[Karunarathna et. al., Vol.6 (Iss.9): September 2018] ISSN- 2350-0530(O), ISSN- 2394-3629(P)
(Received: September 01, 2018 - Accepted: September 29, 2018) DOI: 10.5281/zenodo.1465019

INTERNATIONAL JOURNAL OF RESEARCH – GRANTHAALAYAH
A knowledge Repository

Science

AN EXTENSIBLE APPROACH TO GENERATE FLOWCHARTS FROM SOURCE CODE

Damitha D Karunarathna [*1], Nasik Shafeek [2]
[*1, 2] University of Colombo School of Computing, Sri Lanka

Abstract

Source-code that a developer writes may not definitely make sense to another, the understandability of a source code depends on the proficiency in the language and the logical thinking pattern of the person who has developed the code and who tries to understand it. However, in distributed software development and in software maintenance there is a need to read and understand the source-code probably written by someone else after some time it has encoded. Flowcharts are used to depict the logical flow of processes and can be used as an effective tool in representing the control flow of software programs. This paper presents a novel approach to generate flowcharts from program snippets. It demonstrates that by using an intermediate abstract representation, independent of any programming language, the generation of flowcharts for programs written in any programming language can be achieved. The feasibility of the proposed approach was demonstrated by developing a porotype system of compilers to generate flowcharts for source-codes written in the PHP language.

Keywords: Flowcharts; Compilers; T Diagrams; PHP Programs.

Cite This Article: Damitha D Karunarathna, and Nasik Shafeek. (2018). "AN EXTENSIBLE APPROACH TO GENERATE FLOWCHARTS FROM SOURCE CODE." *International Journal of Research - Granthaalayah*, 6(9), 505-519. https://doi.org/10.5281/zenodo.1465019.

1. Introduction

Programming can be considered both as a science and as an art (Knuth D. E., 1974). Programming languages are built to instruct a computer to perform a sequence of computations. The syntax used by different programming language may vary from a language to language. However, irrespective of the programming language used to code an algorithm, there is only a finite number of abstract constructs that can be used to develop a program. How these constructs are combined to produce the expected result is an art and can be done in many different ways in different programming languages based on the software development maturity and the programming language knowledge of the developer.

The statements of a program define a logical flow of instructions. For a computer what matters is the sequence in which the instructions are to be executed. However, for a developer, who is writing

or reading the code, what is important is the logical flow of the program instructions. It is hard for a developer to conceptualize the logic defined by a program just by reading the code. Even an experienced programmer may find it difficult to understand the logic of a program encoded by him/her at a later time. This is also an inherent problem in distributed software development environments. In a distributed software development environment original author of the code is not the only one responsible in maintaining the code, rather the same code may have to modified by some others.

Visualization is proved to be a good way for understanding a program. Different types of diagrams such as class diagrams, entity-relationship diagrams and Unified Modelling Language (UML) are widely used in application development. Flowchart is a diagram which depicts a complex process by dissecting it into simple steps. Flowcharts are widely used across various disciplines (Wijayasiriwardhane, Wijayarathna, & Karunarathna, "An automated tool to generate test cases for performing basis path testing", 2011) (Wijayasiriwardhane, Wijayarathna, & Karunarathna, "A Method to Generate Test Cases for Performing Basis Path Testing", 2016) (Nassi & Shneiderman, 1973). In Computer Science (CS) flowchart is primarily used to explain algorithms. Thus, flowcharts is an ideal tool for depicting what a particular software source-code is supposed to do. Also, ISO 5807:1985 and BS 4058:1987 has published standard for flow chart drawing (BSI, 1987).

Flowchart representation of a source-code snippet makes it easier for a developer to understand, debug and check the validity of the logic of the code. There are numerous software packages to generate code from flowchart diagrams but there aren't ways to generate flowcharts from source-code. This paper presents a novel approach to generate flowchart for program snippets. The proposed approached is based on the compilers (Aho, Lam, Sethi, & Ulman, 2013) and can be easily extended to generate flowchart for programs snippets written in any programming language. The feasibility of the proposed approach is proved by building a porotype application to generate flowchart for PHP code snippets.

There were very few researches reported in the literature on this domain. Even in the reported literature a generalized approach is not seen on how to translate source-code written in any language to flowcharts. We claim the proposed approach as novel because there was no research found to be done for the translation of source code snippets into flowcharts by constructing a compiler which can be easily extended for any programming language.

One of the applications repoted to generate pictorial representations of source code is AutoDia (AutoDai, 2017). It is designed to generate Unified Modelling Language (UML) class diagrams from source-code written in a selected set of programming languages. The parser of this application searches for specific pre-defined programming language constructs to generating the output on the fly. The output of the sparser is in a proprietary format which is used by the drawing algorithm to generate the class diagrams. The drawing algorithm is tightly-coupled with the application. Since the output generated by the parse does not confirms to a specification, the output cannot be used with any other drawing tools. Also, to extend the application for other source/input languages sub-components have to be developed and the drawing algorithm has to be modified.

An attempt for Automatic Conversion of Flowcharts into language codes by generating program analysis diagrams (PADs) as an intermediate representation is reported in (Xiang-Hu, Qu, & Li, 2012). The authors have identified the basic structures of a flowchart diagram and proposed an algorithm to convert the flowchart into PADs then by using a recursive algorithm into a specific programming language code. This research have identified five control flow structures in any flow chart namely sequence, selection, pre-check loop and post-check loop and multiple selections (Xiang-Hu, Qu, & Li, 2012) which are depicted in Figure 1. They argued that any complex flow chart can be built by combining these five basic control structures.

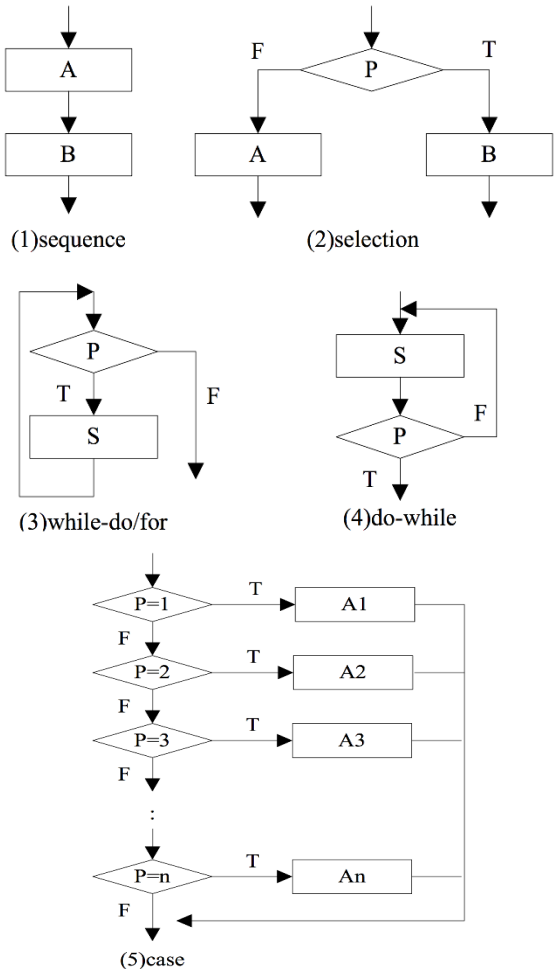

Figure 1: Basic flowchart structures described in (Xiang-Hu, Qu, & Li, 2012).

Lemaitre et al. in (Aur´elie, Harold, Jean, & Bertrand, 2013) presented a technique to recognize handwritten flowcharts by using the structural and syntactic knowledge associated with flowcharts. The authors have described the syntax of flowchart by using a meta-language similar to Backus-Naur Form (BNF).

This research is presented in four sections. In the first section an overview of what is reported relevant to the research provided. In section 2 our proposed methodology is presented. In Section 3 how the proposed methodology can be implemented for program snippets coded in PHP

language is described with examples used to evaluate the prototype implementation. The last section draws conclusions and suggestions.

2. Methodology

An application construction process intend to generate flowcharts for program snippets can take two different approaches.

1) Develop code from scratch to generate flowcharts for programs written in any language.
2) Develop an intermediate representation for flowcharts independent of any programming language, develop a single backend component to generate flowcharts from this intermediate representation and finally develop a frontend translator for each programming language to convert program snippets in that language to the intermediate representation.

Using approach, a) to generate flowcharts for any program language source code by using a programming language like Python can be depicted by using a T diagram as in Figure 1.

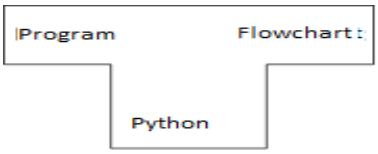

Figure 2: Generating a flowchart for any program using the Python Language.

This is a complex approach and does not allow easy extendibility. Extending of this approach for each new programming language may require complex modification of the code.

Using approach b) for this task offers may advantages over the approach a). Firstly, there are many popular tools available to generate different types of diagrams (Visio, 2018) (Gansner & Ellson, 2017) (Lucidchart, 2018). Thus, there is no need to invest time and money in developing application for data visualization. However, internal representations used by these tools to store data is different. Thus, to visualize a flowchart in the intermediate representation requires a backend component to be developed for each visualization application that uses a different representation of data. Secondly, the approach can be easily extended for program snippets developed in any language. In this case a font end translator must be developed for each programming language.

Using the approach b) to generate flowcharts coded in a programming language such as PHP is depicted in Figure 3.

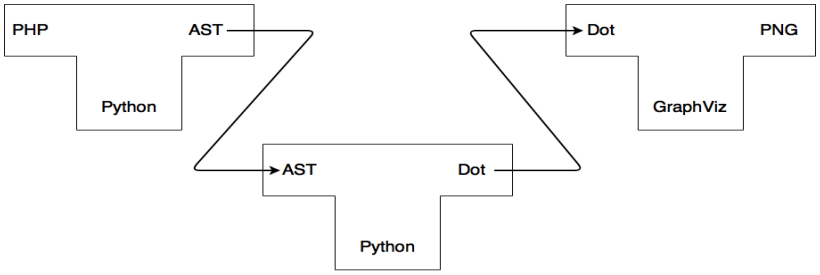

Figure 3: Generating flowchart for PHP programs through an intermediate representation

The application shown in Figure 3 comprises of three compilers. The first compiler, termed as the font-end of the application, converts a given PHP snippets into an Abstract Syntax Tree (AST), which is the intermediate representation of the flowchart. This requires tokenizing and parsing the source-code, which can be done by employing a technique used to develop compilers. Then by using the second compiler the flowchart in AST representation is converted to a representation in Dot Language (Gansner & Ellson, n.d.). Then

An open source visualization software called Graphviz (Gansner & Ellson, 2017) is used as the compiler to generate the final flowchart. The last visualization component is termed as the back-end of this approach. Extending this approach for a different programming language requires only the front-end component to be constructed for the new language. Using a different visualization application to generate flowcharts can also be done easily by developing an appropriate translator as the middle component of the architecture. Thus, this approach can be easily extensible for any programming language and for any visualization application.

Since programs consist of nested code blocks during the generation of the intermediate representation the target code should be analyzed in two phases namely:

- Shallow Analysis
- Deep Analysis

Shallow analyzer, inspired from the shallow parser in compilers, analyzes only the first level of nested code blocks. The shallow analysis does not recursively traverse through the nested code blocks. For example, whilst analyzing an if-else code block it would not look into nested code blocks within the if section or the else section of the code block.

```
1.  <?php
2.  $x = 2;
3.  if ($x == 0) {
4.     if ($y == 0) {
5.       $y = 20;
                          6.  }
7.     $z = $y + 10;
8.  } else {
9.     $y = 10;
10. }
11. echo($y);
```

Listing 1: Example PHP code with nested code blocks

In the example given above as listing 2, the shallow analyzer only identifies the whole if-else block starting at line 3 and ending at line 10. It doesn't evaluate the nested if-else block staring at line 4 and ending at line 6. Thus, the if snippet embedded in the conditionally true block (line 4) of the first if statement in line 3 is not be evaluated by the shallow analyzer. Consequently, the output of the shallow analyzer would be a partially annotated tree of the final AST. This is then required to be passed as input to the deep analyzer.

(Received: September 01, 2018 - Accepted: September 29, 2018) DOI: 10.5281/zenodo.1465019

On the other hand, the deep analyzer evaluates recursively the embedded nested structures within each code block. The output of the shallow analyzer is passed to the deep analyzer which in turn carry out a shallow analysis for each enclosed block followed by a deep-analysis if the code block contains any other code blocks.

Therefor the intermediate representation (AST) used to encode a flowchart needs features to represent nested code blocks recursively.

Any flowchart can be converted to a direct graph data structure where nodes represent the individual items in the flowchart and the arrows represents the flow directions. An example grammar of a language developed to represent the AST of flowcharts (the intermediate representation) is given below. In this AST a program snippet for which a flowchart to be generated is considered as a class.

```
1.  ast = '(', root_node, ')';
2.  root_node = ('"Class"' | "'Class'"), ',', '{', {key_val
    ue}, '}';
3.  key_value = ('"' | "'"), letter, {letter | symbol | num
    ber}, ('"' | "'"), ':', (concrete_values | node), ',';

4.  concrete_values = number | ('"' | "'"), {
5.      letter | symbol | number}, ('"' | "'") | True | Fal
    se;
6.  node = tuple | dict | list;
7.  tuple = '(', ('"' | "'"), node_type, ('"' | "'"), value
    , {',', value}, ')';
8.  list = '[', {concrete_values}, ']';
9.  dict = '{', {key_value}, '}';
10. node_type = 'If' | 'Block' | 'Method' | 'BinaryOp' | 'T
    ernaryOp' |'Input'| 'Output' | 'Else';
```

Listing 2: The grammar of the of a language for AST in BNF

3. Results

The feasibility of the proposed approach was verified by building a prototype to translate source code snippets written in PHP programming language to flowcharts and by evaluating the correctness of the generated flowchart.

The front-end and the middle compiler of the architecture of our prototype was developed by using the tool “Yet Another Compiler-Compiler” (Yacc) and Graphviz software is used as the back-end compiler to generate flowcharts.

The following subset of PHP constructs were used to build our prototype.

- If-else condition
- Switch-cases
- While loop
- Statements (e.g.: variable initialization)
- Function/method calls

ISSN- 2350-0530(O), ISSN- 2394-3629(P)
(Received: September 01, 2018 - Accepted: September 29, 2018)
DOI: 10.5281/zenodo.1465019

```php
<?php
if ($x == 0) {
  $y = 20;
} else {
  $y = 10;
}
```

Listing 3 PHP source-code for standard if else condition.

The experiment designed to evaluate the correctness of the outputs included the following phases: Designing flowchart representation of a logic flow of a program by hand.

Encoding the logic flow by using PHP programming language.

Use the encoded program as the input for the compiler and generating flowchart outputs.
Compare the output generated in step 3 with the initial flowchart generated at step 1 manually, for correctness.

The experiment listed out were performed across number of standard and mixed programming language constructs as described in the following sections.

Test 1: If-Else Construct

An if-else construct comprises a single if block and a single else block. Either of these two blocks can be optional. A manual flowchart and PHP code constructed to represent an if-else construct are shown in Figure 2 and Listing 3 respectively and the flowchart generated by the prototype implementation is shown in figure 3.

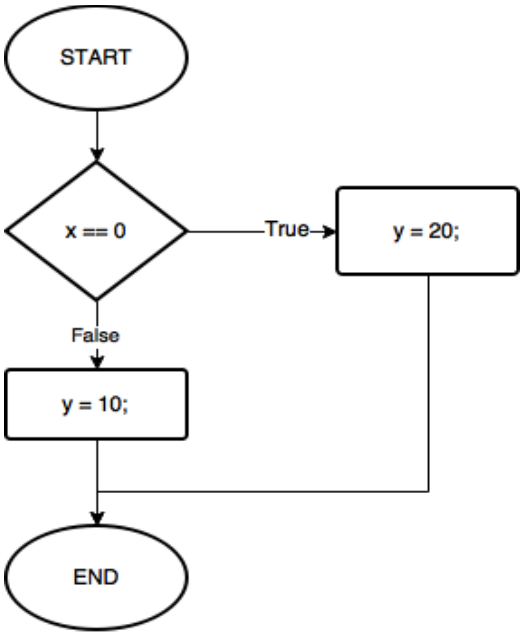

Figure 2: Basic if-else statement drawn beforehand

ISSN- 2350-0530(O), ISSN- 2394-3629(P)
DOI: 10.5281/zenodo.1465019

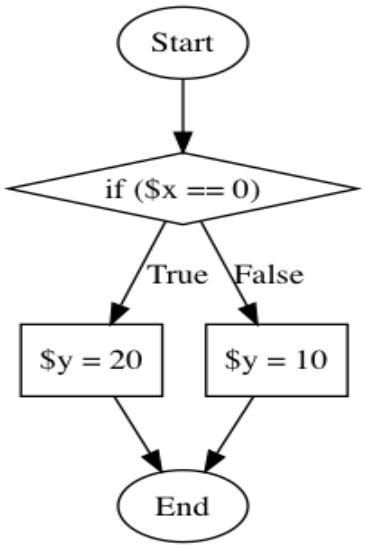

Figure 3: Translated flowchart from the source-code snippet given in Listing 5.

Test 2: Nested If-Else Construct

Nested if-else construct comprises of a embed if-else construct within the if block or else block of a basic if-else construct. A manual flowchart and PHP code constructed to represent a nested if-else construct are shown in figure 4 and Listing 4 respectively and the flowchart generated by the prototype implementation is shown in figure 5.

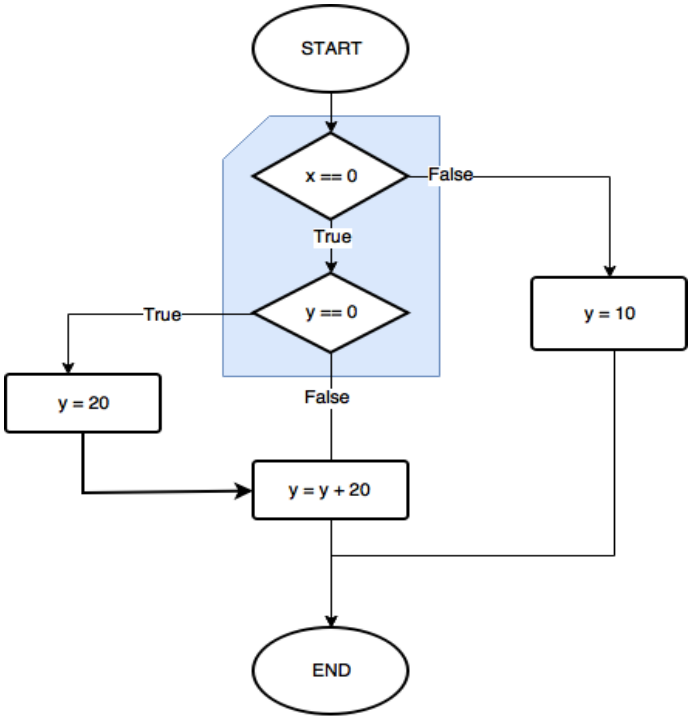

Figure 4: Modelled flowchart for nested if-else conditions.

```
1. <?php
2. if ($x == 0) {
3.   if ($y == 0) {
4.     $y = 20;
5.   }
6.   $y = $y + 20;
7. } else {
```

Listing 4 PHP source-code for standard

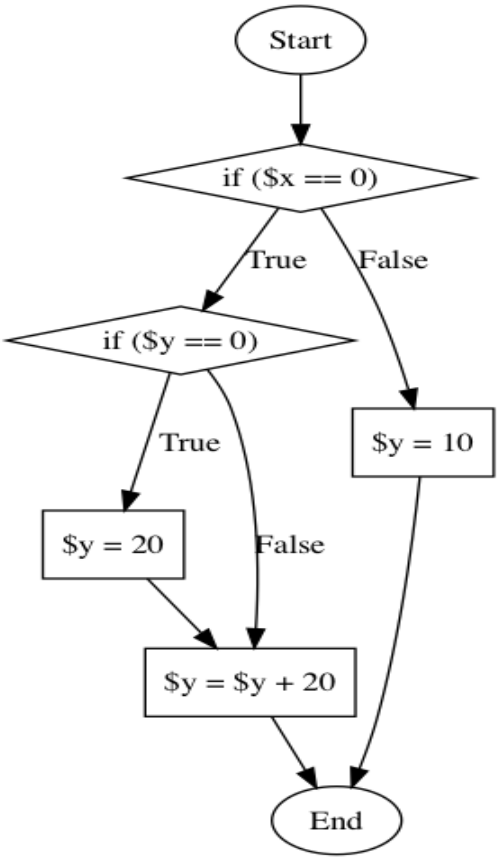

Figure 5: Generated flowchart for the code-snippet in Listing 6.

Test 3: Switch construct

Switch construct can be considered as an extension of nested if-else construct. It typically comprises of multiple logical tests and actions associated with each test. Switch-case can be represented in the flowcharts using a series of if-else constructs. A manual flowchart and PHP code constructed to represent switch construct are shown in figure 6 and Listing 5 respectively and the flowchart generated by the prototype implementation is shown in figure 7.

ISSN- 2350-0530(O), ISSN- 2394-3629(P)
(Received: September 01, 2018 - Accepted: September 29, 2018)
DOI: 10.5281/zenodo.1465019

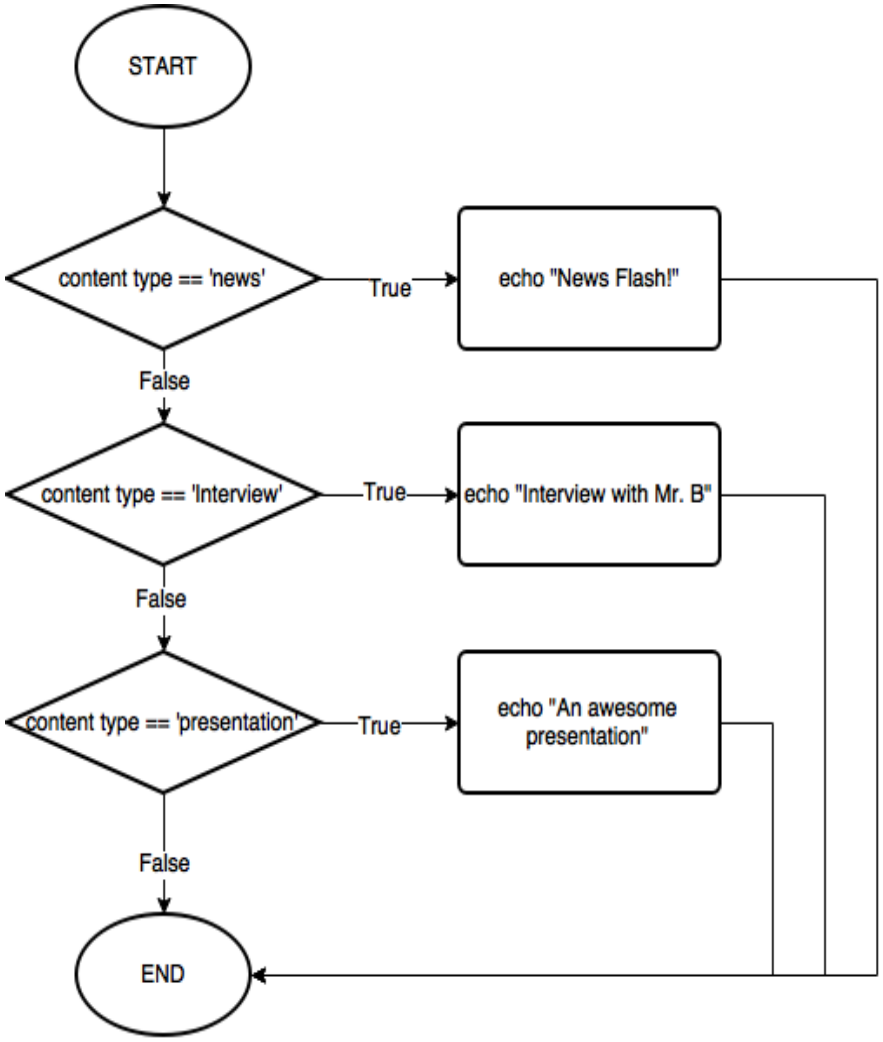

Figure 6 :Typical switch-case construct in flowchart representation.

```
1.  <?php
2.  $content_type = “news”;
3.  switch ($content_type) {
4.    case “news”:
5.      echo “News flash!”;
6.      break;
7.    case “interview”:
8.      echo “Interview with Mr. B”;
9.      break;
10.   case “presentation”:
11.     echo “An awesome presentation”;
12.     break;
13. }
```

Listing 5: Source-code for standard switch-case construct.

ISSN- 2350-0530(O), ISSN- 2394-3629(P)
DOI: 10.5281/zenodo.1465019

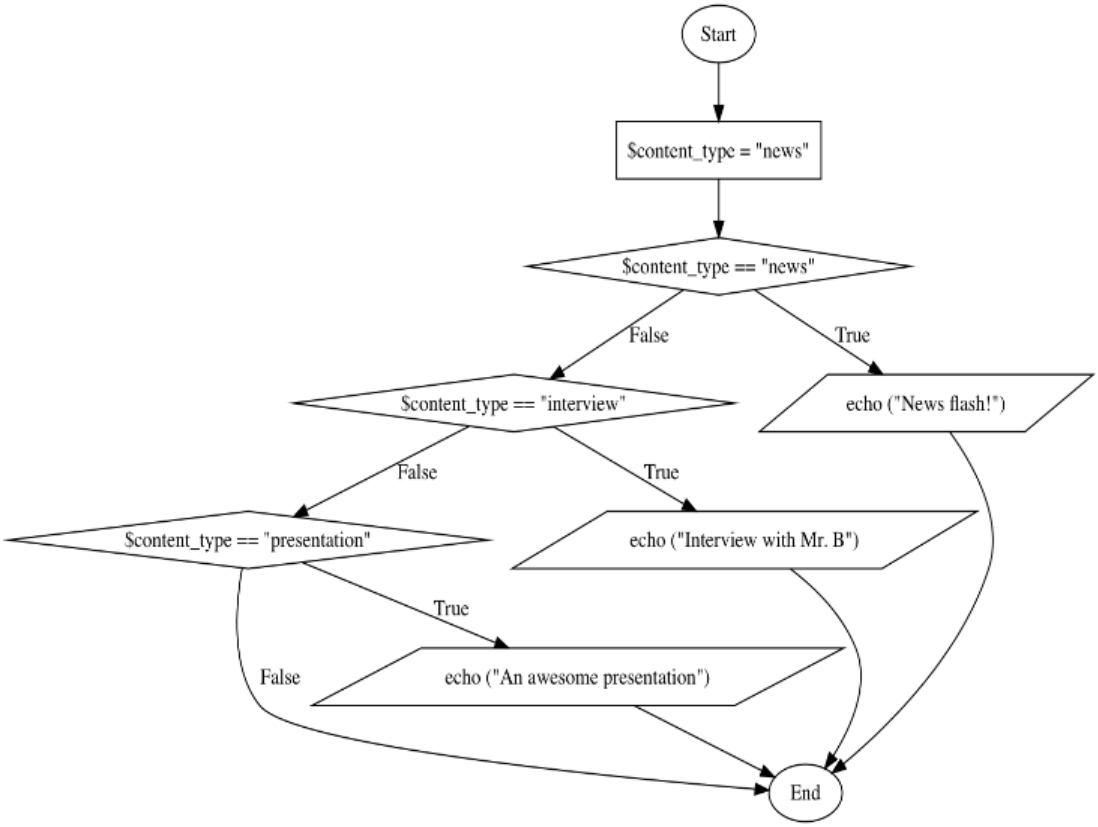

Figure 7: Translated flowchart for Listing 7

Test 4: While Loop Construct

This can be easily represented by using a logical test creating a cyclic to a previous point in the flowchart. A manual flowchart created for a while loop, the corresponding PHP code segment and generated flowchart are shown in fig. 8, listing 6 and fig. 9 respectively.

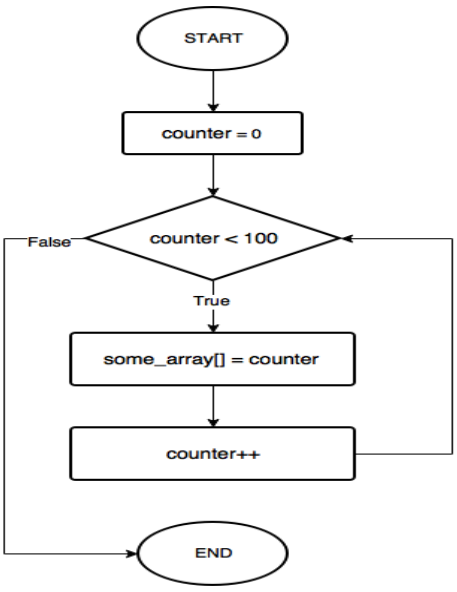

Figure 8: Standard while loop construct in flowchart representat

ISSN- 2350-0530(O), ISSN- 2394-3629(P)
DOI: 10.5281/zenodo.1465019

```php
<?php
$counter = 0;
while ($counter < 100) {
  $some_array [] = $counter;
  $counter++;
}
```

Listing 6 PHP source-code for

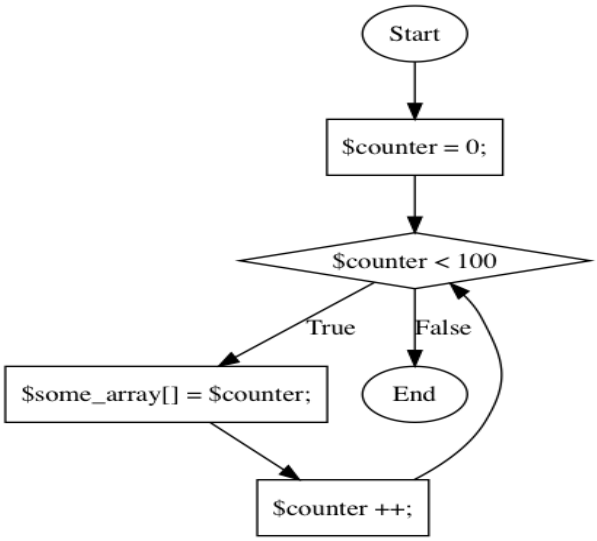

Figure 9:Translated flowchart from the source-code given in Listing 8.

Test 5: Mixed Constructs

A flowchart created by mixing different flow control structures is given in Figure 10. The corresponding source-code written in PHP and the generated flowchart are given in Listing 7 and Figure 11 respectively.

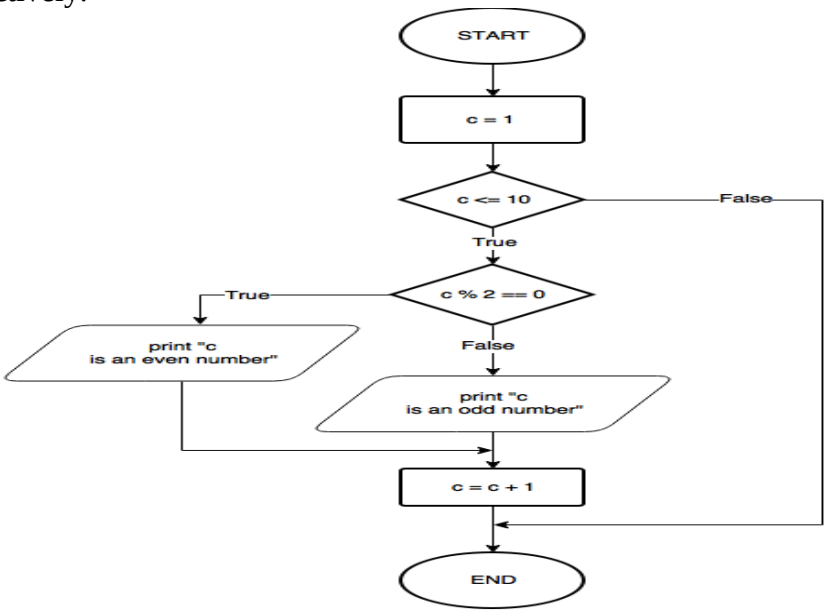

Figure 10: A flowchart comprising a mixture of programming language constructs.

ISSN- 2350-0530(O), ISSN- 2394-3629(P)
(Received: September 01, 2018 - Accepted: September 29, 2018) DOI: 10.5281/zenodo.1465019

```php
<?php
$c = 1;
while ($c <= 10) {
  if ($c % 2 == 0) {
    echo "$c is an even number";
  } else {
    echo "$c is an odd number";
  }
  $c += 1;
}
```

Listing 7: PHP source-code for mixture of basic language constructs.

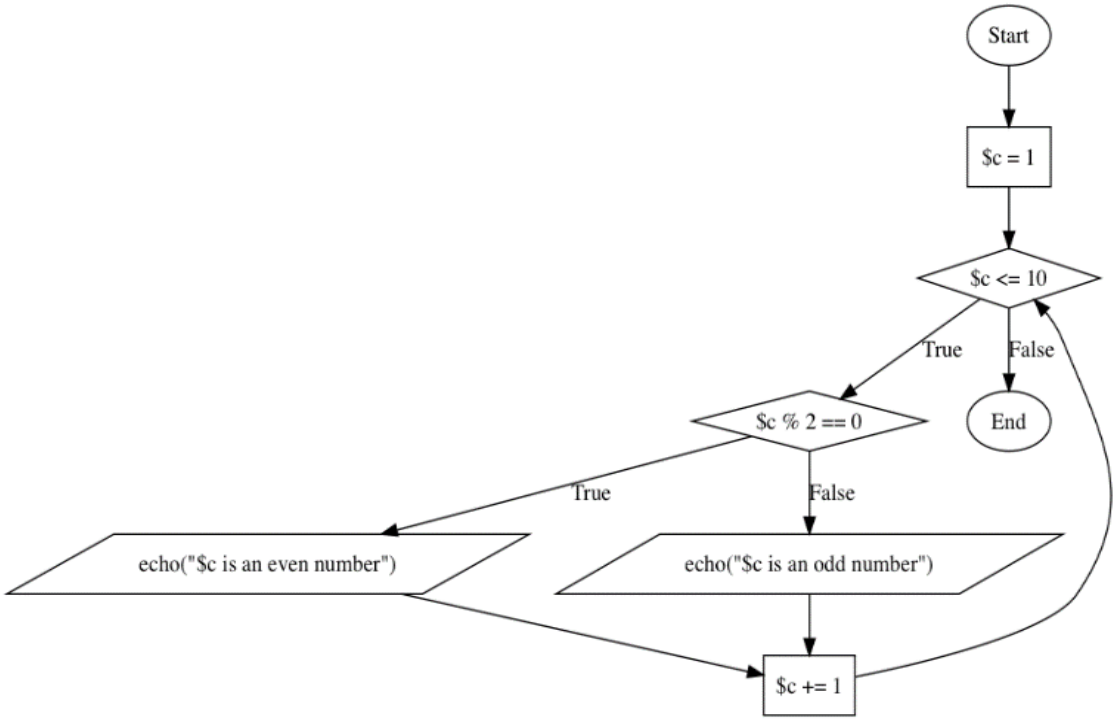

Figure 11: Generated flowchart for mixed constructs.

Table 1 summarized the overall results produced by each of the experiments. This contains the experimentation subject and Boolean values stating whether the flowcharts generated were correct, with regard to the original logic flow and source-code, and whether the flowcharts were valid, with regard to the rules of drawing flowcharts.

Table 1: Overall summary of experiments performed.

Flowchart for section	Is the generated flowchart correct	Is the flowchart valid
If-else construct	True	True
Nested If-else construct	True	True
Switch-case construct	True	True
While construct	True	True
Mixed construct	True	True

(Received: September 01, 2018 - Accepted: September 29, 2018) DOI: 10.5281/zenodo.1465019

4. Conclusion

The experimentations and results verified that the flowcharts are valid and correct. With regard to the experiments and the corresponding results, we can conclude that the proposed architecture is feasible in constructing a compiler to translate source-code to flowcharts.

There are various ways that this research can be extended.

The application can be extended to support various other source languages by writing the appropriate lexers and parsers. The output of the parser should be an AST and it should comply with the specification of AST composed in this research.

Enhancement to the look and feel of the flowchart can be done as a future contribution to the work, either by modifying the GraphViz library or by constructing another library using the Dot language as the input source.

Optimizations of the generated dot language representation hasn't been considered by this research which could be done as an improvement.

The prototype system built to prove the concepts presented only selected set of programming language constructs were used. The rest of the programming language constructs can be made supported by extending the code-generator.

Finally, a User interface (UI) can be implemented for novice developers to use the prototype compiler as a product.

References

[1] Aho, A. V., Lam, M. S., Sethi, R., & Ulman, J. D. (2013). "Compilers: Principles, techniques and tools". London: Pearson Education Inc.

[2] Aur´elie, L., Harold, M., Jean, C., & Bertrand, C. (2013). "Interest of Syntactic Knowledge for On-line Flowchart Recognition". Proceedings of the 9th International Conference on Graphics Recognition: New Trends and Challenges. Berlin.

[3] AutoDai. (2017, 01 22). "AutoDia - Automatic Dia / UML generator". Retrieved 01 22, 2017, from http://www.aarontrevena.co.uk/opensource/autodia/index.html

[4] BSI. (1987). "Specification for data processing flow chart symbols, rules and conventions". BSI.

[5] Gansner, E. R., & Ellson, J. (n.d.). "Dot language specification". (AT&T Labs Research) Retrieved 01 21, 2017, from https://graphviz.gitlab.io/_pages/doc/info/lang.html

[6] Gansner, E. R., & Ellson, J. (2017, 01 01). "GraphViz, Graph Visualization Software". (GraphViz) Retrieved 01 05, 2017, from http://graphviz.org

[7] Gansner, E. R., Koutsofios, E., North, S. C., & Vo, K.-p. (1993). "A Technique for Drawing Directed Graphs". IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, 19(3), 214-230.

[8] Grune, D., Bal, H. E., & G. Langendoes, J. (2012). "Modern Compiler Design". New York: Springer.

[9] Johnson, S. C. (2018, 10). "Yacc: Yet Another Compiler-Compiler". Retrieved 06 30, 2018, from http://dinosaur.compilertools.net/yacc/index.html

[10] Knuth, D. E. (1974). "Computer Programming as an Art". Communications of the ACM, 17(12), 667-673.

[11] Knuth, D. E. (1990). "Literate Programming". Stanford Junior University Press.
[12] Kulkarni, R., Chavan, A., & Hardik, A. (2015). "Transpiler and it's Advantages". International Journal of Computer Science and Information Technologies, 6(2), 1629--1631.
[13] Lesk, M. E., & Schmidt, E. E. (2018, 08). "Lex - A Lexical Analyzer Generator". Retrieved 06 30, 2018, from http://dinosaur.compilertools.net/lex/index.html
[14] Lucidchart. (2018, 10). Retrieved 08 2018, from Lucidchart: https://www.lucidchart.com/
[15] Nassi, I., & Shneiderman, B. (1973, August). "Flowchart Techniques for Structured Programming". SIGPLAN Not., 8(8), 12--26.
[16] Poole, M. (2002). "Compilers, Course notes for module CS 218, 2002". Wales: Department of Computer Science, University of Wales Swansea.
[17] Toal, R. (2002). "Programs, Interpreters and Translators". Marymount: Loyala Marymount University.
[18] Visio. (2018, 10). Visio. (Microsoft) Retrieved 08 2018, from Visio: https://products.office.com/en/visio/flowchart-software
[19] Wijayasiriwardhane, T. K., Wijayarathna, P. G., & Karunarathna, D. D. (2011). "An automated tool to generate test cases for performing basis path testing". International Conference on Advances in ICT for Emerging Regions (ICTer). Colombo, Sri Lanka.
[20] Wijayasiriwardhane, T. K., Wijayarathna, P. G., & Karunarathna, D. D. (2016). "A Method to Generate Test Cases for Performing Basis Path Testing". International Conference on Computer Science Education Innovation & Technology (CSEIT). Global Science and Technology Forum.
[21] Xiang-Hu, L., Qu, M.-c., & Li, Z.-Q. (2012). "Automatic Conversion of Structured Flowcharts into Problem Analysis Diagram for Generation of Codes". Journal of Software, 7(5), 1109--1120.
[22] Zend Community. (2018, 08). "Extension Writing Part-I: Introduction to PHP and Zend". Retrieved 01 03, 2017, from https://devzone.zend.com/303/extension-writing-part-i-introduction-to-php-and-zend

*Corresponding author.
E-mail address: ddk@ ucsc.cmb.ac.lk