Graphene Reference Manual

Ashar Nadeem  an3056
Shengtan Mao  sm4954
Vasileios Kopanas  vk2398
Matthew Sanchez  mcs2307

Table of Contents

1  Introduction
2  Data Types
3  Lexical Conventions
4  Expressions
5  Programs
6  Statements
7  Standard Library
8  Sample Code
1 Introduction

Graphene is an imperative programming language primarily used to easily implement graph algorithms. Our language is at its core a subset of C with built-in graph support for algorithms. Graphene uses C syntax with specified operators for extra built-in data types and functions to make the manipulation of graphs easy. Graphene will support all of the basic C arithmetic and logical operations as well as user-defined functions. We will have a small library of built-in functions to complement the built-in types and enable users to efficiently write, use, and analyze graph algorithms. The language was inspired by looking at the CLRS Algorithms book, and trying to replicate the different types of graph algorithms as efficiently as possible in a C syntax.

2 Data Types

Primitive Types

Integer
An integer is a sequence of decimal digits, always using decimal notation. We will be treating integers as booleans, similarly to how C already does, 0 denotes false, nonzero denotes true.

E.g. 32

Float
A float consists of an integer part, a decimal point, followed by a fraction part. The integer and fraction parts both consist of a sequence of digits. None of these can be missing. Again, strictly decimal.

E.g. 3.2

String
A string is a sequence of characters surrounded by double quotes. Strings are immutable.

E.g. “32”
**Built-in Types**

These include *functions* which return objects of a given type and the following built-in types. The built-in types wrap other types. Currently, user-defined classes and structs are not available.

**Node**

A node contains a key of type int and a value of the declared type. Both the key and value can be altered. It also holds a list of edges.

*Example:*  
\[
\text{node<int> } n = \text{make}(0, 1);
\]

Key access: `node.key`

Value access: `node.val`

**Graph**

A graph is a collection of nodes that hold values of a declared type. Nodes within the graphs are accessed using their keys, so all nodes in the graph must have unique keys. Nodes are accessed using `.get()`.

*Example:*  
\[
\text{graph<int> } g ;
\]

**Edge**

An edge behaves as a struct and holds a float type weight, a node, and an int type indicating whether the edge is traversable. If traversable is 1, it means the edge is directed towards the node that it holds; else it is 0. Edges cannot be directly declared by the user, they are attributes of a node’s edgelist.

**List**

A list is a doubly-linked list that holds elements of a declared type, keyword: “list”.

*Example:*  
\[
\text{list<int> } l ;
\]

---

3 Lexical Conventions

There are five kinds of tokens: keywords, identifiers, literals, expression operators, and separators. Spaces, tab characters, and newline characters are ignored aside from
however they may separate tokens.

Comments

Multi-line comments: /* starts a comment, terminated by */
Single-line comments: // starts a comments, terminated by \n
Keywords

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

```
void    int    float    string
graph   node   edge    list
if      else   for     foreach
while   return continue break
```

Literals

These include the three primitive data types. Integers must consist entirely of decimal digits. Floats must consist of the decimal digits, a decimal, and the fractional value, with no leading zeros. Strings must be enclosed in double quotes.

Identifiers

Identifiers are strings that reference types, they can be declared or assigned freely, although types cannot be changed. Identifiers must start with an upper or lower case letter, and this can be followed by any amount of letters, digits, or underscores.

4 Expressions

Expressions are recursively built out of primary expressions and various operators, which are the following:
Primary Expressions

4.1.1 Identifier
An identifier is a primary expression, provided it has been suitably declared. Its type is specified by its declaration.

4.1.2 Literal
An integer, float, or string constant is a primary expression.

4.1.3 ( expression )
A parenthesized expression is a primary expression whose type and value are identical to those of the unadorned expression.

4.1.4 primary-expression ( expression-list optional )
A primary expression followed by parentheses containing a possibly empty, comma-separated list of expressions, is a function call. The primary expression must be of type function, and the result of the function call corresponds to the function type. All primitive and built-in types are passed by value.

4.1.5 node-id . id
An id expression followed by a dot followed by the name of a member of a structure is a primary expression. This is only used to access keys and values of nodes.

4.1.6 primary-id . method ( expression-list optional )
An id expression followed by a dot followed by the name of a method of its type followed by the arguments for the method. The id expression has to represent a special type since methods are not supported in general.

Operators
Operators act on expressions, and require a sensical value of the expressions. All operators are listed in decreasing order of precedence.

4.2 Unary operators
Expressions with unary operators group right-to-left

4.2.1 ! expression
The result of the logical negation operator ! is 1 if the value of the expression is 0, 0 if the value of the expression is non-zero. The type of the result is int. This operator is applicable only to ints.

4.2.2 - expression
The result of the arithmetic negation operator the value of the int or float it is applied to, multiplied by -1.

4.3 Multiplicative
The multiplicative operators *, /, and % group left-to-right, all have the same precedence.

4.3.1 expression * expression
The binary * operator indicates multiplication. Operands can be int or float, if at least one of them is float, the result will be float.

4.3.2 expression / expression
The binary / operator indicates division. Operands can be int or float, if at least one of them is float, the result will be float. Division by 0 will produce an error.

4.3.3 expression % expression
The binary % operator yields the remainder from the division of the first expression by the second. Both operands must be int, and the result is int. The remainder has the same sign as the dividend.

4.4 Additive
The additive operators + and − group left-to-right, all have the same precedence.

4.4.1 expression + expression
The result is the sum of the expressions. Operands can be int or float, if at least one of them is float, the result will be float.

4.4.2 expression − expression
The result is the difference of the operands. Operands can be int or float, if at least one of them is float, the result will be float.

4.5 Relational
The relational operators group left-to-right, all have the same precedence.

4.5.1 expression < expression
Returns 0 if the expression is false and 1 if the expression is true.

4.5.2 expression > expression
Returns 0 if the expression is false and 1 if the expression is true.

4.5.3 expression <= expression
Returns 0 if the expression is false and 1 if the expression is true.

4.5.4 expression >= expression
Returns 0 if the expression is false and 1 if the expression is true.

4.5.5 expression == expression
The == (equal to) operator, can act on any type.

4.5.6 expression && expression
The && operator returns 1 if both its operands are non-zero, 0 otherwise. && guarantees left-to-right evaluation; moreover, the second operand is not evaluated if the first operand is 0. This operator is applicable only to ints.
4.5.7 expression \| expression

The \| operator returns 1 if either of its operands is non-zero, and 0 otherwise. \| guarantees left-to-right evaluation; moreover, the second operand is not evaluated if the value of the first operand is non-zero. This operator is applicable only to ints.

4.5.8 id = expression

This assignment operator groups left-to-right. The value of the expression replaces that of the object referred to by the id. The operands need to have the same type.

5 Programs

Programs consist of 0 or more of function declarations, and 0 or more variable declarations.

Variable Declarations

Form: type id ;
This is standard C, type denotes type and id will be the name of the variable.

Function Declaration

Form: type id (formals_opt) { vdecl_list stmt_list }
This is standard C, type denotes return type, id denotes the name of the function, formals-opt is an optional list of formals, of the form type id, and stmt-list denotes 0 or more statements. There is a special void type just for function declaration where no value is returned.

Types

The types are
type:

int
float
string

node< type >
6 Statements

6.1 Expression statement
Most statements are expression statements, which have the form

\[ \text{expression} ; \]

Usually expression statements are assignments or function calls.

6.2 Conditional statement
The two forms of the conditional statement are

\[
\text{if ( expression ) statement} \\
\text{if ( expression ) statement else statement}
\]

In both cases, \text{expression} is evaluated, and the following \text{statement} will be evaluated if it is true.

If there is an else and \text{expression} evaluates to false, the \text{statement} following the else will be evaluated. The dangling else problem will be solved by connecting an else to the last elseless if.

6.3 while statement
The while statement has the form

\[
\text{while ( expression ) statement}
\]

\text{expression} is evaluated prior to each iteration, the loop is terminated when it evaluates to false.

\text{statement} is evaluated during each iteration.

6.4 for statement
The for statement has the form:

\[
\text{for (expression}_{opt} ; \text{expression}_{opt} ; \text{expression}_{opt} ) \text{statement}
\]

The first \text{expression} is evaluated upon entering the loop for the first time.
The second expression is evaluated prior to each iteration, terminating the loop when evaluating to false.
The third expression is evaluated after each iteration completes. 
statement is evaluated every iteration.

6.5 foreach statement
The foreach statement has the form
foreach ( declaration : id-expression) statement
The id expression must represent a container, which is a graph or a list. The type of the declaration must match the type of the elements inside the container. The for statement iterates through the container’s elements, and the identifier is set to the corresponding element at each iteration.

6.6 break statement
The statement
break ;
causes termination of the smallest enclosing while, for, or foreach statement. Control passes to the statement following the terminated statement.

6.7 continue statement
The statement
continue ;
causes the current loop iteration to terminate and prompts the next iteration, performing any tests for loop termination.

6.8 return statement
A function returns to its caller with a return statement, which has the form return ( expression ) ;
The value of the expression is returned to the caller of the function, this must match the function return type.
7 Standard Library

The standard library features special types that support the usage of graphs.

7.1 edge. member-of-structure
An edge holds the weight, accessed by w; the node, accessed by n; and traversable, accessed by t.

7.2.1 node.edges()
Returns the list of edges associated with the node.
e.g. list<edge> edge_list = n.edges();

7.2.2 nodeA ~(weight)>> nodeB
Adds a directed edge between nodeA and nodeB, with a defined weight. Weight can be of any primitive data type, to be declared within parentheses.

7.2.3 nodeA ~>> nodeB
Adds a directed edge between nodeA and nodeB of default weight 1, as an integer.

7.2.4 nodeA ~(weight)~ nodeB
Adds an undirected edge between nodeA and nodeB, with a defined weight. Weight can be of any primitive data type, to be declared within parentheses.

7.2.5 nodeA ~~ nodeB
Adds an undirected edge between nodeA and nodeB of default weight 1, as an integer.

7.3 graph.get(key)
Returns the node with the specified key. Accessing a nonexistent node through this operator is undefined behavior.
e.g. node<int> n = g.get(5);
7.3.1 graph.in (key)
    Returns 1 if this graph contains the node with the specified key. Returns 0 otherwise.

7.3.2 graph.add (node)
    Adds the specified node to this graph if it is not already present. The graph and node must contain the same type of value. Returns 1 if the new node is added, 0 otherwise.

7.3.3 graph.add (key, value)
    Automatically constructs the node from key and value and adds the node to the graph as in 7.3.2.

7.3.4 graph.del (key)
    Removes the node with the specified key from this graph if it is present. Returns 1 if the node is removed, 0 otherwise.

7.4.1 list.empty ()
    Returns 1 if the list is empty, 0 otherwise.

7.4.1 list.push_front (e)
    Adds the element to the beginning of the list. The element must be the same type declared to be contained by the list. Returns 1 if the element is added, 0 otherwise.

7.4.2 list.push_back (e)
    Adds the element to the end of the list. The element must be the same type declared to be contained by the list. Returns 1 if the element is added, 0 otherwise.

7.4.3 list.pop_front (e)
    Removes the element at the beginning of the list and returns it. Undefined behavior is list is empty.
7.4.4 list.pop_back (e)
   Removes the element at the end of the list and returns it. Undefined behavior is
   list is empty.
8 Sample Code

// Declare a graph
graph <int> g;

// Adds nodes to the graph
for(int i = 0; i < 10; i++){
    g.add(i, 2*i);
}

// Create an edge of weight i from root node to every other node
for(int i = 1; i < 10; i++){
    g.get(0) ~ (i) >> g.get(i);
}

// Overwrite edge from g[0] to g[2] with weight 10
g.get(0) ~ (10) >> g.get(2);

// BFS search example (Takes in graph and destination)
void bfs(graph<int> g, node<int> n){
    list<node<int>> q;
    graph<int> discovered;
    q.push_back(g.root());
    while(!q.empty()){
        node<int> m = q.pop_front();

        if(m == n){

}}
return m;

```java
foreach (node<int> e : m.edges()){
    if (!discovered.contains(e)){
        q.push_back(e);
    }
}
```