G(roup) A(bstraction) L(anguage) – GAL

By Walter Gress

Group Abstraction Language, or GAL, is an interpreted language with a parser built as a proof of concept and inspired by the requirements documentation I called SELENA. Its parser takes in GROUPS, and performs the following operations on them, ABSTRACTION, INTEGRATION/DEINTEGRATION, SORTING, and SEARCH. It is meant to be a robust yet simple language and was inspired by the simplicity, power, and utility of the AI-specific designed language LISP, LIST PROCESSING.

The most basic element in GAL is a GROUP. The most basic operation for a GROUP is an ABSTRACTION. There are also classes. Classes define the structure of a GROUP and inherit from GROUP which is the universal BASE class for all GROUPs, the simplest derived class from GROUP is LIST. All GROUP derived classes contain the operations SORT, SEARCH, INTEGRATE, DEINTEGRATE, and also the custom operation of FUNCTION. FUNCTIONs can be declared in inheritance to inherit from SORT, SEARCH, INTEGRATE, and DEINTEGRATE just like classes can inherit from GROUP base. FUNCTIONS can inherit from parent functions and GROUPS can inherit from parents.

In order for a group to be used in any way like a common variable it must be ABSTRACTED through the process of ABSTRACTION. Just as an integer variable comes from the GROUP Integer, a GROUP of class Integer must be ABSTRACTED as an Integer before a variable of type Integer can be used.

GROUPs may be constructed of lists of constants or variables. CONSTANTS are GROUPS of values (such as numbers, characters, etc. ), VARIABLES are GROUPS of values (such as variables). In the way that a GROUP of CONSTANTS must be ABSTRACTIONS and a GROUP of VARIABLES must be ABSTRACTIONS a GROUP of GROUPS must be ABSTRACTIONS.

GROUPS are preferred not to be lists but networks of type NODE. Some GROUPS could be flat GROUPS like the list of Integers, a semantic network, a neural network, etc.

Since custom FUNCTIONS are allowed in addition to the functions that must be implemented such as SORT, SEARCH, INTEGRATE, and DEINTEGRATE, VIRTUAL FUNCTIONS are allowed as well. These are functions that can be implemented in OUTER GROUPs or derived classes. FORCED or in C++ pure virtual, FUNCTIONS may be implemented as well to create INTERFACE GROUPS or what would be called in C++ abstract classes.

In GAL there are no arithmetic operators by language definition but they can be implemented on GROUPs as custom FUNCTIONS for ABSTRACTIONs. For example if I wanted to implement the ++ operation on variables of the ABSTRACTION [ Integer ] that had a variable of its type called x for example, I could create a function of that GROUP ABSTRACTION called ++ that could implement code to increment that variable.

There is a significant difference, and it should not be confused between GROUPS and ABSTRACTIONS of GROUPS. A GROUP is literally the collection of pathways established between various NODES in a mesh network. An ABSTRACTION on the other hand is a singular element that abstracts away from that GROUP. An ABSTRACTION Is primarily for the purpose of nesting GROUPS as NODES, instead so that GROUPS of concepts can be nested as NODES, not just CONSTANTS or VARIABLES. The ABSTRACTION of a GROUP allows us to take a GROUP like “dog”, nest it in a group like “canine”, nest it in a group like, “mammal”, nest it in a group like, “animal”, and so on.

The difference between GROUPS and traditional C++ classes are that C++ classes are lists of variables, functions, and other types of data members. GROUPs function as mesh representations that can be operated upon. They do this by declarations of NODES and NODE INTERACTIONS. This is not an easy thing to declare lexically or to parse. That is why I am inventing GAL. Writing code to do this in modern languages or even older languages, like Java, C++, Python, LISP, etc. is a lexical nightmare.

More on OPERATIONS. Besides the default four OPERATIONs, OPERATIONS are important within GROUPS because they are the symbolic representation of relationships between LINKS and SYMLINKS of NODES. SYMLINKS being symmetric or non-directional links between NODES and LINKS being directional links between NODES. Some OPERATIONS can include Boolean operations such as NOT, relational operations such as GREATER or LESS THAN, or constriction operators such as WHILE LESS THAN, or WHILE GREATER THAN. Constriction operators are traditional looping statements, if-else clauses, etc. tied to a LINK. They allow for us to create NEURAL NETWORK like links between NODES. An example LINK with a WHILE LESS THAN constriction could be used to represent a value that is only passed when it is less than 0.5 say, so that we could construct a GROUP that would train a neural network.

NODE is the most basic type. NODE does not mean NULL. NULL means NULL. Nor does it mean UNDEFINED. Everything must start with something; with integers most people base the integers off an increment/decrement of the number one. In GAL our most basic ABSTRACTION is NODE.

GROUP Foo {

Declare:

NODE A

NODE B

NODE C

FORCED Integrate()

FORCED Sort()  
FORCED DeIntegrate()  
FORCED Search()

OPTIONAL Print()

Define:

A SYMLINK B

B SYMLINK C  
C SYMLINK D  
D SYMLINK A

}

This is one of the simplest GROUPS. It *declares* three nodes, A, B, C, and then *defines* SYMLINKS between all of them. Just like a graph in graph theory or a network in communications, this GROUP has connections between them. In this case they are SYMLINKS, the same kind of link used in a semantic network. All we know is there is an association between all of these NODES but no real operations or constrictions occurring between them.

To abstract this GROUP Foo we would simple use the group name:

Foo

To instantiate a variable f of Foo we would use the following syntax:

f : Foo

To create a constant c of Foo we would use the following syntax:

const c : Foo.

To access an individual member of Foo, we would use C++ dot notation like:

Foo.A to access the membership of the NODE A in Foo

f.A to access the membership of the VARIABLE A in the ABSTRACTION of Foo

c.A to access the membership of the CONSTANT A in the CONSTANT

ABSTRACTION

Foo would also have the following operations:

Foo.integrate() { }

Foo.sort() { }

Foo.deintegrate() { }

Foo.search() { }

That are FORCED operations on the base class GROUP

And the OPTIONAL operation (FUNCTION) on the Foo GROUP:

Foo.print() { }

These would all be declared in the GROUP but defined elsewhere using the C++ like notation:

Foo::integrate() { }

Foo::sort() { }

Foo::deintegrate() { }

Foo::print() { }