Graphs Ain't Easy (GaE) Language Reference Manual

Manager: Kevin Zeng (UNI: ksz2109)
Language Guru: Andrew Jones (UNI: adj2129)
System Architect: Jason Delancey (UNI: jrd2172)
Tester: Samara Nebel (UNI: srn2134)

1. About
2. Lexical Conventions
    a. Tokens
    b. Comments
    c. Identifiers
    d. Keywords
    e. Literals
    f. Separators
3. Data types
    a. Primitive data types
    b. Container data types
4. Operators
    a. Boolean operators
    b. Numeric operators
    c. Integer operators
    d. Double operators
    e. Assignment operators
    f. The in operator
5. Control Flow
    a. If statements
    b. Loops
6. Functions
7. Standard Library
    a. Array built-in functions
    b. Map built-in functions
    c. Graph built-in functions
    d. Other built-in functions

About

Graphs are prevalent in many different places across the internet. For example, every social media platform has graphs representing friends (e.g. Facebook) or followers (e.g. Twitter). The goal of this language is to allow the programmer to create a language that provides a much easier way to create and manipulate graphs with any data type stored inside the nodes, including custom structs, compared to conventional programming languages. With easy to use syntax, Dijkstra's algorithm, Kruskal's algorithm, and other commonly used graph algorithms can be easily implemented. Any program that relies on a graph data structure to hold and manipulate data and involves complex graph traversal and manipulation will be ideal for this language. The language is named GaE — Graphs Ain't Easy — in the spirit of the difficulty of writing programs that use graphs, but the language aims to make the experience much easier.

Lexical Conventions

Tokens

There are 5 classes of tokens: identifiers, keywords, operators, literals, and separators. Blanks, horizontal and vertical tabs, newlines, and comments (described below) are ignored when used on their own, though white space is required to separate adjacent keywords and identifiers.

Comments

The characters /* introduce a comment which terminates with the characters */. These comments can span multiple lines, but do not nest, and do not occur within string or character literals.

Identifiers

An identifier consists of a sequence of characters that can be a letter, underscore, or digit. The first character of a variable identifier must be a lowercase letter (e.g. foo), while the first character of a custom struct identifier must be an uppercase letter (e.g. Integer). Identifiers are case sensitive.

Keywords

The following identifiers are reserved for use as keywords and can not be used otherwise:

return	if	else	elseif
for	while	continue	break
in	int	double	char
bool	string	struct	map
graph	func	true	false

Literals

String Literals

A sequence of characters inside of double quotes.

e.g. `"string"`

Character Literals

A single character inside of single quotes.

e.g. `'a'`

Boolean Literals
A boolean literal is either `true` or `false`.

Integer Literals
A sequence of one or more digits representing an unnamed integer. Cannot start with 0.

e.g. `1337`

Double Literals
A double literal consists of an integer part, a decimal point, a fraction part, an `e` or an `E`, and an optionally signed integer component.

e.g. `5.4e-15`

Struct Literals
A struct literal consists of a field and value pair specified by a field name, a colon, and a value. It can contain any number of field and value pairs, and the whole thing is encapsulated by curly brackets. All values must be of the same type, and all fields must be unique. You cannot have an empty struct literal.

e.g. `{value: 5}`

Array Literals
An array literal is a sequence of values (of the same type), separated by commas and encapsulated by square brackets.

e.g. `[1, 2, 3, 4, 5]`

Map Literals
A map literal is a key and value pair separated by a colon, and encapsulated by square brackets. All keys must be the same type, and all values must be the same type, though the key type does not have to be the same as the value type. A key type must be integer, character, or a string. All keys must be unique.

e.g. `["foo":5, "bar": 8]`

Graph Literals
A graph literal is encapsulated by curly brackets and consists of three-tuples that specify the source node, destination node, and edge (in that order) with struct literals. These three-tuples are encapsulated in parentheses and separated by commas. Semantically, each three-tuple defines both the source and destination nodes (if they haven't been already) and a directed edge between them. All nodes must have the same struct type (i.e. they must have the exact same fields), and all edges must have the same struct type, though nodes and edges don't need to have the same

struct type. Each three-tuple must have a unique source-destination combination such that no two three-tuples have the same source and destination. You cannot have an empty graph literal.

e.g.
```
{
        ({value: 1}, {value: 2}, {value: 3}),
        ({value: 2}, {value: 3}, {value: 1})
    }
```

Separators

The following characters are considered separators, i.e. characters that separate tokens

```
( ) [ ] { } ; , :
```

Semicolons are used exclusively for terminating expressions. Use of the other separators will be defined in the following sections.

Data Types

Primitive Data types

Integer

An integer is a number of the set $\mathbb{Z}$ = {-2147483648,..., -2, -1, 0, 1, 2, …,2147483647}. Integers are 32-bit signed integers that can be specified in decimal (base 10). The negation operator can be used to denote a negative integer. The type is denoted with `int`.

e.g. `int i := 101;`

Double

A double is a 32-bit floating-point number that can be specified using the following syntax. The type is denoted with `double`.

e.g. `double i := 2.4;`

Boolean

A boolean expresses a truth value. It can be either true or false. To specify a boolean literal, use the constants true or false. Both literals are case-sensitive.

e.g. `bool lies := true;`

Char

A `char` is an ASCII character that can be any letter, number, punctuation marks, symbols or whitespace. A variable can be declared as a character type and characters can be stored in the variable.

e.g. `char eye := 'i';`

String

A string is an array of characters terminated with a special character '\0'. A string literal can only be specified using double quotes.

e.g. `string greeting := "Hello World";`

Container Data Types

Structs

A struct is a user-defined data type that, unlike arrays, allows for the combination of data items of different kinds. A struct definition is as follows

e.g.
```
struct Int {
    value: int
};
```
This defines a custom struct type that we can use to define variables. For example, given the above struct definition, we can have:

e.g. `Int foo := {value : 5};`

Array

An array is a container of one or more values of the same type. Each value is called an element of the array. The elements of the array share the same variable name. Each element has its own unique index number. An array can be of any type.

e.g.
```
string[] weekdays := [“Saturday”, “Sunday”];
double[] average := [85.6, 91.7];
```

Map

A map is a container of key-value pairings, where a value can be accessed given a key but not vice-versa. The type of the key and value are defined by the type definition on the left-hand side.

e.g. `map<string, double> gpas := [“andrew”: 4.0, “kevin”: 4.0, “samara”: 4.0, “jason”: 4.0];`

Graph

A graph is a collection of nodes and edges. The node type or edge type must be defined on the left-hand side of the declaration.

e.g. `graph<int, int> some_graph := {(1,2,3), (2,1,5)};`

Operators

Boolean Operators

||

This is the logical OR operator. The || operator guarantees left-to-right evaluation: the first operand is evaluated; if it is unequal to false, the value of the expression is true. Otherwise, the right operand is evaluated, and if it is unequal to false, the expression's value is true, otherwise false. The operands need not have the same type. The result is type `boolean`.

e.g.

```
true || false /* returns true */
false || false /* returns false */
false || true /* returns true */
```

&&

This is the logical AND operator. The && operator guarantees left-to-right evaluation: the first operand is evaluated; if it is equal to false, the value of the expression is false. Otherwise, the right operand is evaluated, and if it is equal to false, the expression's value is false, otherwise true. The operands need not have the same type. The result is type `boolean`.

e.g.

```
false && true /* returns false */
true && false /* returns false */
true && true /* returns true */
```

!

This is the logical NOT operator. The ! operand of the operator must have an boolean type. The operator negates the boolean value of the operand. The result is type `boolean`.

e.g.

```
!true /* returns false */
!false /* returns true */
```

Numeric Operators

<, >, <=, >=

These are the RELATIONAL operators. These operators evaluate from left-to-right: if the relation is true, the operator returns true; and if the relation is false, the operator returns false. The result is type `boolean`. These operators compare numeric values, not pointers.

e.g. `1 > 2 /* returns false */`

==, !=

These are the EQUALITY operators. These operators are analogous to the relational operators, except they have lower precedence. Thus, a<b == c<d would be evaluated: (a<b) == (c<d) where the operands on either side of the equality operators are evaluated first.

e.g. `1 == 1 /* returns true */`

e.g. `2 > 3 == 3 < 5 /* returns false */`

Integer Operators

+, -, *, /, %, ++, --

These are the INTEGER operators. The **+** operator denotes addition, the **-** operator denotes subtraction, the ***** operator denotes multiplication, the **/** operator denotes division, the **%** operator yields the remainder, the **++** operator increments, and the **--** operator decrements. The result is type `int`. All of these operators may be used *only* in **integer** arithmetic; if used incorrectly, the compiler will throw a "type mismatch" error.

e.g.

```
5 + 6 /* returns 11 */
5.0 + 6 /* error */
```

Double Operators

+., -., *., /., %.

These are the DOUBLE operators. The **+.** operator denotes addition, the **-.** operator denotes subtraction, the ***.** operator denotes multiplication, the **/.** operator denotes division, and the **%.** operator yields the remainder. The result is type `double`. All of these operators may be used *only* in **double** arithmetic; if used incorrectly, the compiler will throw a "type mismatch" error.

e.g.

```
5.0 +. 6.0 /* returns 11.0 */
5.0 +. 6 /* error */
```

Assignment Operators

:=

This is the INITIALIZATION operator. The := operator evaluates from right-to-left and the value of the expression becomes the object referred to by the left operand. Both operands must have the same type. The type of the left operand precedes the variable name, and the type of the right operand is inferred based on the syntax.

e.g. `int i := 1;`

=

This is the REASSIGNMENT operator. The = operator evaluates from right-to-left and the value of the expression replaces the object referred to by the left operand. Both operands must have the same type. The type of the left operand precedes the variable name, and the type of the right operand is inferred based on the syntax.

e.g.

```
int i := 1;
i = 2;
```

The `in` Operator

The `in` operator is evaluated from left to right and takes two arguments in the form

```
arg1 in arg2
```

and returns a boolean value for whether `arg1` is contained by `arg2`. Note that use of this operator is valid only when `arg2` is an array, map or graph. Here is how the operator's behavior is defined in each case:

Array

When `arg2` is an array, `arg1 in arg2` returns true if and only if `arg1` is an element of `arg2`.

e.g. `5 in [1,2,3,4,5] /* returns true */`

Map

When `arg2` is a map, `arg1 in arg2` returns true if and only if `arg1` is a key of `arg2`.

```
e.g. 5 in [1:2,3:4,5:6] /* returns true */
     6 in [1:2,3:4,5:6] /* returns false */
```

Graph

When `arg2` is a graph, `arg1 in arg2` returns true if and only if `arg1` is a node of `arg2`.

```
e.g. {value: 5} in {({value: 1}, {value:5}, {value: 3})}
     /* returns true */
```

Control flow

If-statements

If-statements are used to define a block of code whose execution is conditional based on an expression that returns a boolean value. They follow the form

```
if expr {
     /* some code */
}
```

They may optionally contain an else-statement, which is executed in the event that expr evaluates to false:

```
if expr {
     /* some code */
} else {
     /* some more code */
}
```

We may also have an elseif-statement, which must follow an if-statement and is used to define a block of code whose execution is conditional based on a given expression and on the premise that all previous if or elseif-statements are false. An elseif-statement cannot follow an else-statement, if the else-statement exists.

```
if expr1 {
     /* some code */
} elseif expr2 {
     /* some more code */
} elseif expr3 {
     /* even more code */
} else {
     /* last bit of code */
}
```

Here is an example piece of code making use of if statements.

```
int foo := 5;
if foo > 10 {
     print(“your number is greater than 10”);
} elseif foo > 7 {
     print(“your number is greater than 7”);
```

```
} else {
     print(“your number is less than or equal to 7”);
}
```

Loops

There are two kinds of loops we support: while-loops and for-loops. While-loops define a block of code that is repeatedly executed as long as a given expression continues to evaluate as true. While-loops follow the form

```
while expr {
     /* some code */
}
```

An example while-loop that prints the numbers 1 to 10 is as follows:

```
int x := 1;
while x <= 10 {
     x += 1;
     print(x);
}
```

For-loops are similar to while-loops in that they define a block of code that is repeatedly executed while a given expression continues to evaluate as true, but there is syntactic support for variable instantiation and modification after every execution of the defined block of code. For-loops follow the form:

```
for initialize; test; step {
     /* some code */
}
```

An example for-loop that prints the odd numbers from 1 to 20 is as follows:

```
for int x := 1; x < 20; x+=2 {
     print(x);
}
```

We also support nested loops, where a while or for-loop is embedded within an outer while or for-loop. An example of nesting loops to find all prime numbers under 100 is as follows:

```
for int x := 2; x < 100; x++ {
     bool is_prime := true;
     for int y := 2; y < x; y++ {
          if x % y == 0 {
               is_prime = false;
          }
     if is_prime {
          print(x);
     }
}
```

In loops, we can make use of the `continue` and `break` keywords. When a `continue` is reached within a block of code inside a for or while-loop, execution of the current loop halts and execution of the next loop begins. For example, a `continue` statement is used in this loop such that all numbers from 1 to 10 are printed except for 5:

```
for int x := 1; x <= 10; x++ {
     if x == 5 {
          continue;
     }
     print(x);
}
```

When a `break` is reached, the entire loop is terminated. For example, a `break` statement is used in this loop such that only numbers 1 to 5 are printed, even though the loop would have otherwise printed 1 to 10.

```
for int x := 1; x <= 10; x++ {
     if x == 6 {
          break;
     }
     print(x);
}
```

Functions

A function declaration has the form:

```
func func_name(parameter-list) return-type
```

The parameter list specifies the types of the parameters, which are separated by commas. A function is then followed by curly brackets. Inside the curly brackets is the code for the function, which must include a return statement denoted by the `return` keyword, returning the correct type as specified in the function declaration. All functions must specify a return type. All programs must have a main function (`func main() int`) which must follow this format. The code within this function will be executed when the program is run.

e.g. `func add(int i, int j) int { return i+j; }`

Standard Library

Array built-in functions

`len(array)`

Returns the length (type int) of a given array.

e.g.
```
len([1,2,3,4]); /* returns int of value 4 */
```

`array[i]`

Returns the ith element in the array (starts at index 0). If there is no ith element, throw an error.

e.g.
```
int[] arr := [1,2,3,4];
   arr[1]; /* returns int of value 2 */
```

Map built-in functions

`len(map)`

Returns the total number of key-value pairs in the map.

e.g.
```
len([1:2,3:4,5:6]); /* returns int of value 3 */
```

`map[k]`

Returns the value mapped to given key k. If key k doesn't exist in map, throw an error.

e.g.
```
map<int, int> mymap := [1:2,3:4,5:6];
   mymap[3]; /* returns int of value 4 */
```

Graph built-in functions

`addNode(g, value)`

Adds a node to the graph with the specified value. If a node with the same value exists, return false. Otherwise return true.

`nodes(g)`

Returns an array of the values of the nodes.

`addEdge(g, src, dest, edge)`

Adds edge to a graph with specified source, destination, and edge value. If an edge with the same source and destination exists, return false, else return true. Note that type of `src` and `dest` must

be the same as type of node specified by the graph. The type of `edge` must also be the same type of edge specified by the graph.

`edges(g)`
Returns an array of the values of the edges.

`getEdges(g, node)`
Returns edges starting from node (described by its value).

Other built-in functions

`print(value)`
Prints the contents of `value` to standard output.

`string_of_struct(struct)`
Returns a string-formatted version of the struct.

`string_of_array(array)`
Returns a string-formatted version of the array.

`string_of_map(map)`
Returns a string-formatted version of the map.

`string_of_graph(graph)`
Returns a string-formatted version of the graph.

`string_of_int(int)`
Returns a string-formatted version of the int.

`string_of_double(double)`
Returns a string-formatted version of the double.

`int_of_double(double)`
Returns the value of a double as an int. The digits that follow the decimal point are truncated.
e.g. `int_of_double(5.3) /* returns 5 */`

`double_of_int(int)`
Returns the value of an int as a double.
e.g. `double_of_int(5) /* returns 5.0 */`