Ahsanullah University of Science and Technology

Department of Computer Science and Engineering

CSE 4130

Formal Languages and Compiler Lab

Project Report

Submitted To

Md. Aminur Rahman

Lecturer, CSE, AUST

Submitted By

Inzamum Ul Haque

17.02.04.087

Scanning and Filtering a Source Program – Part 1

The program here can scan a source program and filter all the unnecessary whitespace characters, comments and new lines from the code.

Necessary Functions

```
void removeEmptyLines(FILE *p1, FILE *p2);
int isEmpty(char *str);
void removeComments(FILE *p1, FILE *p2);
void singleLineComments(FILE *p1);
void multiLineComments(FILE *p1);
void removeNewLines(FILE *p1,FILE *p2);

int isEmpty(char *str)
{
    char ch;
    do
    {
        ch = *(str++);
        //printf("%c ",ch);
        if(ch != ' ' && ch != '\t' && ch != '\n' && ch != '\r' && ch != '\0')
            return 0;
    }
    while(ch != '\0');
    return 1;//returns 1 if string is empty
}

void removeEmptyLines(FILE *p1, FILE *p2)
{
    char str[Buffer_Size];
    while((fgets(str,Buffer_Size,p1))!= NULL)
    {
```

```
        if(!isEmpty(str))
        {
            fputs(str,p2);
        }
    }
}

void removeComments(FILE *p1,FILE *p2)
{
    char ch,d,e;
    while((ch=fgetc(p1))!=EOF)
    {
        if(ch == '/')
        {
            if((d=fgetc(p1)) == '/')
            {
                singleLineComments(p1);
            }
            else if(d == '*')
            {
                multiLineComments(p1);
            }
            else
            {
                fputc(ch,p2);
                fputc(d,p2);
            }
        }
        else if(ch == '"')
        {
            fputc(ch,p2);
```

```
            while((e=fgetc(p1))!='"')
            {
                fputc(e,p2);
            }
            fputc(e,p2);
        }
        else
        {
            fputc(ch,p2);
        }
    }
}

void singleLineComments(FILE *p1)
{
    char s;
    while((s=fgetc(p1))!=EOF)
    {
        if(s == '\n')
        {
            return;
        }
    }
}

void multiLineComments(FILE *p1)
{
    char m,e;
    while((m=fgetc(p1))!=EOF)
    {
        if(m == '*')
```

```
        {
            e=fgetc(p1);
            if(e == '/')
            {
                return;
            }
        }
    }
}

void removeNewLines(FILE *p1,FILE *p2)
{
    char ch;
    while((ch=fgetc(p1))!=EOF)
    {
        if(ch!='\n')
        {
            fputc(ch,p2);
        }
    }
}
```

Working of the Functions

The function **removeEmptyLines(FILE *p1,FILE *p2)** detects if a line is empty and removes all the whitespace characters, spaces from the code.

Function **isEmpty(char *str)** checks if a line is empty or not.

Functions **removeComments(FILE *p1, FILE *p2), singleLineComments(FILE *p1)** and **multiLineComments(FILE *p1)** detects and removes comments from the code whether it's single line or multi line comment.

Function **removeNewLines(FILE *p1,FILE *p2)** removes the new lines from the code to makes it into a one line code.

Then from the main function we call all the functions. We take input from a file and each time generate a new file to write out the output for the next input.

Lexical Analysis – Part 2

Now, we take the previous file output of part-1 as input. It was scanned and filtered. Now, we separate the lexemes first and then mark the lexemes as different types of tokens like keywords, operators, separators, parentheses, numbers etc.

Necessary Functions

```
bool isDelimiter(char ch);
bool isOperator(char ch);
bool isSeparator(char ch);
bool isParanthesis(char ch);
bool isKeyword(char* str);
bool isInteger(char* str);
bool isFloat(char* str);
bool validIdentifier(char* str);
char* substring(char*str, int left, int right);
void iterate(char* str);


bool isDelimiter(char ch)
{
    if(ch == ' ' || ch == '+' || ch == '-' || ch == '*' ||
```

```
            ch == '/' || ch == ',' || ch == ';' || ch == '>' ||
            ch == '<' || ch == '=' || ch == '(' || ch == ')' ||
            ch == '{' || ch == '}' || ch == '[' || ch == ']')
    {
        return true;
    }
    return false;
}

bool isOperator(char ch)
{
    if(ch == '+' || ch == '-' || ch == '/' || ch == '*' ||
            ch == '<' || ch == '>' || ch == '=')
    {
        return true;
    }
    return false;
}

bool isSeparator(char ch)
{
    if(ch == ';' || ch == ',' || ch == ':' || ch == '\'' || ch == '\"')
    {
        return true;
    }
    return false;
}

bool isParanthesis(char ch)
{
```

```
    if(ch == '(' || ch == ')' || ch == '{' || ch == '}' || ch == '[' || ch ==
']')
    {
        return true;
    }
    return false;
}

bool isKeyword(char* str)
{
    if(!strcmp(str,"if") || !strcmp(str,"else") || !strcmp(str,"while") ||
       !strcmp(str,"do") || !strcmp(str,"break") || !strcmp(str,"continue")
||
       !strcmp(str,"int") || !strcmp(str,"double") || !strcmp(str,"float") ||
       !strcmp(str,"return") || !strcmp(str,"char") || !strcmp(str,"case") ||
       !strcmp(str,"sizeof") || !strcmp(str,"long") || !strcmp(str,"short")
||
       !strcmp(str,"typedef") || !strcmp(str,"switch") ||
!strcmp(str,"unsigned") ||
       !strcmp(str,"void") || !strcmp(str,"static") || !strcmp(str,"struct")
|| !strcmp(str,"goto"))
    {
        return true;
    }
    return false;
}

bool isInteger(char* str)
{
    int i,len=strlen(str);
    if(len==0)
    {
        return false;
```

```
    }
    for(i=0; i<len; i++)
    {
        if(str[i]!='0' && str[i]!='1' && str[i]!='2'
                && str[i]!='3' && str[i]!='4' && str[i]!='5'
                && str[i]!='6' && str[i]!='7' && str[i]!='8'
                && str[i]!='9' || (str[i]=='-' && i>0))
        {
            return false;
        }
    }
    return true;
}

bool isFloat(char* str)
{
    int i,len=strlen(str);
    bool point=false;
    if(len==0)
    {
        return false;
    }
    for(i=0; i<len; i++)
    {
        if(str[i]!='0' && str[i]!='1' && str[i]!='2'
                && str[i]!='3' && str[i]!='4' && str[i]!='5'
                && str[i]!='6' && str[i]!='7' && str[i]!='8'
                && str[i]!='9' && str[i]!='.' || (str[i]=='-' && i>0))
        {
            return false;
        }
```

```
        if(str[i]=='.')
        {
            point=true;
        }
    }
    return point;
}

bool validIdentifier(char* str)
{
    if (str[0] == '0' || str[0] == '1' || str[0] == '2' ||
            str[0] == '3' || str[0] == '4' || str[0] == '5' ||
            str[0] == '6' || str[0] == '7' || str[0] == '8' ||
            str[0] == '9' || isDelimiter(str[0]) == true)
    {
        return (false);
    }
    return (true);
}

char* substring(char*str, int left, int right)
{
    int i;
    char* substr = (char*)malloc(sizeof(char) *(right-left+2));
    for(i=left; i<=right; i++)
    {
        substr[i-left] = str[i];
    }
    substr[right-left+1]='\0';
    return substr;
}
```

```
void iterate(char* str)
{
    int left = 0,right = 0;
    int len = strlen(str);
    char opening_bracket = '[';
    FILE *file;
    file = fopen("Output_Assignment2_2.txt","w");
    char ch;
    while(right<=len && left<=right)
    {
        if(isDelimiter(str[right])==false)
        {
            right++;
        }
        if(isDelimiter(str[right])==true && left==right)
        {
            if(isOperator(str[right])==true)
            {
                printf("[%c operator]  ",str[right]);
                fputc(opening_bracket,file);
                fputc(str[right],file);
                fputs(" operator]  ",file);
            }
            else if(isSeparator(str[right])==true)
            {
                printf("[%c separator]  ",str[right]);
                fputc(opening_bracket,file);
                fputc(str[right],file);
                fputs(" separator]  ",file);
            }
```

```
            else if(isParanthesis(str[right])==true)
            {
                printf("[%c paranthesis]  ",str[right]);
                fputc(opening_bracket,file);
                fputc(str[right],file);
                fputs(" paranthesis]  ",file);
            }
            right++;
            left=right;
        }
        else if(isDelimiter(str[right])==true && left!=right || (right==len
&& left!=right))
        {
            char* sub = substring(str, left, right-1);
            if(isKeyword(sub)==true)
            {
                printf("[%s keyword]  ",sub);
                fputc(opening_bracket,file);
                fputs(sub,file);
                fputs(" keyword]  ",file);
            }
            else if(isInteger(sub)==true)
            {
                printf("[%s number]  ",sub);
                fputc(opening_bracket,file);
                fputs(sub,file);
                fputs(" number]  ",file);
            }
            else if(isFloat(sub)==true)
            {
                printf("[%s number]  ",sub);
```

```
                fputc(opening_bracket,file);
                fputs(sub,file);
                fputs(" number]  ",file);
            }
            else if(validIdentifier(sub)==true && isDelimiter(str[right-
1])==false)
            {
                printf("[%s identifier]  ",sub);
                fputc(opening_bracket,file);
                fputs(sub,file);
                fputs(" identifier]  ",file);
            }
            else if(validIdentifier(sub)==false && isDelimiter(str[right-
1])==false)
            {
                printf("[%s unknown]  ",sub);
                fputc(opening_bracket,file);
                fputs(sub,file);
                fputs(" unknown]  ",file);
            }
            left=right;
        }
    }
    fclose(file);
    return;
}
```

Working of the Functions

We first take the input and then separate the lexemes first by delimiters to distinguish the character tokens and then we categorize the tokens.

Function **isDelimiter(char ch)** checks if a given character is a delimiter or not. Delimiters can be ;, ‘, ’’, |, <, >, <=, >=, =, +, -, *, / etc.

isOperator(char ch) checks if the character is an operator or not. Operators can be +, -, *, /, <, > etc.

isSeparator(char ch) checks if the given character is a separator. Separators can be ;, :, \ etc.

isParanthesis(char ch) checks if the character is a parenthesis like (,), {, }, [,].

Function **isKeyword(char* str)** checks the given string is a keyword or not.

Function **isInteger(char* str)** checks if a given character is integer or not.

isFloat(char* str) checks if the character is a floating point number. It also checks the decimal points.

validIdentifier(char* str) checks the valid or invalid identifier name. Like the function names, variable names etc.

This function **substring(char*str, int left, int right)** takes a string input and it splits the substring format to check if it is an operator, a separator or a valid identifier etc.

Function **iterate(char* str)** iterates through the strings, splits them into the substrings and then categorizes the tokens under each category whether it’s operator, separator, parentheses, numbers, keywords etc.

We call them all by taking the input in files from the main function and then generate each file for the output to use it for the next input. Each time we call a function opening a file pointer from the main function, we need to close the file pointer before calling another function.

Symbol Table Construction and Management – Part 3

Here we generate a symbol table where all the identifiers are stored with necessary information about them. When a new variable is entered the program puts it in a new entry in the table. When a variable is referred, it first checks up to the table if that variable is defined in the table and takes the necessary information from the table.

Necessary Functions

```
struct Table
{
    int sl;
    char name[100];
    char datatype[100];
    char idtype[100];
    char scope[100];
    char value[100];
} symbolTable[100];
void display(FILE *f2);
bool searchId(char str[],int i);
char *findDatatype(int i);
int findScope(int i);
void implementing_Symbol_Table(FILE *f1,FILE *f2);
void removing_Lexemes_Except_Id(FILE *f1,FILE *f2);
void final_Output(FILE *f1,FILE *f2);

void display(FILE *f2)
{
    char ch;
    while((ch=fgetc(f2))!=EOF)
    {
```

```
        printf("%c",ch);
    }
    printf("\n\n");
}

bool searchId(char str[],int i)
{
    for(int cnt=0; cnt<i; cnt++)
    {
        if(!strcmp(str,tokens[cnt]))
        {
            return false;
        }
    }
    return true;
}

char *findDatatype(int i)
{
    char *dt;
    while(i>=0)
    {
        if((!strcmp(tokens[i],"float")))// ||
(!strcmp(tokens[i],"double")) || (!strcmp(tokens[i],"float"))
(!strcmp(tokens[i],"long")) || (!strcmp(tokens[i],"char")))
        {
            strcpy(dt,tokens[i]);
            return dt;
        }
```

```
        i--;
    }
}

int findScope(int i)
{
    for(int j=i;j>=0;j--)
    {
        if(!strcmp(tokens[j],"("))
            return j-1;
        else if(!strcmp(tokens[j],")"))
            return 0;//global
    }
    return 0;//means global scope
}

void implementing_Symbol_Table(FILE *f1,FILE *f2)
{
    char ch;
    char str[100];
    int i=0;
    while((ch=fgetc(f1))!=EOF)
    {
        if(ch!='[')
        {
            if(ch==']')
            {
                fileInput[i]=' ';
```

```
                i++;
            }
            else
            {
                fileInput[i]=ch;
                i++;
            }
        }
    }
    /*for(int i=0; i<strlen(fileInput); i++)
    {
        printf("%c",fileInput[i]);
    }*/
    int j=0,ctr=0;
    for(int i=0; i<strlen(fileInput); i++)
    {
        if(fileInput[i]==' ' || fileInput[i]=='\0')
        {
            tokens[ctr][j]='\0';
            ctr++;
            j=0;
        }
        else
        {
            tokens[ctr][j]=fileInput[i];
            j++;
        }
    }
```

```
    /*for(int i=0; i<ctr; i++)
    {
        printf("%d. %s\n",i,tokens[i]);
    }*/
    int serial=0;
    //Inserting serial number and name in table
    for(int i=0; i<ctr; i++)
    {
        if(!strcmp(tokens[i],"id"))
        {
            symbolTable[serial].sl=serial+1;
            strcpy(str,tokens[i+1]);
            if(searchId(str,i+1))
            {
                strcpy(symbolTable[serial].name,tokens[i+1]);
                serial++;
            }
        }
    }
    //Inserting datatype in table
    char datatype[100];
    serial=0;
    for(int i=0; i<ctr; i++)
    {
        if(!strcmp(tokens[i],"id"))
        {
            if(!strcmp(tokens[i+1],symbolTable[serial].name))
            {
```

```
                strcpy(symbolTable[serial].datatype,tokens[i-1]);
                serial++;
            }
        }
    }
    //Inserting Idtype
    serial = 0;
    for(int i=0; i<ctr; i++)
    {
        if(!strcmp(tokens[i],symbolTable[serial].name))
        {
            if(!strcmp(tokens[i+1],"("))
            {
                strcpy(symbolTable[serial].idtype,"func");
                serial++;
            }
            else
            {
                strcpy(symbolTable[serial].idtype,"var");
                serial++;
            }
        }
    }
    j=serial;
    //Inserting value
    serial = 0;
    for(int i=0; i<ctr; i++)
    {
```

```
        if(!strcmp(tokens[i],symbolTable[serial].name) &&
!strcmp(symbolTable[serial].idtype,"var"))
        {
            if(!strcmp(tokens[i+1],"="))
            {
                strcpy(symbolTable[serial].value,tokens[i+2]);
                serial++;
            }
            else
            {
                strcpy(symbolTable[serial].value,"-");
                serial++;
            }
        }
    }
    //Find scope
    int scope;
    serial=0;
    for(int i=0;i<ctr;i++)
    {
        if(!strcmp(tokens[i],"id"))
        {
            scope = findScope(i);
            if(scope==0)
            {
                strcpy(symbolTable[serial].scope,"global");
                serial++;
            }
```

```
            else
            {
                strcpy(symbolTable[serial].scope,tokens[scope]);
                serial++;
            }
        }
    }
    printf("---Symbol Table---\n");
    printf("Sl\tName\tIdtype\tDatatype\tScope\tValue\n");
    for(int i=0; i<j; i++)
    {
printf("%d\t%s\t%s\t%s\t\t%s\t%s\n",symbolTable[i].sl,symbolTable[i].n
ame,symbolTable[i].idtype,symbolTable[i].datatype,symbolTable[i].scope
,symbolTable[i].value);
    }
}


void removing_Lexemes_Except_Id(FILE *f1,FILE *f2)
{
    char ch,c,space=' ';
    char string[100],separate[100][100];
    int ctr,j;
    while((ch=fgetc(f1))!=EOF)
    {
        if(ch=='[')
        {
            fputc(ch,f2);
            int i=0;
            memset(string,'\0',sizeof(string));
```

```
do
{
    ch=fgetc(f1);
    if(ch!=']')
    {
        string[i]=ch;
        i++;
    }
}
while(ch!=']');
c=ch;
string[i]='\0';
j=0,ctr=0;
for(int i=0; i<=(strlen(string)); i++)
{
    if(string[i]==' ' || string[i]=='\0')
    {
        separate[ctr][j]='\0';
        ctr++;
        j=0;
    }
    else
    {
        separate[ctr][j]=string[i];
        j++;
    }
}
if(!strcmp(separate[0],"id"))
```

```
            {
                fputs(separate[0],f2);
                fputc(space,f2);
                fputs(separate[1],f2);
            }
            else
            {
                fputs(separate[1],f2);
            }
            fputc(c,f2);
        }
    }
}

void final_Output(FILE *f1,FILE *f2)
{
    int serial=0;
    for(int i=0;i<strlen(fileInput);i++)
    {
        if(fileInput[i]=='i' && fileInput[i+1]=='d')
        {
            fileInput[i+3]=(char)symbolTable[serial].sl;
            serial++;
        }
    }
    for(int i=0;i<strlen(fileInput);i++)
    {
        fputc(fileInput[i],f2);
```

```
    }
}
```

Working of the Functions

First, we define a structure data type here to build a symbol table or hash table, where we can store name, value, scope, datatype, serial number of a variable.

display(FILE *f2) function displays strings of characters from a file.

searchId(char str[],int i) function searches for id in the symbol table or hash table. If a declared variable is previously found on the hash table, it returns true, otherwise false.

***findDatatype(int i)** function finds the datatype of the declared variable from the hash table.

Function **findScope(int i)** finds the scope of the variable whether it is declared globally or locally inside a function.

implementing_Symbol_Table(FILE *f1,FILE *f2) this function generates the symbol table or hash table for each new variable in the code. Before generating, it first checks if the variable is declared before of the same name, if a value is present there or not, then updates it, searches its scope and datatype. Same datatype of the same variable can't be declared if their scopes are same.

Function **removing_Lexemes_Except_Id(FILE *f1,FILE *f2)** removes the lexemes from the tokens except the identifiers. Because we need to entry the identifiers into the hash table and update their values, datatypes and scopes accordingly.

final_Output(FILE *f1,FILE *f2) prints the final output from the hash table.

Here we take the output of part 2 and use them as input for this part. In the main function, we call all the functions accordingly to generate the hash table.

Detecting Simple Syntax Errors – Part 4

Here we find the simplest syntax errors from the codes and generate error message for the syntax errors.

Necessary Functions

```
struct Stack
{
    int top;
    int parentheses[buffer];
    int line[buffer];
} st;
void initialize();
bool isFull();
bool isEmptyy();
void push(int par,int line);
int pop1(int line);
int pop2();
void printLineNumbers(FILE *f1,FILE *f2);
void checkSemicolon();
void checkParantheses();
void checkIfElse();
void checkKeyword();

void initialize()
{
    st.top=-1;
}
```

```
//Check to see if stack is full
bool isFull()
{
    if(st.top >= buffer-1)
    {
        return true;
    }
    else
    {
        return false;
    }
}

//Check to see if stack is empty
bool isEmptyy()
{
    if(st.top == -1)
    {
        return true;
    }
    else
    {
        return false;
    }
}

//Add element to stack
```

```
void push(int par,int line)
{
    if(isFull())
    {
        printf("Stack error\n");
    }
    else
    {
        st.parentheses[st.top+1] = par;
        st.line[st.top+1] = line;
        st.top++;
    }
}

//Pop parentheses from the stack
int pop1(int line)
{
    if(isEmptyy())
    {
        printf("Mismatched parentheses at line %d,\n",line);
    }
    else
    {
        st.top=st.top-1;
        return st.parentheses[st.top+1];
    }
}
```

```
//Pop line number from the stack
int pop2()
{
    return st.line[st.top+1];
}

void printLineNumbers(FILE *f1,FILE *f2)
{
    int line=1;
    char ch;

    int i=0;
    while((ch=fgetc(f1))!=EOF)
    {
        temp[i]=ch;
        i++;
    }
    /*for(int i=0; i<strlen(temp); i++)
    {
        printf("%c",temp[i]);
    }*/
    int j=0,ctr=1;
    for(int i=0; i<strlen(temp); i++)
    {
        if(temp[i]=='\n')
        {
            addline[ctr][j]='\0';
            ctr++;
```

```
            j=0;
        }
        else
        {
            addline[ctr][j]=temp[i];
            j++;
        }
    }
    for(int i=1; i<ctr; i++)
    {
        printf("%d. %s\n",i,addline[i]);
        fprintf(f2,"%d %s\n",i,addline[i]);
    }
    arraylen=ctr;
}


void checkSemicolon()
{
    printf("\n\n");
    for(int i=1; i<arraylen; i++)
    {
        for(int j=0; j<strlen(addline[i]); j++)
        {
            if(addline[i][j]=='f' && addline[i][j+1]=='o' &&
addline[i][j+2]=='r' && addline[i][j+3]=='(' && addline[i][j+4]==';'
&& addline[i][j+5]==';' && addline[i][j+6]==')')
            {
                continue;
            }
```

```
            if(addline[i][j]==';' && addline[i][j+1]==';')
            {
                printf("Duplicate token at line %d,\n",i);
                break;
            }
        }
    }
}

void checkParantheses()
{
    initialize();
    for(int i=1; i<=arraylen; i++)
    {
        for(int j=0; j<strlen(addline[i]); j++)
        {
            if(addline[i][j]=='(' || addline[i][j]=='{' ||
addline[i][j]=='[')
            {
                push(addline[i][j],i);
            }
            else if(addline[i][j]==')' || addline[i][j]=='}' ||
addline[i][j]==']')
            {
                pop1(i);
                pop2();
            }
        }
    }
```

```
    if(!isEmptyy())
    {
        printf("Unmatched parentheses at line
%d,\n",st.line[st.top+1]);
    }
}

void checkIfElse()
{
    char ifElse[buffer][buffer];
    int x=1,ifElseArr[buffer];
    for(int i=1; i<arraylen; i++)
    {
        for(int j=0; j<strlen(addline[i]); j++)
        {
            if(addline[i][j]=='i' && addline[i][j+1]=='f')
            {
                strcpy(ifElse[x],"if");
                ifElseArr[x]=i;
                x++;
            }
            if(addline[i][j]=='e' && addline[i][j+1]=='l' &&
addline[i][j+2]=='s' && addline[i][j]=='e')
            {
                strcpy(ifElse[x],"else");
                ifElseArr[x]=i;
                x++;
            }
```

```
            if(addline[i][j]=='e' && addline[i][j+1]=='l' &&
addline[i][j+2]=='s' && addline[i][j]=='e' && addline[i][j]==' ' &&
addline[i][j]=='i' && addline[i][j]=='f')
            {
                strcpy(ifElse[x],"else if");
                ifElseArr[x]=i;
                x++;
            }
        }
    }
    for(int i=1; i<=x; i++)
    {
        if(!strcmp(ifElse[i],"else") && !strcmp(ifElse[i+1],"else"))
        {
            printf("Unmatched 'else' at line %d,\n",ifElseArr[i+1]);
        }
        if(!strcmp(ifElse[i],"else") && !strcmp(ifElse[i+1],"else
if"))
        {
            printf("Unmatched 'else if' at line
%d,\n",ifElseArr[i+1]);
        }
    }
}


void checkKeyword()
{
    char separate[buffer][buffer];
    int i,j,ctr=0,x=0;
```

```
    for(i=1; i<=arraylen; i++)
    {
        for(j=0; j<strlen(addline[i]); j++)
        {
            if(addline[i][j]=='\0' || addline[i][j]==' ')
            {
                separate[ctr][x]='\0';
                ctr++;
                x=0;
            }
            else
            {
                separate[ctr][x]=addline[i][j];
                x++;
            }
        }
    }
}
```

Working of the Functions

In order to detect the errors, we need to add the line numbers in the codes from where we can learn in which line number the error has occurred. We take a 2d matrix to store the code strings and then iterate each line for matching errors.

printLineNumbers(FILE *f1,FILE *f2) function adds the required line number in the codes and prints them.

initialize(), isFull(), isEmptyy(), push(int par,int line), pop1(int line), pop2() all the functions are part of the stack to push and check for parenthesis errors in each line. Notice, there are 2 pop function, pop1 and pop2. Pop1 pops the parenthesis from the stack and pop2 pops the line number from the stack. If a parenthesis is mismatched then we print an error message for that.

Function **checkSemicolon()** checks for semicolon error in each line of the code. For more than one semicolon in a line or don't have semicolon in a line we print error message.

checkParantheses() checks for parenthesis errors by calling the push, pop functions and matches the parenthesis into the stack.

checkIfElse() checks for if else errors. It detects any unmatched if else errors and print out the error message.

checkKeyword() checks for unidentified keywords in the lines.

For this, we take the input from the first part of the code in a file, and call them from the main function.