Project Proposal: NodalCompML (NCML)

Team: Oscar Batori (ojb2109)
Nina Berg (nb2555)
Victor Frenkel (vgf2103)
Venkata Yamajala (vsy2104)

About Nodal Computation

The main idea behind this language is to allow **nodal computation (NC)**. NC is a way to design programs around nodes that contain data, a compute function, and input/output connections that define dependencies between nodes and flow of data.

A program in NodalCompML will contain a set of *StartNode*s, along with a series of regular *Nodes*. Each Node (including the StartNode) will have their own *compute* function. The output of each node then becomes the input to the next node. Program execution ends when the exit() function is called.

START NODES
We require an explicit set of start nodes that do not have dependencies on any other nodes. The start node may receive arguments from the command line. It then forwards its data to its adjacent nodes to start computation.

DATA PART OF NODE/ NODE STRUCTURE
Every node can contain an arbitrary number of local variables to be used by the compute function. A separate area is used to store the result of the compute function, to be distributed to other nodes via output connections from the node. Nodes consist of a required *compute* function, as well as any necessary local variables or helper functions.

NODE I/O
The input and outputs of every node must be defined using the basic types given in NCML. (double, float, int, string, char, boolean, **but not the node type**). All compute functions accept a list of identified input parameters. The output of each compute function is made via the *forward* command. Data outputs are forwarded to the specified input parameter of a chosen node. The compiler will verify that these types match.

The various input parameters to a Node's compute function can come from multiple nodes. Additionally, nodes can selectively forward their data based on boolean conditions using if-else constructs. This functionality may result in a situation where a node does not receive all the inputs it needs, or received multiple inputs for the same input parameter.

Each forward command can send one output value to an input value. Multiple copies of the same data can be sent to different nodes. Forwards may also send the same value to multiple

input parameters of the same node. Nodes can choose to forward data back to themselves recursively.

COMPUTE FUNCTION
The compute function accepts a list of input data, acts on the data, and forwards output to the next node in the program. The language allows for helper functions to be defined, breaking the compute function for each node into various pieces. However, all data i/o must be done within the compute function.

The function syntax will resemble C++/Java syntax, however, we will not be using semi-colons to end statements. Individual statements will be separated by a newline.

DEPENDENCIES/RE-EVALUATION
When a node receives a forwarded message from another node, its compute function is rerun on the changed field along with previous values for unchanged fields. If a node does not receive input for a field it simply blocks. If a node receives multiple inputs for a field, it computes using the most recent value for that field. This can occur in the case of conditional forwarding, where two nodes are inputting into the same field.

LANGUAGE FEATURES
The language will support creation of arrays out of any of the basic types (useful for working with images, matrices, etc.)

Problem Being Addressed

Nodal Computation, or rather DAGs, allow for thinking about computation from a different perspective. While O'Caml does not do anything that C cannot, it offers programmers a new, and in some respects more elegant, perspective on pre-existing concepts. In the way O'Caml encourages a programmer to think about the function in a mathematically rigorous fashion, NodalCompML encourages the programmer to conceptualize the relationships between function applications in a visual way.

However, NodalCompML is not just an elegant way of thinking about function composition like algorithmic procedures, it is also a computational paradigm that has been heavily employed in the graphics processing applications.

Ultimately we believe that this language delivers a valuable and expressive way to describe certain types of serieses of operations, while at the same time being mathematically elegant.

Node Template::
Node <node_name> {

    // local variables

```
        fun compute(<type> <field_name> , ...) {
                float a
                int b

                //do work - If/else, while, do-while

                forward (a) to (<node>.<field_name>)
                forward (b) to (<node>.<field_name>)
        }

        <type> fun <function_name> (params...) {
                // do work

                return <type>
        }
}
```

Representative Program

```
Node StartNode {
        fun compute(int argc, string argv) {
                //We can do some error checking here
                forward (argv[1]) to (Read.filename)
                forward (argv[2]) to (Write.filename)
        }
}

Node Read {
        fun compute(string filename) {
                file f = open(filename)
                if (f != NULL)
                        forward (f) to (Filter.f)
                else
                        forward (“error opening file”) to (Error.message)
        }
}

Node Filter {
        fun applyFilter(file f) {
                // do some stuff to file
                return modifiedFile;
        }
```

```
        fun compute(file f) {
                file mf = applyFilter(f)
                forward (mf) to (Write.f)
        }
}

Node Write {
        fun compute(string filename, file f) {
                if write(filename, f)
                        exit()
                else
                        forward (“error writing file”) to (Error.message)
        }
}

Node Error {
        fun compute(string message) {
                print message
                exit()
        }
}
```

The concept of the program is demonstrated graphically here: