VGM

Beeram Sandya - CS21BTECH11006

Challa Akshay Santoshi - CS21BTECH11012

Hema Sri Cheekatla - CS21BTECH11013

Potta Vennela - CS21BTECH11046

November 25, 2023

Chapter 1

Introduction

Motivation for designing VGM

VGM is suitable for applications involving graphs, matrices, vectors. It also contains the primary data types present in C in addition to string data type. This language is more efficient and intuitive for tasks in these domains (graph, matrices especially) compared to general purpose languages like C and C++. It has more concise and readable syntax for operations involving the above. VGM provides abstraction for working with graphs, matrices and vectors, so it can save the time and effort compared to manually implementing these functionalities, allowing them to focus more on their algorithms. It is more user-friendly and expressive for developers working on these tasks. If not all, there are some basic implementations which are handled by this language. For example graph bfs, dfs traversal, shortest path and its value between two nodes, matrix arithmetic operations, transpose, trace, matrix to graph (and vice-versa) conversion along with string and vector C++ STL features.

Chapter 2

Lexical Conventions

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	add_matx
sub_matx	mult_matx	print	call	for	goto
true	false				

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 :

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

```
!! This is a
 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.

```
        declr int abc;
        expr abc = 12;

```

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

```
        declr float a, b;
        expr b = 12.32;
        expr a = 9;

```

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

```
        declr char z;
        expr z = 'a';

```

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

```
        declr string str;
        expr str = "Compilers Course!!";

```

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

```
        declr bool val, res;
        expr val = true;
        expr res = false;

```

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

```
        struct node
        {
            declr string name;
            declr char grade;
            declr int marks;
            declr float cgpa;
            declr bool feeStatus;
        };

```

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.

```
        class vehicle
        {
            declr string type;
            declr int regNo;
            declr bool pollutionCheck;

            func int getRegNo()
            {
                return regNo;
            }
            func void updatePollStatus ( bool val )
            {
                pollutionCheck = val;
                return null;
            }
        }
```

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.

```
        declr int arr[10];
        expr arr = [1,2,3,4,5,6,7,8,9];

        declr string str[3];
        expr str = ["Hema", "Sandya", "Vennela"];
```

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.

```
        declr graph G[4];
        expr G =
        {
            1 : 2,3;
            2 : 1,4;
            3 : 1;
            4 : 2;
        };

        declr graph G[3];
        expr G =
        {
            1 : (2,8),(3,12);
            2 : (3,5);
        };

```

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.

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

```

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,...};`

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

```

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
8. unary statements
9. improvisation statements

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

```
func int foo()
{
    !! write statements here (must contain atleast one return)!!
}

func string boo(int a, int b)
{
    !! write the statements here (must contain atleast one return) !!
}
```

NOTE: Functions must contains atleast one return statement returning the same data type as that of function
NOTE: Function body can be comprised of statements, or function bodies
NOTE: Function CAN have empty scopes

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
- functions of improvisations

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

```
    add(mul(a, b), incr(c))
    add(call fun(), 3)
    incr(incr(a))
    incr(add(i, j))
    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:

```
    (a and b)
    (a or (b and c))
    (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> ;`

```
func int main()
{
    declr int a;
    declr float f1, f2;
    declr int arr[a];
    declr string str1, str2;
    declr char c;
    declr bool ans;
    declr vect<int> myVec; !! declaration of vector !!
    declr graph G[10]; !! declaration of graph !!
    return 0;
}
```

5.1.2 struct declarations

The declaration of user defined structs is as follows:
`declr struct <identifier> <identifier> ;`

```
func int main()
{
    !! assume that the struct name node is defined !!
    declr struct node A;
    declr struct node b, c;
    return 0;
}
```

5.1.3 class declarations

The declaration of user defined class is as follows:
`declr class <identifier> <identifier> ;`

```
func int main()
{
    !!  assume that the class name node is defined !!
    declr class node A;
    declr class node vertex1, vertex2;
    return 0;
}
```

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

```
        func int main()
        {
            declr int a;
            expr a = 10;

            declr string str;
            expr str = "Hello, World!"

            declr vect<int> myVec;
            expr myVec = {1, 2};

            return 1;
        }

```

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

```
        func int main()
        {
            declr int ans, a, c, d;
            expr ans = add(a, mul(c, d));
            return ans;
        }

```

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

```
        func int main()
        {
            declr bool ans, c;
            declr int a, b;
            expr ans = ((a and True) or (b and c));
            return ans;
        }

```

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

```
        func int main()
        {
            declr int ans;
            expr ans = call foo();
            return ans;
        }

```

5. `expr variable = <functions of improvisations>` Details of `functions of improvisations` are given in section 6. They can return integers, strings, graphs, matrices or vectors

```
        func int main()
        {
            declr vect<int> myVec;
            expr myVec = {1, 2, 3};
            expr myVec = myVec.append(4);
            expr myVec = myVec.append(5).remove(4);
            declr int length;
            expr length = myVec.length();
            return length
        }

```

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
- functions of improvisations

```
        func int main()
        {
            !! assumes that the identifier used it these statments are already declared !!
            if ( (a and b) or (add(1, 2)))
            {
                !! write statements here !!
            }

            return a;
        }

```

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

```
        func int main()
        {
            !! assumes that the identifier used it these statments are already declared !!
            if ( (a and b) and (call foo(1, 2)))
            {
                !! write statements here !!
                return a;
            }
            else
            {
                !! write statements here !!
                return b;
            }
        }

```

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

```
    func int main()
    {
        !! assume that the identifiers that are used are already declare before !!
        switch(id1)
        {
            case 1:
                !! write statements here !!
            default:
                !! write statements here !!
        }

        return id1;
    }
```

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>`}
`loop for(<exprssion statement> ; <logical operation> ; <arithmetic operation>)`{`<statements>`}

```
func int main()
{
    !! write statements here (optional)!!
    loop for (i=0 ; (i lt 10); incr(i))
    {
        !! write statements here !!
    }

    loop for(j = 0; (j lt i); expr i = i + 2)
    {
        !! write statements here !!
    }
    return i;
}
```

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
- functions of improvisations

```
func int main()
{
    !! write statements here (optional) !!
    loop while((i<10))
    {
        !! write statements here !!
    }

    loop while(decr(i))
    {
        !! write statemens here !!
    }

    return i;
}
```

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>`

```
func int main()
{
    declr int a;
    expr a = 0;
    label1 : incr(a);

    !! write statements here !!

    if(a){
        goto label1;
    }

    !! write statemnts here !!

    return a;
}
```

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
- functions of improvisations

```
func int main()
{
    !! write statements here (optional) !!
    call foo();
    call fun(a, incr(n));
    expr a = call fan(i);
    if(call ben(s, str, 10))
    {
        !! write statements here !!
    }
    return 0;
}
```

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

```
func int main()
{
    !! assume that a class named node is defined and there is a
        function name Print_node, add_nums exists in that node class !!

    declr class node A;
    call A.Print_node();
    declr int ans;
    expr ans = call A.add_nums(12, 13);
    return ans;
}
```

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
- functions of improvisations

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

```
    func int main()
    {
        declr int a;
        expr a = 10;

        if(a)
        {
            return 0;
        }
        else{
            return incr(a);
        }
        return a;
    }
```

5.8 unary operation statements

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

```
func int main()
{
    declr int a;
    expr a = 10;
    incr(incr(a));
    return incr(a);
}
```

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. It returns itself, which is modified. This can be used as standalone statement or as RHS of expression statement or it can be used in return statements.

```

expr myvec = {1,2,3}; !! expression statement !!

myvec.append(4); !! append method of vector !!

!! myvec becomes {1,2,3,4} !!

expr myvec = myvec.append(5).append(6);

!! first myvec becomes {1, 2, 3, 4, 5} and then it becomes {1, 2, 3, 4, 5, 6} !!

return myvec;
```

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.It returns itself, which is modified. This can be used as standalone statement or as RHS of expression statement or it can be used in return statements.

If the non-existed element is asked to remove, then no change takes place in the vector

```
!! myvec = {10,25,35} !!

myvec.remove(1);

!! myvec becomes {10,35} !!

expr myvec2 = myvec.remove(10);

!! now myvec2 as well as myvec are {35} !!

return myvec.remove(35);
```

6.1.3 length

The function 'length' returns the number of the elements in the vector.
syntax : `<vector_resultant> . length ( )`
vector_resultant can be identifier of vector type or anything that returns vector type

```
!! myvec = {'c','h','e','z'} !!

expr a = myvec.length();

!! The value 4 gets assigned to variable 'a' !!
```

6.1.4 sort

The function 'sort' by default sorts the vector in increasing order. syntax : `<vector_resultant> . sort ( )`
vector_resultant can be identifier of vector type or anything that returns vector type

```
!! myvec = {34,23,1,-123} !!

myvec.sort();

!! myvec becomes {-123,1,23,34} !!
```

6.1.5 clear

The function 'clear' removes all the elements in the vector making its size 0. syntax : `<vector_resultant> . clear ( )`
vector_resultant can be identifier of vector type or anything that returns vector type

```
!! myvec = {1,2,3,4} !!

myvec.clear();

!! 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.

syntax : `<vector_resultant> . at ( <integer_resultant> )`
vector_resultant can be identifier of vector type or anything that returns vector type and integer_resultant can be identifier of integer type or anything that returns integer type

```
!! myvec = {20,100,67,69} !!

expr a = myvec.at(3);

!! 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.
syntax: `<add_matx> (<matrix_resultant> , <matrix_resultant>)`
Here matrix_resultant can be identifier of type matrix or anthing that returns matrix type

```
    declr matrix A[2][3], B[2][3], C[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.
syntax: `<sub_matx> (<matrix_resultant> , <matrix_resultant>)`
Here matrix_resultant can be identifier of type matrix or anthing that returns matrix type

```
    declr matrix A[2][3], B[2][3], C[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.

syntax: `<mult_matx> (<matrix_resultant> , <matrix_resultant>)`
Here matrix_resultant can be identifier of type matrix or anthing that returns matrix type

```
    declr matrix A[5][6], B[6][10], C[5][10];

    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.

syntax: `<matrix_resultant> . transpose ()`
Here matrix_resultant can be identifier of type matrix or anthing that returns matrix type

```
    declr matrix A[1][2], B[2][1];

    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.
syntax: `<matrix_resultant> . trace ()`
Here matrix_resultant can be identifier of type matrix or anthing that returns matrix type

```
    declr matrix A[2][2];

    expr a = A.trace(); !! 'a' gets assigned the value of trace of A !!
```

6.2.6 matx_to_graph

This function converts a matrix into a graph. The matrix here represents an adjacency matrix with the restrictions that it has to be square matrix and should contain non-negative values. If there is a 1 at the position of row_1 and col_2, it means that there is a directed edge from vertex_1 to vertex_2.

```
declr matrix A[4][4];

expr A =
[
    [12.4,1,1,0];
    [1,5,0,1];
    [1,0,0,0];
    [0,1,90,0];
];
expr a = A.matx_to_graph(); !! 'a' gets the graphical type of matrix A !!
```

6.2.7 printMatrix

This function is used to print the matrix.

```
declr matrix M[2][1];
expr M =
[
    [3.14];
    [2.73];
];
M.printMatrix();
```

6.2.8 addVal

This function is used to update or set the value at a position in the matrix by giving the row number, column number and the value to be added.

```
declr matrix A[3][3];
M.addVal(1,2,7.14);
```

6.2.9 getVal

This function is used to get the value which is present at the i, jth position.

```
declr matrix A[2][2];
M.getVal(1,1);
```

6.2.10 getRows

This function is used to get the number of rows of the matrix;

```
declr matrix M[3][3];
declr int a;
expr a = M.getRows();
```

6.2.11 getCols

This function is used to get the number of columns of the matrix;

```
declr matrix M[3][3];
declr int a;
expr a = M.getCols();
```

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.
syntax : `<identifier> dfs ( <integer_resultant> )`
Here integer_resultant is any thing that returns item of integer type

```
declr vect<int> myvec;

declr graph G[12];

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.
syntax : `<identifier> bfs ( <integer_resultant> )`
Here integer_resultant is any thing that returns item of integer type

```
declr vect<int> myvec;

declr graph G[5];

expr myvec = G.bfs(2); !! myvec gets assigned the bfs order of nodes starting at 2 !!
```

6.3.3 graph_to_matx

This function converts a graph into a matrix. If there is a edge from vertex_1 to vertex_2, then in the resulting matrix, we would have a 1 at position (row_1, column_2). Weighted Graph would have the weight in that position instead of 1.

```
declr graph G[4];
declr matrix M[4][4];

expr M = G.graph_to_matx(); !! M gets the adjacency matrix type of representation of graph G !!
```

6.3.4 shortest_path_value

This function returns the shortest path value between two nodes of a graph. If there is no path between the two edges, the function returns -1. For weighted graphs, this function has a limitation that the weights have to be non-negative.

```
declr graph G[3];
declr int x;

expr x = G.shortest_path_value(1,3); !! x contains the shortest distance value between 1 and 3 !!
```

6.3.5 shortest_path

This function returns a vector containing vertices covered while traversing in the shortest path between the asked nodes. If no path exists, then an empty vector is returned.

```
declr graph G[3];
declr vect<int> myvec;

expr myvec = G.shortest_path(1,3); !! myvec gets the shortest route between 1 and 3 !!
```

6.3.6 printGraph

This function is used to print the graph which the identifier is representing.

```
declr G[3];
expr G =
{
    1:2,3;
    2: 3;
};
G.printGraph();
```

6.3.7 adjNodes

This function returns a vector containing the adjacent nodes to the node sent as parameter to this function.

```
declr G[3];
expr G =
{
    1:2,3;
    2: 3;
};
declr vect<int> myVec;
expr myVec = G.adjNodes(1);
```

6.4 strings

In this langauage we have some special functions that can be called on strings. They perform some string operations

6.4.1 strlen

This function `strlen` returns the length of the string (it returns an integer). It is to be called from the variable/identifier that store the string value. Here is the corresponding syntax
`<identifier> . strlen ( )`

```
declr string str;
expr str = "Hello, world\n";
declr int length;
expr length = str.strlen();

!!Now the identifier length stores the value 13!!
```

6.4.2 strcmp

This function `strcmp` compares if the given strings are identical or not and return the boolean value 'true' or 'false' / returns integer value '1' or '0'.
'true'/'1' indicates the given two strings are identical, otherwise they are unidentical. Here is the corresponding syntax
`strcmp ( <identifier> , <identifier> )`
Note: The argument is need not be an identifier, it can be anything that returns string, like a function call that returns string, or any operation that returns string

```
declr string str1, str2, str3;
expr str1 = "Hello, world\n";
expr str2 = "hello ,world\n";
expr str3 = str1;
declr int a;
declr bool b;
expr a = strcmp(str1, str2); !! now a contains the value  0!!
expr b = strcmp(str1, str3); !! now b contains the value true !!

expr a = strcmp (str1, call foo()); !! This is valid if the function foo() returns a string type !!
```

6.4.3 strjoin

This function '`strjoin`' performs string concatenation and returns the concatenated string. It takes two arguments both are string type and returns a string type. Here is the corresponding syntax.
`strjoin ( <identifier> , <identifier> )`

```
declr string str1, str2, str3, str4;
expr str1 = "Hello, ";
expr str2 = "world\n";
expr str3 = strjoin(str1, str2); !! now str2 contains "Hello, world\n" !!
expr str4 = strjoin(strjoin(str1, str2), str2);
!! now str4 contains "Hello, world\nworld\n"
expr str4 = strjoin(strjoin(foo(), boo()), zoo());
!! This is also valid as long as those functions return string type !!
```

6.4.4 strcut

This function `strcut` cuts strings with respect to the indices. We give the starting and ending indices, as long as they are valid, this function cuts that part of string and returns it. Here is the corresponding syntax
`<identifier> .  strcut ( <number> , <number> )`
In the above '`<number>`' indicates any integer constant or identifier of integer type or any operation or function call that return an integer. If those indices are valid, this function would give corresponding substring, else it returns empty string?
Note: The resultant string is the substring of original string from starting index to ending-1 index. It does not include the ending index. And the indexing starts from 0.

```
declr string str1, str2, str3;
declr int a, b;
expr str1 = "abcdefgh";
expr str2 = str1.strcut(2, 6);
!! now str2 is "cdef"
expr str3 = str1.strcut(a, call foo()); !!as long as they are integer type and has valid integers, the
  funciton gives the string!!
```