GRAPL Language Reference Manual

1 Introduction

GRAPL (**GRA**ph **P**rocessing **L**anguage) is a user-friendly language designed to simplify the creation and navigation of directed graphs. Creating graphs in existing languages generally requires building node objects and maintaining arrays or lists of pointers to other nodes (along with weights); keeping track of this information can be cumbersome. In addition, the structure of the graph (in the general case where any node may be connected to any other node) takes a good deal of code to set up. GRAPL is intended to hide the complexity of graph navigation and to make creating graphs and adding nodes as simple as possible.

This manual describes the syntax and use of the GRAPL language. It is an unabashed plagiarism of the C Language Reference Manual (CLRM), published as Appendix A in Kernighan and Ritchie's "The C Programming Language." Where GRAPL and C share common features, we have sometimes reproduced the corresponding text from the CLRM verbatim rather than attempt to paraphrase an identical idea.

2 Lexical Conventions

There are six classes of tokens: identifiers, keywords, constants, strings, operators, and other separators. As in C, whitespace characters are ignored except insofar as they serve to delineate other tokens in the input stream. If the input stream has been parsed into tokens up to a given character, the next token is taken to include the longest string of characters which could possibly constitute a token.

2.1 Comments

GRAPL uses C-style, un-nested comments; that is, the characters `/*` introduce a comment, which terminates with the characters `*/`.

2.2 Identifiers

A GRAPL identifier must meet the same requirements as a C identifier; that is, it consists of a sequence of letters, digits, and underscore characters, where the first character must be either a letter or an underscore. Identifiers are case-sensitive.

2.3 Keywords

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

```
boolean         head            length
else            if              list
forEach         isVisited       node
from            graph           number
```

```
print               unvisit
return              unvisited          with
tail                visit
to                  visited
```

2.4 Literals

The only literals in GRAPL are of type `number` and `boolean.` A number may consist of any valid `float` or decimal `int` representation in C. (C hexadecimal or octal integers are not supported.)

3 Expressions

Expressions, along with operators, are essential elements in the formation of code in the GRAPL language. Here we try to specify all of the expressions with operators.

The precedence of all the operators is the same as the order of the subsections here, that is to say, all operators in sections 3.1 – 3.5 have higher precedence in comparison with operators in section 3.6. Besides, all operators in the same subsection have the same precedence.

We don't assume either left or right associativity for the expressions. For evaluation of expressions, the order is also set to be undefined.

For handling arithmetic problems, we define that overflow of numbers will be discarded. In addition, problems like “divided by zero” will cause exception and be handled in library functions.

The following table summarizes the rules:

Tokens	**Operators**	**Associativity**
Identifiers, parenthesized expression	Primary	
`( )`	Function calls	Left
`head`	List access (head)	Left
`tail`	List access (tail)	Left
`visit unvisit isvisited`	Node Visitation	Right
`!`	Negation	Right
`* /`	Multiplicative	Left
`+ -`	Additive	Left
`< <= > >=`	Relational	Left
`== !=`	Equality	Left
`::`	List concatenation	Right
`=`	Assignment	Right

3.1 Primary Expressions

Primary expressions are identifiers, constants, or expressions in parentheses.

primary-expression:
    identifier
    literal
    (expression)

An identifier is a primary expression, with its type specified by its declaration.

A literal constant is a primary expression.

An expression inside parentheses is a primary expression.

3.2 Function calls

Expressions involving function calls associate from left to right. The following is the syntax used:

fcall-expression:
    expression(argument-list-opt)

argument-list-opt:
    nothing
    argument-list

argument-list:
    expression
    argument-list , expression

The function call will return the type specified by the function declaration. In order to be correctly evaluated the function has to be in scope.

3.3 Node operators

There are three unary node operators, `visit, unvisit, isVisited`. The are all right-associative. The first two operators mark a node as having been visited or unvisited, respectively, while the last returns a Boolean value indicating the visitation status of the node.

visit-expression:
    `visit` *node-identifier*

unvisit-expression
    `unvisit` *node-identifier*

isVisited-expression:
`isVisited` *node-identifier*

3.4 Negation Operator

The negation operator returns the opposite boolean value of its argument. The right hand side must evaluate to a Boolean type.

negation-expression:
! expression

3.5 Multiplicative Operators

The multiplicative operators * and / group left-to-right.

multiplicative-expression:
*expression * expression*
expression / expression

The operands of * and / must have number type.

Operations are carried on the operands and the result value is returned.

The binary * operator denotes multiplication of the left and right operands.

The binary / operator denotes division. The left operand is divided by the right operand. If the right operand is 0, the value is undefined and an exception will be caught.

3.6 Additive Operator

The additive operators + and - group left-to-right.

additive-expression:
expression + expression
expression - expression

The operands of + and – should have number type.

Operations are carried on the operands and the result value is returned.

The binary + operator denotes adding the two operands and getting the sum.

The binary – operator denotes subtracting the right operand from the left operand and getting the difference between the two.

3.7 Relational Operator

The relational operators group left-to-right.

relational-expression:
expression < expression
expression > expression
expression <= expression
expression >= expression

The operands must have `number` type here. The return value also has type `number.` For all of the operators < (less than), > (greater than), <= (less than or equal to), >= (greater than or equal to), the return value is set to be 1 if the specified relation is true, and it's set to be 0 if the specified relation is false.

3.8 Equality Operator

The equality operators group left-to-right.

equality-expression:
expression == expression
expression != expression

The operands must have `number` type here. The return value also has type `number`. For the operators == (equal to) and != (not equal to), the return value is set to 1 if the specified relation is true, and it's set to 0 if the specified relation is false.

In addition, equality operators have lower precedence than relational operators.

3.9 Logical AND Operator

The logical AND operators group left-to-right.

logical-and-expression:
expression && expression

The operands and the result should have `number` type.

The evaluation of the expressions is also left-to-right. The left operand is firstly evaluated. If its value is 0, then the result value is set to 0. If its value is 1, then the right operand is evaluated. If the value of the right operand is also 1, then the result value is set to 1. If the value of the right operand is 0, then the result value is set to 0.

3.10 Logical OR Operator

The logical OR operators group left-to-right.

logical-or-expression:
expression || *expression*

The operands and the result should have `number` type.

The evaluation of the expressions is also left-to-right. The left operand is firstly evaluated. If its value is 1, then the result value is set to 1. If its value is 0, then the right operand is evaluated. If the value of the right operand is also 0, then the result value is set to 0. If the value of the right operand is 1, then the result value is set to 1.

3.11 Assignment Expressions

The assignment operators group right-to-left.

assignment-expression:
identifier = *expression*

Assignments can be made to identifiers of types `list`, `number`, `boolean`, or `node`. The expression on the right-hand side must evaluate to the same type as the identifier on the left. Besides, the value of the assignment expression is the value of the left operand after the assignment process.

3.12 List Concatenation Operator

Additional nodes may be concatenated onto the front of an existing list using the :: operator.

list-concatenation-expression:
node-identifier :: list-identifier

Example:

```
list l = a :: l; // a is a node; l is a list
```

3.13 List Access Operators

The first node in a list can be accessed with the `head` operator. The expression returns a node type.

head-expression:
`head` *list-identifier*

Example:

```
node h = head l; // h is a node; l is a list
```

The tail of a list can be accessed with the tail operator. The expression returns a list type.

tail-expression:
`tail` *list-identifier*

Example:
```
list t = tail l; // t and l are lists
```

4 Declarations

A variable should always be declared first before it can be used. Declarations have the form:

declaration:
type-specifier declarator ;

type-specifier:
`node`
`graph`
`list`
`number`
`boolean`

A variable name cannot be declared twice. It is also illegal to declare multiple variables in the same line.

The declaration contains exactly one type specifier and one declarator. Each declarator contains exactly one identifier. Each declarator is taken to be an assertion that when a construction of the same form as the declarator appears in an expression, it yields an object of the indicated type.

4.1 Node

Node objects are created automatically in the creation of a new graph, without explicit declaration (see 4.2). They can also be declared individually. A node identifier is implicitly a reference to a node, and multiple references may point to the same node.

Node-declaration:
`node` *identifier*
`node` *identifier = node-identifier*

Example:
```
node A; // declare node individually
```

Nodes can be initialized in the same line in which they are assigned; for instance, by setting the identifier to point to the same node as another identifier.

Example:

```
node x = lastNode; // lastNode is already in existence
```

4.2 Graph

In our compiler, a graph declaration is actually an initialization of the graph at the same time. Each program has one or zero graphs; because of this, the type-specifier `graph` is not followed by an identifier in the declaration/initialization. To define a `graph`, one or more nodes must be simultaneously defined, as follows. If more than one `node` is defined, then edge definitions are required between each sequential `node` in the declaration.

A graph can be declared as follows:

graph-declaration :
graph [constructor-declarations];

constructor-declarations:
constructor-declaration
constructor-declaration, constructor-declarations

constructor-declaration:
node-id
node-id edge-definitions

edge-definitions:
edge-definition
edge-definitions edge-definition

edge-definition:
>> *node-identifier*
<< *node-identifier*
<> *node-identifier*
>> *literal node-identifier*
<< *literal node-identifier*
<>*literal node-identifier*

Example:

```
graph [A <<3 B];
graph [A <>2 B >>3 C <>4 D <>5 B, C <>1 E];
// A, B, C, D, E are nodes
```

4.3 List

List are like arrays. They are assemblies of Nodes. Lists are declared with the keyword `list`:

List-declaration:
`list` *identifier*
`list` *identifier* = *[list]*

list:
<nothing>
node-identifiers

node-identifiers:
node-identifier
node-identifiers, node-identifier

Example:

```
list l = [A, B, C, D];
```

5 Statements

In GRAPL, statements are of five different types:

- Expression
- Compound
- Conditional
- Iteration
- Return

5.1 Expression

An expression statement has the form:

Expression-statement:
expression ;

It is important to note that the expression is not optional, so an instruction consisting of a single semicolon is not considered correct.

5.2 Compound Statements

Compound statements (blocks) are used to group a set of different type of one or more statements. They follow the form:

compound-statement:
{ statement-list }

statement-list
statement

statement statement-list

Each of the statements in this compound-statement will be executed sequentially starting from left to right.

5.3 Conditional Statements

Selection statements are used to select which block of statements to execute based on the evaluation of a controlling expression. They follow the following form:

conditional-statement:
if (expression) then { statements } else { statements }

The `else` clause in the if-then-else statement is not optional and cannot be omitted; however, an empty statement may follow the `else`.

Example1:

```
if (node1 == node2) then
{
print(node1);
}
else { }
```

Example2:

```
if (length(list1) < length(list2)) then
     print(list2);
else
{
node1 :: list1;
print(list1);
}
```

5.4 Iteration

GRAPL has a single iteration statement to iterate through the nodes of a given graph: the `foreach` statement. This statement allows a user to specify an instruction set to execute for each of the nodes on the other side of an edge with a weight that satisfies certain conditions (the predicate) inside a graph. The general form of the `foreach` loop is:

iteration-statement:
`foreach` *node-control { statements }*

node-control:
qualifiers_opt id fromTo id withStmt

qualifiers_opt:

```
visited
unvisited
```

fromTo:

```
from
to
```

withStmt:

`with` *(predicate)*

predicate:

binary-operator expression

Example:

```
forEach visited leaf from root with (<= 1000)
{
/*do something*/
}
```

5.5 Return Statement

The return statement returns a function's value to its caller by means of one of the following forms:

return-statement:

`return ;`

`return` *expression ;*

The expression must evaluate to the correct return type for the function.

6 Built-in Functions

GRAPL has several built-in functions, including `length` and `print`. The `length`() function operates on a list and returns the number of nodes in a list. The `print`() function prints out a useful representation of its argument (a literal, node, list, etc.).

7 Scope

The scope rules in GRAPL follow these guidelines. An identifier is available after it has been declared. Additionally, an identifier that was used in a graph initialization need not be initialized. Every identifier is available from the point where it was used in the graph initialization until the end of the statement group enclosing it. Subsequent nested blocks have access to identifiers defined by any parent block. A variable with the same name as in a parent block cannot be declared.

A function identifier is available at any point after the left bracket of the function definition starts. After a function is declared, the identifier is locked and cannot be used to define a variable.