GrAlgo

Language Specification Document
Team 5

Sahil Chandra - CS20BTECH11033
P Ganesh Nikhil Madhav - CS20BTECH11036
Gorantla Pranav Sai - CS20BTECH11018
Suraj Telugu - CS20BTECH11050
Umesh Kalvakuntla - CS20BTECH11024
Vanga Aravind Shounik - CS20BTECH11055
Adepu Vasisht - CS20BTECH11002

August 25, 2022

Contents

Introduction

1.1 Description of Language

GrAlgo is language build to check graph related properties and make use of graph algorithms in an easier and faster way. Graph data structure is one of the non linear data structure with nodes and edges which can implemented as adjacency list or adjacency matrix. Graph are frequently used data structures and their algorithms play a crucial role in solving real life problems.

There are many handy graph algorithms such as Depth for Search (DFS), Breadth for Search (BFS), Cycle Detection, Shortest Path Algorithm (Dijkstra's Algorithm) ...etc. These algorithms are hard to remember and implement, since they are hard to understand student face problem to solve graph related problems. This language tackles this problem by implementing an inbuilt graph data structure which has in built methods as algorithms of graph. User do not need to understand the complexity of algorithm just by following the syntax user can use GrAlgo to find solutions to to graph related problems. GrAlgo is a language focused on graphs and its algorithms.

1.2 Uses

- Since C/Cpp does not have a dedicated library for graphs we have created a dsl solely for the usage of graphs.
  - Using a new data type named 'graphs' to represent graphs.
  - We are trying to implement 4 types of commonly used graphs and they are, Directed Graphs, Undirected Graphs, Weighted Graphs, and Unweighted Graphs.
  - Common kinds of graph traversals like DFS and BFS are implemented and need to be written by the user. We return a sequence of nodes in these traversal algorithms.
  - We are going to implement functions which take arguments as paths, such as append operations.
  - Find the shortest path between two nodes, and also find the number of paths between two nodes.
  - Checking connectivity types between graphs like connected, strongly connected and **disjoint**.
  - Three internal interpretations are used: **adjacency matrix, adjacency list** and **vertex and edge set**.
  - Has algorithms to detect whether a cycle exists or not, additionally we can also check whether there is a cycle of given length n.

Syntax

2.1 Comments

The comments in this language follow the general C comment syntax.

- With the line ending comments starting with two forward slashes `// This is a valid comment`
- MultiLine comments are written in the following way **"/* Matter */"**.

```
// This is a comment
int a=3; // This is also a comment

/*
This is a multiline comment
*/
```

- Nested comments and comments between strings are not allowed

```
 string a = "Aravind /* This is not a comment*/ Shounik" // This is considered a
comment
```

2.2 Whitespaces

- WhiteSpace characters that are not matched in any regex of the grammar are ignored.
- "\t" is used for a single tab-space while "\n" is used for new line is used in string literals similar to C.

```
func (int) main()
 {
     int a = 5;
     if(a < 50)
     {
         a++;
     }

     return 0;
 }
```

ff Graph dGraph Node nodeSet edgeSet nodeSeq

```
func(int) main()
{int a = 5;
if(a<50)
{a++;}
return 0;
}
```

2.3 Punctuation

Statements should be terminated with a semicolon (;)

2.4 Datatypes

2.4.1 Basic Datatypes

Datatype	Size	Example	Default Value
int	4B	`int a;`	0
large or large int	8B	`large int a;` `large b;`	0
small or small int	2B	`small int a;` `small b;`	0
float	4B	`float a;`	0.0
double	4B	`double a;`	0.0
char	1B	`char c;`	\0
string		`string s;`	„„

2.4.2 Graph related Datatypes

- **Graph:** This data structure represents an undirected graph

```
Graph G = {2:3:3, // v1 : v2 : w
           3:1:1,
           1:2,
           2:4,
           1:3,
           }
           //Here graph can take input from a list like above where edges are
 separated by , and source and destination of edge are separated by : Here G
ignores edge 1:3 because it is redundant to 3:1

           // v1, v2 represents the nodes Node v1, v2. And their weight is
represented by w. Default value of weight is 1 (if is not provided).
```

- **dGraph:** This data structure represents an directed graph.

```
dGraph G ={
           2:3:3,
           3:1:1,
           1:2,
           }
           //Here graph can take input from a list like above where edges are
 separated by , and source and destination of edge are separated by : and also
order of the nodes matter here.
```

- **Node:** can represent a node in a particular graph.

```
Node<G> n=3; //
nodeSet<G> s;
s.add(n);
```

- **Edge:** can represent an edge in a particular graph.

```
Edge<G> e;
```

- **nodeSet:** This data structures represents an set of nodes.Here nodeSet will not contain any repeated nodes and stores them without order

```
nodeSet<G> N = G.Nodes;
```

- **edgeSet:** edgeSet stores edges as tuples and does contain redundant edges(It varies according to dGraph and Graph).

```
edgeSet<G> E = E.Nodes;
```

- **nodeSeq:** nodeSeq stores a sequence of nodes in a given order.nodeSeq can only exist along with the graph it describes.Path,cycle,trail,walk can be represented in nodeSeq.

```
nodeSeq<G> N; // N stores nodes of a graph in a order
```

2.5 Operators

2.5.1 Unary Operators

Purpose	Symbol	Associativity	Valid Operands
Parentheses for grouping of operations	( )	left to right	int, float
Member access operator	.	Left to Right	struct uGraph dGraph
Unary negation	!	Right to Left	all bool
Increment	++	Right to Left	int, float
Decrement	--	Right to Left	int, float

Example:

```
int a = ++b;              // returns b added by 1
int a = --b;              // returns b substracted by 1
bool x = !(3<5);          //logical complement operator which negates boolean variables
```

2.5.2 Arthimetic Operators

Purpose	Symbol	Associativity	Valid Operands
Exponent	**	Left to Right	int, float
Modulo	%	Left to Right	int, float
Multiplication	*	Left to Right	int, float
Division	/	Left to Right	int, float
Addition	+	Left to Right	int, float, string
Subtraction	-	Left to Right	int, float

Example:

```
int a = 3 + 5;                  // adds 3 and 5 to 8
float b = 1.2 + 6.5;            // adds 1.2 and 6.5 to get 7.7
float c = 1.2 + a;              // adds 1.2 and a an int to get 9.2
int a = 5 - 2;                  // substracts 2 from 5
int b = 9*5 ;                   // multiplies 9 and 5 to get 45
int c = 8/3;                    // divides 8 by 3 and returns the quotient 2
int d = 8%3;                    // returns the remainder 2 when 8 is divided by 3
int a = (3+5)*4 + 3             // returns 35 as '()' is taken precedence over * and
    this over +
```

2.5.3 Other Operators

Purpose	Symbol	Associativity	Valid Operands
Relational Operators	<=, <, >=, >	left to right	all data types
	==, !=	left to right	all data types
Shift operators	>>	Left to Right	int, long
	<<	Left to Right	int,long
Bitwise Operators	&, |, ^	left to right	bool
Logical Operators	AND OR	left to right	bool
Assignment operators	=, *=, +=, /=, **=, %=	right to left	wherever the operator is valid
cons operation	::	left to right	vSeq,eSeq
append operation	@	left to right	vSeq,eSeq

Example:

```
//Relational operators
bool a = (3 < 5);                 // returns true if 3 is less than 5
bool b = (17 <= 8);               // returns false as 17 is not less than or equal to 8
bool d = (4 == 4);                // return true as 4 is equal to 4
bool a = (2 > 1);                 // return true as 2 is greater than 1
bool b = (4 >= 3);                // return true as 4 is greater than or equal to 3

// Logical Operators
bool a = (3<5)&&(4>2);            // returns AND of both the boolean expressions - true
bool b = (3>5)||(4<2);            // returns OR of both the boolean expressions - false
bool c = (3>5)^^(3<5);            // returns XOR of both the boolean expression - true

// Bitwise operators
int a = 3&5;                      // returns the bitwise AND of both the operators - 1
int b = 3|5;                      // returns the bitwise OR of both the operators - 7
int c = 3^5;                      // returns the bitwise XOR of both the operators - 6
int d = 4<<1;                     // returns the number when operand1 shifted to the left
    by operand 2 times
                                  // the first bits are lost - 8
int e = 1>>1;                     // similar to the above it returns 0
int a = ~4;                       // negates every bit of the number,can be considered
   unary

//Assignment operators
int a = 1;                        // Assigns the value of 1 to the variable
int a += 1;                       // This adds 1 to the value of a and assigns it again
   giving 2
```

```
int a *= 4;                          // Assigns (a=2)*4 to a
int a /= 2;                          // Assigns (a=8)/2 to a
//Similar with all the bitwise operators except '~' -(&=,|=,>>=,<<=)

//String additon
string c = "abc" + "def";            // returns the concatenation of both operands "abcdef"
```

2.6 Keywords:

if	else	int	float	static	struct
extern	const	for	while	void	char
string	bool	NULL	large	small	func
return	break	continue	typedef	Graph	dGraph
Node	nodeSet	edgeSet	nodeSeq		

2.7 Identifiers:

Identifiers in our language need to follow certain rules, and they are

- The identifier should start with a letter (both capital and small included) or an underscore.
- Then any of the following can be used
  - Another character.
  - Another underscore.
  - Digit
- The regex for the identifier is given as follows `[_a-zA-Z][_0-9a-zA-Z]*`.
- But they are still words which follow these rules and cannot be used as names for identifiers because they serve some other purpose; which were shown in 2.6

2.8 Expressions:

Expressions in GrAlgo are valid sequence of operands and operators, where operands can be identifiers or constants on which operations are performed. The sequence denotes the action of operation to be executed.

```
a*5     //  Expression using binary operator , operands are identifier and constant
a + b   //  Expression using binary operator,  both operands are identifier
a >= 10 //  Expression using relational operator
b /= 10 //  Expression using binary operator
```

2.9 Declarations:

2.9.1 Variable Declaration:

A variable declaration would start with the datatype name, then the variable name with initialization optional, could come any number of times separated by ‘,’.
For Example,

```
int a = 5,b;
```

2.9.2 Function Declaration:

The function declaration starts with the keyword “func” and then the return type closed within the parentheses “()”. Then comes the name which contains the similar rules of syntax as an identifier. The closed in the “()” parentheses come the function parameters with types and names with no initialization.
For Example,

```
func(int) add(int a,int b);
```

2.9.3 Array Declaration:

An array declaration is almost similar to a variable declaration. The only differences are size and initialization. The size should be mentioned in “[ ]” immediately after the identifier. The initialisation should be done enclosing the values in “” seperated by ‘,’. There should be as many as values as mentioned in the size if initialized.
For Example,

```
int a[10],b[5] = {1,3,5,6,19};
```

2.9.4 Declaration Scope:

Declaration scope in GrAlgo is defined by “”. Any variable, function and an array can not be used outside of that scope. Any variable which has already been in a larger scope cannot be redefined in a smaller scope that is they cannot have same names.
For Example,

```
int a;

  int main(){
      int a,b;     // Here 'a' cannot be redeclared in the local scope.
  }
```

2.9.5 Initialisations:

Every variable when declared would be initialized to a default value. The variable could be assigned any value of choice which is in the range of that datatype. The value to be assigned can also be an expression which constitutes to a value.
For Example,

```
    int a = 5, b;
```

2.10 Statements:

Types of Statements in GrAlgo are:

1. Labeled statements
2. Compound statements
3. Expression statements
4. Selection statements
5. Iteration statements

Labeled statements are used in switch statements. A simple identifier followed by a ":" is a label.
For example, in switch statements, there are case and default labeled statements.

2.10.1 Labeled Statements:

```
case constant-expression : statement
```

If the value of the switch statement matches the value of the constant expression, then the control will pass to this statement.
If the value of the switch statement does not match any of the case statements, then the control will pass to the default statement unless the switch is in another switch.

2.10.2 Compound Statements:

A compound statement is an optional list of declarations and optional list of statements enclosed within braces.
For Example,

```
{
  int a; // declaration
  a = 5; // statement
}
```

2.10.3 Expression Statements:

It is an optional expression followed by a semicolon. If there is some expression, it might have a value. If there is no expression and just a semicolon, it is called an empty or null statement.
Statements which just have a function call followed by a semicolon come under this category.

2.10.4 Selection Statements:

There are three types of selection statements in GrAlgo:

1. if (expression) statement
2. if (expression) statement else statement
3. Switch statements are also type of selection statements
   `switch (expression) statement`

2.10.5 Iteration Statements:

There are two types of iteration statements in GrAlgo:

1. While Loop

```
while( expression )
{
  // statements go here
}
```

2. For Loop

```
for ( initialisation_statement; condition_check_expression; update_statement )
{
  // statements go here
}
```

3. Range based for loops

```
for ( range_declaration : range_expression )
{
  // statements go here
}
```

2.11 Inbuilt Utilities:

- **Breadth First Search**:Works similar to BFS algorithm

```
BFS(Node d:Graph.Nodes, Node  src){//BFS syntax here is similar to range based
 for loops we use in cpp
    //code goes here
}

BFS(NodeSeq d:Graph.Levels, Node  src){//BFS syntax here is similar to range
based for loops we use in cpp
    //code goes here
}
```

- **depth First Search**:Works similar to DFS algorithm

```
DFS(Node d:Graph.Nodes, Node  src){//DFS syntax here is similar to range based
 for loops we use in cpp
    //code goes here
}
```

Non-Trivial Code

```
func(int) main(){

    Graph G = {2:3:2, 3:1:4, 1:2:1, 2:4:5, 1:3:2, 4:5:1, 6:2:2};

    Node src = 5;

    // BFS use case:
    BFS(Node d : G.Nodes, Node src)
    {
        print(d);
    }
    // Above code prints all the nodes in BFS order of graph : 5, 4, 2, 1, 3, 6

    DFS(Node d : G.Nodes, Node src)
    {
        print(d);
    }
    // Above code prints all the nodes in DFS order of graph : 5, 4, 2, 3, 1, 6

    return 0;
}
```

Grammar

Type: INT | FLOAT | STRING | BOOL | STRUCT | NUMSET | STRSET | GRAPH | MATRIX | VOID | LONG
Operator: PLUS | MINUS | MULTIPLY | DIVIDE | MODULO | EXPONENT | LPAREN | RPAREN | SHIFT_LEFT | SHIFT_RIGHT | TRANSPOSE | DECREMENT | INCREMENT | UNARY_NEGATION | MEMBER_ACCESS | UNION | INTERSECTION | SET_DIFFERENCE | RELATIONAL | LOGICAL | ASSIGNMENT

Relational : `'<='`
| `'<'`
| `'>'`
| `'>='`
| `'=='`
| `'!='`

Logical : `'&'`
| ’|’

Assignment : `'='`
| `'+='`
| `'-='`
| `'*='`
| `'/='`
| `'^='`
| `'%='`

Decls : /*nothing*/
| `Decls var_decl`
| `Decls fun_decl`

fun_decl : `Type IDENTIFIER LPAREN Args RPAREN`

Args : /*nothing*/
| `Type IDENTIFIER`
| `Type IDENTIFIER ‘,’ Args`

var_decl : `Sc_specifier Type id_list ‘;’`

id_list : IDENTIFIER
| id_list ‘,’ IDENTIFIER

Expression : expr Operator expr
| NOT expr
| ABS expr ABS
| FALSE
| FLOAT_LITERAL
| STRING_LITERAL
| LITERAL