Flowgorithm: Principles for Teaching Introductory Programming Using Flowcharts

Devin D. Cook

California State University, Sacramento, CA

Abstract

For students, the task of learning their first programming language can be compounded by the challenges of syntax, semantics and superfluous code. Historically, programming languages had a gentle learning curve requiring little syntactic or semantic overhead. Modern object-oriented languages, however, create a conceptual hurdle. Even the trivial Hello World program contains syntactic and semantic complexity far beyond the level of a beginning student.

This paper introduces Flowgorithm – a programming environment which allows students, with little to no programming experience, create programs using the visual metaphor of flowcharts. These flowcharts can be executed directly by the built-in interpreter allowing students to learn programming concepts before being confronted with language-specific syntax and semantics.

Flowgorithm provides an integrated learning path so students can apply their knowledge to a "real" programming language. The flowcharts can be interactively translated to over 10 programming languages. These include: C#, C++, Delphi/Pascal, Java, JavaScript, Lua, Python, Ruby, Visual Basic .NET, and Visual Basic for Applications. This allows a natural transition from the simple procedural logic of flowcharts to the more common object oriented languages used by universities today.

Beginning Programmers

Even the most gifted computer programmer, at one point, was a beginner. And as beginners, they have to struggle with the inherit challenges of their first language. Whether the language was BASIC, Pascal, C, Java, etc… they had to first handle the issue of syntax. Many languages have a syntax closely related to natural pseudocode while others can be either obtuse or symbolic. After basic syntax is understood, the programmer can then learn the semantics of the language. These can be implied by the syntax itself or can be, in many cases, unrelated. In this environment, students learn the basics of programming logic.

This learning curve can be gentle – allowing students to learn and understand concepts one at a time. Only the source code, that demonstrates the concept, is required. For example, the following is the solution for Hello World in QuickBASIC. There is little syntactic overhead which results in a program that is simple, short, and intuitive. Students can understand the program even though they may have never had any training in this language.

```
PRINT "Hello, world!"
```

Figure 1: Hello World in QuickBasic

Unfortunately, the original procedural paradigm is being supplanted by the object-oriented paradigm. Under object oriented programming, everything is an object and all written code is related to class definitions and methods. While this approach offers flexibility and scalability, it requires students to define class constructs before even basic programming concepts are understood – such as expressions, variables, conditional logic, loops, etc… This results in a learning curve that has increased rather than decrease.

For example, the following is same Hello World example in the Java Programming Language. This code declares a class called *HelloWorld* with a single static method called *main*. Under the semantics of Java, the static *main* method is called when the Java runtime engine starts. The method then calls the *System* object's *println* method which prints the text to the console.

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, world!");
    }
}
```

Figure 2: Hello World in Java

While this is a valid solution for an object-oriented language, the beginner student is confronted with considerable syntactic overhead and a large number of advanced concepts. For beginning programmers, the emphasis should be on the *printlln*, the concept of output, and string literals. However, the student must type (and ignore) structures that define: a static function, a void return value, parameters, arrays, access modifiers, and more… This is compounded with the terse and symbolic notation of the C-family languages. While Java is both a robust and popular language, the rudimentary Hello World program is intimidating and overly complex for the beginner programmer.

Solution Requirements

The solution to this problem is a programming environment that allows students to learn programming concepts without burdening them with the nuances of a specific language. This includes syntax and language-specific semantics. However, it must be acknowledged that students will eventually learn another language. So, any solution must incorporate a learning path to other languages. As a result, the solution must adhere to the following principles:

1. Minimizes syntactic overhead
2. Clear semantics
3. Provides a mechanism to transition students to a major programming language

1. Minimalizing Syntactic Overhead

To address the issue of syntactic overhead, this solution creates a programming environment that incorporates a graphical programming metaphor. Students will construct programs using graphical shapes to represent different components of an algorithm. This concept is not new, and has been implemented successfully by a myriad of applications.

When comes to the design of the graphical elements, should they be original or based on an existing industry standard? If the graphical elements are customized, such as in MIT's Scratch[1], then their use cannot be applied elsewhere.

Alternatively, flowcharts have been used throughout the industry since their original inception by IBM[2]. They are easy to understand and conceptualize. And, most importantly, they are able to represent all the common language constructs such as If, While, Do, For, etc… Introductory programming books often make use of flowcharts. This is the case with "Starting Out with Programming Logic and Design" by Tony Gaddis[3].

To adhere to a standard, flowcharts were selected as the graphical metaphor. This programming environment was named "Flowgorithm" which is a portmanteau of the words "flowchart" and "algorithm".

Mapping Flowchart Shapes

Most of the shapes found in modern flowcharts follow directly from the original IBM standard. Variations do exist, but the IBM format is the de-facto standard used in industry and academics. Flowgorithm maps each of the common programming logic tasks directly to the shape that best corresponds to its semantics. The Terminal Shape is used to represent the beginning and ending of a function. The Input/output shape is be used to represent information being read from the keyboard and information displayed to the screen's terminal. The Decision Shape represents an If Statement with the two branches – true and false (alternatively yes and no) – to represent the "then" and "else" clauses of the statement. The Subroutine Shape will represent a call to a function (i.e. procedure or subroutine).

The Calculation Shape represents variable assignment. It is important to note that mathematical expressions can be used throughout a program – such as output, variable assignment, operands of a Boolean expression, etc… So, the designation "Calculation Shape" is inheritably vague and misleading. As a result, this shape, in the context of this solution, will be referred to as the "Assignment" shape.

In addition, it is important for programs to contain internal documentation in the form of comments. In classical flowcharts, this information is provided using the Annotation Shape. To emphasize the inert nature of comments and to contrast from other shapes, Flowgorithm uses as variation with dashed lines.

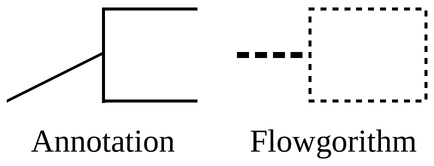


Figure 3: Comment Shape

In classic flowcharts, both loops and If Statements are constructed using the diamond "Decision" shape. While the two concepts are related, it requires the student to visually trace the lines to determine of the construct is loop or If Statement. To make the semantics clear upon visual inspection, Flowgorithm will use an elongated hexagon to represent a looping structure.

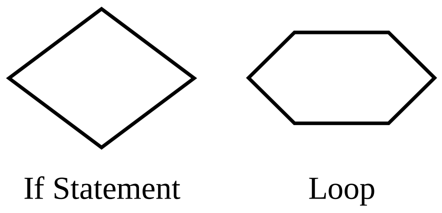


Figure 4: Conditional Branch vs. Loop Shape

This shape is classically used in flowcharts to represent a "preparation"[4] task - i.e. an action required for the flowchart to proceed. However, this shape is commonly used to represent looping in existing solutions. These will be covered later in this paper.

Finally, Flowgorithm uses explicit variation declaration. To represent a declaration, one of the original (yet rarely used) IBM flowchart symbols is used called "Internal Storage"[5]. The symbol is used to represent the computer's core memory – which is where variables are stored. Visually, the shape is similar to the Assignment Shape which underscores that both incorporate variable management.

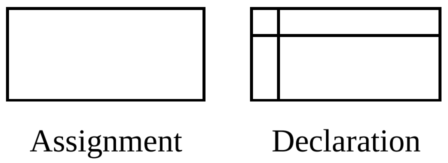


Figure 5: Variable Management Shapes

Flowchart Structure

There is no official standard to how the different graphical elements of a flowchart are arranged. Variations are common with shapes moving to the left, to the right, or bounding back to the top of the page. To make the flowcharts consistent in Flowgorithm, the arrangement of shapes will follow a set number of rules.

1. The flowchart will start at the top of the page
2. Sequences of shapes (blocks of statements) will move downward.
3. The flowchart will move to the left or right only based on conditional logic.
4. Flow will move upwards only to complete a loop.

Loops and If Statements both contain a block of statements that are conditionally executed. To make these blocks visually distinctive, each block will either be located to the left (or right) of the shape. Hence, any blocks of statements should be discernable, to the beginner programmer, upon inspection. To keep the logical flow of pre-test loops, post-test loops and If Statements consistent, the flowchart will always branch to the right if the expression is true. The false branch will either fall straight down (for loops) or to the left (for If Statements).

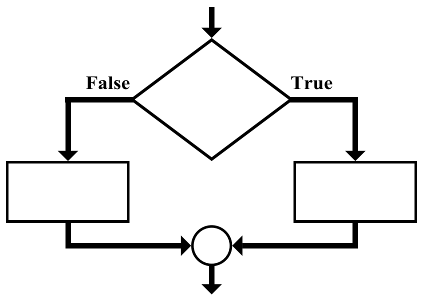


Figure 6: If Statement Layout

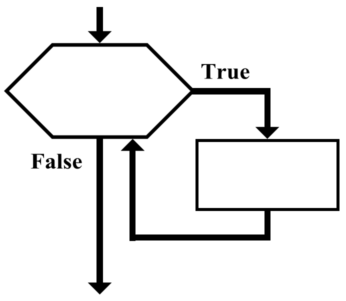
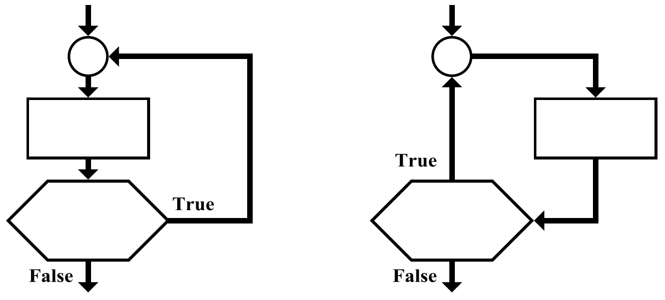


Figure 7: Pre-Test Loop Layout

Flowgorithm uses a different post-test loop layout than is used in typical flowcharts. To maintain consistency with the other two layouts, the post-test loops are arranged such that the body of the loop branches to the right. This allows the "block" to be visually distinct and also creates a clockwise motion which is consistent with the pre-test loop.



Classic Flowgorithm

Figure 8: Post-Test Layouts

2. Clear Semantics

Languages used by novice programmers should avoid semantic ambiguity and unpleasant "gotcha" moments. Beginning students have not had the training or the experience to logically dissect a logical error. If the error itself is caused by unintuitive operator behavior or an unexpected side-effect, it can be both frustrating and disheartening to a student.

For example, one of the classic "gotchas" in the C Programming Language is the behavior of the division operator. The usual arithmetic conversions of the C language emphasize integer operations. This is a logical choice given that the language was developed in an era where many processors did not support floating point logic and C's emphasis on speed and efficiency[6]. The following C-Style program provides an incredibly difficult "gotcha" moment.

```
volume = 4 / 3 * M_PI * pow(radius, 3);
```

Figure 9: Incorrect Volume of a Sphere in C

While the example uses the correct formula to calculate the volume of a sphere, it contains a fatal flaw. Under the usual arithmetic conversions of C, if both operands are integers, then integer math is used. The "4 / 3" contains two integer literals resulting in the integer value "1" rather than "1.33333". For a novice programmer, it is not obvious upon initial inspection.

Moreover, some operators have different semantics based on the data type used. The most common form is the use of the "+" operator to both represent addition and string concatenation. Abstractly, the concepts can be seen as related, but for a student attempting to create a string from string literals and numeric values (either a variable or literal), the expression can be difficult to write and contain unintended side effects.

```
1 + 2 + "value"
```

```
"value" + 1 + 2
```

Figure 10: Java Concatenation Variations

In the first example, the integers 1 and 2 are being mathematically added and then concatenated with the text *"value"*. The result of the expression will be a string containing "3value". The second example, with only the order of operands changed, results in a drastically different result. In this case, *"value" + 1+ 2* will return the string "value12" rather than "value3".

The cause of the unexpected result is the order of the operands. The Java Programming Language uses the left operand to define the semantics of the operator. In the second case, "value" caused the addition operator to be interpreted as concatenation. This also chained into the second addition operator (since the first returned a string). It should also be noted that, in mathematics, the addition operator is defined as commutative and associative. Java's usage violates both properties.

Expressions

While the use of flowcharts minimalizes syntax (in fact, nearly eliminates it), any programming environment needs to implement mathematical and Boolean expressions. Unlike the shapes used in flowcharts, the symbols, used to represent operators, vary greatly between different programming languages. Expressions tend to fall into one of two categories depending on what other languages influenced them. The BASIC Programming Language uses readable operators such as AND, OR, etc. This contrasts with the C Programming Language which uses more symbolic notation such as &&, ||, etc.

If students plan to transition to major language, the question arises: which family should be used? Depending on the student's learning path, one set of operators will be compatible, while the other will not. To make it easier for students to take either path, Flowgorithm supports both sets. This will result in a number of redundant operators. The instructor can teach using either set – depending on the target language.

Operator	C	BASIC
Modulus	`%`	`mod`
Equality	`==`	`=`
Inequality	`!=`	`<>`

Operator	C	BASIC
Logical And	`&&`	`and`
Logical Or	`||`	`or`
Logical Not	`!`	`not`

Table 1: Redundant operators

To maintain simple semantics, the "+" operator will only be allowed for numbers. String concatenation will be achieved by using the Visual Basic styled ampersand "&". The C-Family languages also lack an exponent operator. Like the ampersand, Flowgorithm borrows the operator from the Visual Basic language. Both of these operators have clean semantics which also aid in code generation.

Variables

Flowgorithm uses strong typing in conjunction with explicit variable declaration. There a number of benefits to this approach:

1. Undeclared variables will cause an error – hence catching minor typos in the variable name. This is a common mistake by both novice an expert programmers.
2. The type of a variable is explicitly known.
3. Its usage of the variable is consistent throughout the program since its stored data cannot deviate from its declared type.

While not all languages make use of explicit variation declaration, it is an important concept to stress. If a student's learning path takes them to Python, for example, they will not need to type variable declarations. The student simply needs to omit typing a construct that they have been trained to understand. The student could be encouraged to create comments in the place of these declarations further documenting the program. However, if a student learns in an environment that uses implicit variable declaration, then the concept will be new when they are exposed to languages such as Java, C#, Visual Basic, and Lua. In other words, is far more beneficial for a student to learn disciplined programming and then by exposed to a weak-typed system, than to allow an undisciplined programmer to be exposed to a strong-typed system.

One Way In, One Way Out

Ever since Dijkstra wrote his paper letter concerning the use of the Go-To Statement[7], the concept of Go-To Statements has been widely discredited for use in high-level languages. In fact, most high-level languages, such as Java, do not feature it. Flowcharts, like modern structured languages, are inherently one-way-in, one way out.

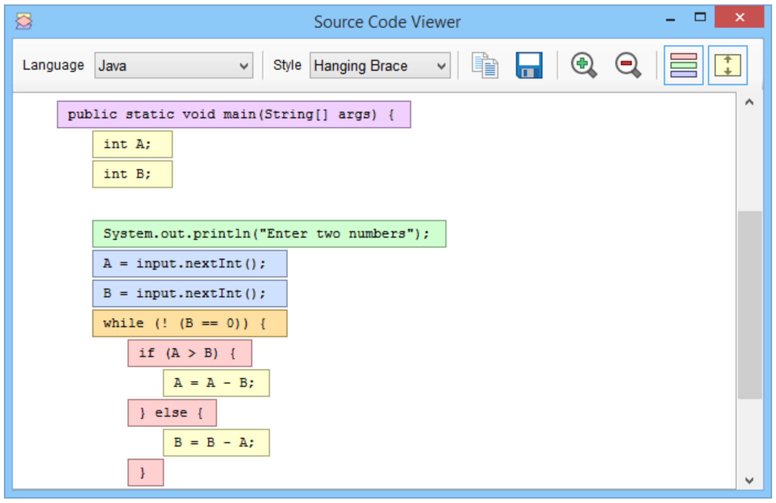
However, it is possible to add constructs that violate this principle. Quite notably, many languages feature Break Statements and Return Statements. Both of these concepts allow execution to jump directly out of a control-structure. They essentially function as a go-to, but only jump forward and prevent the "spaghetti-code" that caused derision with the Go-To. This is also true of the Return Statement except it jumps to the end of the function.

To maintain a strict adherence to one-way-in, one-way-out concept, Flowgorithm does not contain Go-To Statements, Break Statements or Return Statements.

3. Transition to a Major Language

While flowcharts are a standard method of describing an algorithm, they are not actually used to write programs. Rather, applications are written in Java, C#, and another major language. The problem then arises that if a student learns to program using flowcharts, how does it translate to actual knowledge in a University-level language?

Flowgorithm was designed to provide a learning path so students can take the knowledge and apply it to actual source code. Once the basic concepts are understood, students can generate source code from the flowchart. This code will not be exported separately to a file, but, instead displayed in an onscreen window that automatically updates as the flowchart is changed.



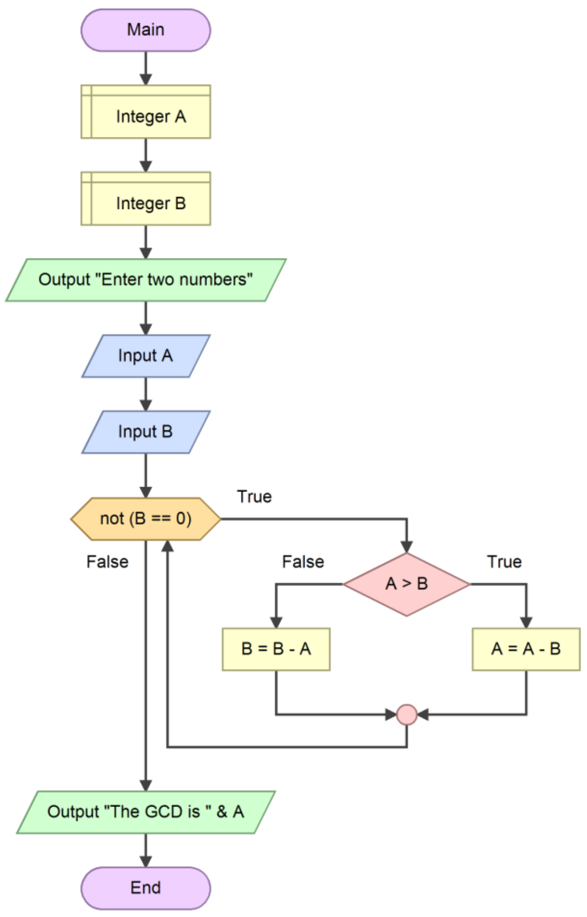
Language: Java
Style: Hanging Brace

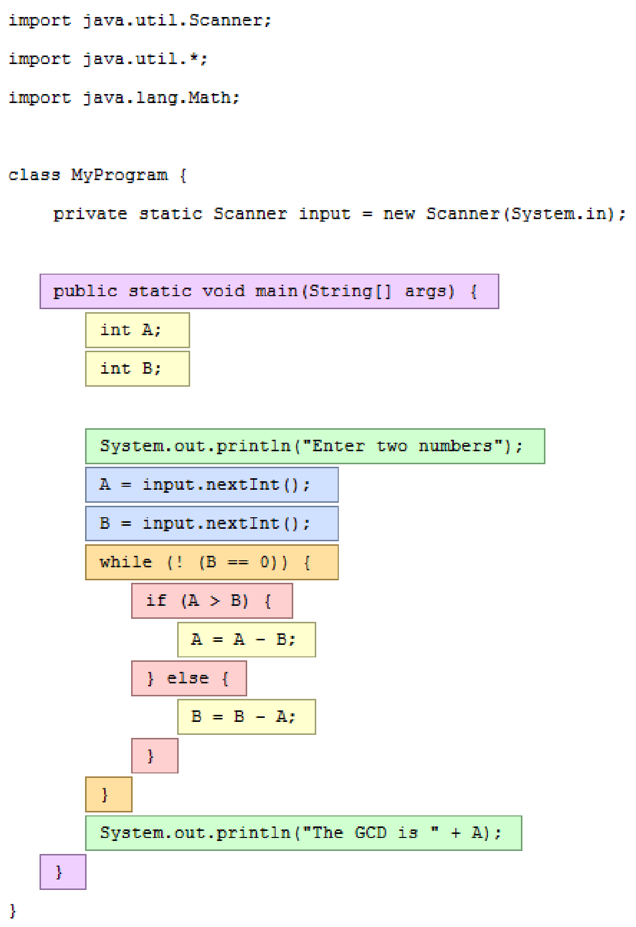
```
public static void main(String[] args) {
    int A;
    int B;

    System.out.println("Enter two numbers");
    A = input.nextInt();
    B = input.nextInt();
    while (! (B == 0)) {
        if (A > B) {
            A = A - B;
        } else {
            B = B - A;
        }
```

Figure 11: Source Code Viewer

To aid students visualize how a shape relates to its generated source code, both shapes and source code are color coded based on their functional category.

If the programmer is steps through the code, the current shape in the flowchart is highlighted as well as the corresponding source code. Each line of source code is linked directly to its parent shape – and will change colors based if the shape is selected, in error, or the current shape in an executing program.





```
import java.util.Scanner;
import java.util.*;
import java.lang.Math;

class MyProgram {
    private static Scanner input = new Scanner(System.in);

    public static void main(String[] args) {
        int A;
        int B;

        System.out.println("Enter two numbers");
        A = input.nextInt();
        B = input.nextInt();
        while (! (B == 0)) {
            if (A > B) {
                A = A - B;
            } else {
                B = B - A;
            }
        }
        System.out.println("The GCD is " + A);
    }
}
```

Figure 12: Flowchart of Euclid's GCD and Generated Code

As of this writing, the following programming languages are supported: C#, C++, Delphi/Pascal, Java, JavaScript, Lua, Python, Ruby, Visual Basic .NET, and Visual Basic for Applications.

Existing Solutions

Using the visual paradigm of flowcharts is not an original approach. Flowcharts were created for this very purpose, so creating a programming environment based them is merely a logical extension of the concept. The following are the three most notable:

- Visual Logic[8]
- LARP[9]
- RAPTOR[10]

Each of these solutions successfully minimalizes syntax by using flowcharts. The design of the flowchart layouts and shapes varies between solutions, but, overall the notation is compatible. Unfortunately, these solutions do not sufficiently address the issues regarding semantics nor do they provide a learning path for students.

Future Work

Currently the software is only available on Microsoft Windows. Since the source was written in Microsoft C#, efforts will be made to cross-compile it to both Macintosh and Linux. Also, future versions will feature multi-lingual support. The project website, www.flowgorithm.org, will continue to be enhanced with examples, documentation, and relevant information.

Additional programming languages will be supported, as needed, by the Source Code Viewer. These include Perl, Objective-C, and Ada. The language itself can also be further enhanced with additional intrinsic functions and simple file I/O. The latter will only be added if it will produce valid programs in all the supported programming languages.

Conclusions

Besides addressing the aforementioned issues, great effort was made to make the application user-friendly and self-documenting. This includes features not mentioned in this paper such as the Variable Watch Window and Console Window. Student reaction to the software, as well as comments from the website, have been extremely positive.

The software was introduced in fall 2014, so more time is needed to fully measure any long term effects on student performance and retention. However, given student performance in "CSC 10: Introduction to Programming Logic", it is expected to be a beneficial tool.

Bibliography

[1] Scratch Website. (1/20/2015). Retrieved from http://scratch.mit.edu
[2] IBM. (1970). *Flowcharting Techniques*, Publication: C20-8152-1
[3] Tony Gaddis. (2012). *Starting Out with Programming and Design (3 edition)*, Addison-Wesley. 978-0132805452
[4] IBM. (1970). *Flowcharting Techniques*, Publication: C20-8152-1
[5] IBM. (1970). *Flowcharting Techniques*, Publication: C20-8152-1
[6] Dennis Ritchie (1993). *The development of the C language*, ACM SIGPLAN Notices, Vol 28 Issue 3, March 1993, pg 201-208
[7] Edsger W. Dijkstra. (1968). *Go-to statement considered harmful*, Letter to the Editor, ACM 11, 3: 147–148
[8] D Gudmundsen, L Olivieri, N Sarawagi. (2011). *Using visual logic©: three different approaches in different courses - general education, CS0, and CS1*. Journal of Computing Sciences in Colleges, Vol 26 Issue 6, June 2011, pg 23-29. Website: www.visuallogic.org
[9] LARP Website. (1/20/2015). Retrieved from http://www.marcolavoie.ca/larp/en/default.htm
[10] M Carlisle, T Wilson, J Humphries. (2004). *RAPTOR: introducing programming to non-majors with flowcharts*, Journal of Computing Sciences in Colleges, Vol 19 Issue 4, Apr 2004, pg 52-60. Website: raptor.martincarlisle.com