lexeme, "*") == 0 ||
strcmp(currentToken->lexeme, "/") == 0 || strcmp(currentToken-
>lexeme, "%") == 0)
   {
       currentToken = getNextToken(fp), tokenDebug();
       return;
   }
   else
   {
       printf("* / or mod expected \n");
       invalid();
   }
}

void decision_statement()
{
   if (strcmp(currentToken->lexeme, "if") == 0)
   {
       currentToken = getNextToken(fp), tokenDebug();
       if (strcmp(currentToken->lexeme, "(") == 0)
       {
           currentToken = getNextToken(fp), tokenDebug();
           expn();
           if (strcmp(currentToken->lexeme, ")") == 0)
           {
               currentToken = getNextToken(fp), tokenDebug();
               if (strcmp(currentToken->lexeme, "{") == 0)
               {
                   currentToken = getNextToken(fp), tokenDebug();
                   statement_list();
                   if (strcmp(currentToken->lexeme, "}") == 0)
                   {
                       currentToken = getNextToken(fp),
tokenDebug();
                       dprime();
                   }
                   else
                   {
                       printf("} expected\n");
                       invalid();
                   }
               }
               else
               {
                   printf("{ expected\n");
                   invalid();
               }
           }
           else
           {
               printf(") expected\n");
               invalid();
           }
       }
       else
       {
           printf("( expected\n");
           invalid();
       }
   }
   else
   {
       printf("if expected\n");
       invalid();
   }
}
void dprime()
{
   if (strcmp(currentToken->lexeme, "else") == 0)
   {
       currentToken = getNextToken(fp), tokenDebug();
       if (strcmp(currentToken->lexeme, "{") == 0)
       {
           currentToken = getNextToken(fp), tokenDebug();
           statement_list();
           if (strcmp(currentToken->lexeme, "}") == 0)
           {
               currentToken = getNextToken(fp), tokenDebug();
               return;
           }
           else
           {
               printf("} expected\n");
               invalid();
           }
       }
       else
       {
           printf("{} expected\n");
           invalid();
       }
   }
}

void looping_statement()
{
   if (strcmp(currentToken->lexeme, "while") == 0)
   {
       currentToken = getNextToken(fp), tokenDebug();
       if (strcmp(currentToken->lexeme, "(") == 0)
       {
           currentToken = getNextToken(fp), tokenDebug();
           expn();
           if (strcmp(currentToken->lexeme, ")") == 0)
           {
               currentToken = getNextToken(fp), tokenDebug();
               if (strcmp(currentToken->lexeme, "{") == 0)
               {
                   currentToken = getNextToken(fp), tokenDebug();
                   statement_list();
                   if (strcmp(currentToken->lexeme, "}") == 0)
                   {
                       currentToken = getNextToken(fp),
tokenDebug();
                       return;
                   }
                   else
                   {
                       printf("  }  expected \n");
                       invalid();
                   }
```

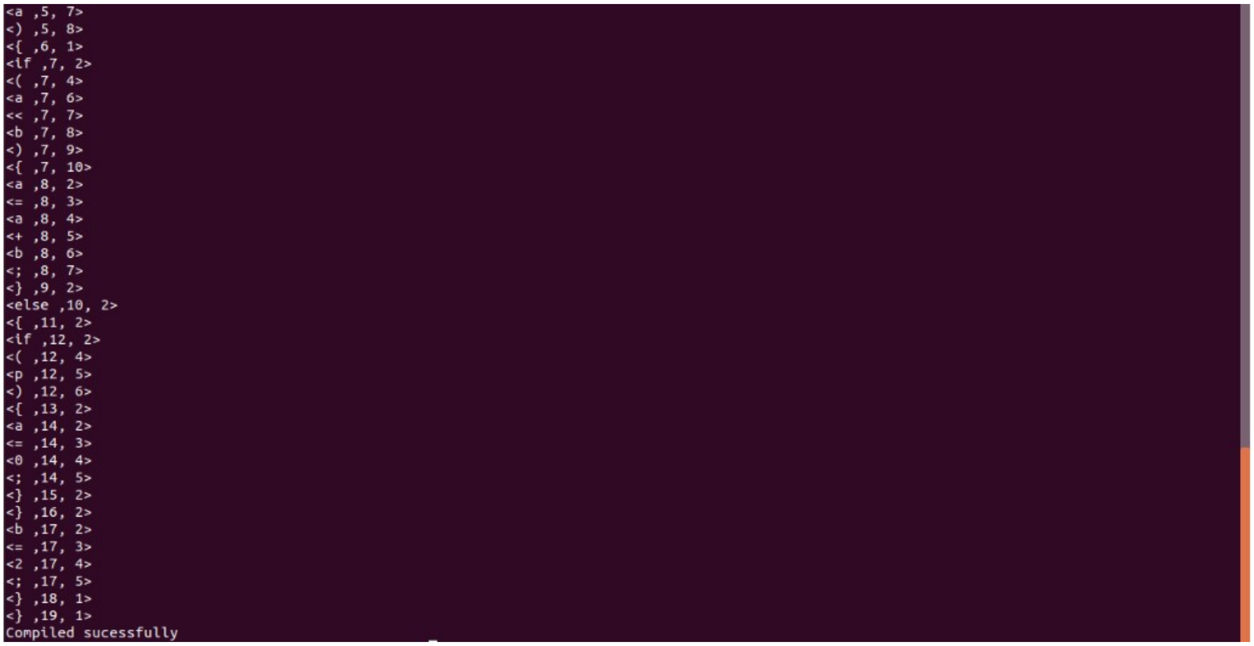
```
                }
                else
                {
                    printf("  { expected \n");
                    invalid();
                }
            }
            else
            {
                printf("  )  expected \n");
                invalid();
            }
        }
        else
        {
            printf("  (  expected \n");
            invalid();
        }
    }
    else if (strcmp(currentToken->lexeme, "for") == 0)
    {
        currentToken = getNextToken(fp), tokenDebug();
        if (strcmp(currentToken->lexeme, "(") == 0)
        {
            currentToken = getNextToken(fp), tokenDebug();
            assign_stat();
            if (strcmp(currentToken->lexeme, ";") == 0)
            {
                currentToken = getNextToken(fp), tokenDebug();
                expn();
                if (strcmp(currentToken->lexeme, ";") == 0)
                {
                    currentToken = getNextToken(fp), tokenDebug();
                    assign_stat();
                    if (strcmp(currentToken->lexeme, ")") == 0)
                    {
                        currentToken = getNextToken(fp),
tokenDebug();
                        return;
                    }
                    else
                    {
                        printf(" ) expected\n");
                        invalid();
                    }
```

```
               }
               else
               {
                   printf(" ; expected\n");
                   invalid();
               }
           }
           else
           {
               printf(" ; expected\n");
               invalid();
           }
       }
       else
       {
           printf(" ( expected\n");
           invalid();
       }
   }
   else
   {
       printf("for or while expected \n");
       invalid();
   }
}

int main(int argc, char const *argv[])
{

   fp = fopen("lab09_RDP_input.c", "r");
   // freopen("lab07_RDP_output.txt", "w", stdout);

   if (fp == NULL)
   {
       printf("Cannot open file \n Exiting.. \n");
       exit(0);
   }

   currentToken = getNextToken(fp), tokenDebug();
   Program();
   success();

   printf("\n*******************Finished Recursive Decent
Parsing*******************\n");
```

```
    return 0;
}
```



```
<main ,1 ,1 >
<( ,1, 5>
<) ,1, 6>
<{ ,2, 1>
<int ,3, 1>
<a ,3, 5>
<, ,3, 6>
<b ,3, 7>
<, ,3, 8>
<p ,3, 9>
<[ ,3, 10>
<25 ,3, 11>
<] ,3, 13>
<; ,3, 14>
<char ,4, 1>
<c ,4, 6>
<; ,4, 7>
<while ,5, 1>
<( ,5, 6>
<a ,5, 7>
<) ,5, 8>
<{ ,6, 1>
<if ,7, 2>
<( ,7, 4>
<a ,7, 6>
<< ,7, 7>
<b ,7, 8>
<) ,7, 9>
<{ ,7, 10>
<a ,8, 2>
<= ,8, 3>
<a ,8, 4>
<+ ,8, 5>
```

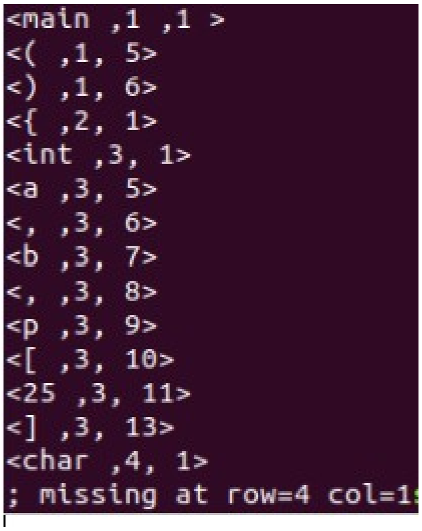


```
<a ,5, 7>
<) ,5, 8>
<{ ,6, 1>
<if ,7, 2>
<( ,7, 4>
<a ,7, 6>
<< ,7, 7>
<b ,7, 8>
<) ,7, 9>
<{ ,7, 10>
<a ,8, 2>
<= ,8, 3>
<a ,8, 4>
<+ ,8, 5>
<b ,8, 6>
<; ,8, 7>
<} ,9, 2>
<else ,10, 2>
<{ ,11, 2>
<if ,12, 2>
<( ,12, 4>
<p ,12, 5>
<) ,12, 6>
<{ ,13, 2>
<a ,14, 2>
<= ,14, 3>
<0 ,14, 4>
<; ,14, 5>
<} ,15, 2>
<} ,16, 2>
<b ,17, 2>
<= ,17, 3>
<2 ,17, 4>
<; ,17, 5>
<} ,18, 1>
<} ,19, 1>
Compiled sucessfully
```

Error input

```
main()
{
int a,b,p[25]
char c;
while(a)
{
 if( a<b){
 a=a+b*c;
 }
 else
 {
  if(p)
  {
   a=0;
  }
 }
 b=2*c;
}
}
```

Error output



```
<main ,1 ,1 >
<( ,1, 5>
<) ,1, 6>
<{ ,2, 1>
<int ,3, 1>
<a ,3, 5>
<, ,3, 6>
<b ,3, 7>
<, ,3, 8>
<p ,3, 9>
<[ ,3, 10>
<25 ,3, 11>
<] ,3, 13>
<char ,4, 1>
; missing at row=4 col=1
```