Flex And Bison:

FLEX(Fast Lexical Analyzer Generator) is a complex program that generates a lexical analyzer. This lexical analyzer divides the input from a '.l' file into meaningful units(tokens), which are used to find identifiers or keywords in a c program. It scans a '.l' file using "yylex()" function and generates yy.lex.c file, which is then compiled by gcc compiler to generate the lexical analyzer.

Bison in a general purpose parser generator. It takes the tokens generated from flex and grammar rules written in the .y file and generates a syntax tree. In the '.y' file, the grammar rules are written in BNF form. YACC(Yet Another Compiler Compiler) is used to generate a '.h' file and '.c' file from the '.y' file. The '.h' file contains the definitions of the tokens the '.l' file returns. The function "yyparse()" is called by yacc which calls the yylex() function to get the tokens. Bison stores the parsed tokens in two stacks: parse stack, and value stack. In value stack the values of the parsed tokens are assigned automatically starting from ASCII value 258. The parsed tree generated by Bison is calculated from bottom to top.

Using Bison, a tab.h and a tab.c file is created. This tab.h contains the definitions of the tokens and used by flex to identify tokens from the .l file, which generates lex.yy.c file. Finally these two .c files are used by the gcc compiler to apply the grammar rules in the input file and generate output.

TOKENS

Tokens or non-terminals are sequence of characters that can not be divided in any other terminals. Keywords, identifiers, constants, operators etc. are considered tokens.

The tokens used in this language are of two types:

1) Tokens having type
2) Tokens having no type

Typed tokens are under the structure data “data_type”. “data_type” has 6 variables:

1) int integer: it contains the integer value of the token given input by the user.
2) double floating: contains the floating value of the token given input by the user.
3) char name[1000]: contains the name of the variable declared by the user.
4) int hash_value: contains the hashing value of the variable name declared by the user.
5) char str[1000]: contains the string value initialized to the variable.
6) int type: contains the type of the variable(Integer, floating, String)

The details of the tokens are described below:

Tokens having type:

***1. VAR_NAME**:*

It is used to identify a variable name when the user declares a variable. A variable name must start with a ‘$’ or ‘#’ and small letter [a-z] followed by any number of capital letters, small letters, digits. This token also returns the variable name declared by the user.

Sample variable name: **#mark, #e1, $AM, $Am3.**

***2. INT, DOUBLE, STRING Tokens**:*

INT identifies the keyword ‘int’, DOUBLE identifies the keyword ‘double’, STRING identifies the keyword ‘string’ from the input files. They also return respective types. Type is set ‘1’ when ‘int’ is found in the input file. Similarly type ‘2’ is set for double and type ‘3’ is set for ‘string’.

3.INTEGER_VALUE:

It is used to identify a integer number given by the user in the input file. The format of the integer number is: '+'/ '-' / 'none' followed by any number of digits([0-9]+). This token converts the yytext into a positive or negative integer number according to the + or – sign and returns it. It also returns the type of the integer value along with the number which is 1.

Sample integer value: **+100, -934,12, 3, 0**

4.FLOATING_VALUE:

Identifies a floating type value and returns the value. The format of floating type value is:

'+'/ '-'/ ' ' followed by [0-9]*"."[0-9]+. It converts the yytext into a floating value and returns it. It also returns the type which is 2 for floating values.

Sample floating value: **-1.2 4324.445 10.322 1.2**

5.STRING_VALUE:

Used to identify a string value from the input file. A string value starts with " ' " symbol and ends with " ' " symbol. Inside this the format will be '[a-zA-Z0-9@$%^&*(){}_+-,.:\|?><]*'.

It returns the yytext without the ' and ' symbols and the type is returned 3.

Sample string value: **'hello' 'abcd' '@tHEre' 'A9bc'**

6.ARRAY:

Used to identify the keyword 'array' from the input file. 'array' is used to declare a array type variable. It also returns the type of the array.

7.PRINT:

Used to identify the keyword 'print' from the input file. It is used as a function to print different variable values, constants, expressions etc. It also gives the type of the content it is printing.

Tokens without type:

These tokens are used to identify brackets, operators, some keywords. Details are given below:

1.PS: Identifies '[' from the input.

2.PE: Identifies ']' from the input.

3.MAIN: Used to identify the keyword ‘Start’ from the input. It indicates the beginning of the main user program.

4.STOP: Identifies ‘.’ from the input. It indicates the end of a statement line.

5.COLON: Identifies ‘:’ from the input.

6.DASH: Identifies ‘-’ from the input. It is also used as a operator and when used as operator it means subtraction of two floating or integer values or variables.

Sample subtraction: **1.2-3 3-2 #ab-2 #ab-#bc #ab-1-2-#bc (Here #ab #bc are floating or integer variables), associativity is left associative.**

7.COMMA: Identifies ‘ , ’ from the input. Used when multiple assignment or declaration operations are done in a single line to separate them.

8.DECLARE: Identifies keyword ‘Declaring’ from the input. Used when declaring a variable.

Sample: **Declaring #ab< int >.**

9.BAAM and DAAN: BAAM identifies ‘<’ from the input and DAAN identifies ‘>’ from the input. Used for declaration purposes to define the type of the variable described above.

10.EQUALS: Identifies ‘==’ from the input. Used to assign value to a variable

Sample: **#ab == 2 #dc2A == 1.5 #str1 == ‘hello’**

11.FBO and FBC: FBO identifies ‘(’ and FBC identifies ‘)’. They are used in print(), loop structure, conditional structures, switch case structures, constructor for class.

Sample: **print() while() for() if() case() class< #class1 > #aaabb (1,1.5).**

12.PLUS: Identifies ‘+’. It means addition of two integer or floating values.

Sample: **1+2 4.5+3.5+2 #ab+2+1.5+#bc (#ab and #bc are variables), associativity is left associative.**

13.DIV: Identifies ‘/’ form the input file. It means division operation of two integer or floating values. Associativity is **left associative**.

14.MUL: Identifies ‘*’ form the input file. It means multiplication operation of two integer or floating values. Associativity is **left associative**.

15.POW: Identifies ‘^^’ form the input file. It means power operation of two integer or floating values. Associativity is **left associative**.

16.LOG: Identifies keyword ‘log’ from the input. It is used to do log 10 based operations on variables, integer or floating numbers.

17.SIN COS TAN: SIN identifies ‘sin’, COS identifies ‘cos’, TAN identifies ‘tan’ from the input file. They are used for sine, cosine and tangent operations on variables or integer or floating values.

18.Relational operators: GTR identifies ‘>>’ and means ‘greater than’, LESS identifies ‘<<’ and means ‘less than’, GEQ identifies ‘>=’ means ‘greater or equals than’, LEQ identifies ‘<=’ means ‘less equals than’, EQS identifies ‘=’ means ‘equals to’. They are all left associative.

18.MOD: identifies ‘%%’ and used to do mod operation. It is **left associative**.

19.LCM: identifies ‘lcm’ keyword, used to do lcm operation.

20.GCD: identifies ‘gcd’ keyword and does gcd operation.

21.INPUT: identifies ‘input’ keyword and used for taking input from user.

22.TAKE: identifies ‘take’ keyword from input file and used for input syntax purposes.

23Loops: FOR is used to identify ‘for’, WHILE to identify ‘while’, DO to identify ‘do’ keywords from the input file. These are used for loop syntax purposes.

24.DCRM: identifies ‘---’ keyword from the input and used to decrement the value of an expression by 1.

25.Conditionals: IF identifies ‘if’, ELSE identifies ‘else’ and are used in conditional syntax structures.

26.Switch-case structure: SWITCH is used to identify ‘switch’, CASE is used to identify ‘case’, DEFT identifies ‘default’. They are used in switch case structure.

27.CLASS: identifies ‘class’ from the input file and used while declaring classes.

FEATURES

Class definition:

This section is kept for defining a class. A class can have two type of variables: integer and double. The sample code for defining a class is given below:

Sample 1: class #class1 [

#i1 < int >, #i2 < double >.

].

Beginning of the program:

The program starts with ‘Start:-’ followed by ‘[ ]’ braces. All the statements of the program must be in the ‘[ ]’ braces.

Variable Declaration:

There are four types of data types : **int double string array.** User can declare any type of variable in a single line. User can declare multiple variables with different data types in a single line.

Declaration style: Starts with keyword ‘Declaring’, followed by the variable name, **(definition given in token section)**, then the type of the variable (**int, double or string**) inside the ‘< >’ braces. There will be a ‘.’(full stop) after each declaration line. Any number of variables can be declared in a single line, the types can be different, they have to be separated by a ‘ , ’.

Sample 1: Declaring #Am2 < int >.

Sample 2: Declaring $AM2 < double >, $Bm3 < string >, #i89A2d < int >.

Values can also be assigned to variables while declaring. To do this, ‘==’ is to used after the variable name is declared, and the assigned value will be given after this. The assigned value can be a value, or a variable. Int type data can be assigned to float type data. Else data types must be same.

Sample 3: Declaring $a1< int > == 1, $a2< double > == 1.534, $a3< string > == 'str’

Sample 4: Declaring #Am2 < int > == $a1, #bC4 < double > == $a1+#Am2+1.5.

Sample 5: Declaring #sst < string > == ‘hello’+ ‘world’.

Variables that are already declared can not be redeclared, error will be shown is such case in the form **'variable x already declared'**.

Array declaration: Array declaration also starts with the keyword 'Declaring' followed by the keyword 'array' followed by the type of the array in the '< >' braces followed by the array name and finally the size of the array in the '[ ]' barces.

Sample 1: Declaring array < int > $Az3[10], array < double > $tf2[10].

The type of the array must be int or double as string itself is an array type variable and size can be maximum 1000. Unlike other variables, arrays can not be assigned value during declaration. Multiple arrays having multiple types can be declared in a single line and there must be a '.' At the ending of each line.

Variables that are already declared can not be redeclared, error will be shown is such case in the form **'variable x already declared'**.

Class declaration: Starts with the keyword 'class' followed by the type of the class in the '< >' braces, the type of the class must be defined in the class definition section. After that comes the class name which also follows the variable name structure described in the token section. Then there must be a **constructor** in the '( )' braces containing the integer and double values. There must be '.' At the end of the line. If the class type is not defined then 'no class named classtype' will be shown and declaration will not be successful.

Sample 1: class< #class1 > #aaabb (1,1.5).

Also same goes for class, variables that are already declared can not be redeclared, error will be shown is such case in the form **'variable x already exists'**.

Assignment operations:

Assignment operations are used for assigning values to variables. Multiple values of different types can be assigned a value in a single line. If multiple variables are assigned in a single line they must be separated with a ','. The line must end with a '.'.

Type conventions: In case of double type variables, both double and integer values can be assigned to it. In case of other type of variables, type must be same.

Assignment style: Each assignment will start with the variable name, followed by the '==' sign and followed by an expression. This expression will contain the value that is to be assigned to the variable. Type convention must be followed. If the user fails to follow type convention error message **'Invalid Assignment Operation'** will be shown. Also the variable must be declared before, if user tries to assign value to an undeclared variable error message **'Variable not Declared yet! Assignment is not possible'** will be shown.

Sample 1: $Am2 == 1, $AM2 == 2.5, $ms3 == 'HELLO' .

Sample 2: #am == 1.5+2+$Am2, $Am3 == 1+$Am2

Sample 3: $ms3 == 'ab'+ 'cd', #ms4 == 'fg'+ '$ms3'.

Array assignment: Assigning values to an array variable will also start with the variable name, followed by the index number inside which the value will be assigned. The index will be given inside '[ ]' braces. The index must be an integer type expression. This will be followed by '==' and followed by the expression. Type conventions are same as described above. Type convention failure will produce error message. Multiple values can be assigned in a single line, in such cases user must separate them with a ','. In case of undeclared variable, error message **'Variable 'x' is not an array'** will be shown.

Sample 1: $Az3[1]==3, $tf2[0] == 1.5.

Sample 2: $Az3[1]==3+#ab, $tf2[0] == 1.5+2.

Sample 3: $Az3[#index]==3. (Here #index is a variable of integer type containing the index value in which value is to be assigned)

Class assignment: Explicit class assignment option is not available, user must assign values to a class variable using **constructor** described above.

Expression part:

Expression is the most vital part of this language. It is used in every feature of the language to control the flow of the program in a desired way, such as loops to determine how many times statements will be executed, conditional and switch case statements to determine if a condition is true, assigning values in variables, printing values, determining the type of a constant value, performing mathematical operations and so on.

Expression is defined as a non-terminal under the structure data 'data_type'(discussed in token section). Expression can be deduced into following things:

1.Constant values: These are integer, floating and string type value given directly by the user in the input file. Type for integer is 1, double is 2 and string is 3.

Sample 1: 1 232 232(int), 1.232 42.4234(float), 'Hello' (string)

The size of an integer can be at most **10^9,** double **2^64,** string **10^4.**

2. Variable names: Expression can also be variable names which are already declared. In such case it will contain the value of the variable and the type of the variable. In case of array variable, the expression will contain the value in the described index

Sample 1: #a1 #a2[10] (here these names contains the type of the variable and the assigned value)

3. Mathematical operations: There are a few available operations:

Addition:

Used for adding two values.

type convention:

1. **int +int**
2. **int + double**
3. **double +int**
4. **double + double**
5. **string +string**

Sample: 1+2 2.5345+1323 'hello'+ 'world' #am + 12 + $ij (here #am, $ij are variables)

Subtraction:

Used for subtracting two values.

type convention:

1. **int -int**
2. **int - double**
3. **double -int**
4. **double - double**

Sample: 4-2 2.5345-1.323 #am - 12 - $ij (here #am, $ij are variables)

Multiplication:

Used for multiplicating two values.

type convention:

1. **int *int**
2. **int * double**
3. **double * int**
4. **double * double**

Sample: 4*2 2.5345*1.323 #am * $ij (here #am, $ij are variables)

Division:

Used for division operation.

type convention:

1. **int /int**
2. **int / double**
3. **double / int**
4. **double / double**

Sample: 4/2 6.25*2.5 #am / $ij (here #am, $ij are variables)

Power operation:

Used for power operation.

type convention:

1. **int ^^ int**
2. **double ^^ int**

Sample: 4^^2 (will return 16 and type will be int) 6.25^^2 #am ^^ $ij (here #am, $ij are variables and type of $ij must be int)

Mod operation:

Used for modulas operation.

type convention:

1. **int %% int**

Sample: 4%%2 (will return 16 and type will be int) #am %% $ij #am%%3 (here #am, $ij are variables and type of both must be int)

Precedence: POWER > MOD > DIVISION,MULTIPLICATION > PLUS, MINUS

Associativity: All are left associative

Logarithm:

Used for log 10 based operations. Takes input **int or double** type data and **returns double** type data.

Sample 1: log(12) log(30.5) log(#ab) (here #ab is a variable which type must be int or double)

Trigonometry:

Used for mathematical sine, cosine or tangent function. Takes **int or double** data and returns **double** type data.

Sample 1: sin(90) cos(150) tan(45) sin(#ab) cos(#bc) (Here #ab and #bc are variables of type int or double)

GCD and LCM:

Used to find GCD and LCM of two **int** data types. It returns an **int** data.

Sample 1: gcd(3,6) lcm(12,6) gcd(#ab, 4) lcm(#a, #c) (Here #ab #a #c are int variables)

4. Relational operations:

There are 5 types of relational operators used in this language that user can use:

1. **'>>' meaning greater than**
2. **'<<' meaning less than**
3. **'>=' meaning greater or equals than**
4. **'<=' meaning less or equals than**
5. **'=' meaning equals to**

Type convention:

1. **int OP int**
2. **double OP double**
3. **string OP string**

This operation returns 1 or 'True' if the relation holds and 0 or 'False' if the relation is not correct.

Sample 1: 1<<2 2.5 >>1.0 3<<2

Sample 2: 1=2 2.5 >= 3.5 3 <= 18

Sample 3: 'a' >> 'A' 'C' << 'd' 'a' >= 'z' 'a' <= 'f'

Sample 4: 'string' << 'STRING' 'tHis' >> 'This'

Sample 5: 'this'= 'this' 'this' >= 'That' 'This' <= 'that'

Relational operators have **less precedence than mathematical operators** and they are all **left associative.**

Input and Output:

User can give input to variables from terminal and also print values of different expressions described above in the terminal.

Input syntax: Starts with keyword ‘take’ followed by keyword ‘array’ if it is array, followed by variable name, followed by ‘[ ]’ and desired index expression if it is an array, followed by ‘==’ followed by keyword ‘input’ and ‘( )’ braces. The variable must be declared and assigned previously.

Sample 1: take #b2 == input().

take #b1 == input().

Sample 2: take array #d1[0] == input().

take array #d2[3] == input().

Output syntax: The user can print any expression value described above in the terminal.

Syntax: print (‘any expression’).

Samples:

print(array $tf2[2]).

print(1+3).

print($a3+$ms3).

print(1.5+2).

print($CD3+2).

print($CD3+1.5).

print(5-2).

print(#x).

print(3%%2).

print(gcd(3,6)).

print('ab' >= 'ab').

print('ab' <= 'Ab').

print(2.5 << 10.9).

print('a' = 'a').

print('a' = 'A').

print(2.5=3.5).

Conditional Statements:

Conditional statements are used to check a condition, if it is true then the associated statement is executed. In this language, user can use **if, else and if-else if** block for conditional purposes.

If , else if and else each statement starts with keywords ‘if’, ‘else if’, ‘else’ keywords respectively. It is then followed by ‘( )’ braces and inside them boolean expressions can be used to check the conditions. Here bool expressions have type ‘8’. It is followed ‘[ ]’ braces inside which statements are written. There must be ‘.’ At the end of the statement.

In if else-if block, the execution will enter the first block that satisfies the condition and ignore other blocks.

Sample 1: if:

```
 if(#in%%2=1)[
     print('OddNumber').
  ]
```

Execution will enter this block if #in is an odd number

Sample 2: if else:

```
if(#in%%2=1)[
     print('OddNumber').
  ]
  else [
          print('EvenNumber').
 ]
```

Execution will enter the if block if #in is odd and will enter the else block otherwise.

Sample 3: if else-if block:

```
if(#mark>=90)[
    print('A').
 ]
 else if(#mark>=80)[
        print('B').
]
 else if(#mark>=70)[
        print('C').
]
 else if(#mark>=60)[
        print('D').
]
 else if(#mark>=50)[
        print('E').
]
 else [
        print('Failed').
]
```

User will give input the value for #mark variable. If #mark >= 90 then the execution of the program will enter the if(#mark >= 90) block and execute it's statement. It will ignore the latter else if blocks though it fulfills each of the conditions. Similarly execution will only enter the #mark >= 70 when input 75 is given and will ignore other blocks. Execution will enter else block if #mark <= 59.

Comment:

User can give comments to their input files. Single line and multi line comments can be used. The syntax is given below:

Single line comment: !! single line comment

Multiple line comment:

! mul

ti

line

!

Loops:

User can use three types of loops: **for, while and do while** loop. Details are described below:

For loop:

The syntax of the for loop starts with the keyword ‘for’. It is followed by ‘( )’ braces. Inside the braces, three conditions are given:

1. The value from which the loop will start iterating
2. The value till which the loop will iterate
3. Increment amount

These values are to be separated by ‘ , ’. Then user can give statements inside the ‘[ ]’ braces. There must be ‘ . ’ at the end of statement. Using for loop user can do the following things:

1.Taking array input: User can take input for an array in different index from the terminal using for loop. The syntax is given below:

Sample 1:

```
#in == 0.
 for(#in, 5, 1)[
  take array #d1[#in] == input().
 ]
```

Here the loop will iterate 5 times taking input from the terminal in the array #d1 in indexes 0 to 4.

2.Printing array values: User can also use the for loop for printing array values. The syntax is given below:

Sample 2:

```
#in == 0.
 for(#in, 5, 1)[
        print(#d1[#in]).
]
```

Here the loop will iterate 5 times and print the contents of the array from indexes 0 to 5.

3. Printing any expression: Besides user can print any expression multiple times using for loop.

Sample 3:

```
#in == 0.
 for(#in, 5, 1)[
        print('Hello').
]
```

Here the loop will iterate 5 time and 'Hello' will be printed 5 times.

While loop:

The syntax of while loop starts with the keyword 'while' followed by '( )' braces. Inside the braces there will be a decremental statement, which will decide how many times the loop will iterate. The loop will iterate until the expression value becomes 0. After that user can give statements inside '[ ]' braces.

Sample:

```
#in == 10.
 Declaring #e1 < int >, #e2 < int >.
 while(#in---)[
        #e1 == input().
        #e2 == input().
```

print(#e1*#e2).

]

Here the loop will execute 10 times. Each time user can give input values for variables #e1 and #e2 and the output will be their multiplication.

Do while loop:

This loop is similar to while loop, the difference is in while loop execution will not go inside the loop if the condition is false. But in case of do while loop execution will go inside the loop at least one time even if the condition given in the while section is false.

The syntax starts with the keyword ‘do’ followed by statement inside ‘[ ]’ braces, followed by the keyword ‘do’ followed by the condition inside ‘( )’ braces.

Sample 1:

#in == 1.

do[

#e1 == input().

print(#e1).

]while(#in---).

Here the loop will iterate one time.

#in == 2.

Sample 2:

do[

#e1 == input().

print(#e1).

]while(#in---).

Here the loop will iterate 3 times.

Switch-Case statements:

This is kind of a conditional statement. The syntax is as follows:

It starts with ‘switch’ keyword, followed by an expression, followed by ‘:’ . After that there is a statement. Switch is followed by multiple ‘case’, each having own conditional expressions and statements. In the end there is a ‘default’ statement which does not have any conditional expression but has a statement.

The condition check starts from top and goes to bottom. Execution will enter the block the first condition which is true and execute the statement associated with it. It will ignore the conditions bottom of the first true conditions even if they are true. If all the conditions are false then program will enter the default block and execute the statement.

Sample 1:

```
take #in == input().


switch (#in%%2=1) : print('OddNumber').
default : print('EvenNumber').
```

Here if the #in variable is odd then the execution will enter the switch block and ignore the default block. Else it will enter the default block.

Sample 2:

```
take #mark == input().


switch (#mark>=90) : print('A').
case (#mark>=80) : print('B').
case (#mark>=70) : print('C').
case (#mark>=60) : print('D').
default : print('Failed').
```

User will give input the value for #mark variable. If #mark >= 90 then the execution of the program will enter the switch block and execute it’s statement. It will ignore the latter blocks though it fulfills each of the conditions. Execution will enter default block if #mark <= 59.