1.

```python
import string
bad_chars = [';', ':', '!', "*"]
test_string = " <a-sequence> ::= a | <a-sequence> a "
print ("Original String : " + test_string)
delete_dict = {sp_character: '' for sp_character in string.punctuation}
delete_dict[' '] = ''
table = str.maketrans(delete_dict)
test_string = test_string.translate(table)
print ("Resultant list is : " + str(test_string))
```

2.

The weakest precondition is

y = 2 x + 1

y = 3 x − 1 { y > 3}

3.

```c
#include<stdio.h>
int main()
{  int n = 1;
 while(n != 1)
{    if(n % 2 == 0){ //for even numbers
   n = n / 2;
    printf("%d\n", n);
   }    else{ //for odd numbers
    n = n * 3 + 1;
    printf("%d\n", n);
 } }
  return 0;}
```

4.

```
#include <bits/stdc++.h>
using namespace std;
int convert(int m, int n)
{
    if (m == n)
        return 0;
    if (m > n)
        return m - n;
    if (m <= 0 && n > 0)
        return -1;
    if (n % 2 == 1)
        return 1 + convert(m, n + 1);
    else
        return 1 + convert(m, n / 2);
}
int main()
{
    int m = 3, n = 11;
    cout << "Minimum number of operations : "
         << convert(m, n);
    return 0;
}
```

5.

```
#include<iostream>
#include<fstream>
#include<stdlib.h>
```

```
#include<string.h>
#include<ctype.h>

using namespace std;

int isKeyword(char buffer[]){
char keywords[32][10] = {"auto","break","case","char","const","continue","default",
"do","double","else","enum","extern","float","for","goto",
"if","int","long","register","return","short","signed",
"sizeof","static","struct","switch","typedef","union",
"unsigned","void","volatile","while"};
int i, flag = 0;
for(i = 0; i < 32; ++i){
if(strcmp(keywords[i], buffer) == 0){
flag = 1;
break;
}
}
return flag;
}

int main(){
char ch, buffer[15], operators[] = "+-*/%=";
ifstream fin("program.txt");
int i,j=0;
if(!fin.is_open()){
cout<<"error while opening the file\n";
exit(0);
}
while(!fin.eof()){
  ch = fin.get();

for(i = 0; i < 6; ++i){
  if(ch == operators[i])
  cout<<ch<<" is operator\n";
  }

  if(isalnum(ch)){
  buffer[j++] = ch;
  }
  else if((ch == ' ' || ch == '\n') && (j != 0)){
  buffer[j] = '\0';
  j = 0;

  if(isKeyword(buffer) == 1)
  cout<<buffer<<" is keyword\n";
  else
  cout<<buffer<<" is indentifier\n";
  }

}
fin.close();
return 0;
}
```

6.

```
#include <stdbool.h>
#include <stdio.h>
```

```
#include <string.h>
#include <stdlib.h>

// Returns 'true' if the character is a DELIMITER.
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);
}

// Returns 'true' if the character is an OPERATOR.
bool isOperator(char ch)
{
if (ch == '+' || ch == '-' || ch == '*' ||
ch == '/' || ch == '>' || ch == '<' ||
ch == '=')
return (true);
return (false);
}

// Returns 'true' if the string is a VALID IDENTIFIER.
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);
}

// Returns 'true' if the string is a KEYWORD.
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, "char")
|| !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);
}

// Returns 'true' if the string is an INTEGER.
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);

}


// Returns 'true' if the string is a REAL NUMBER.

bool isRealNumber(char* str)

{

int i, len = strlen(str);

bool hasDecimal = 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] == '.')
```

```
hasDecimal = true;
}
return (hasDecimal);
}

// Extracts the SUBSTRING.
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);
}

// Parsing the input STRING.
void parse(char* str)
{
int left = 0, right = 0;
int len = strlen(str);

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' IS AN OPERATOR\n", str[right]);

right++;
left = right;
} else if (isDelimiter(str[right]) == true && left != right
|| (right == len && left != right)) {
char* subStr = subString(str, left, right - 1);

if (isKeyword(subStr) == true)
printf("'%s' IS A KEYWORD\n", subStr);

else if (isInteger(subStr) == true)
printf("'%s' IS AN INTEGER\n", subStr);

else if (isRealNumber(subStr) == true)
printf("'%s' IS A REAL NUMBER\n", subStr);

else if (validIdentifier(subStr) == true
&& isDelimiter(str[right - 1]) == false)
printf("'%s' IS A VALID IDENTIFIER\n", subStr);

else if (validIdentifier(subStr) == false
&& isDelimiter(str[right - 1]) == false)
printf("'%s' IS NOT A VALID IDENTIFIER\n", subStr);
left = right;
}
}
return;
```

```
}

// DRIVER FUNCTION
int main()
{
// maximum length of string is 100 here
char str[100] = "int a = b + 1c; ";

parse(str); // calling the parse function

return (0);
}
```

7.

Symbol Table is an important data structure created and maintained by the compiler in order to keep track of semantics of variable i.e. it stores information about scope and binding information about names, information about instances of various entities such as variable and function names, classes, objects, etc.

It is built in lexical and syntax analysis phases.

The information is collected by the analysis phases of compiler and is used by synthesis phases of compiler to generate code.

It is used by compiler to achieve compile time efficiency.

It is used by various phases of compiler as follows :-

Lexical Analysis: Creates new table entries in the table, example like entries about token.

Syntax Analysis: Adds information regarding attribute type, scope, dimension, line of reference, use, etc in the table.

Semantic Analysis: Uses available information in the table to check for semantics i.e. to verify that expressions and assignments are semantically correct(type checking) and update it accordingly.

Intermediate Code generation: Refers symbol table for knowing how much and what type of run-time is allocated and table helps in adding temporary variable information.

Code Optimization: Uses information present in symbol table for machine dependent optimization.

Target Code generation: Generates code by using address information of identifier present in the table.

Symbol Table entries – Each entry in symbol table is associated with attributes that support compiler in different phases.
Items stored in Symbol table:

Variable names and constants

Procedure and function names

Literal constants and strings

Compiler generated temporaries

Labels in source languages

Information used by compiler from Symbol table:

Data type and name

Declaring procedures

Offset in storage

If structure or record then, pointer to structure table.

For parameters, whether parameter passing by value or by reference

Number and type of arguments passed to function

Base Address

Operations of Symbol table – The basic operations defined on a symbol table include:

Implementation of Symbol table –
Following are commonly used data structure for implementing symbol table :-

List –

In this method, an array is used to store names and associated information.

A pointer "available" is maintained at end of all stored records and new names are added in the order as they arrive

To search for a name we start from beginning of list till available pointer and if not found we get an error "use of undeclared name"

While inserting a new name we must ensure that it is not already present otherwise error occurs i.e. "Multiple defined name"

Insertion is fast O(1), but lookup is slow for large tables – O(n) on average

Advantage is that it takes minimum amount of space.

Linked List –

This implementation is using linked list. A link field is added to each record.

Searching of names is done in order pointed by link of link field.

A pointer “First” is maintained to point to first record of symbol table.

Insertion is fast O(1), but lookup is slow for large tables – O(n) on average

Hash Table –

In hashing scheme two tables are maintained – a hash table and symbol table and is the most commonly used method to implement symbol tables..

A hash table is an array with index range: 0 to tablesize – 1.These entries are pointer pointing to names of symbol table.

To search for a name we use hash function that will result in any integer between 0 to tablesize – 1.

Insertion and lookup can be made very fast – O(1).

Advantage is quick search is possible and disadvantage is that hashing is complicated to implement.

Binary Search Tree –

Another approach to implement symbol table is to use binary search tree i.e. we add two link fields i.e. left and right child.

All names are created as child of root node that always follow the property of binary search tree.

Insertion and lookup are O(log2 n) on average

8.

```
x = 1;          # x is a variable name assigned to 1

y = 3;          # y is a variable name assigned to 3

z = 5;          # z is a variable name assigned to 5

def sub1():     # sub1 is a function

a = 7;          # a is a variable name assigned to 7

y = 9;          # y is a variable name assigned to 9

z = 11;         # z is a variable name assigned to 11

# . . . Line 1
```

```
def sub2():        # sub2 is a function

global x;          # x is a globally defined variable name

a = 13;            # a is a variable name assigned to 13

x = 15;            # x is a variable name assigned to 15

w = 17;            # w is a variable name assigned to 17

# . . . Line 2

def sub3():        # sub3 is a function

nonlocal a;        # a is non local variable can be inner and outer of the function

a = 19;            # a is a variable name assigned to 19

b = 21;            # b is a variable name assigned to 21

z = 23;            # z is a variable name assigned to 23

# . . . Line 3

def sub2():        # sub2 is a function

nonlocal a;        # a is non local variable can be inner and outer of the function

global z;          # x is a globally defined variable name

a = 19;            # a is a variable name assigned to 19

b = 21;            # b is a variable name assigned to 21

z = 23;            # z is a variable name assigned to 23

# . . . Line
```

9.

With dynamic scope, a global identifier refers to the identifier associated with the most recent environment, and is uncommon in modern languages. In technical terms, this means that each identifier has a global stack of bindings and the occurrence of a identifier is searched in the most recent binding.

In simpler terms, in dynamic scoping the compiler first searches the current block and then successively all the calling functions.