GPL
Language Reference Manual

Qingxiang Jia (qj2125)
Peiqian Li (pl2521)
Ephraim Donghyun Park (edp2114)

December 17, 2014

Contents

1 Introduction

Graph is a very powerful data structure that can be used to model a variety of systems in many fields. Graph is such a fundamental model that people have developed libraries dedicated to graphs in almost all general-purpose high-level programming languages. However, implementing graph-related algorithms in languages like Java or C++, even with the benefit of using third-party graph libraries, entails manual manipulation of nodes and edges. This could prove to be error-prone (with pointer manipulations in C++), tedious (verbose especially in Java), and daunting (to people new to the programming world).

The Graph Programming Language (GPL) is a domain-specific language that attempts to remedy these problems. GPL strives to hide most logic behind graph handling under the hood, so that programmers are able to focus more on using graphs, instead of implementing them.

The primary goal of GPL is to allow programmers to create, use, and manipulate graphs in a natural, flexible and intuitive way. All graph-based algorithms should be easier to implement in GPL, e.g. shortest path, spanning tree, strong connectivity. Because all trees are graphs, GPL is automatically suitable for applications involving tree structures, such as priority queues (min/max heaps), binary search trees, or any kind of hierarchical data representation.

2 Lexical Conventions

2.1 Comments

Two slashes "//" introduce one-line comment, which is terminated by the newline character. For multiline comment, /* is used to start commenting and */ is used to terminate commenting. Nested commenting is not supported by the language.

2.2 Identifiers

An identifier consists of a sequence of letters, digits, and the underscore character; the first character of an identifier cannot be a digit. Upper and lower case letters are considered different.

2.3 Keywords

The following identifiers are reserved by the language to have specific meanings and may not be used otherwise:

boolean	break	char
continue	edge	else
for	graph	if
int	node	return
string	while	

2.4 Object Type

In GPL, graph, edge, node, and string are object types. ”Object” implies that they are not primitive data types, and they can have member functions and fields which can be invoked or accessed through the dot operator. Graph, node, edge, and string are the only four object types in GPL; GPL does not support user-defined objects, but does support user-defined functions.

2.5 Literals

2.5.1 Integer literals

Integer literals are decimal (base 10). An integer literal is just a sequence of digits.

2.5.2 Character literals

A character literal is one or two characters enclosed by single quotes. Two characters enclosed by single quotes are for escape characters. The first character must be back-slash, and the second one can be back-slash, single quote, ’n’, ’r’, ’t’; they represent backslash, single-quote, line-feed, carriage-return, and tab, respectively. The only valid representation of the single-quote character is a back-slash followed by a single-quote, enclosed in two single quotes.

2.5.3 Boolean literals

There are exactly two boolean literals: **true** and **false**.

2.5.4 String literals

A string literal is a sequence of characters enclosed by double quotes. Backslash, double-quote, line-feed, carriage-return, and tab characters need to be escaped by a preceding back-slash, similar to character literals.

2.5.5 Graph literals

A graph literal is a list of weighted directed edges enclosed by square brackets. All weights must be integers. Only directed edges are supported by GPL.

3 Expressions

3.1 Primary Expressions

3.1.1 identifier

An identifier is a unique (in its own scope) name used to identify an entity in the program. It can be the name of a function, parameter, or variable. A reserved keyword cannot be used as an identifier.

For array identifier, the following member functions can be used:

len()	this member function returns the size of the array
sort()	this member function can only be used for int, char, or edge arrays and it sorts the element

3.1.2 literal

Literals include strings, characters, numbers (integer), and graphs.

3.1.3 string

String is an object type of the language. String is immutable.

String has the following two member functions:

len()	this member function returns the length of the string
substr(a, len)	this member function returns the substring starting at index **a**, and includes **len** characters.
empty()	this member function returns true if the string is empty
at(i)	this member function returns the character at the specified index
append(s)	this member function appends s to the called string

3.1.4 node

Node is an object type of the language. It represents a node in a graph.

Node has member function:

getID()	returns the unique integer ID of the specified node

3.1.5 edge

Edge is an object type of the language. It connects two nodes.

Edge has member functions:

getWeight()	returns the weight of the edge.
getSrc()	returns the source node of this edge.
getDst()	returns the destination node of this edge.

3.1.6 graph

Graph is an object type of the language. It represents a directed graph which consists of nodes and directed edges with integer weights.

The nodes in a graph can be accessed like fields. For example, if graph g has node v, then the node is represented by the expression ”g.v”.

Graph has member functions:

getNode(int id)	returns the node with specified id
getEdge(node src, node dst)	returns the specified edge. If the specified edge does not exist, a run-time exception is raised.
getAllEdges()	returns the list of edges in the graph as an array
getAllNodes()	returns the list of nodes in the graph as an array
getWeight(node src, node dst)	returns the weight of the edge that goes from src to dst node
getEdgeCount()	returns the number of edges in the graph
getNodeCount()	returns the number of nodes in the graph
getInNeighbours(node n)	returns an array of nodes that have edge to the specified node
getOutNeighbours(node n)	returns an array of nodes that this node has edges to

3.1.7 (expression)

The parenthesized expression is the same as expression. Including an expression in a pair of parentheses does not imply any precedence of the expression.

3.1.8 primary-expression [expression]

The primary-expression in this part can only be array. The expression can only be integers within the range of the array. primary-expression [expression] means to access the expression-th element in the array.

3.1.9 primary-expression (expression)

This expression means a functional call, where primary-expression is an identifier that is a name of a function. The expression in the pair of brackets is parameter(s) to in the call. It can be single parameter. If there are more than one parameters, they should be separated by a comma.

3.1.10 primary-lvalue . member-function-of-object-type

An lvalue expression followed by a dot followed by the name of a member function of object-type is a primary expression.

3.2 Graph Definition Operators

3.2.1 identifier − > identifier

The − > binary operator connects two nodes with zero-weighted directed edge. The direction goes from first node identifier to second node identifier.

3.2.2 identifier :− > identifier, identifier, ...

The :− > operator connects first node identifier with all the other node identifier that follow with zero-weighted directed edge.

3.2.3 identifier −(expression)> identifier

The −(identifier)> operator connects two nodes with weighted directed edge. The direction goes from first node identifier to second node identifier. The weight of the edge equals the expression in the middle. The expression must be of int type.

3.2.4 identifier : −(expression)> identifier, identifier, ...

The : −(expression)> operator connects first node identifier with all the other node identifier what follow with weighted directed edge. The weight of the edge equals the expression in parentheses. The expression must be of int type.

3.3 Unary Operators

Unary operators are grouped from right to left.

3.3.1 − expression

The − unary operator can be applied to an expression of type int or char, and results in the negative of the expression.

3.3.2 ! expression

The ! unary operator can only be applied to an expression of boolean type, and results in the opposite of the truth value of the expression

3.4 Multiplicative Operators

3.4.1 expression * expression

The binary operator * indicates multiplication between expression and expression. The expression pair can be in the following combinations. 1) int int 2) char char 3) int char 4) char int. In case 3, and 4, all the parameters will be treated as int.

3.4.2 expression / expression

The binary operator / indicates division between expression and expression. The expression pair can be in the following combinations. 1) int int 2) char char 3) int char 4) char int. In case 3, and 4, all the parameters will be treated as int.

3.4.3 expression % expression

The binary % operator outputs the remainder from the division of the first expression by the second. The expression pair can be in the following combinations. 1) int int 2) char char 3) int char 4) char int. In case 3, and 4, all the parameters will be treated as int.

3.5 Additive operators

3.5.1 expression + expression

The binary + operator outputs the addition of the first expression and the second expression. The expression pair can be in the following combinations.

1) int int 2) char char 3) int char 4) char int. In case 3, and 4, all the parameters will be treated as int.

3.5.2 expression - expression

The binary - operator outputs the result of the first expression minus that of the second expression. The expression pair can be in the following combinations. 1) int int 2) char char 3) int char 4) char int. In case 3, and 4, all the parameters will be treated as int.

3.6 Relational operators

3.6.1 expression < expression

3.6.2 expression > expression

3.6.3 expression <= expression

3.6.4 expression >= expression

3.6.5 expression == expression

3.6.6 expression ! = expression

The relational operators < (less than), > (greater than), <= (less than or equal to), >= (greater than or equal to), == (equal to), ! = (not equal to) all yield boolean **true** or **false**. The two expressions being compared must be of the same type, and they can be int, float, char or string. Characters are compared by ASCII values; strings are compared lexicographically.

3.7 Assignment operators

3.7.1 variable = expression

The binary = operator indicates that the result of the expression on the right side is stored in the variable on the left. If there is already data stored in the variable, the data will be replaced. The variable can be any legal type defined in the language.

3.7.2 variable += expression

The binary += operator indicates that the value of the variable on the right side will be incremented by the quantity of the result of the expression on the left side. This operator requires the two expressions to be in the same numerical type, i.e. either both in int, or both in char.

3.7.3 variable -= expression

The binary -= operator indicates that the value of the variable on the right side will be decremented by the quantity of the result of the expression on the left side. This operator requires the two expressions to be in the same numerical type, i.e. either both in int, or both in int.

3.8 Logical operators

3.8.1 boolean-expression && boolean-expression

3.8.2 boolean-expression || boolean-expression

The logical operators && (and) and || (or) can be applied to two boolean expressions, and results in the logical AND or OR of the truth values of the two boolean expressions.

3.9 Function calls

Function calls are made by function identifier followed by the list of arguments separated by commas enclosed by parentheses. Function overloading, functions with the same name but different set of argument types, is supported.

4 Declarations

4.1 Type specifiers

The type-specifiers are

```
type-specifier:
                    int
                    char
                    string
                    graph
                    node
                    edge
                    type-specifier [ ]
```

4.2 Variable Declarators

```
declarator:
        type-specifier identifier
        type-specifier identifier = expr
```

4.3 Graph Expression

```
graph-expr
        [ graph-body ]

graph-body
        edge-declaration-list

edge-declaration-list:
        edge-declaration;
        edge-declaration; edge-declaration-list

edge-stmt:
        node-declarator
        node-declarator -> edge-stmt
        node-declarator - ( expr ) > edge-stmt

edge-declaration:
        edge-stmt
        node-declarator : - ( expr ) > node-declarator-list
        node-declarator : -> node-declarator-list

node-declarator-list:
        node-declarator
        node-declarator node-declarator-list

node-declarator:
        identifier
```

4.4 Function declarations

```
function-decl:
        retval formals_opt ) { stmt_list }


procdecl: /*procedure (aka. void function) declarator*/
        void identifier ( formals_opt ) { stmt_list }
```

```
retval:
        vartype identifier (

formals_opt:
        /* nothing */
        formal_list

formal_list:
        vardecl
        formal_list , vardecl
```

5 Statements

5.1 Expression statement

Expression statement is an expression followed by semicolon.

5.2 Compound statement

The compound statement is a list of statements surrounded by parentheses.

5.3 Conditional statement

There are two types of conditional statements:

- Type 1: if (expression) statement
- Type 2: if (expression) statement else statement

In type 1, if expression is evaluated to be true, the statement will be executed. In type 2, if expression is evaluated to be true, the first statement will be executed, otherwise the second statement will be executed.

5.4 While statement

The while statement can be described as: while (expression) statement

As long as the expression is evaluated to be true, the statement will be executed repeatedly. The expression is evaluated before the execution of statement.

5.5 For statement

The for statement can be expressed as:

for (expression-1; expression-2; expression-3) statement

expression-1 defines the initialization of the loop. expression-2 is the test that will be evaluated for truth in each loop. expression-3 defines what to do after each loop has been executed.

5.6 Return statement

Return statement can be described as: return (expression)

The expression can be either a simple expression, which will be evaluated to a value and then be returned to the calling function. Alternatively, the expression can be consisted of one or more function calls, then the return statement will be executed after all function calls have been returned.

5.7 Null statement

A null statement consists a single semicolon. It is useful in a for loop where one or more of the three expressions is not defined (or unneeded to define).

6 Scoping

There are three rules of scoping. The first rule states that the global variables and functions can be referred from anywhere in the code even before it is declared. The second rule states that the variables declared in a function can be referred only after the declaration. The third rule states that the variables declared in a function bind closer than the variables declared outside the function. For example, there is a variable named a in a function, even though, outside the function, there may be a variable a, because of the stronger binding of the variable declared in the function, if one refers a in the function, he or she refers to the one declared in the function.