Syntax, Semantics, and Pragmatics of a Visual Programming Language*

By: Nicholas DiPasquale
Faculty Advisor: Vijay Gehlot
Computing Sciences Department
Villanova University

In a recent report, the National Research Council, or NRC, advocated teaching a modest set of programming ideas even to non-science majors in order to prepare them for the challenges of the future and to develop algorithmic thinking and reasoning. This paper summarizes a work in progress of a flowchart-based visual programming language suitable for non-science majors. One of the key design goals is that the language for the aforementioned purposes should have "minimal syntactic burden". The contention of this paper is that the visual aspect of this language makes it easy to understand the control structure without delving into the peculiarities of syntax. Furthermore, in this setting, the notions of functional abstraction and application can be visualized as "boxing" and "plugging", activities, and this operational understanding of computation can be gained in a very visual manner.

In designing a language some of the first items to consider are syntax and semantics of that language, as well as target users. First, it is important to define the syntax of the language, defining how the user goes about programming in this language. Then based on the syntax it is important to define the semantics. When designing a language it is important to ensure it is syntactically complete. In order to do so, one must understand what it means to be syntactically complete. Syntactically complete means that a language contains several different types of statements. These statement types are an assignment operation, an expression, a list of expression structure, a branch structure, and a loop structure. These are the essential statement types that are included in this flowchart language; however, it is important to note how they are structured as this is a graphical language and statements often are not obvious at first.

The most important of these statements is the assignment operation. In many languages is an expression as well as a statement, this is also true for the flowchart language. The assignment operation is defined as the binding of some key to an associated data value to that key. This flowchart language is designed for non-science major, this requires the assignment operation to be simple and something common in mathematics. In order to do this, first the mathematical meaning of assignment should be analyzed. In mathematics, assignment is a general operation. Its meaning can be

* Proceedings of MASPLAS'04
Mid-Atlantic Student Workshop on Programming Languages and Systems
Seton Hall University, April 3, 2004

stated as the evaluated result of the left-hand side of the operator, the dependant variable, is evaluated equal to the evaluated result of the right hand side of the operator, the independent variable. A regular expression for the mathematical assignment looks like this:

```
<expression> = <expression>
```

Note: This is not to be confused with mathematical equality

For use in a programming language, this is too general. In order for this to translate easily and to avoid any confusion in the compiler this must be constrained. Most compilers today use a much simpler meaning for assignment, that is the left hand side key is assigned the evaluated result of the right hand side of the operator. The regular expression looks like this:

```
<identifier> = <expression>
```

This is constrained enough for use in this flowchart language, it also follows along with the mathematical definition. Once more, this definition of assignment gives a reasonable representation for non-science majors; they will be familiar with such constructs already.

Expressions are the heart of any modern programming language. Many statements derive their effects from expressions. Expressions are simple mathematical phrases. These include the use of binary and unary operations on variables. An expression in this flowchart language is typed in by the user in most programming languages; however, expressions are contained in programming blocks. Some sample expressions in this language look like this:

```
my_variable + 5
number1 < number2
```

Non-science majors are familiar with many of these operations from basic mathematics. This leads into the representation of list of expression. A list of expressions is simply a method to represent a group of expressions associated with one another. Under the flowchart language, there are two ways to represent a list of expression. The first is by using multiple blocks. The user can create basic programming blocks and enter in an expression for each block connecting them in the proper logical sequence. The second representation of a list of expression in this flowchart language is by use of a comma ',' after each expression. Thus, a sample list of expression looks as follows:

```
result = variable1 * variable2,
my_boolean = result < 100,
!my_boolean
```

This list of expression would appear in one programming block as opposed to three, this allows simplification of charts by more advanced users. It is not necessary to end an expression with a comma; however, it must be the only expression or the last expression in the list.

Branch structures are vital to the completeness of a language. These structures are the center program flow and control. A branch structure is a method of selecting one program flow branch from another based on a specified condition. The most common form of this is *if* statements in modern programming languages. In a visual programming language like this flowchart-based language, such constructs as if statements are represented graphically. Also for non-science majors keywords like *if* become a syntactic burden. A sample representation of a branch structure in this language is:

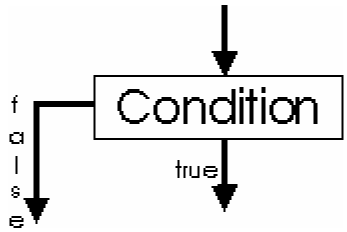

Diagram 5.1: A branch statement

The branch structure enters a specialized block with a condition expression that is evaluated. Then if that condition expression evaluates true, or non-zero the true branch is followed, otherwise the condition evaluated false, or zero, and the false branch is evaluated.

Derived from the branch structure is the loop structure. There are two major categories of loops in modern programming languages. These are classified by when the loop condition is evaluated. The first is loop evaluate first, this loop type evaluates the body of the loop before evaluating the condition for the loop. The second is loop evaluate after, this evaluates the loop condition first then possibly evaluates the body of the loop. The major difference between the two classifications is how many times the loop body will evaluate. The first is guaranteed to evaluate one or more times. The second is guaranteed to evaluate zero or more times. In modern programming languages, they are separated by the keyword used. A pseudo C sample looks like this:

Table 5.1: **Loop Structure**

Loop evaluate first:
 	
`while(loop condition) {`
`        loop block;`
`}`

Loop evaluate after:
 	
`do {`
`        loop block;`
`} while(loop condition);`

In a flowchart language, this must be represented graphically. Diagram 5.2 demonstrates this for the two loop classifications.

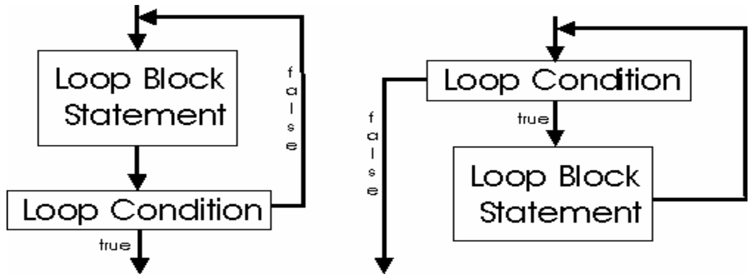

Diagram 5.2: A) Left Loop Evaluate after B) Right Loop evaluate first

The body of the loop consists of one or more block components followed by a condition block or a condition block followed by one or more blocks that form the body of the loop. The only other requirement is that a branch from the condition connects back to an earlier block, or a later block connects back to the condition.

Now that the syntax and semantics are established firmly for the flowchart language, a sample is in order. This section will discuss the representation of a well-known algorithm using the syntax and semantics discussed above for the flowchart language. Diagram 5.3 shows the sample algorithm of shellsort.

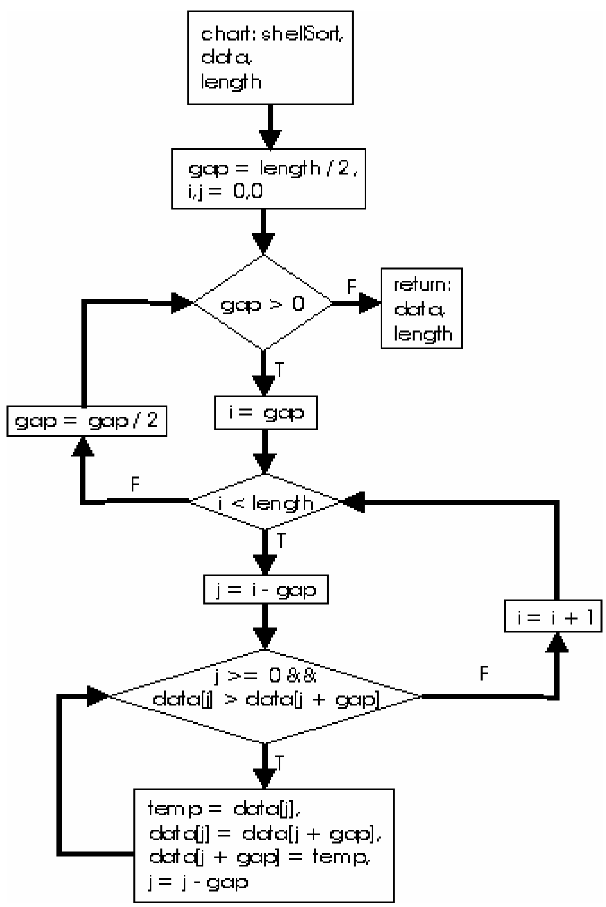

Diagram 5.3: Sample Shellsort Algorithm

The chart entry point contains the name of the chart and the parameters, if any, the chart requires. Shellsort requires two parameters, an array of data to be sorted and the length of the data array. An important note is all variables are not typed, this relieves the syntax burden from the user; however, datatypes must be checked either at compile time using aspect checking or at runtime using runtime type information, or RTTI. The entry point contains no actual code, but a header for the algorithm, the next block contains the first expressions to be executed. The first standard block contains two expressions, the compiler recognizes the assignment of the variables named *gap*, *i*, and *j* and declares them in the symbol table appropriately. Each standard block contains a combined list of expression, instead of multiple block list of expressions, this minimizes the amount of blocks required for the chart; however, it is acceptable to separate these list of expression. The diamond shaped blocks are the branch structures, each of these contains a true branch and a false branch out. The chart is guaranteed to terminate in the halt block, labeled return, as long as length is greater than zero and data is a valid array. The program follows the algorithm in the standard procedural manner moving from one block to the next as dictated by the connections in the chart and the evaluated conditions at branch points.

This language offers a Turing complete tool set in a few specialized blocks, thus minimizing learning time for users. Expression syntax is based on general mathematical expressions and from general programming language pragmatics. Using these rules users do not have to remember complex syntax for expressions, thus simplifying the language further. The specialized program blocks that users have available to them to create programs are defined by the following Standard ML type definition:

```
datatype Block = start of StartBlock | halt of HaltBlock |
                 cond of IfBlock | call of CallBlock |
                 basic of StandardBlock
```

Each of these sub-types represent a component that can be placed in a program by the user. The StartBlock represents the chart entry point. All valid charts require this block and each chart may only have one such block. A StartBlock has the following type definition:

```
datatype StartBlock = out of Block * name of string *
                 params of string list option
```

This definition provides all of the information that a chart requires upon entry. The name of the chart is maintained here as well as a lit of parameters for that chart, this is used to distinguish this chart from other charts, of differing name or same name differing parameter lengths. It is not necessary for a chart to have a parameter list, yet all charts must have a name. This name is suggested to be a unique identifier; however, as the parameter list length is used in identifying a chart it can be the same as another with a different set of parameters. StartBlocks maintain only one outgoing connection to other blocks and no incoming connection can be made. The next block type is the HaltBlock type, it represents a possible program termination point. Unlike the StartBlock there can be multiple copies of this block, however they must maintain the same information. A HaltBlock has the following definition:

```
datatype HaltBlock = in of Block list * nil *
                 ret_types of string list option
```

The HaltBlock contains a list of return values for a particular chart. It is not necessary for a chart to return any values. The most important idea that must be stressed on this block is that within any program each copy of this block must look identical to all other copies of this block. The values passed at any given point in the program need not be the same, however the list length, must be the same. There is not limit on incoming connections to the HaltBlock; however, a HaltBlock can not have any outgoing connections. The next block type is the IfBlock. This block type allows users to create conditions in the flow of the chart. The IfBlock has the following definition:

```
datatype IfBlock = in of Block list * out of Block list *
                   loopin of Block * expr of Expression
```

The IfBlock maintains a list of incoming blocks and outgoing blocks. There are two possible outgoing connections from any IfBlock, a true connection and a false connection. Following that there is a special loop connection, this connection is where users can create loops in their programs. IfBlocks are the only place where users can create loops. After the loop block there is the expression that is to be evaluated on every run of the IfBlock. After the IfBlock the last special block is the CallBlock. The CallBlock is a special block that acts as a method or function call would act in a standard programming language. The syntax for the CallBlock is as follows:

```
datatype CallBlock = in of Block list * out of Block *
                     name of string * params of string list option
```

This block maintains a list of incoming connections and an outgoing connection as well. Following the connections is the name of the chart to call or execute along with a parameter list for that chart. A parameter list is not required; however, if a chart requires parameters, the list from its StartBlock, then a list of values of that length must be given. A chart may call itself, recursively, as long as the parameter list in the CallBlock match those of the StartBlock. Lastly, there is the StandardBlock. This block type makes up most of the components of any program. This definition of a StandardBlock is as follows:

```
datatype StandardBlock = in of Block list * out of Block *
                         expr of Expression list
```

This block type, again, maintains a list of incoming and outgoing connections, as do all of the other blocks; however, it maintains a list of expressions. This list of expressions will be executed whenever the flow of the chart enters this block. Each expression is executed first come first serve, as per the normal semantics of standard programming languages.

The sample algorithm above can be easily translated from charts into these SML datatypes. Starting with the entry point the sample program can be translated into the following SML datatypes:

```
StartBlock(StandardBlock0, “shellsort”, “data”, “length”)
StandardBlock0(StartBlock, IfBlock0, Expression(“gap = length / 2”,
       Expression(“i, j = 0, 0”))
IfBlock0(StandardBlock0, StandardBlock1, HaltBlock, StandardBlock5,
       Expression(“gap > 0”))
StandardBlock1(IfBlock0, IfBlock1, Expression(“i = gap”))
IfBlock1(StandardBlock1, StandardBlock2, StandardBlock5,
```

```
        StandardBlock4, Expression( "i < length" ))
    StandardBlock2(IfBlock1, IfBlock2, Expression( "i = i -  gap" ))
    IfBlock2(StandardBlock2, StandardBlock3, StandardBlock4,
        StandardBlock3,
        Expression( "j  >= 0 && data[j] > data[j + gap]" ))
    StandardBlock3(IfBlock2, IfBlock2, Expression( "temp = data[j]" ,
        Expression( "data[j] = data[j + gap]" ,
            Expression( "data[j + gap] = temp" ))))
    StandardBlock4(IfBlock2, IfBlock1, Expression( "i = i + 1" ))
    StandardBlock5(IfBlock1, IfBlock0, Expression( "gap = gap / 2" ))
    HaltBlock(IfBlock0,  "data" ,  "length" )
```

The numbers are added in for ease of reading. These datatypes form the complete abstract syntax tree for the shellsort chart example. This abstract syntax tree then is used to translate the chart from its graphical form to be executed on the computer.

The flowchart language is designed as a graphical language, this gives the users a GUI to construct programs. The GUI feature of drag and drop add simplicity in the creation of a program and its components. In this manner a user selects a particular component from a list and then drops that into their program. Each component also offers the ability of customization for specific usage in that program. This customization does not alter how the component works but rather what the component does in that particular program. Then the component can be "plugged" into the application as seen fit by the user. The "plugging" action formalizes the flow of the program components, keeping in mind that this is a flowchart-based language.

The design of the components in this language is structured for ease of translation and simplicity of use. An algorithm derived from basic graph traversal and parsing algorithms is designed to translate a chart into CHart Intermediate Level Language, or CHILL. CHILL is based from the ANSI C standards and is very much like C in design; however, there are some basic differences. From CHILL, it is then possible to interpret, generate, or optimize the program from its original chart form. The chart is maintained in an almost abstract syntax tree, or AST, like data structure that facilitates its parsing and translation. This also allows for error checking while the user creates the program. On the fly error checking is important as this language is designed for non-science majors. Using this method the GUI can alert the user that they have connected the chart in an improper manner or attempting to perform an illegal operation, as they attempt to do so.

The translation algorithm is based on Abstract Syntax Trees and depth first search, or DFS, of a directed graph. The data structure that any given chart is maintained in is much like an AST. This facilitates the translation process, which is governed by the DFS since this is not exactly a tree structure. This structure must contain cross cut branches and back branches in order to allow looping in the chart. The sample algorithm representation in CHILL looks as follows:

```
shellsort(data, length) : _data, _length {
    gap = length / 2;
    i, j = 0, 0;
    while(i < length) {
        i = gap;
        while(i < length) {
            j = i -  gap;
```

```
                        while(j >= 0 && data[j] > data[j + gap]) {
                                temp = data[j];
                                data[j] = data[j + gap];
                                data[j + gap] = temp;
                                j = j - gap;
                        }
                        i = i + 1;
                }
                gap = gap / 2;
        }
        return(data, length);
}
```

The SML translates starting from the StartBlock generating the function header, the return types are generated later upon encountering of the HaltBlocks. The first StandardBlock is encountered and the expressions are output without modification. The first IfBlock is encountered and the special *loopin* member is checked to determine if the condition should be translated as a loop or an *if* statement. The *loopin* is non-null in this instance and thus a while loop is created with the expression as the condition of the while loop. This process then continues until every node of the tree is visited.

With a clear syntax and semantics derived from the syntax, the definition of a flowchart-based language is complete. The potential of the language is displayed through a shellsort sample. The graphical representation is presented along with its various stages in translation. The language offers a complete set of tools; they are simple enough for non-science majors to use intuitively. Still there are some areas left unexplored by this paper. For instance, the branch structures are restricted to only a true and false branch, this can be expanded to *n* branching, to simulate a switch block. Another area left to be explored is in the language library, most successful languages offer an extensive library. One of the important areas for this library to cover is in the area of multimedia. Most non-science majors would appreciate the results of a multimedia-enhanced application as opposed to a math or text oriented application. For example, image rotation, color filtering, or even movie playback. In the end a graphical language for non-science majors must come down to two factors, the first is ease of use and the second is tool set completeness. In the current state, this flowchart-based language offers simplicity of use and a complete set of programming components.

Appendix 5.1: CHILL regular Expressions

```
CHILL Regular Expressions Describing Language Syntax:

<program>                      ->  (<function>)+
<function>                     ->  <identifier> '(' <identifier list> ')'
                                   [':' <identifier list>] <block>
<identifier list>              ->  <identifier> (',' <identifier>)*
<block>                        ->  '{' (<statement>)* '}'
<statement>                    ->  <if statement>
                               |   <while statement>
                               |   <do statement>
                               |   <return statement>
                               |   <expression> ';'
```

```
                                   |   <block>
<if statement>                     ->  'if' '(' <expression> ')' <statement>
                                       [ 'else' <statement> ]
<while statement>                  ->  'while' '(' <expression> ')' <block>
<do statement>                     ->  'do' <block> 'while' '('
                                       <expression> ')'
<return statement>                 ->  'return' <identifier list>
<expression>                       ->  <assignment>
                                   |   <arithmetic expression>
<assignment>                       ->  <identifier list> ('=' | '+=' | '-=' |
                                       ...) <arithmetic expression list>
<expression list>                  ->  <expression> (',' <expression>)*
<arithmetic expression list>       ->  <arithmetic expression> (','
                                       <arithmetic expression>)*
<arithmetic expression>            ->  <logical or>
<logical or>                       ->  <logical and> ('||' <logical and>)*
<logical and>                      ->  <bitwise or> ('&&' <bitwise or>)*
<bitwise or>                       ->  <bitwise xor> ('|' <bitwise xor>)*
<bitwise xor>                      ->  <bitwise and> ('^' <bitwise and>)*
<bitwise and>                      ->  <equality>  ('&' <equality>)*
<equality>                         ->  <relational> (('==' | '!=')
                                       <relational>)*
<relational>                       ->  <shift> (('<' | '<=' | '>' | '>=')
                                       <shift>)*
<shift>                            ->  <addition> (('<<' | '>>') <addition>)*
<addition>                         ->  <multiplication> (('+' | '-')
                                       <multiplication>)*
<multiplication>                   ->  <unary> (('*' | '/' | '%') <unary>)
<unary>                            ->  ('!' | '~' | '-' | '+' | '--' | '++' |
                                       'sizeof') <primary>     <postfix>
<primary>                          ->  <identifier>
                                       ['(' <arithmetic expression list> ')' |
                                          '[' <expression> ']' ('['
                                          <expression> ']')*]
                                   |   '(' <expression> ')'
                                   |   <numerical literal>
                                   |   ('true' | 'false')
                                   |   'null'
<postfix>                          ->  ['--' | '++']
```

Designer's NOTES:

Some important factors that differentiate this language from the design template language of C are the lack of types and clear declarations. This is due to the super-language that is associated with this intermediate level language. The super-language does not support variable typing nor declarations. Another note is that this language differs from C in function declarations. The function header places the return types after the name and formal parameter list of the function. This is a Pascal convention; however, it simplifies the reading of multiple return values. Operator precedence follows exactly that of the template language C. The lack of the 'for' statement can also be noted, this is again attributed to the structure of the super-language. The 'sizeof' operator may or may not appear in a production version because of the lack of types in either of these languages, this or the super-language. The <arithmetic expression> non-terminal is left for possible further expansion of the language with other expression types. All variables are passed by value not by reference.

Bibliography

- Sedgewick, Robert, *Algorithms in C: Part 5 - Graph Algorithms*, third edition, Addison Wesley 2002.
- Grune, Dick, et al. *Modern Compiler Design*, Wiley, 2000
- Mak, Ronald, *Writing Compilers and Interpreters: An Applied Approach Using C++*, second edition, Wiley, 1996.
- MIT Logo, http://el.media.mit.edu/logo-foundation/index.html
- CMU ALICE, http://www.alice.org/
- Ullman, Jeffrey D., *Elements of ML Programming*, Prentice-Hall, 1998.