Compilers-II

White Paper
GraphIT

भारतीय प्रौद्योगिकी संस्थान हैदराबाद
Indian Institute of Technology Hyderabad

White Paper for *GraphIT*

Produced by:

Prabhath Chellingi - CS20BTECH11038

Venkata Datta Sri Harsha Kusampudi - CS20BTECH11028

Chitneedi Geetha Sowmya - CS20BTECH11011

Danda Sai Pravallika - CS20BTECH11013

Kodavanti Rama Sravanth - CS20BTECH11027

Mannam Sarandeep - CS20BTECH11030

Chinasani Asish Sashank Reddy - CS20BTECH11010

Instructor:
Ramakrishna Upadrasta

Happy Programming!

Contents

1 Introduction

1.1 Brief Description

This Language contains graphs as an inbuilt data type with easy initialisation and operation. It can be dynamically modified and worked upon. We can add some more features and functions. It also supports lists and complex value data types. Graphs mainly constitute weighted type, where every edge will have a weight.

1.2 Motivation behind the Design

The main idea behind this language is to provide easy experimentation on graphs with different algorithms and operations. We strive to produce more efficient and easy going algorithms with this language, where it supports easy addition of new operations to graph.

1.3 Design Goals

- This language comes with a different style of initialisation for graphs (built-in graph datatype), which makes the user visualize things easily and also perform operations such as finding shortest paths between nodes etc in a single line of code.
- The language provides the complex data types, which eases the user to code using complex numbers ($\mathbb{C}$) rather than limiting to reals ($\mathbb{R}$)

2 Basic Syntax

2.1 Whitespaces

White spaces contains the space for tabs, blanks and newlines. These characters are ignored except when they are used to separate other tokens. Blanks and tab spaces are not ignored in strings.

2.2 Comments

There are two types of comment representations:

- Single line comments are started with "#"

```
# This a single-line comment
```

- Multi line comments are enclosed within "#!" and "!#".

```
#! This is a
multi line comment !#
```

Note: All the information enclosed in the comments are ignored by the compiler during compilation

2.3 Reserved Keywords

These are the reserved keywords which cannot be used as identifiers.

int	float	if	else	break
for	while	graph	switch	case
bool	str	void	return	continue
struct	const	Alias	true	false
AND	OR	XOR	struct	enum
i				

Note: All the other data types discussed in Section 3 are also reserved keywords

2.4 Identifiers

- Identifiers must start with a letter or an underscore (**A-Z,a-z,_**).
- The remaining characters must be letters or digits or underscores (**A-Z,a-z,_,0-9**)

Note:

- The identifiers are case-sensitive in GraphIt. Hence **abc, Abc** are two different identifiers
- Identifiers cannot be one among the keywords listed above

2.5 Punctuators

All the statements should end with a semi-colon (;)

```
Statement1; # Correct
Statement2  # Gives an error if it is not ended with a semi-colon (;)
```

3 Data types

Data type is a value, set to a variable. It tells the compiler how the programmer intends to use the data.

3.1 Fundamental data types

3.1.1 Integer

Declaration keyword: *int*

It is used store the integers ranging from -2^{63} to $2^{63}-1$

Initialization

```
....
int a = 10;
int b = -20;
....
```

3.1.2 Unsigned Integer

Declaration keyword: *uint*

It is used to store non-negative integers ranging from 0 to $2^{64}-1$

Initialization

```
....
uint a = 10;
....
```

3.1.3 Floating Point

Declaration keyword: *float*

It is used to store floating point numbers - decimals and exponentials - with size of 8 bytes.

```
Initialization
....
float a = 164.74534;
float b = 45E12 #45E12 is equal to 45x10^12
....
```

3.1.4 Boolean

Declaration keyword: ***bool***

It takes only two values True or False.

```
Initialization
....
bool a = True;
....
```

3.1.5 Character

Declaration keyword: ***char***

It stores the all characters and letters. Only a single character or letter is stored and initialized using (‘ ’).

```
Initialization
....
char c = ‘a’;
....
```

3.2 Derived data types

3.2.1 Array

Declaration keyword: [...] (after the variable name with length of array)

It is a group of similar data types. They are stored in a contiguous memory location. Need to give the size of array(or group) while initializing.

```
Initialization
....
int codes[10] = {23, 45, 6, 5, 4, 3, 2, 1, 54, 63};
char labels[] = {‘a’, ‘b’, ‘c’, ‘x’, ‘y’};
....
```

3.2.2 Pointer

Declaration keyword: $*$ (before the variable name)

It is variable where memory address to object is stored rather than it's value.

Initialization
```
....
char *s = 'q';
....
```

3.2.3 string

Declaration keyword: *str*

Internally, it is sequence of memory location of char datatype. The variable contains the first element memory address. The size is fixed at the point initialization.

Initialization
```
....
str name1 = 'Krishna';
str name2 = "Hello";
....
```

3.2.4 Graph

Declaration keyword: *graph*

This is main data type of this language. This stores the graph data structure, where node names use str data type and weights use float datatype. It is a dynamically operated data type.

Initialization
```
....
# Here the graph is represented as adjacency list
graph G1 = ["A":"B"(10),"C"(50),"D"(24.6) | "B":"C"(15)];
....
```

3.2.5 Complex type

Declaration keyword: *compx*

This datatype stores complex numbers, where the real and imaginary part use float data type

Initialization

```
....
compx x = 10.2 + i 25.6
....
```

3.3 User definable data types

3.3.1 Structure

Declaration keyword: ***struct***

It is used to create user defined data type. It is basically used to group the required fundamental data types using different variable names.

Initialization

```
....
# Here Name and Number are stored into single structure

struct Contact
{
str Name;
int Number;
};
....
```

3.3.2 Enumeration

Declaration keyword: ***enum***

It is a user defined datatype, where each string value is assigned to consecutive integer number, where the start integer can be given by user.

Initialization

```
....
# Here Aerial will be given the default integer value 0, where others are given the integers
consecutively.(Aerial = 0, Aquatic = 1, Terrestrial = 2, Amphibian = 3)

enum classify {Aerial, Aquatic, Terrestrial, Amphibian}
....
```

4 Operators

4.1 Assignment Operators

Assignment operations copy a graph's reference to a variable, assign a value to a graph node's value, or store primitive values in variables. The assignment operator in the GraphIt is =. It is a right-associative binary operator. When an assignment is made, the expression's value the left value, is assigned to the right value, which is returned with its new value.
Assignment might take some of these types, for instance:

```
# where a is declared as int a;
a = 4;
#! where b is declared as graph b; and c is previously
declared and defined graph !#
b = c;
```

4.2 Arithmetic Operators / Graph Operators

In this part, we outline the GraphIt language's built-in operators and clarify what an expression is.

Standard arithmetic operations including addition (+), subtraction (−), multiplication (∗), and division (/), as well as modular division (*mod*), and negation ( ), are all supported by GraphIt. Using primitive types simplifies the use of these operators. With the bool type, arithmetic operations are invalid. Only the addition and subtraction operators work with two char operands. Here are some examples of how to use primitives and arithmetic operators.

```
x = 5 + 3; # where x is of type int
y = 10.23 + 37.332; # where y is of type float
z = 'a' + ('c' - 'a'); # where z is of type char
```

To get the remainder by dividing its two operands, you use the modulus operator mod. Only two integer values may be used with the mod operator.

```
x = 5 mod 3;
```

You use the negation operator on a float or int type

```
x = -4;
```

The main Operator used on the Graph is @. The syntax of the operator is:

```
Graph g;
g  @ ("a", "b")
```

It calculates the shortest distance from one node to the other.

4.3 Comparison Operators

The comparison operators are used to determine the relationship between two operands, such as whether they are equal, whether one is larger than the other, whether one is smaller, and so forth. Any of the comparison operators will produce a Boolean value of true or false when used. All comparison operators are left-associative binary operators. The following definition of a comparison in the context of Graphs is provided:

Operator	Primitive Types Definition	Tree Type Definition
>	Greater than	LHS #nodes > RHS #nodes
>=	Greater than or equals	LHS #nodes >= RHS #nodes
<	Less than	LHS #nodes < RHS #nodes
<=	Less than or equals	LHS #nodes <= RHS #nodes
==	Equal to	LHS Graph Structure and data equals to RHS Graph Structure and Data
!=	Not equal to	LHS Graph Structure and data is not equal to RHS Graph Structure and Data

4.4 Logical Operators

A pair of operands' truth value is tested by logical operators. The left associative binary operators && (logical and) and || (logical or) are the two logical operators. They require two Boolean operands and return a Boolean value.
The unary operator ! can be found on the operand's left side. Both the operand's type and the return type must be of the Boolean data type.

4.5 Operator Precedence

The following is a list of expressions, presented in order of highest precedence first. Sometimes two or more operators have equal precedence; all those operators are applied from left to right.

```
()
@
!
* / mod
+ -
> < >= <=
== !=
&&
||
=
```

,

5 Declarations

Declarations are the statements that describes an identifier such as naming a variable or function with their data type . Each one variables , functions used in the program must be declared before . Sample program for understanding declarations in GraphIT .

```
int val(int a)
{
    return a-7;
}
int main(){
    int x = 0;
    int y = 2;
    int z = val(y);
}
```

5.1 Declaration of variables :

Every variable which was used in the program must be declared before .Here declaration of the variable mean denoting the data type of the variable and the name of the variable . The declaration should be in a statement with a semicolon . Examples for declaring a variable is as follows

- *int* x;
- *int* r=10;
- *uint* p;
- *bool* q;
- *char* ch='a';
- *str* s = "bheem";
- *list* L;
- *graph* g;
- *compx* c;

5.2 Scope of a declaration :

1. Local variables can only be used within a function/scope i.e. variables which are created by a declaration are valid within the function/scope where the declaration which created the variable occurs . In a given function/scope there should not be more than one identifier with same name .

2. Global variables are the one which are declared in global scope . They can be used anywhere in the program after their declaration occurs .These global variables should have unique identifiers . Below are some sample codes .

```
int funct(int a , int b){
    return a+b;
}
int main(){
    int x = 2 ;
    int y = 4 ;
    int z = funct(x,y);
    int x ; #! This line gives an error as identifier x has been
    declared before cannot be declared again !#
```

- In the above program variable x has been declared twice in a local scope which is invalid .

```
bool function(int a , int b , int c){
    if (a+b == c) return true;
    else return false;
}
int x ; #Global variable declaration .
int main(){
    int a = 0 ;
    int b = 2 ;
    int c = 2;
    bool p = function(a,b,c);
    int x = 3; #! This line gives an error as x has been before
    declared as global variable !#
}
```

- The above program gives an error as identifier x has been declared as global variable before & It can't be declared again as local variable in any other scope .

5.3 Alias:

We can use Alias keyword to change the name of the data type . This will be mainly used in user defined data types .The following sample code is an illustartion for this alias .

```
struct identifier1 {
   int val;
   str name;
   list p;
   graph g;
};
Alias struct identifier1 identifier2;
#!Now after this we can use the name of the data type
struct identifier1 as identifier2!#
```

5.4 Const:

The keyword const tells that the variable value is a constant .Now variable declared in this form can't be modified in the entire program .For example

```
int main(){
    const str ="Lakshman";
    str="Ram"; #This line gives an error
}
```

5.5 Declaration of a function :

A function which are used in the program should be declared / defined before calling it . Declaration of a function is just giving the return type of the function & it's arguments data types Where as definition of a function is what function exactly do . The below code is an illustration of function declaration .

```
data_type_of_function name_of_function ( data_type arg1 , data_type arg2 ,
..... )
```

5.6 Intializations:

Initially when variables are declared but not initialized they were given default value i.e. 0 for integers data type , "" (empty string) for str data type , NULL for pointer data type . Variables are initialized by assigning the identifiers to a value . Or we can assign a value of initialized variable to the variable that we are initializing now . Below codes are illustartion for this

```
int main(){
    str s;      # s is initialized with ""
    int * p;    # p is initialized with NULL
    int x;      # x is initialized with 0
    graph g ;   # graph is initialized as an empty graph i.e. with no nodes.
    int a = 2; # Variable 'a' is initialized with value 2.
    int b = a;  #!  Variable 'b' is initialized with value of 'a' which
    was already initialized to 2 . !#
}
```

6 Statements

Most of these statements provided by Graphit will be similar to those of C++.Similar to C++ every statement ends with semicolon(;).Graphit supports the following statements.

- labeled statements
- declaration statements
- expression statements
- conditional statements
- jump statements
- iteration statements

6.1 Conditional Statements

6.1.1 if statement

If there is only one statement it can be written as

```
if(boolean_expression)          #simple if statement
    statement;
```

If the expression evaluates to true then the statement executes. For multiple statements the code is enclosed within a pair of curly braces and when the expression evaluates to true the whole code within the curly braces executes.

6.1.2 if-else statement

An else statement can be included with the if statement and the code within the else block executes if the expression in the if statement evaluates to false. We can also have else if clauses in between the if and else statements when required. Nested if-else statements(if-else within if-else statements) are also allowed.

```
if(boolean_expression1)           # if-else
{
 if(boolean_expression2)          # nested if-else
     statement1;
 else if(boolean_expression3)     #else if
     statement2;
 else
     statement3;
}
else
{
     statement4;
}
```

6.2 Jump Statements

when you encounter these statements in a program you will be jumped from one line of execution to an other line of execution.

6.2.1 break statement

The **'break'** keyword is used to exit from a loop without executing the further iterations and code below it in the current iteration. The updated line of execution is the immediate line after the loop containing the break statement.

```
while(i>0)
{
 x=foo(i);
 if(x==3)
   x=x+1;
 else
   break;  # break from loop without executing remaining iterations
 i--;
}
```

6.2.2 continue statement

The **'continue'** keyword is used to end the current iteration of the loop and program control is passed from continue statement to end of loop body.

```
while(i>0)
{
 x=foo(i);
 if(x==3)
   x=x+1;
 else
   continue; #when this statement is encountered
             #the statement i--; is not executed
 i--;
}
```

6.2.3 return statement

The **'return'** keyword is used to exit from a function and while doing so we can return a value(can be a fundamental data type (or) derived data type (or) user defined data type) depending upon the return type of the function. If the return type is void then it will return nothing.

```
int foo(int&x,int&y)
{
  return x+y;
}
```

6.3 Iteration Statements

Loops are used to execute same set of statements any number of times according to our requirement.

6.3.1 for loop

Format:

```
for(initialization;condition;expression)
{
   statement1;
   statement2;
   .
   .
   .
}
```

It has two statements followed by an expression. First statement is for initialization, second is a condition statement and expression is usually used for increment or decrement the variables. If the conditional statement evaluates to true then the code enclosed with in the braces executes and stops when condition becomes false.

6.3.2 while loop

Format:

```
while(conditional expression)
{
   statement1;
   statement2;
   .
   .
   .
}
```

If the condition evaluates to true,the code inside the while loop executes and it executes until the condition becomes false.

7 Built-in functions

Graphit provides many built-in functions, But as this langauge mainly focuses on graphs, we have some functions which can only be used by graph datatype members.

7.1 Built-in functions for any datatype

7.1.1 print function

This function takes 1 or more arguments of any datatype and prints them. For graphs, it will print it in its adjacency list form.

```
int n = 2;
graph g = ["a":"b"(10),"c"(20)];

print("Hello",n); # prints Hello2
print(g); # prints ["a":"b"(10),"c"(20)]
```

7.1.2 scan function

This function takes 1 or more arguments of any datatype and scan the input (which are separated by 1 or more white spaces) into corresponding variables. If input format doesn't match with the datatype format, then we get an error. For graphs, any representation that is of correct format is considered valid

```
int n;
graph g;
scan(n,g); # scans an integer, graph from console.
```

7.2 Built-in functions for Graph datatype

7.2.1 get_adjacent_nodes function

This function takes a node (string) as argument and returns the array of adjacent nodes.

```
graph g = ["a":"b"(10),"c"(20)];
str adj1[] = g.get_adjacent_nodes("a"); # adj1 = {"b"};
str adj2[] = g.get_adjacent_nodes("b"); # adj2 = {"a","c"};
```

7.2.2 get_edge_weight function

This function takes two nodes as argument and returns the edge weight from first node to second node.

```
graph g = ["a":"b"(10),"c"(20)];
print(g.get_edge_weight("a","b")); # prints 10
```

7.2.3 set_edge_weight function

This function takes two nodes and a number as argument and sets the edge weight from first node to second node with given value.

```
graph g = ["a":"b"(10),"c"(20)];
g.set_edge_weight("a","b",20);
print(g.get_edge_weight("a","b")); # prints 20
```

7.2.4 get_nodes function

This function returns array of all the nodes in the given graph.

```
graph g = ["a":"b"(10),"c"(20)];
str nodes[] = g.get_nodes(); # nodes = {"a","b","c"};
```

7.2.5 get_components_count function

This function returns the number of components present in the given graph

```
graph g = ["A":"B"(10),"C"(50),"D"(24.6)];
int s = g.get_components_count(); # s = 1 here
```

8 Language design

8.1 Key Tokens

INT | UNSIGNEDINT | FLOAT | GRAPH | BOOLEAN | CHAR | STRING | VOID | STRUCT | COMPLEX | ENUM | CONSTANT | ALIAS

CASE | DEFAULT | IF | ELSE | SWITCH | WHILE | FOR | CONTINUE | BREAK | RETURN

LOGICAL_AND_OP | LOGICAL_OR_OP | ADD_OP | SUB_OP | MULT_OP | DIV_OP | MOD_OP | LEFT_OP | RIGHT_OP | LT_OP | GT_OP | EQ_OP | NE_OP | PTR_OP | DIST_OP

8.2 Grammar

Basic Grammar for the **Graph** data-type

```
graph
        : '[' adjacency_list_expression ']'
        ;

adjacency_list_expression
        :  each_node_adjacency_list
        |  each_node_adjacency_list '|' adjacency_list_expression
        ;
```

```
each_node_adjacency_list
        :  STRING_LITERAL ':' node_list
        ;

node_list
        : node_list ',' STRING_LITERAL '(' CONSTANT ')'
        | STRING_LITERAL '(' CONSTANT ')'
        ;

shortest_distance_expression
        : IDENTIFIER DIST_OP '(' STRING_LITERAL ',' STRING_LITERAL ')'
```

9 Sample Codes

9.1 Code I

```
int main()
{

    graph g = ["a":"b"(10),"c"(12) |
                "b": "a"(11),"d"(9) |
                "c": "b"(16), "a"(5) |
                "d": "a"(4)];
    # Gives shortest path between the nodes a and c in the graph
    int short_dist = g @ ("a","c");
}
```

9.2 Code II

```
int main()
{
    compx c1 = 1 + i 2;
    compx c2 = 1 + i 3;

    c3 = c1 * c2;
}
```