VGM

Beeram Sandya - CS21BTECH11006

Challa Akshay Santoshi - CS21BTECH11012

Hema Sri Cheekatla - CS21BTECH11013

Potta Vennela - CS21BTECH11046

October 11, 2023

Chapter 1

Introduction

Motivation for designing VGM

Many of us, programmers, start our coding journey with one of the very good and basic language C. At least, we started our coding with C. C makes us learn good code writing. We get to write the code from basics. Often we felt, why we were writing the things again and again. Especially, when it comes to cases like matrix operations, where we have to write nested loops every time. At some point of coding journey in C, most of us hope for the existence of inbuilt functions for performing those operations. So that we can just write all the necessary code in a single line.

Also, there are times, where we would like to add to or delete an element from an array. In some cases, we don't know the size of the array beforehand. So to make sure our data fits into array, we may declare an array with larger size, which leads to wastage of the memory allocated. This also might happen that the size we declared might be smaller than required. In either way, we don't want to declare the size beforehand. But C doesn't allow us to do it.

When it comes to implementation of the graphs, the traversals DFS and BFS are widely used. So our lives will be easier if we have them inbuilt.

So the main goal of this language is to reduce some burden if not all.

Chapter 2

Lexical Conventions

```
1 #include <stdio.h>
2 int main(){
3     int a;
4     printf("Hello World!\n");
5 }
```

2.1 Identifiers

The identifiers in this language are same like in C language.
1. The first letter of an identifier can be alphabet or underscore.
2. Alphabet can be uppercase or lowercase.
3. Identifier name cannot start with a digit.

2.2 Reserved Keywords

These are some reserved words which cannot be used as identifiers.

int	char	float	string	void	bool
return	true	false	if	else	continue
break	matrix	graph	vect	switch	case
class	struct.	and	or	gte	gt
eq	neq	lte	lt	add	sub
mul	div	not	exp	incr	decr
declr	expr	func	null	loop	

Table 2.1: reserved keywords

2.3 Comments

Comments can be single line or multi line. But both of these should include in '!!' symbol.
Example :

```
1  !! This is a single line comment !!
```

```
1 !! This is a
2  multi line comment!!
```

2.4 whitespaces

All white spaces including space, tab, new line will be ignored except for places where it is used to separate the tokens

2.5 Punctuation

These include ';', ':', ',', "'", "''", "." .
Statements should end with semicolon ';'.

```
func int main()
{
    int a = 4; !! for ending statements !!
    string s = "hi"; !! for string constants !!
    char c = 'c'; !! for character constants !!
    case a: !! for labels and cases !!
    call obj.function(); !! for calling object functions !!

    return a;
}
```

Chapter 3

Data Types

3.1   Primary datatypes

1. **int**: This int datatype is similar to that of C, i.e. 64-bit signed integer.

```
1        declr int abc;
2        expr abc = 12;
3
```

2. **float**: It is used to store floating-point values. It is a 64-bit signed floating-point number similar to C.

```
1        declr float a, b;
2        expr b = 12.32;
3        expr a = 9;
4
```

3. **char**: It is used to store only a single character.

```
1        declr char z;
2        expr z = 'a';
3
```

4. **string**: This datatype is used to store a sequence of characters.

```
1        declr string str;
2        expr str = "Compilers Course!!";
3
```

5. **bool**: It represents the truth conditions and can take two possible values, wither true or false.

```
1        declr bool val, res;
2        expr val = true;
3        expr res = false;
4
```

6. **struct**: This datatype follows the syntax: `struct identifier{ declaration statements };` Struct datatype can only contain declaration statements.

```
1        struct node
2        {
3            declr string name;
4            declr char grade;
5            declr int marks;
6            declr float cgpa;
7            declr bool feeStatus;
8        };
9
```

7. **class**: This datatype is similar to C++. It can contain declaration statements as well as functions. This datatype follows the syntax: `class identifier { declr stmts; functions,....};` However expression statements cannot be present inside a class.

```
 1      class vehicle
 2      {
 3          declr string type;
 4          declr int regNo;
 5          declr bool pollutionCheck;
 6
 7          func int getRegNo()
 8          {
 9              return regNo;
10          }
11          func void updatePollStatus ( bool val )
12          {
13              pollutionCheck = val;
14              return null;
15          }
16      }
17
```

8. **void**

3.2   Introduced datatypes

1. **array**:

   To declare an array the following syntax is used: `declr <datatype> <identifier>[num];`
   The expression statement for this array should follow the condition that the number of values inside the array equals the number which is the size with which it was initialized. It can be represented as follows: `exp <array identifier> = { value1, value2, value3,...  };`
   All of these values should have the same datatype and should match with it's declaration datatype.

```
 1      declr int arr[10];
 2      expr arr = {1,2,3,4,5,6,7,8,9};
 3
 4      declr string str[3];
 5      expr str = {"Hema", "Sandya", "Vennela"};
 6
```

2. **graph**:

   Two types of graphs can be implemented. It can be a weighted graph or it can be an unweighted graph. By default the graphs are taken as directed graphs.
   The following syntax is used for declaration: `declr graph <identifier>[num of nodes];`
   Here by default the node labels will be from 1 to n, where n is the number of nodes given in the declaration statement. Now, the edges of the nodes can be represented as follows: `expr <identifier> = { stmts };`
   The 'stmts' present here depend on the type of graph you want to construct. For an unweighted graph, it can be represented as `node_label :  node1_label, node2_label,....;`
   This states that there is a directed edge from node_label to node1_label, from node_label to node2_label and so on. If you want to represent an undirected graph, then it can be represented as 2 directed edges between the same two nodes but in reverse direction. For example, if there is an undirected edge between nodes 1 and 2, then you represent it by having a directed edge from 1 to 2, as well as a directed edge from 2 to 1.

   Now, if you want to represent a weighted graph, 'stmts' takes the form: `node_label :  (node1_label, weight1), (node2_label, weight2),...;`

Here also, by default we take them as directed edges. Inside the pair, the first value is the node label and the second value is the weight value associated with the edge. For an undirected graph, the same rule of taking the edge as a combination of two directed edges applies.

Here, there is a directed edge from node_label to node1_label with a weight value of weight1 and so on.

```
1      declr graph G[4];
2      expr G =
3      {
4          1 : 2,3;
5          2 : 1,4;
6          3 : 1;
7          4 : 2;
8      };
9
10     declr graph G[3];
11     expr G =
12     {
13         1 : (2,8),(3,12);
14         2 : (3,5);
15     };
16
```

3. **matrix**:

   The declaration statement of a matrix takes the form `declr matrix <identifier> [no of rows][no of columns];`
   The matrix itself can be represented as follows: `expr <identifier> = [ [val_11, val_12,...,val_1n]; [val_21,val_22,val_23,...,val_2n]; ..... [val_m1,val_m2,...,val_mn] ];`
   Here 'm' represents the number of rows and 'n' represents the number of columns.

```
1      declr matrix mat [2][3];
2      expr mat =
3      [
4          [1,2,3];
5          [4,5,6];
6      ];
7
```

4. **vector**:

   The size of a vector is not mentioned in our declaration. The declaration of a vector is represented as follows: `declr vect<datatype> <identifier>;`
   This is similar to the C++ vector datatype.
   Its values maintain the constraint that their datatype matches with the datatype of the vector during its declaration. Their values are represented as follows: `expr <identifier> = {val1, val2,...};`

```
1      declr vect<int> myvec;
2      expr myvec = {1,2,3};
3
```

6

Chapter 4

Defining function

Functions start with a key word `func`. The declaration of functions follows the following syntax:
`func <data_type> ( <arguments> )  <selected_statements>`
The arguments follows the following syntax:
for single argument: `<data_type> variable`
for multiple arguments: `<data_type> variable1, <data_type> variable2, <data_type> variable3`
functions can have only the following statements

1. call statements

2. variable declaration statements

3. expression statements

4. conditional statements

5. loop statements

6. jump statements

7. return statements

Function can have no arguments too. Here is its corresponding syntax
`func <data_type> ( )  <selected_statements>`

```
1 func int foo()
2 {
3     !! write statements here (must contain atleast one return)!!
4 }
5
6 func string boo(int a, int b)
7 {
8     !! write the statements here (must contain atleast one return) !!
9 }
```

NOTE: Functions must contains atleast one return statement returning the same data type as that of function

Chapter 5

Statements and their syntax

Operations

arithmetic operations

Here are the binary operators: **add, sub, mul, div**
They have the following syntax:
`<b_operator> (<resultant>, <resultant>)`
In this the `<resultant>` represents the following

- constants (constants of data types int, float, bool , etc)

- arithmetic operations

- logical operations

- call statements

Here are unary operators: **incr, decr**
They have the following syntax:
`<u_operator>(<resultant>)`
`<resultant>` is same as mentioned above Examples:

```
1    add(mul(a, b), incr(c))
2    add(call fun(), 3)
3    incr(incr(a))
4    incr(add(i, j))
5    decr(call fun())
```

logical operations

Here are the logical operators: **and, or, not**
s Statements exists only within the scopes. Here are the types of statements:
`(<resultant> <logical_operator> <resultant>)` Examples:

```
1    (a and b)
2    (a or (b and c))
3    (call fun() or incr(c))
```

Note: The logical operations should lie within braces

5.1   Declaration Statements

Declaration statements are used to declare. These statements begin with the keyword `declr`

5.1.1 pre-defined datatype declarations

We can declare single or multiples variables in a single declare statement Declaration of primary datatypes.

The declaration of pre-defined datatypes is as follows:

`declr <data_type> <variables> ;`

```
1  func int main()
2  {
3      declr int a;
4      declr float f1, f2;
5      declr int arr[a];
6      declr string str1, str2;
7      declr char c;
8      declr bool ans;
9      declr vect<int> myVec; !! declaration of vector !!
10     declr graph G[10]; !! declaration of graph !!
11     return 0;
12 }
```

5.1.2 struct declarations

The declaration of user defined structs is as follows:

`declr struct <identifier> <identifier> ;`

```
1  func int main()
2  {
3      !! assume that the struct name node is defined !!
4      declr struct node A;
5      declr struct node b, c;
6      return 0;
7  }
```

5.1.3 class declarations

The declaration of user defined class is as follows:

`declr class <identifier> <identifier> ;`

```
1  func int main()
2  {
3      !!  assume that the class name node is defined !!
4      declr class node A;
5      declr class node vertex1, vertex2;
6      return 0;
7  }
```

5.2   Expression statements

Expression statements start with the key word `expr`. It follows the following syntax

1. `expr variable = <constant>`:
   Here `<constant>` is constant of any data types such as int, float, string, char, vect, matrix, bool

   ```
   1      func int main()
   2      {
   3          declr int a;
   4          expr a = 10;
   5
   6          declr string str;
   7          expr str = "Hello, World!"
   8
   9          declr vect<int> myVec;
   10         expr myVec = {};
   11
   12         return 1;
   ```

```
13            }
14
```

2. `expr variable = <arithematic operations>`
   Details about `<arithmetic operation>` are provided earlier

```
1        func int main()
2        {
3            declr int ans, a, c, d;
4            expr ans = add(a, mul(c, d));
5            return ans;
6        }
7
```

3. `expr variable = <logical operations>`:
   Details about `<logical operations>` are provided earlier

```
1        func int main()
2        {
3            declr bool ans, c;
4            declr int a, b;
5            expr ans = ((a and True) or (b and c));
6            return ans;
7        }
8
```

4. `expr variable = <Call statements>`:
   Here call statements are the ones that are described in the later section.

```
1        func int main()
2        {
3            declr int ans;
4            expr ans = call foo();
5            return ans;
6        }
7
```

5.3 Conditional statements

5.3.1 if-conditionals

Here is the syntax for if-conditionals

1. `if (<resultant>) { <statements>}`:
   The `<resultant>` includes following kinds

   - constants (constants of data types int, float, bool , etc)
   - arithematic operations
   - logical operations
   - call statements

```
1        func int main()
2        {
3            !! assumes that the identifier used it these statments are already declared !!
4            if ( (a and b) or (add(1, 2)))
5            {
6                !! write statements here !!
7            }
8
9            return a;
10        }
11
```

2. if (<resultant> { <statements>}) else { <statements>}:

```
1      func int main()
2      {
3          !! assumes that the identifier used it these statments are already declared !!
4          if ( (a and b) and (call foo(1, 2)))
5          {
6              !! write statements here !!
7              return a;
8          }
9          else
10         {
11             !! write statements here !!
12             return b;
13         }
14     }
15
```

5.3.2 switch-conditionals

Here is the syntax for the switch conditionals
switch (<resultant>) {
case <constant> :
<statements>
case <constant> :
<statement>
default :
<statements>
}

Note: There can be any number of cases. The `default` case is optional

```
1   func int main()
2   {
3       !! assume that the identifiers that are used are already declare before !!
4       switch(id1)
5       {
6           case 1:
7               !! write statements here !!
8           default:
9               !! write statements here !!
10      }
11
12      return id1;
13  }
```

5.4 Loops

The loop start with a key word 'loop'

5.4.1 for loops

The syntax for 'for loops'
loop for(<expression statement> ; <logical operations>; <increment operations>) { <statements>}

```
1   func int main()
2   {
3       !! write statements here (optional)!!
4       loop for (i=0 ; (i<10); incr(i))
5       {
6           !! write statements here !!
7       }
```

```
8        return i;
9    }
```

5.4.2   while loops

The syntax for while loops
`loop while (<resultant>) {<statements>}`
The `<resultant>` above includes the following

- constants (constants of data types int, float, bool , etc)

- arithmetic operations

- logical operations

- call statements

```
1    func int main()
2    {
3        !! write statements here (optional) !!
4        loop while((i<10))
5        {
6            !! write statements here !!
7        }
8
9        loop while(decr(i))
10       {
11           !! write statemens here !!
12       }
13
14       return i;
15   }
```

5.5   Jump statements

To write the jump statements, the label to which the program should jump to should be declared first.
Here is its syntax assuming that the corresponding label is declared before
`goto <identifier>`

```
1    func int main()
2    {
3        declr int a;
4        expr a = 0;
5        label1 : incr(a);
6
7        !! write statements here !!
8
9        if(a){
10           goto label1;
11       }
12
13       !! write statemnts here !!
14
15       return a;
16   }
```

The labeled statements looks like the following
`<identifier> :  <statement>`
or `<identifier>:`
It is not necessary to have statements right to the label.

5.6 Call statements

Call statements are used to call the functions. They can be used as statements or as RHS of expressions or as conditionals in conditional statements too. They must start with the keyword `call`. Here is their syntax:
`call <identifier>(<call_arguments>)`
The `<call_arguments>` follow the following syntax:
`<resultant>`
or
`<resultant>, <resultant>, <resultant>`
The call arguments are none or more `<resultant>` objects. The `<resultant>` means the following

- constants (constants of data types int, float, bool , etc)

- arithmetic operations

- logical operations

- call statements

```
1   func int main()
2   {
3       !! write statements here (optional) !!
4       call foo();
5       call fun(a, incr(n));
6       expr a = call fan(i);
7       if(call ben(s, str, 10))
8       {
9           !! write statements here !!
10      }
11      return 0;
12  }
```

And here is the syntax for calling functions that belong to some class
`call <identifier>.<identifier>(call_arguments)`

```
1   func int main()
2   {
3       !! assume that a class named node is defined and there is a
4           function name Print_node, add_nums exists in that node class !!
5
6       declr class node A;
7       call A.Print_node();
8       declr int ans;
9       expr ans = call A.add_nums(12, 13);
10      return ans;
11  }
```

5.7 return statements

The return statement are used in functions to return the result of the function. Here is the syntax for the return statements:
`return <resultant>`
As mentioned in earlier the `<resultant>` represents the following

- constants (constants of data types int, float, bool , etc)

- arithmetic operations

- logical operations

- call statements

The thing the return statement return should match the data type of the function, which is checked by syntax analyser. A function should contain atleast one return statement

```
1    func int main()
2    {
3        declr int a;
4        expr a = 10;
5
6        if(a)
7        {
8            return 0;
9        }
10       else{
11           return incr(a);
12       }
13       return a;
14   }
```

5.8   unary operation statements

The unary operation itself can act as statement. The syntax is already defined previously.
`<u_operator> (<resultant>)`

```
1  func int main()
2  {
3      declr int a;
4      expr a = 10;
5      incr(incr(a));
6      return incr(a);
7  }
```

They can also be given as function arguments, conditional arguments, RHS of expression statements also.

Chapter 6

Improvisation

6.1 Vectors

Vectors in this language works similar to the vectors in C++. There is no beforehand declaration of the size as in the case of arrays.
Vectors are mutable.
We have some inbuilt functions like append, remove, pop, length, sort for various uses.

6.1.1 append

The function 'append' adds the associated element at the end of the vector.

```
1
2    expr myvec = {1,2,3}; !! expression statement !!
3
4    myvec.append(4); !! append method of vector !!
5
6    !! myvec becomes {1,2,3,4} !!
```

6.1.2 remove

The function 'remove' takes in an argument of the index which has to removed. It must have the constraint that the index must be within the size of that vector.

```
1    !! myvec = {10,25,35} !!
2
3    myvec.remove(1);
4
5    !! myvec becomes {10,35} !!
```

6.1.3 length

The function 'length' returns the number of the elements in the vector.

```
1    !! myvec = {'c','h','e','z'} !!
2
3    expr a = myvec.length();
4
5    !! The value 4 gets assigned to variable 'a' !!
```

6.1.4 sort

The function 'sort' by default sorts the vector in increasing order.

```
1    !! myvec = {34,23,1,-123} !!
2
3    myvec.sort();
4
5    !! myvec becomes {-123,1,23,34} !!
```

6.1.5   clear

The function 'clear' removes all the elements in the vector making its size 0.

```
1    !! myvec = {1,2,3,4} !!
2
3    myvec.clear();
4
5    !! myvec becomes {} !!
```

6.1.6   at

This function takes an argument of index and returns the value which is present at that index in the vector.

```
1    !! myvec = {20,100,67,69} !!
2
3    expr a = myvec.at(3);
4
5    !! The variable 'a' gets assigned the value of 69. !!
```

6.2   Matrices

We can declare matrix directly using this 'matrix' datatype.
We have some inbuilt functions for regularly using matrix operations like add, sub, transpose, trace.

6.2.1   add_matx

'add_matx' is a function which takes two matrices as input and return their addition only if their dimensions match. Else gives an error.

```
1    declr matrix A[2][3], B[2][3], C[2][3];
2
3    expr C = add_matx(A,B); !! C gets assigned the matrix obtained by the addition of A and B !!
```

6.2.2   sub_matx

'sub_matx' is a function which takes two matrices as input and return their subtraction only if their dimensions match. Else gives an error.
By returning the subtraction here we mean, the matrix which is given as a second argument is subtracted from first.

```
1    declr matrix A[2][3], B[2][3], C[2][3];
2
3    expr C = sub_matx(A,B); !! C gets assigned the matrix obtained by the subtraction of B from A !!
```

6.2.3   mult_matx

'mult_matx' is matrix multiplication function. Here two matrices can qualify for multiplication only if the number of columns of first matrix is equal to the number of rows of the second matrix.
This returned matrix has rows equal to the number of rows of the first matrix and the columns equal to that of the second matrix.

```
1    declr matrix A[5][6], B[6][10], C[5][10];
2
3    expr C = mult_matx(A,B); !! C gets assigned the matrix obtained by the multiplication of A with B. !!
```

6.2.4 transpose

This takes a matrix as an input and return its transpose.

```
1    declr matrix A[1][2], B[2][1];
2
3    expr B = A.transpose(); !! B gets assigned the transpose of A !!
```

6.2.5 trace

This function checks whether the matrix is a square matrix. If it is, returns the trace else gives an error.

```
1    declr matrix A[2][2];
2
3    expr a = A.trace(); !! 'a' gets assigned the value of trace of A !!
```

6.3 Graphs

Graphs has inbuilt functions for BFS and DFS traversals.

6.3.1 dfs

This 'dfs' function implements the DFS traversal of a graph. It takes an argument which is the node at which the dfs search has to be implemented. This returns a vector.

```
1    declr vect<int> myvec;
2
3    declr graph G[12];
4
5    expr myvec = G.dfs(3);  !! myvec gets assigned the dfs order of nodes starting at 3 !!
```

6.3.2 bfs

This 'bfs' function implements the BFS traversal of a graph. It takes an argument of the node and returns the bfs search from there as a vector.

```
1    declr vect<int> myvec;
2
3    declr graph G[5];
4
5    expr myvec = G.bfs(2); !! myvec gets assigned the bfs order of nodes starting at 2 !!
```