GVL Reference Manual

Minhe Zhang (mz2864)
Yaxin Chen (yc3995)
Aster Wei (aw3389)
Jiawen Yan (jy3088)

Oct 19 2021

Contents

1 Introduction

This reference manual aims for a comprehensive description of Graph Visualization Language(GVL). GVL is an imperative, strong-typed, and indentation-insensitive programming language which has a C-style syntax and is specialized to visualize graph data structures and algorithms. For now, Object-Oriented Programming features are not introduced into GVL, but programmers can use structure to mimic.

2 Lexical conventions

There are six classes of tokens: identifiers, keywords, constants, string literals, operators, and separators. For adjacent identifiers, keywords, and constants, they must be separated with white space. Then white spaces and any characters within comments are ignored by the compiler.

2.1 Comments

For single line comment, we use two backslashes `//`. The characters after `//` in the same line will be ignored by compiler.
The characters `/*` introduce a multi-line comment and `*/` end it.
Comments do not nest and they do not occur within string or character literals.

2.2 Identifiers (Names)

An identifier is a sequence of letters and digits (the underscore `_` counts as a letter). Letters are case-sensitive and the first character of an identifier must be a letter. Identifiers can have any length. It can represent names of variables, functions, structures, and members of structure. A name has a scope. The same name in different scopes should refer to different data or functions.

2.3 Keywords

Following identifiers are reserved as keywords and may not be used otherwise:

```
int       break
float     continue
char      if
bool      else
string    for
struct    while
node      return
edge      true
graph     false
```

2.4 Constants

There are four kinds of constants in GVL, which are integer constant, floating-point number constant, character constant, and string constant.

2.4.1 Integer Constants

An integer constant consisting of a sequence of digits is taken to be decimal.

2.4.2 Floating-point number Constants

A floating constant consists of an integer part, a decimal point, and a fraction part. It should look like `123.456` or `1.23456e2`.

2.4.3 Character Constants

A character constant is a sequence of one of more characters which can only represent ASCII enclosed in single quotes. For example, a character constant might be `'a'`.

2.4.4 String Literals

String literals or string constant is a sequence of ASCII characters surrounded by double quotes such as `"Hello World"`.

2.5 Separators

There are four kinds of separators.

- Semicolon `;` means the end of a variable declaration, an expression as a statement. `;` can also separate expressions in parentheses of `for` statement.
- A pair of curly braces `{}` surround and must surround block of statements and expressions. It is used to determine the block of function implementation, block of statements after branch like `if` and `else`, and block of statements after looping like `for` or `while`.
- A pair of parentheses surround and must surround the conditional checking expression after `if` or `while`. It also surround the initialization, condition checking, and updating statements after `for` and the arguments when defining and calling functions. Parentheses also change the precedence explicitly when evaluating an expression.
- Comma `,` separates arguments of function declaration or call.
  For example, `int fun(int a, int b) {...}` or `int a = fun(b, c);`.

3 Types

There are two categories of type: basic type and derived type.

3.1 Basic Types

There are five basic types of which the keywords are:

```
int         float       bool        char        string
```

3.1.1 Integral Number

Integral numbers are represented by 32 bits and can contain integer from `-2147483648` to `2147483647`. They are declared or initialized using keyword `int`.

```
int a = 1;
```

3.1.2 Floating-point Number

Floating-point numbers are represented by 32 bits and range from `-1.2E-38` to `3.4E+38`. They are declared using keyword `float`.

```
float a = 1.0;
```

3.1.3 Boolean

Boolean type contains boolean value and are represented by 8 bits. It can be either `true` or `false` declared using keyword `bool`.

```
bool a = true;
```

3.1.4 Character

Character type is able to contain a single ASCII character. The value of character is represented by 8 bits. It is declared using keyword `char`.

```
char a = 'a';
```

3.1.5 String

String type is used to indicate that a variable can contain or a function can return a sequence of ASCII characters with maximum length of 65534. We use keyword `string` to declare a string type.

```
string a = "Hello World!";
```

3.2 Derived Types

There are six derived types.

```
array       structure       function
node        edge            graph
```

3.2.1 Array

The array type indicates that a variable should hold or a function should return an array of a certain type. We define it of a fixed length(at most 2147483647) and can access an item in an array by the index of that item.

```
int[5] a = {1, 2, 3, 4, 5};
int[2][2] a = {{1, 2}, {3, 4}};
```

3.2.2 Structure

Structure is a type of one or more variables. The types of variables it contains need not be the same. Structure is declared using keyword `struct`. For example,

```
struct account {
    string id;
    float balance;
    ...
}
```

Inside the `{}` of `struct` declaration, there can only be variable declarations. The names of members in the same structure must be different. Initialization and assignment are not allowed.

3.2.3 Node

Node is a built-in compound type. It must has coordinate`(x,y)`, radius`(radius)`, and color`(r,g,b)`. It is declared using keyword `node`. For example,

```
node n1 = node(...);
```

Inside the parenthesises are the parameters of `node` constructor. The signature of `node` constructor is

```
node(float x, float y, float radius, int r, int g, int b);
```

Also, node is able to carry extra payloads. Programmers can set attributes beyond the ones in the signature such as `set_node_attr(n1, "visited", true)`. Attributes and the corresponding values are key-value pairs. The type of key must be string.

3.2.4 Edge

Edge is a built-in compound type. It has mandatory attributes containing endpoints`(start, end)`, thickness`(t)`, and color`(r, g, b)`. It is declared using keyword `edge`:

```
edge e1 = edge(...);
```

Inside the parenthesises is the parameters of `edge` constructor which has a signature

```
edge(node n1, node n2, float t, int r, int g, int b);
```

Also, edge is able to carry extra payloads. Programmers can set attributes beyond the ones in the signature such as `set_edge_attr(e1, "weight", 1.0)`. Attributes and the corresponding values are key-value pairs. The type of key must be string.

3.2.5 Graph

Graph is a built-in compound type. It has data containing nodes and edges and is declared using keyword `graph`:

```
graph g1 = graph();
```

Inside the parenthesises is the parameters of `graph` constructor which has a signature

```
graph(node[] nodes, edge[] edges);
```

4 Expressions

Expressions are a combination of literals, identifiers, operators, and function calls to be evaluated. The precedence of expression operators follow the order of the subsections in this section. Left/Right associativity is specified in each subsection.

4.1 Primary Expressions

Primary expressions are identifier, constant or expressions in parentheses.

```
primary-expression:
    identifier
    constant
    (expression)
```

4.2 Postfix Expressions

The operators in postfix expressions group left to right.

```
postfix-expression:
    primary-expression
    postfix-expression [ expression ]
    postfix-expression ( argument-expression-list-option)
    postfix-expression . identifier

argument-expression-list-option:
    empty
    argument-expression-list

argument-expression-list:
    assignment-expression
    argument-expression-list, assignment-expression
```

4.2.1 Array Reference

A postfix expression followed by an expression in square brackets is a postfix expression denoting an array reference. The first expression must be a type `T[]` and the second one must be integral. The type of this reference expression is `T`.

4.2.2 Function Calls

A function call is a postfix expression. It is followed by parentheses containing comma-separated list of assignment expressions(arguments of the function). The arguments could be empty. The type of function call is the same as the return type of that function.

4.2.3 Structure References

A postfix expression followed by a dot followed by an identifier is a postfix expression. The first expression must be a structure and the identifier must be a name of one of the structure members.

4.3 Unary Operators

Expression with unary operators group right-to-left.

```
unary-expression:
    postfix-expression
    unary-operator expression
unary-operator: one of
    + - !
```

4.3.1 Unary Plus Operator

The operand of the unary `+` operator must have arithmetic type, and the result is the value of the operand.

4.3.2 Unary Minus Operator

The operand of the unary `-` operator must have arithmetic type, and the result is the negative of the operand.

4.3.3 Logical Negation Operator

The operand of the unary `!` operator must be `bool` type, and the result is the negation boolean value of the operand.

4.4 Multiplicative Expressions

The multiplicative expressions group left-to-right.

```
expression * expression
expression / expression
expression % expression
```

The `*` operator denotes multiplication of arithmetic type operands.
The `/` operator denotes quotient of the first expression over the second expression.
The `%` operator denotes remainder calculation when the first expression is divided by the second expression.

4.5 Additive Expressions

The additive expressions group left-to-right.

```
expression + expression
expression - expression
```

4.6 Relational Expressions

The relational expressions, `>` (greater than), `<` (less than), `>=` (greater than or equal to), and `<=` (less than or equal to), group left-to-right. The result of relational expression is `true` if the relation is true, and `false` otherwise.

```
expression > expression
expression < expression
expression >= expression
expression <= expression
```

4.7 Equality Operators

Equality Operators `==` (equal to) and `!=` (not equal to) group left-to-right.The result of equality expression is `true` if the relation is true, and `false` otherwise.

```
expression == expression
expression != expression
```

4.8 Logical Expressions

Logical expressions group left-to-right.

```
expression && expression
expression || expression
```

The `&&` operator returns 1 if both operands are non-zero, 0 otherwise. The `||` operator returns 1 if one of the operands are non-zero, 0 otherwise. Each of the two operands must have one of the basic types.

4.9 Assignment Expressions

Assignment expressions group right-to-left. The left operand is an lvalue and the right operand is an expression. Two operands must have the same type. The result of assignment expression is the value stored in the left operand after assignment.

```
lvalue = expression
```

The above form of assignment expressions assigns the value of the expression on the right hand side to the lvalue on the left hand side.

```
lvalue += expression
lvalue -= expression
lvalue *= expression
lvalue /= expression
lvalue %= expression
```

There are five other binary operators `+=, -=, *=, /=, %=` that can construct an assignment expression. They are in a form of `e1 op = e2`, which is equivalent to `e1 = e1 op e2`.

5 Declarations

Declarations in GVL have the following form:

```
declaration:
    type-specifier declarator
```

where the type-specifier are specified type in Section 3 Types and the declarator are defined identifiers in section 2.

5.1 Struct Declaration

Struct in GVL are declared as follow:

```
struct identifier { declaration-list }
```

where declaration-list are defined as:

```
declaration-list:
    declaration
    declaration ; declaration-list
```

5.2 Node Declaration

Node in GVL are declared as follow:

```
node identifier = node ( argument-list )
```

where argument-list is defined as:

```
argument-list:
    float x, float y, float radius, int r, int g, int b
```

Node can be extended with more data members by:

```
struct identifier : node { declaration-list }
```

5.3 Edge Declaration

Edge in GVL are declared as follow:

```
edge identifier = edge ( argument-list )
```

where argument-list is defined as:

```
argument-list:
    node n1, node n2, float t, int r, int g, int b
```

Edge can also be extended with more data members by:

```
struct identifier : edge { declaration-list }
```

5.4 Graph Declaration

Graph in GVL are declared as follow:

```
graph identifier = graph ( )
```

6 Statements

Statements are a sequence of GVL code that usually end with semicolon `;` or is delimited by braces `{}`.

6.1 Expression Statement

An expression statement is formed by an expression followed by a comma. For example, `y = x + 1;` and `y += 5;` are expression statements.

```
expression ;
```

6.2 Compound Statement

An compound statement is formed by list of statements delimited by braces.

```
{ statement-list }
```

where

```
statement-list:
    statement
    statement statement-list
```

6.3 Control Flow

For simplicity, usually, statement inside the control-flow statement can only be compound statement in GVL.

6.3.1 If/Else Statement

If/Else statement executes statements conditionally. It follows the following forms

```
if ( expression ) statement
if ( expression ) statement else statement
```

For the first form, statement is executed if expression is evaluated as true. For the second form, if expression is evaluated as true, the first statement is executed; otherwise, the second statement is executed. An exception here is that the second statement in the second form can not only be compound statement but also if/else statement.

6.3.2 While Statement

While statement is a looping statement; it repetitively executes statement as long as the expression is evaluated as true.

```
while ( expression ) statement
```

6.3.3 For Statement

For statement is also a looping statement.

```
for ( expression1 ; expression2 ; expression3 ) statement
```

All of the expressions are optional. Before looping, expression1 is evaluated. Statement followed by expression3 are repetitively executed as long as expression2 is evaluated as true.

6.3.4 Break Statement

Break statement jumps out of the loop of for/while.

```
break ;
```

6.3.5 Continue Statement

Continue statement skips execution of remaining statements in the current iteration of a for/while loop and continues to execute the next iteration if condition is satisfied.

```
continue ;
```

6.3.6 Return Statement

A function uses return statement to return to its caller, which has the following form:

```
return expression ;
```

An expression is evaluated and the result value is returned to the caller. GVL does not accept the form that no value is returned.

7 Graph Functions and Attributes

7.1 Node Functions

7.1.1 Node Constructor

Construct a node with built-in constructor:

```
node node(float x, float y);
node node(float x, float y, float radius, int r, int g, int b);
```

`x` and `y` are the x-coordinate and y-coordinate of the node when visualization.
`radius` is a float type number larger than 0, which decides the radius of the node.
`r, g, b` are int type numbers in range $[0, 255]$ used to decide the node's color.
The arguments in the constructors are all mandatory. If programmer uses the first constructor, the default `radius` will be set to 1 and `r`, `g`, and `g` will all be set to 255.

7.1.2 Node Attributes

Use `set_{attribute}` to change a node's built-in attribute in x, y, radius, r, g, b. These attributes cannot be removed by users.

```
set_node_attr(n1, "x", xval);
set_node_attr(n1, "y", yval);
set_node_attr(n1, "radius", rval);
set_node_attr(n1, "r", rval);
set_node_attr(n1, "g", gval);
set_node_attr(n1, "b", bval);
```

use `set_node_attr(node, "{user-defined attribute}", value)` to set a user-defined attribute.

```
set_node_attr(node, "key", val);
```

Use `node.{attribute}` to get a node's built-in attribute.

```
float x_value = n1.x;
float y_value = n1.y;
int radius_value = n1.radius;
int red = n1.r;
int green = n1.g;
int blue= n1.b;
```

7.2 Edge

7.2.1 Edge Constructor

Construct a node with `edge() start` and `end` are node type, indicate the start and the end in the directed edge.
`bold` is int type in range [1, 100], indicate the thickness of the edge.r, g, b are int type numbers in range [0,255] used to decide the edge's color.

```
edge e1 = edge(node start, node end);
edge e1 = edge(node start, node end, int bold, int r, int g , int b);
```

7.2.2 Edge Attributes

Use `set_{attribute}` to change a edge's built-in attribute in start, end, bold, r, g, b. These attributes cannot be removed by users.

```
set_edge_attr(n1, "start", startnode);
set_edge_attr(n1, "end", endnode);
set_edge_attr(n1, "bold", thickness);
set_edge_attr(n1, "r", rval);
set_edge_attr(n1, "g", gval);
set_edge_attr(n1, "b", bval);
```

use `set_edge_attr(edge, "{user-defined attribute}", value)` to set a user-defined attribute.

```
set_edge_attr(e1, "key", val);
```

Use `node.{attribute}` to get a node's built-in attribute.

```
node start_node = e1.start;
node end_node = e1.end;
int bold_val = e1.bold;
int red_val = e1.r;
int blue_val = e1.b;
int green_val = e1.g;
```

Use `edge.get("{user-defined attribute}")` to get a user-defined attribute.

7.3 Graph

Use "`graph()`" to build a new graph.

7.3.1 Graph Constructor

```
graph g1 = graph();
```

7.3.2 Graph Modification

A node or an edge can be added to a graph using "`++`" operator

```
add_node(g1, n1);
g1 ++ n1;
add_edge(g1, e1);
g1 ++ e1;
```

A node or an edge can be deleted from a graph using "`--`" operator

```
remove_node(g1, n1);
g1 -- n1;
remove_edge(g1, e1);
g1 -- e1;
```

7.3.3 Get nodes of a graph

Use `graph.nodes()` to get the node set in a graph.

```
node[] nodes = g1.nodes();
```

7.3.4 Get adjacent nodes from a graph

Use `get_adj_nodes(graph g, node n)` to get the adjacent nodes in a graph.

```
node[] nodes = get_adj_nodes(g1, n1);
```

7.3.5 Get edges of a graph

Use `graph.edges()` to get the edge set in a graph.

```
edge[] edges = g1.edges();
```

8 Examples

```
int N = 1000;

struct person {
    string id;
    int age;
    string major;
};

int add_int(int x, int y) {
    return x + y;
}

int bfs(graph g, node s, node t) {

    set visited = set();
    queue q = queue();
    q.add(s);
    int count = 0;

    while (q.size() > 0) {

        node curr = queue.pop();
        count += 1;
        visited.add(curr);
        set_node_attr(curr, "r", 127);
        set_node_attr(curr, "g", 127);
        set_node_attr(curr, "b", 127);
        if (curr == t) {
            return count;
        }
        node[] adj = get_adj_nodes(g, curr);
        for (int i = 0; i < adj.length; i += 1) {
            if (! visited.has(adj[i] && queue.has(adj[i]) {
                queue.add(adj[i]);
            }
        }
    }
    return -1;
}

int main() {

    graph g1 = graph();

    // Node initialization.
    node[N] nodes;
    for (int i = 0; i < N; i += 1) {
        nodes[i] = node(i, i);
```

```
        add_node(g1, nodes[i]);
    }

    // Edge initialization.
    edge[N - 1] edges;
    for (int i = 0; i < N - 1; i += 1) {
        edges[i] = edge(nodes[i], nodes[i + 1]);
        add_edge(g1, edges[i]);
    }

    bfs(g1, node[0], node[N - 1]);

    show(g1);

    return 0;
}
```