# GAL Reference Manual

Anton: ain2108, Donovan: dc3095, Macrina: mml2204, Andrew: af2849

July 21, 2016

## 1 Introduction

Graph Application Language or GAL is designed to make definitions, operations and manipulations on graphs easier. A lot of real world problems can be modeled by graphs and complex graph algorithms can be applied to solve them. Although graph algorithms are easily coded in several programming languages, a language with special data structures, syntax and semantics to allow the user to easily interact with graphs and their algorithms is desirable.
The syntax of GAL is similar to C. It has some special syntactic and syntax elements to deal more easily with graph applications. The compiler will be implemented in OCAML. It will compile to LLVM.

## 2 Lexical Conventions

Six type of tokens exist in GAL: identifiers, keywords, constants, strings, expression operators and other forms of separators. Common keystrokes such as blanks, tabs, newlines and comments are ignored and used to separate tokens. At least one of these common keystrokes are required to separate adjacent tokens.

### 2.1 Comments

The characters /*introduce a comment which terminates with the characters */. There are no single line comments (such as // in C).

### 2.2 Identifiers (Names)

An identifier is a sequence of letters and digits; the first character must be alphabetic. The underscore counts as alphabetic. Upper and lower case letters are considered different. The maximum number of characters in an identifier is seven.

### 2.3 Keywords

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

1. int

2. list
3. string
4. break
5. continue
6. if
7. else
8. for
9. return
10. def
11. node
12. edge

### 2.4 String

A string is a sequence of ASCII characters surrounded by double quotes i.e. one set of double quotes " begins the string and another set " ends the string. For example, "GAL" represents a string. Individual characters of the string cannot be accessed. There are no escape characters within strings. All ASCII characters between the quotes are stored in the string with a \n string terminator character appended automatically to the end.

### 2.5 Constants

There are several kinds of constants, as follows:

#### 2.5.1 Integer Constants

It is a sequence of digits. It is taken to be in decimal.

#### 2.5.2 String Constants

It is a constant of the string type defined above.

## 3 More About Identifiers

The storage associated with all identifiers is local to the function the identifier is defined in. There are no global variables.The meaning of the value(s) stored in the identifier is determined by its type. The fundamental types of objects in GAL are integers and strings. GAL defines several derived data types namely list, node, edge and function.The fundamental as well as derived types are referred to as 'type' in the rest of this manual.

1. List: They comprise several objects of other types such as integers, strings, edges, nodes or even other lists. However, a list cannot contain functions. All objects in a list must be of the same type. For example, a list can contain all edges or all lists.
2. Node: It encodes all the information present in a graph vertex i.e. it contains the string names of the vertex, the set of all vertices reachable from the vertex it defines as well as the weights of corresponding outgoing edges from it. It contains string and integer datatypes. The vertices connected to it are represented by strings with their corresponding weights as integers. Floating point weights are not accepted.
3. Edge: It contains three elements namely two strings corresponding to the two vertices the edge connects and an integer representing its weight.
4. Function: It takes one or more input objects of node, edge, list, integer or string type and returns a single object of a given type, namely, node, edge, list or integer. Functions cannot take as input or return other functions. If a function is required to return multiple objects, we can combine them into one of our derived data types which the function is allowed to return.

Methods of constructing objects can be applied recursively. For example, we can have a list containing lists of lists.

## 4 Conversions

Operands cannot be converted automatically from one type to another by any operators.

## 5 Expressions denoted by *expression*

The two major considerations are precedence and associativity of expression operators. In the description of the expressions below, the subsections are ordered in decreasing level of precedence. The expressions in the same subsection have the same level of precedence. The operators which can act on the expressions in each subsection and their associativity has been described.

### 5.1 Primary Expressions denoted by *primary*

These include identifiers, strings, constants, nodes, edges, parenthesized expressions of any type, function calls and subscripts. Primary expression involving subscripts and function function calls are left associative.

#### 5.1.1 Identifiers denoted by *identifier*

It is the name given to any integer, string, list, node, edge or function according to the naming convention described earlier.

#### 5.1.2 Constant denoted by *constant*

These include integer denoted by *intConst* and string denoted by *strConst* constants.

### 5.1.3 Node denoted by *node*

A node is a primary expression of the form

$$|string : (integer, string)(integer, string)(integer, string).....(integer, string)| \quad (1)$$

The *string* and *integer* maybe constants and/or identifiers of string and integer types respectively.The first *string* denotes the vertex represented by the node and the (integer, string) pairs denote the weights and vertex names of the vertices connected to it by edges in the graph. An expression evaluating to an integer is not permitted in place of *integer* in equation (1). The order in which the (*integer*,*string*) pairs are arranged is random. The pairs need not be unique. For example, we can have 2 edges from node A to node B both having the same weight.However, every outgoing edge from a graph vertex has exactly one (*integer*,*string*) pair associated with it.

### 5.1.4 Edge denoted by *edge*

An edge is a primary expression of the form

$$|string, integer, string| \quad (2)$$

The *string* and *integer* maybe constants and/or identifiers of string and integer types respectively. The first *string* denotes the source vertex followed by the *integer* weight and the destination vertex representing an edge in the graph. An expression evaluating to an integer is not permitted in place of *integer* in equation (2).

Thus when our language encounters a |, it checks for the subsequent pattern and accordingly decides if an edge or node is being defined.

### 5.1.5 Parenthesized expressions

Any expression in GAL can be parenthesized. The format is

$$(expression) \quad (3)$$

. Parenthesis cannot be inserted or removed from within the node or edge definitions.

The type and value of the expression remains same - only the parenthesized expression of primary expression type.

### 5.1.6 Subscripts

The form is

$$identifier[constant] \quad (4)$$

The *identifier* must be of type list and the *constant* must be an integer greater than or equal to 0. Subscript expressions output the *constant*th element of the list. Lists are indexed from 0. In order for the expression to be valid, the *identifier* must have been previously defined having length at least one greater than the value of the constant.

### 5.1.7 Function calls

Functions previously defined may be called. GAL matches the return type (according to the expression the function is present in), the function name (represented by an identifier), the number of inputs and the input types with the previously defined function. Identifiers used in function names may not be used in other function names or as variable names except in the following case : An identifier may be used as a variable name local to a function other than the calling function and the function being called. The form of a primary-expression indicating a function call is

$$identifier(optionalexpressionlist) \tag{5}$$

The *optionalexpressionlist* denotes a comma separated set of inputs to the function and may be absent if the function is defined as containing no inputs. Any expression is acceptable as long as it evaluates to the required type as mentioned in the function definition. Thus nested function calls can exist. All inputs of the functions must be explicitly listed in the function call and none of the inputs can take on inferred values. Thus an identifier followed by open parenthesis matching the requirements above is a function. If it is previously undefined or does not meet the above requirements an error is returned. If it is not declared at any point in the program, a compile time error occurs at the point where the function is called. If the defined function has no input arguments, the called function must also not have any. All function parameters are passed by value i.e. changes to the input parameters within the function will not be reflected in the calling function unless the parameter is returned. GAL passes arguments by value.

## 5.2 Unary Operators

Our language has two unary operators - unary minus and logical negation. They are right associative.

### 5.2.1 Unary minus

The form is

$$-integer \tag{6}$$

The integer may be a constant or an identifier of integer type.

### 5.2.2 Logical negation

This expression is of the form

$$!integer \tag{7}$$

The integer may be an identifier or a constant of integer type. Its ouput is the integer 1 if the input integer is 0 and is 0 for any non-zero integer. GAL does not have the boolean variable type. 1 and 0 are represented as integers.

## 5.3 Multiplicative Binary Operators

The operators of this type are * and / They are left associative.

### 5.3.1 Binary Multiplication

This is of the form

$$expression * expression \tag{8}$$

Both the expressions in the above must evaluate to an integer type.

### 5.3.2 Binary Division

This is of the form

$$expression/expression \tag{9}$$

Both the expressions in the above must evaluate to an integer type. GAL does not have or require floating point numbers since its primary application is graphs. Hence binary division outputs the integer quotient of the two operands.

## 5.4 Additive Binary Operators

The operators of this type are + and - and are left associative.

### 5.4.1 Binary Addition

This is of the form

$$expression + expression \tag{10}$$

Both the expressions must evaluate to integers.

### 5.4.2 Binary Subtraction

This is of the form

$$expression - expression \tag{11}$$

Both the expressions must evaluate to integers.

## 5.5 Binary Graph Operators

The operators of this type are +. and -. They are left associative. Graphs are defined as a list of edges in GAL using the standard library function define Graph which takes a list of edges or list of nodes as input.

### 5.5.1 Graph Edge Addition

The expression is of the form

$$expression + .\ expression \tag{12}$$

Here both expressions must evaluate to a single edge, a list of edges or a single edge on one side and a list of edges on the other.

### 5.5.2 Graph Edge Subtraction

The expression is of the form

$$expression - .\ expression \tag{13}$$

Here both expressions must evaluate to a single edge, a list of edges or a single edge on one side and a list of edges on the other.

### 5.5.3 Graph Node Addition

The expression is of the form

$$expressionA + .\ expressionB \tag{14}$$

Here expressionA must evaluate to a list of edges (which is a graph in GAL) and expressionB must evaluate to a single node.

### 5.5.4 Graph Node Subtraction

The expression is of the form

$$expressionA - .\ expressionB \tag{15}$$

Here expressionA must evaluate to a list of edges (which is a graph in GAL) and expressionB must evaluate to a single node.

When graph node addition or subtraction is encountered, all the edges associated with that node are removed from the graph denoted by expressionA.

## 5.6 Binary Relational Operators

These are left associative. Each expression of this type evaluates to integer 1 if true and integer 0 is false. The expressions on both sides of the operator must evaluate to integers. The operators are of the following types:

### 5.6.1 Less than

Expressions of the form

$$expression < expression \tag{16}$$

### 5.6.2 Greater than

Expressions of the form

$$expression > expression \tag{17}$$

### 5.6.3 Less than equal to

Expressions of the form

$$expression <= expression \tag{18}$$

### 5.6.4 Greater than equal to

Expressions of the form

$$expression >= expression \tag{19}$$

## 5.7 Equality Operator

These are left associative. Each expression of this type evaluates to integer 1 if true and integer 0 is false. The expressions on both sides of the operator must evaluate to integers. Its form is

$$expression == expression \tag{20}$$

## 5.8 Graph Equality Operator

This is left associative. Each expression of this type evaluates to integer 1 if true and integer 0 is false. Its form is

$$expression == .\ expression \tag{21}$$

Here both expressions must evaluate to a single edge, a list of edges or a single edge on one side and a list of edges on the other. In the later case the result of the expression is clearly 0. Alternatively, both expressions may evaluate to nodes.

## 5.9 AND Operator

It is of the form

$$expression \&\& expression \tag{22}$$

It is valid only if the left and right side expressions both evaluate to integers. First the left hand expression is evaluated. If it returns, a non-zero integer, then the right side is evaluated. If that too returns a non-zero integer the AND operator expression evaluates to integer 1. If the left side expression evaluates to integer 0, the right side expression is not evaluated and the AND operator expression evaluates to 0.

## 5.10 OR Operator

It is of the form

$$expression || expression \tag{23}$$

It is valid only if the left and right side expressions both evaluate to integers. First the left hand expression is evaluated. If it returns, a zero integer, then the right side is evaluated. If that returns a non-zero integer the OR operator expression evaluates to 1 but if it returns a 0 integer, the OR operator expression evaluates to integer 0. If the left side expression evaluates to integer 1, the right side expression is not evaluated and the OR operator expression evaluates to 1.

## 5.11 Assignment Operator

This is right associative and is of the form

$$expressionA = expressionB \tag{24}$$

*expressionA* must an identifier, a subscript expression, a parenthesized identifier or a parenthesized subscript expression. *expressionB* may be an expression of any type. *expressionA* must have been declared to have the same type as the value generated by *expressionB*. If *expressionA* is a subscript or parenthesized subscript expression, the Assignment Operator expression is valid only if the list in expressionA has all the elements of the same type as expressionB.

# 6 Declarations

There are two types of declarations

## 6.1 Function Declaration

All functions must be declared at the start of the program. A function cannot be declared within another function. A function cannot be defined or called unless it is declared. A function declaration has the following form

```
def type-specifier identifier(type1 input1, type2 input2, .... typen inputn);
```

The above is a statement. The significance of ';' will be discussed later. The *typek inputk* for k = 1 to n denotes the kth input to the function. A function may also have no inputs. *typek* is one of our types described earlier while input is an identifier of that type. The identifier(name) used in the function declaration, function definition and function call (function call can contain a more complex expression as well) may be different as long as the type is same.*type-specifier* denotes the return type of the function. A function must have exactly one definition and exactly one declaration. The uniqueness requirements of the identifier have been described in Section 6.1.8.

## 6.2 Variable Declaration

All variables used in a function must be declared at the start of the function. They may be (re)defined at any point within the function in which they are declared. The value of the variable at any point is the most recent one as defined within the function irrespective of the program block structure. The scope of the variable is limited to the function in which it is declared. Variables cannot be declared or defined outside functions. variables can be of type integer, string, list, node or edge. The type of every identifier within a function does not change throughout the function. No two identifiers within the same function can have the same name. Storage space is allotted to all variables at the time of declaration. If the contents of any declared variable are printed before definition, a random value is printed.

1. Variables of type integer, string, node and edge and declared as follows:

   ```
   def declared-type identifier;
   ```

   *declared-type* assigns a type from among integer, string, node or edge to the identifier. For these types, space is not allocated at the time of the declaration but only at the time of definition.

2. Variables of the list type must be provided with a length at the time of declaration. This is done as follows:

   ```
   def list (identifier,type,integer);
   ```

   The identifier is declared to be of type list and space is allotted to it according to the integer constant or identifier (third term in the bracket) mentioned. The length of the list must be provided at the time of the declaration. Every element of the list must be of the type according to the second term in the bracket.

# 7 Definitions

These are of two types:

## 7.1 Function Definition

This takes the following form:

```
type-specifier identifier(type1 input1, type2 input2 ... typen inputn){
/*first define (and optionally declare) the variables*/

/*set of simple and compound statements*/

/*return (return_value);*/
}
```

The first line is identical to the function definition except for the def keyword. The return statement can occur anywhere within the function provided it is the last statement within the function according to the control flow. Statements occurring after return in the control flow will cause errors. A function that is declared and called but not defined will produce a compile-time error. Any function can be defined anywhere in the program provided it is declared at the beginning.

## 7.2 Variable Definition

A variable definition is simply an assignment. Note that if a definition is an assignment of the form in (24) and if *expressionA* is a subscript or parenthesized subscript where the other elements of the list have not been defined but the list has been declared, the other elements of the list still contain random (garbage) values.

1. integer, string, node and edge definitions consist of

   ```
   identifier = expression;
   ```

   The left hand side identifier must be of the same type that the right hand side expression evaluates to.

2. list definition A single element of a list may be defined as follows:

   ```
   subscript-expression = expression;
   ```

   with matching types. However, an entire list is defined as follows:

   ```
   list-identifier = [element1;element2;....elementn];
   ```

   All the elements must be of the same type according to the type of the identifier on the left side as specified in the list declaration.

Variable declaration and definition must be done in separate statements.

# 8 Statements denoted by *statement*

Execution of statements are carried out in order unless specified otherwise. There are several types of statements:

## 8.1 Expression Statement

Most statements are expression statements which have the form

$$\textit{expression}; \tag{25}$$

Usually expression statements are assignment or function calls.

## 8.2 Compound statement

Several statements *statement* of any statement type may be enclosed in a block beginning and ending with parentheses as follows:

```
{statment-list};
```

The entire block (along with the parentheses) is called a compound statement. Statment-list can comprise a single statement (including the null statement) or a set of statements of any statement type. Thus compound statements may be nested.

## 8.3 Conditional Statement

The form is:

```
if (expression)
compound statement
else
compound statement
```

This entire form is called a conditional statement. Every *if* must be followed by an expression and then a compound statement. This in turn must be followed by an *else* which must be followed by a compound statement (even if it is the null statement). Thus there is no else ambiguity. The *expression* must evaluate to an integer. If it evaluates to 1, the compound statement immediately after if is evaluated and the block following else is not executed and the flow proceeds to the next statement(following the conditional statement). If the expression following *it* evaluates to 0, the else block is executed. Thus conditional statements may be nested.

## 8.4 For Loop Statement

The statement has the form

```
for(expression1;expression2;expression3)
compound statement
```

None of the expression statements can be omitted. Identical to C, the first expression specifies the initialization of the loop, the second specifies a test made before each iteration such that the loop is exited when the expression becomes the integer 0; the third expression specifies an incrementation which is performed after each iteration.

### 8.5 Break Statement

The format is

```
break;
```

It causes termination of the smallest enclosing for loop. Control passes to the statement just outside this terminated for loop.

### 8.6 Continue Statement

The format is

```
continue;
```

It causes the control to pass to the next iteration of the smallest enclosing for loop. The for loop counter variable is incremented just as it would be if the control had reached the end of the for loop.

### 8.7 Return Statement

The *return* statement is a function return to the caller. Every function must have a return type. This return value may or may not be collected by the calling function depending on the statement containing the function call. The format is:

```
return (expression);
```

; In the above, *expression* must evaluate to the same type as that in the function definition it is present in.

### 8.8 Null Statement

This has the form

```
;
```

## 9 File Inclusion

Other GAL files containing user defined function declarations or definitions and standard library functions used in the current program must be included at the start of the program. The syntax for this is

```
# include "filename"
```

The *filename* must contain the path to the file. *filename* must be a string constant. If the file is not found an error is returned.

## 10 Built-In Functions

GAL has six built in functions:

## 10.1 Opening a File denoted by *fopen*

The format is:

```
special_identifier = fopen(filename);
```

This is a statement. Since printing to files is not allowed, fopen always opens the file in read mode. It returns a *special_identifier*. The rules for naming it are the same as the previously defined identifier. However, its type is a pointer to the file *filename*. *filename* must contain a constant or identifier of type string. No operations can be performed on this special identifier and it is only used in two places - as an output of the fopen function and as an input to the scan function.

## 10.2 Printing denoted by *print*

The format is:

```
print(identifier_1 , identifier_2 .... identifier_n)
```

This is a statement. *identifier_1* to *identifier_n* are the comma separated list of identifiers of any type which we want to print. *n* can take any value from 1 to a maximum of 10. Thus only upto 10 different strings can be printed with a single print statment. Each identifier is internally converted to a string type for printing. The *print* function automatically prints a single space between the value of every identifier. Every print statement automatically prints a newline on completion to the standard output.

## 10.3 Scanning denoted by *scan*

The format is

```
scan(special_identifier ,type ,delimiter);
```

This is a statement comprising a built-in function call. If the special_identifier does not exist an error is returned. The return type of *scan* is the same as the input *type* and must be captured by an identifier of the same type. Delimiter may be a space denoted by ' ' or a newline denoted by '\n'. Thus every scan statement must be preceeded by an fopen statement.

Printing and scanning are both I/O functions.

## 10.4 Length denoted by *len*

This computes the length of a list. Its format is

```
identifierA = len(identifierB)
```

Here, *identifierA* is of type integer returns the length of the list. For a non-zero value of identifierA, the maximum allowable list index (for the subscript expression) is clearly length - 1. *identifierB* is of type list.

### 10.5 Finding source vertex denoted by *src*

This computes the source vertex in an edge. Its format is

```
string = src(edge);
```

where string is an identifier of type string and edge is an identifier or expression of type edge.

### 10.6 Finding destination vertex denoted by *dest*

This computes the destination vertex in an edge. Its format is

```
string = dest(edge);
```

where string is an identifier of type string and edge is an identifier or expression of type edge.

No identifier can have the names print, scan, openfile, len, src or dest.

## 11 Standard Library Functions

The standard library functions included in GAL are *pop* and *append* on lists, and *create_graph_node* on a list of nodes.

1. Pop: input is a list and output is a new list and an element of the same type as those present in the list. The element in the 0th index position of the list is popped. If the input list is empty an error is returned.

2. Append: input is a list and an element of the same type as those present in the list. Output is a new list of length one greater than the input list with the element appended at the end.

3. create_graph_node outputs a list of edges. Its input is a list of nodes.

The definitions of the standard library functions are beyond the scope of this reference manual. Additional functions may be subsequently added to the standard library if deemed necessary.

## 12 Structure of the Program

Our program has the following structure:

1. The files to be included

2. The function declarations of all the functions used in the current program

3. The main function: Every GAL program must have a main function which returns an int. It is the only function which need not be declared in the function declaration step. It takes no input arguments. It must contain a

   ```
   return (1);
   ```

   at the end of its control flow. The absence of this return statement will generate an error. The definition of the main function is thus as follows:

```
int main(){
/*compound statements, loops etc*/
return (1);
}
```

Its optional definition is

```
def int main();
```

4. Other function definitions (only functions declared earlier can be defined). These definitions may occur anywhere in the program after the function declaration section, before or after the definition of the main function.

The main function is the entry point and its return statement is the exit point of every GAL program which successfully executes.

## 13 Sample Code

A sample function in GAL for a breadth first search with the root as the first element of the input list is shown below:

```
def list BFS(list list_in);/*The BFS function is defined*/

int main(){/*The main function; Note how the main function was
not explicitly declared*/
        /*variables are declared*/
    def edge e1;
    def edge e2;
    def edge e3;
    def list (new_list,edge,4);
    def list (list_op,edge,4);

    /*variables are defined*/
        e1 = |"A",2,"B"|;
        e2 = |"B",3,"C"|;
        e3 = |"C",1,"D"|;
        e4 = |"C",4,"E"|;
        new_list = [e1;e2;e3;e4];

        /*new_list is a list of edges so create_graph_node need not be
    called to create a graph consisting of a list of edges*/

    list_op = BFS(new_list);/*function call to BFS*/
    /*note how new_list and list_in which is used in the function
    declaration are different names but of same type*/

    return (1);/*return statement in main*/
    }/*main ends*/

def list BFS(list input_list){/*function definition. input_list and
list_in which was used in the function declation are different names
```

```
but of same type*/

        /*variables are declared*/
    def list (a,edge,0);
    def list (visited,edge,0);
        def list (root,edge,0);
    def list (queue,string,0);
    def string node1;
    def int j = 0;
    def int i = 0;
    def int k = 0;

    a = input_list;

        append(root,src(a[0]));
        queue = root;

    for(i=0;i<9999999;i=i+1){
      if (len(queue)!=0){
                node1 = pop(queue);
                append(visited,node1);

                for (j= 0; j < len(a); j = j+1){

                                        if (src(a[j]) == node1){
                        for ( k = 0; k < len(visited); k++){
                            if (dest(a[j]) != visited[k]){
                                append(queue,dest(a[j]));
                            }
                            else{
                            ;
                            }

                        }

                                        }
                                        else{
                    ;
                    }
                }
      }


          else{
        break;
      }
   }
```

```
        return (visited);
}
```