# Graphical Block Structured Programming: A Visual Programming Paradigm

**Jason P. Sermeno** [1]

**Abstract:** This paper discusses the concept and design of a graphical block-structured programming paradigm that presents a model for constructing computer programs using a set of graphical objects that resembles the existing lexical instructions in a C language. The design of the paradigm was motivated by the results from studies investigating the previous designs and the acquisition of existing visual programming languages. Studies showed that most people are having trouble expressing the structures that they cannot write or verbally describe due to their limited grasp of natural language. The aim of this proposed programming paradigm is to improve the user's ability to create programs by making programming more accessible to some particular audience and improving the correctness and speed with which people perform programming tasks.

**Keywords:** Computer programming, Programming languages, Graphical block-structured programming, Visual programming

## 1. Introduction

A programming language is defined as a machine-readable artificial language that is specifically intended to implement computations that can be performed by a machine (*i.e.*, a computer) [1][2]. Programming languages are utilized in creating programs, software, scripts, or any set of instructions that can manipulate machines and devices, precisely express and implement algorithms, or as a mode of human communication [3].

In general, most people are frustrated in learning the basic concepts of constructing programs because it is difficult for them to conceptualize the structures that they cannot describe verbally or in writing due to their limited grasp of natural language, particularly in the depth of abstraction [4]. Many users especially beginners find it difficult and time-consuming in constructing and dealing with the syntactical structure of the code because most of its instructions or commands are written in text and are presented from an abstract point of view. It can be difficult to identify or distinguish the functionality of certain features of the programming language because of its vague syntactic rules. Thus, more efficient programming language features can be misused. Debugging is a natural way of finding and fixing a bug. But troubleshooting a code with a large number of syntax errors may add a burden to users and would

[1]* College of Computer Studies, University of Antique, Sibalom, Antique, Philippines
Email: jason.sermeno@antiquespride.edu.ph

*Received [December 21, 2018]; Revised [February 18, 2019]; Accepted [April 2, 2019]*

Published by InnoCon Publishing.
ISSN 2704-4440

somehow introduce different types of errors that would result in unexpected side effects to a program if not properly managed.

Designing programming languages through the combination of the best features is still necessary despite the fact that many features of existing programming languages can be simulated in other languages [5]. There are several programming languages that are designed not only for computer experts but for everyone. These programming languages may have some other goals and their models may not suit the amateur's needs, although there are lessons to be learned by looking at them.

The general objective of this study is to make programming more accessible and ease users in performing the programming task by designing an interactive visual programming paradigm that will aid them in constructing computer programs.

## 1.1 Objectives

The following are the specific objectives of the proposed graphical block-structured programming.

1. To ease and speed up the process of creating and managing the syntactical structure of a program by designing a visual programming environment that is capable of handling and integrating a group or set of graphical tokens to construct a functional program visually.
2. To aid users to manage the functional features of the language by designing a distinctive set of graphical tokens that could easily be identified, distinguished from other sets of components, and efficiently use to suit the designed algorithm.
3. To restrict the occurrences of syntax errors by designing a mechanism that could recognize, semantically analyze and parse the syntactical structure of a graphical program to generate a formal predefined line of instruction in a particular form of a language.

## 1.2 Scope and Limitations

Since the aim of this research is to improve the ways through which the user constructs computer programs through visual programming [6][7]. The following are the scope and limitations of the proposed graphical block-structured programming.

- The target code through which it is converted into actual or formal code is in the form of C language. Although any language could be used as a target code, the C language has become more popular through its structured programming style.
- Just like any other programming language, graphical screen objects are hard-coded and compiled in packages that serve as an ingredient in constructing the program.
- The mechanisms for recognizing, parsing, and generating the actual code are built within an interactive editor to provide immediate visual feedback for users or programmers.
- The external systems software (*i.e.*, Borland's compiler, linker, and external libraries) will participate in the programming tasks as a tool for generating an executable package of a graphical constructed program.

## 1.3 Significance of the Study

This proposed programming paradigm is designed to improve the current ways of constructing structured programs through visual modeling which could be used by anybody to build applications. Users could gain knowledge in learning the concepts of program structures and syntax of a certain language as well as managing the code efficiently. It could ease and speed up the user in creating programs using a range number of features to manipulate data.

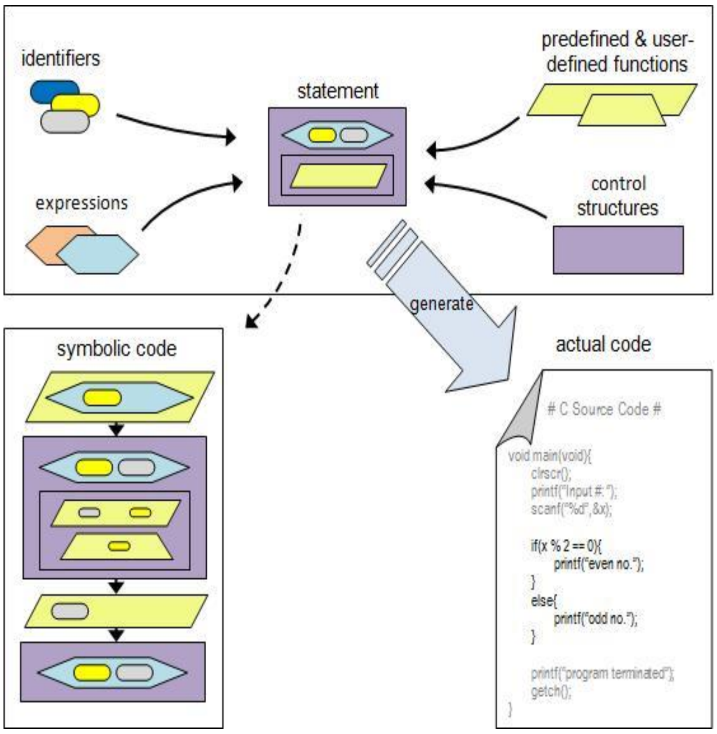
It will also help the user improve the correctness of creating programs without worrying about the grammatical rule of the language. Thus, this proposed graphical block-structured programming reduces the effort in creating programs as well as the cost.

Developers and other researchers can also benefit from this programming paradigm to design or build a more interactive programming environment for other types of programming languages.

The rest of this paper is organized as follows: Section 2 outlines the Extended Backus-Naur Form (EBNF) theory for language recognition and generation; the block-structured programming is presented in Section 5; and the concluding remarks is presented in Section 6.

## 2. EBNF Theory for Language Recognition & Generation

This section discusses the concept and theory of language recognition and code generation of the proposed programming paradigm. Figure 1 illustrates how graphical symbols can be used to construct computer programs through recognition and conversion of symbolic code to formal or actual code (*i.e.*, textual form). In this case, the target is the source code in C language form.



```
# C Source Code #

void main(void){
    clrscr();
    printf("Input #: ");
    scanf("%d",&x);

    if(x % 2 == 0){
        printf("even no.");
    }
    else{
        printf("odd no.");
    }

    printf("program terminated");
    getch();
}
```

**Figure 1.** Logical View of Symbolic Code Transformation

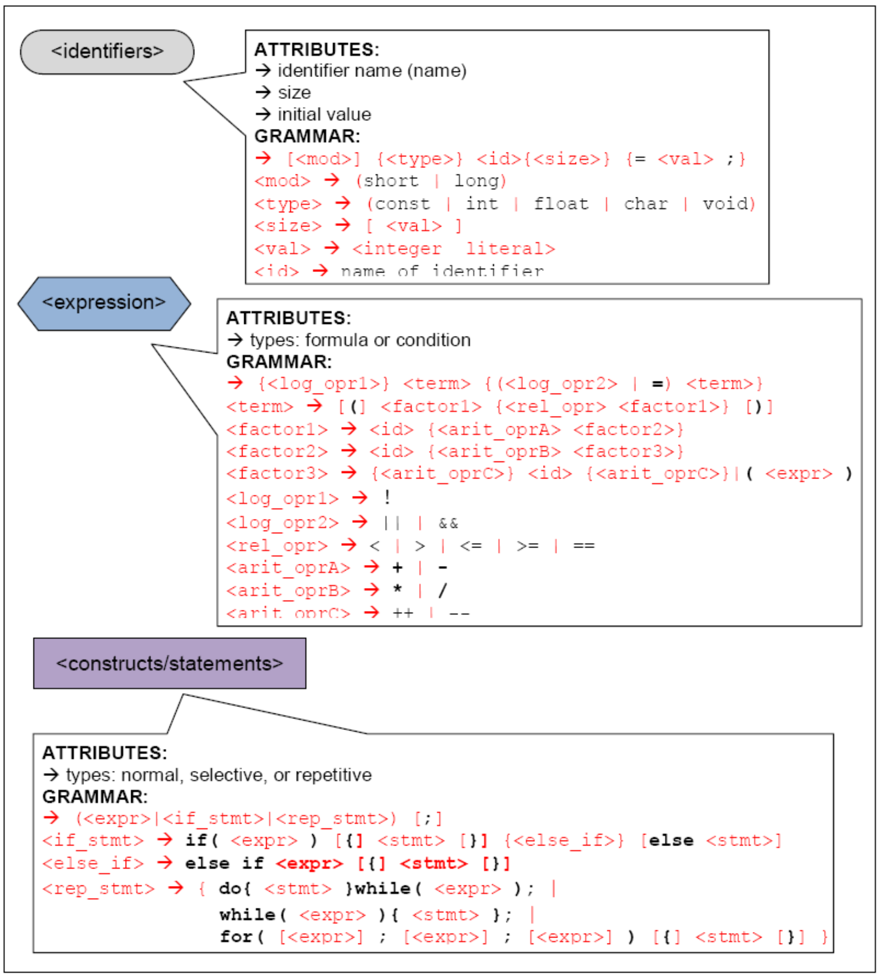
In general, languages can be formally defined in two distinct ways, recognition and generation [8]. Both language recognizers and generators play an important role in the syntax analysis part of a compiler. Language recognizer acts like filters, separating correct sentences from those that are incorrectly formed while a language generator is used to generate the sentence of a language.

The Backus-Naur form (or BNF) is a very natural notation for describing the syntax of computer programs that is nearly identical to Chomsky's generative devices for context-free languages, called context-free grammar. The limitations incurred in BNF have led to its extension in several ways. Most extended versions are called Extended BNF, or simply EBNF [4][9].

In Figure 1, all identifiers, expressions, control structures, predefined and user-defined functions are expressed in graphical tokens and will be used to construct a block of symbolic statements through the drag-and-drop operation. In this way, users won’t have to worry about the syntax of the language since the graphical code is converted to C language each time an object is being placed or removed from its workspace area.

In this study, the arrangement of graphical tokens is responsible for constructing the sequence of a program and is affixed with encapsulated signatures such as its attributes and grammar in EBNF form. These attributes will serve as placeholders for properties like the type and size of an identifier, type of expression, and other supporting properties that a token can hold. The list of grammar serves as a syntactic rule for parsing connected objects to generate the actual code.

Figure 2 describes the contents of some selected graphical tokens used in constructing the code graphically. It is vital that the grammar must be constructed carefully to avoid ambiguous statements that could be generated during the parsing process. This is why grammar holds a list of possible syntactic statements of a program. In addition to this, recursive descent parsing is applied. This is because in many cases, the language being parsed has syntactic units like expressions and statements, so parsing subprograms are often recursive.



**ATTRIBUTES:**
→ identifier name (name)
→ size
→ initial value
**GRAMMAR:**

```
→ [<mod>] {<type>} <id>{<size>} {= <val> ;}
<mod> → (short | long)
<type> → (const | int | float | char | void)
<size> → [ <val> ]
<val> → <integer  literal>
<id> → name of identifier
```

**ATTRIBUTES:**
→ types: formula or condition
**GRAMMAR:**

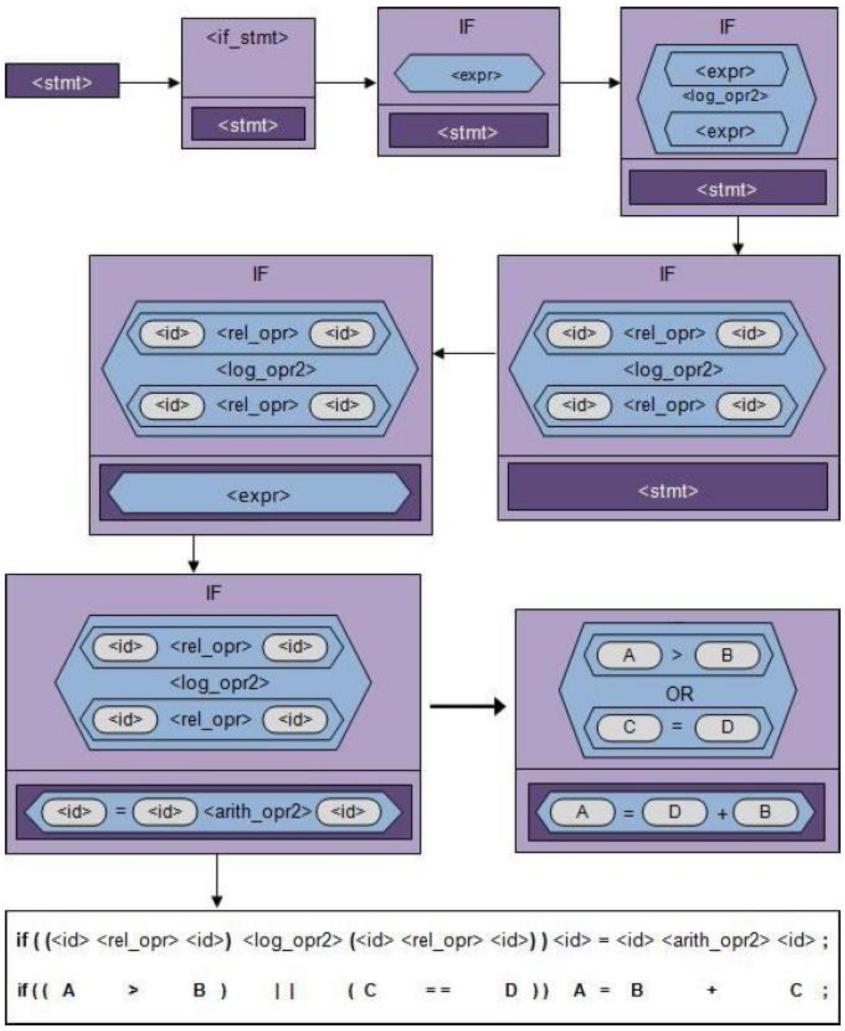
```
→ {<log_opr1>} <term> {(<log_opr2> | =) <term>}
<term> → [(] <factor1> {<rel_opr> <factor1>} [)]
<factor1> → <id> {<arit_oprA> <factor2>}
<factor2> → <id> {<arit_oprB> <factor3>}
<factor3> → {<arit_oprC>} <id> {<arit_oprC>}|( <expr> )
<log_opr1> → !
<log_opr2> → || | &&
<rel_opr> → < | > | <= | >= | ==
<arit_oprA> → + | -
<arit_oprB> → * | /
<arit_oprC> → ++ | --
```

**ATTRIBUTES:**
→ types: normal, selective, or repetitive
**GRAMMAR:**

```
→ (<expr>|<if_stmt>|<rep_stmt>) [;]
<if_stmt> → if( <expr> ) [{] <stmt> [}] {<else_if>} [else <stmt>]
<else_if> → else if <expr> [{] <stmt> [}]
<rep_stmt> → { do{ <stmt> }while( <expr> ); |
                while( <expr> ){ <stmt> }; |
                for( [<expr>] ; [<expr>] ; [<expr>] ) [{] <stmt> [}] }
```

**Figure 2.** Attributes and Grammar of Tokens

How does a graphical code being recognized and generated another form of code? These graphical objects vary their grammar through another type of graphical object that they are connected with. The recognizer performs grammar tracing starting with the outer object. Once another object is transformed or placed within another object, the grammar expands to its appropriate syntactic structure allowing the parser to construct a parse tree for a given input structure as depicted in Figure 3. In this figure, a simple graphical if-statement is parsed and converted into an abstract C language statement.

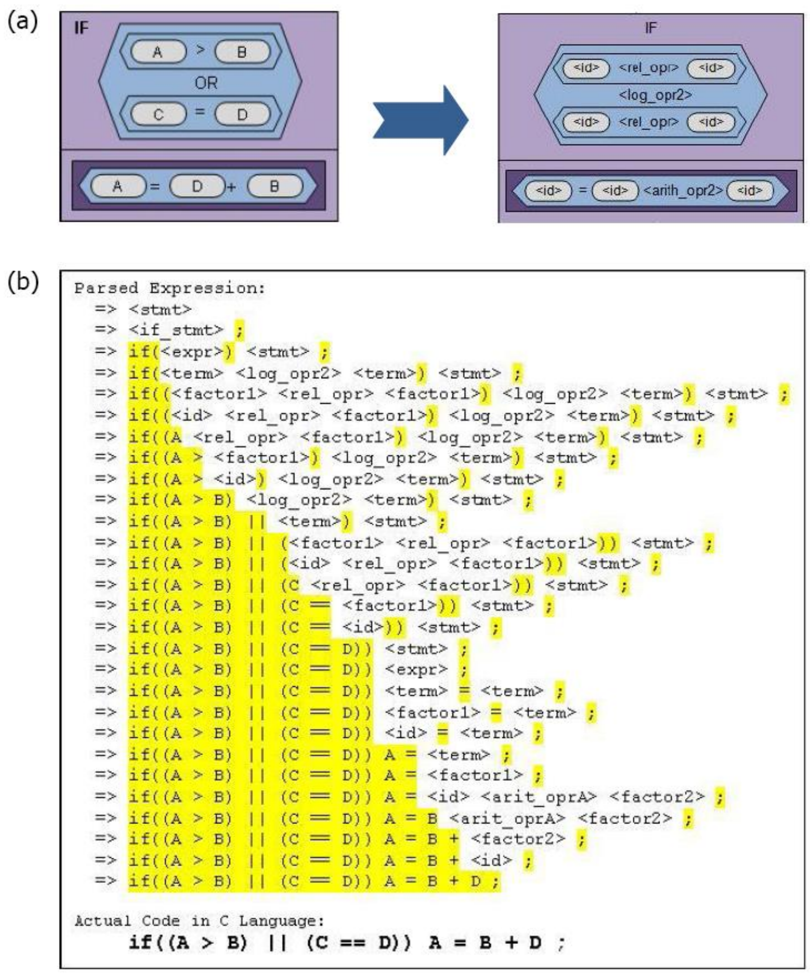


```
if ( (<id> <rel_opr> <id>)  <log_opr2> (<id> <rel_opr> <id>) ) <id> = <id> <arith_opr2> <id> ;
if((  A      >      B  )     | |      ( C      = =       D  ))   A  =  B       +        C  ;
```

**Figure 3.** The logical process of parsing a graphical if-statement

Another view of the if-statement parsing process is illustrated in Figures 4(a) and 4(b). Abstractions in the EBNF description are often called non-terminals while the lexemes (*i.e.*, the lowest level of the syntactic unit) and tokens of the rules are called terminals. Each of the strings of the derivation, including <stmt>, is called a sentential form. In the derivation in Figure 4(b), the replaced non-terminal is always the leftmost non-terminal in the previous sentential form. The order of derivation used in this example is the leftmost derivation. The derivation process continues until there will be no non-terminals contained in the sentential. In addition, aside from the leftmost derivation, a rightmost derivation is performed or in an order that is neither leftmost nor rightmost. The order of derivation has no effect on the language generated by the grammar.

Each time a symbolic object is either added or being replaced by another symbol, the non-terminal of the grammar expands or collapses to an appropriate grammar to generate the right syntax form for the actual code. If an object or group of objects is removed, the corresponding line(s) linking with that object(s) is also removed.



```
Parsed Expression:
  => <stmt>
  => <if_stmt> ;
  => if(<expr>) <stmt> ;
  => if(<term> <log_opr2> <term>) <stmt> ;
  => if((<factor1> <rel_opr> <factor1>) <log_opr2> <term>) <stmt> ;
  => if((<id> <rel_opr> <factor1>) <log_opr2> <term>) <stmt> ;
  => if((A <rel_opr> <factor1>) <log_opr2> <term>) <stmt> ;
  => if((A > <factor1>) <log_opr2> <term>) <stmt> ;
  => if((A > <id>) <log_opr2> <term>) <stmt> ;
  => if((A > B) <log_opr2> <term>) <stmt> ;
  => if((A > B) || <term>) <stmt> ;
  => if((A > B) || (<factor1> <rel_opr> <factor1>)) <stmt> ;
  => if((A > B) || (<id> <rel_opr> <factor1>)) <stmt> ;
  => if((A > B) || (C <rel_opr> <factor1>)) <stmt> ;
  => if((A > B) || (C == <factor1>)) <stmt> ;
  => if((A > B) || (C == <id>)) <stmt> ;
  => if((A > B) || (C == D)) <stmt> ;
  => if((A > B) || (C == D)) <expr> ;
  => if((A > B) || (C == D)) <term> = <term> ;
  => if((A > B) || (C == D)) <factor1> = <term> ;
  => if((A > B) || (C == D)) <id> = <term> ;
  => if((A > B) || (C == D)) A = <term> ;
  => if((A > B) || (C == D)) A = <factor1> ;
  => if((A > B) || (C == D)) A = <id> <arit_oprA> <factor2> ;
  => if((A > B) || (C == D)) A = B <arit_oprA> <factor2> ;
  => if((A > B) || (C == D)) A = B + <factor2> ;
  => if((A > B) || (C == D)) A = B + <id> ;
  => if((A > B) || (C == D)) A = B + D ;

Actual Code in C Language:
     if((A > B) || (C == D)) A = B + D ;
```

**Figure 4.** (a) Skeletal view of the graphical if-statement; (b) Derivation of the parsing process of the given if-statement

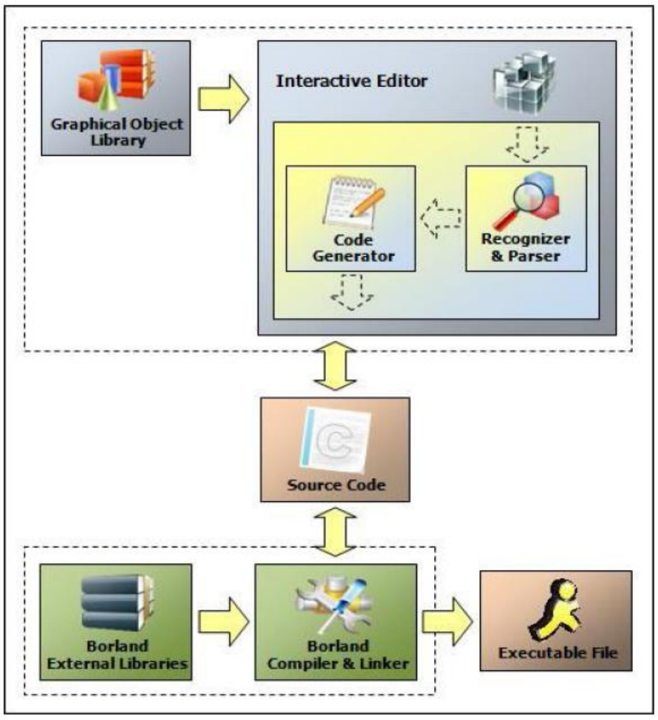


**Figure 5.** Overview Design

Figure 5 illustrates the overview design of the project. The upper dotted box is the scope through which the project is to be constructed. The bottom dotted box is the external system software that will participate in generating the executable file out of the source code that has been generated by the Interactive Editor.

## 3. Block-Structured Programming

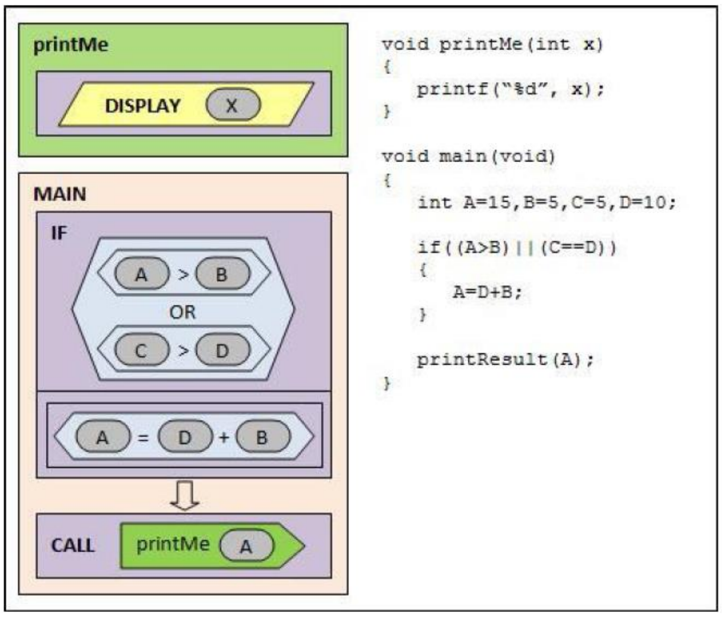
In the early development of high-level languages, the introduction of blocks has enabled the construction of programs in which a group of statements and declarations could be treated as if they were a single statement [10]. This, alongside the introduction of subroutines, enables the complex structures to be expressed by hierarchical decomposition into simpler procedural structures.

Normally, blocks are delimited by the use of keywords or symbols to mark the beginning and end of a particular set of instructions. For example, in Pascal and Delphi, begin and end keywords are used to group the statements. In C/C++, C#, and Java, the special characters { and } (curly braces) are used to mark a block of instructions.

Indentions of a clause also play a role in block-structured programming style. It allows programmers a virtual line of sight to match the delimiters of the clause. Thus, making the program readable, allowing the programmer to quickly determine the program flow and easily work with it by updating the code.

This influenced many programming languages including object-oriented programming languages to support the block-structured programming style. It breaks down a program into a hierarchy of tasks. It enhances the manageability of the problem by decomposing it into the hierarchy of problems. The more visually apparent the blocks are within a program, the easier it is to interpret the program flow, thereby further increasing manageability.

In this study, the researcher came up with the idea of illustrating block-structured programming by embedding the graphical symbols within another symbol. Figure 6 illustrates a sample model of graphical block-structured programming. The user-defined function printMe is separated from the MAIN block. Within these blocks are several graphical components that make up the executable instructions. Variables are embedded within these components and are used as either passing parameters or simply as formulas.



```
void printMe(int x)
{
   printf("%d", x);
}

void main(void)
{
   int A=15,B=5,C=5,D=10;

   if((A>B)||(C==D))
   {
      A=D+B;
   }

   printResult(A);
}
```

**Figure 6.** Example of a Graphical Block-Structured Program

There will be an often trade-off between more powerful software versus more understandable software. While this study is aimed to improve the ways through which the user builds an application through graphical block-structured programming, the researcher decided to make the software more understandable and therefore categorize and shape all complicated structures of the program virtually.

## 4. Conclusion

This paper had presented the proposed graphical block-structured programming as a visual programming paradigm. This programming paradigm breaks down a program into a hierarchy of tasks, thus, the manageability of the problem has been enhanced through decomposing it into a hierarchy of problems. It can be said that the more visually apparent the blocks are within a program, the easier it is to interpret the program flow, thereby, further increasing the program manageability. It has made programming more accessible and eases programmers in performing the programming task by designing an interactive visual programming paradigm that will aid them in the efficient construction of computer programs.

In the future, a sample program development will be simulated with the proposed graphical block-structured programming as a visual programming paradigm.

## References

[1] M. Gabbrielli and S. Martini, "*Programming Languages: Principles and Paradigms*", United Kingdom: Springer-Verlag London Limited, 2010, ISBN: 978-1-84882-913-8.

[2] M. Balaban, "*Principles of Programming Languages*", https://www.cs.bgu.ac.il/~mira/ppl-book.pdf (Accessed December 10, 2018).

[3] McGill School of Computer Science, "*Programming language*", www.cs.mcgill.ca/~rwest/wikispeedia /wpcd/wp/p/Programming_language.htm (Accessed December 10, 2018).

[4] R. W. Sebesta, "*Concepts of programming languages*", United Kingdom: Pearson; W. Ross MacDonald School Resource Services Library, 2015, ISBN: 9781292100555.

[5] Structures of Programming Language, "*Reasons for Studying Concepts of Programming Languages*", https://structuresofprogramminglanguage.wordpress.com/2011/07/04/reasons-for-studying-concepts-of-programming-languages/ (Accessed December 10, 2018).

[6] M. M. Burnett, M. J. Baker, C. Bohus, P. Carlson, S. Yang, P. van Zee, "*Scaling Up Visual Programming Languages*", Computer, vol. 28, no. 3, March 1995, pp. 45-54, ISSN: 0018-9162, https://web.engr.oregonstate.edu/~burnett/Scaling/ScalingUp.html.

[7] X. Shen, G. Wang, J. Gu, X. Dong, "*A Novel Visual Programming Method Designed for Error Rate Reduction*", 2008 International Symposium on Computer Science and Computational Technology, Shanghai, China, December 20-22, 2008, pp. 280-283, ISBN: 978-0-7695-3498-5/08, doi: 10.1109/iscsct.2008.146.

[8] K. Yang, "*Describing Syntax and Semantics*", https://www2.southeastern.edu/Academics/Faculty/kyang/ 2017/Fall/CMPS401/ClassNotes/CMPS401ClassNotesChap03.pdf (Accessed December 10, 2018).

[9] R. S. Scowen, "*Extended BNF — A generic base standard*", https://www.cl.cam.ac.uk/~mgk25/iso-14977-paper.pdf (Accessed December 10, 2018).

[10] Estudies4you, "*Organization for Block Structured Languages*", http://estudies4you.blogspot.com/2017/09/ organization-for-block-structured.html (Accessed December 10, 2018).