Converter to Translate Program Source-Code to Flowchart

A dissertation submitted for the Degree of Master of Computer Science

M. S. Nasik
University of Colombo School of Computing
2018

Declaration

The thesis is my original work and has not been submitted previously for a degree at this or any other university/institute.

To the best of my knowledge it does not contain any material published or written by another person, except as acknowledged in the text.

Student Name: M. S. Nasik

Registration Number: 2015/MCS/050

Index Number: 15440502

.....
Signature: Date:

This is to certify that this thesis is based on the work of

Mr. M. S. Nasik

under my supervision. The thesis has been prepared according to the format stipulated and is of acceptable standard.

Certified by:
Supervisor Name: Dr. Damitha Karunaratne

.....
Signature: Date:

Abstract

Modern software development is so dynamic. Each and everyday a new technology flows in. When it comes to programming languages, numerous extensions, number of new programming languages, and major updates to the existing programming languages are released frequently. Moreover the process of development, along with the new advancement like micro-services architecture and distributed software development methodologies, is very dynamic too. Hence developers have a need for understanding, debugging and modifying source-code written by themselves sometime back, which they may have forgotten, or written by their peers or someone they may not know of. It is hard for developers at such instances to read the code and understand the whole logic behind the code at once.

Flow control is considered the logical flow of software programs. Over the years flowcharts are proven to be standard in depicting the logical flow of a process. It is an effective way of representing the flow control of software programs. It would be interesting if there's a tool which could translate the source-code of a programming language to flowcharts.

A prototype compiler named $\_toFlowchart$ which translates Hypertext Preprocessor (PHP) source-code to flowcharts is modelled and implemented. This is so modelled it can translate the written source-code to flowcharts when the source-code of the language and the corresponding language specification are given. This is a novel approach for solving the problem.

The results produced by the model compiler when implemented with the source-code written by hand were evaluated. The generated flowcharts were proven to be correct representations of the source-code and found to be valid according to flowchart specifications.

This approach in generating flowcharts from source-code using a compiler architecture is found to be first of its kind when referring to the previous work done and the literature reviewed. Though the scope of the compiler implemented was minimal, improvements can be built upon it to widen its scope much further. Moreover, someone interested may follow the same approach in designing a compiler for other visualizations like use-case and class diagrams. The designed and implemented model is referred to as $\_toFlowchart$ throughout this work.

To my
Mom and Dad,
for you are
the brightest light in my darkest nights.

Acknowledgement

Thank God!

I would take the opportunity to wholeheartedly thank my supervisor Dr. Damitha Karunaratne, for you have been both my inspiration and the agent of guidance in every aspect of the research.

I'm so grateful to my mom and dad, for you have always been my pillars of support and agents of motivation. It is my absolute pleasure in thanking my dad for doing the proof reading and providing feedback on the thesis. I also thank my siblings for being kind enough in providing the best advice and motivation throughout the course of the degree program.

I would also take the opportunity in thanking all circle of my friends who have supported and motivated me.

I take great pride and pleasure in thanking the whole academic, administrative and non-academic staff of UCSC. You have given me the one of the best times in my life.

Last but not least, I would like to thank Mr. Yaseer Sheriff and Mrs. Zeenath Hidaya of Abstraction Software (Pvt) Ltd. who have been inspiring me and have supported financially to start and complete the Masters degree program.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

API Application Program Interface. 12, 35, 37

AST Abstract Syntax Tree. 17–24, 33, 35–37, 47, 48

BNF Backus Naur Form. 10, 11, 14, 20, 21

DMOS Description and MOdification Segmentation. 11

DSL Domain Specific Language. 36

EBNF Extended Backus Naur Form. 5, 14, 21, 37, 47

EPF Enhanced Position Formalism. 11

GPU Graphics Processing Unit. 5

GraphViz Graph Visualization Software. 17, 22–24, 37

JVM Java Virtual Machine. 20

LALR Look-Ahead LR. 20

MDA Model Driven Architecture. 8, 14

O/S Operating System. 12

PAD Program Analysis Diagram. 8

PHP Hypertext Preprocessor. ii, 4, 5, 15, 17, 18, 20, 23, 34–36, 40–45, 49, 50

phply PHP Hypertext Preprocessor Python Lex YACC. 35

PL Programming Language. 3–5, 18, 23–25, 40, 45–47

PLY Python Lex-Yacc. vi, 23, 24, 35, 48

regex Regular Expression. 34

SDLC Software Development Life Cycle. 1

UI User Interface. 48

UML Unified Modeling Language. 7, 36

VM Virtual Machine. 20

Yacc Yet Another Compiler-Compiler. 20, 23, 35

Chapter 1

Introduction

"The best programs are written so that computing machines can perform them quickly and so that human beings can understand them clearly. A programmer is ideally an essayist who works with traditional aesthetic and literary forms as well as mathematical concepts, to communicate the way that an algorithm works and to convince a reader that the results will be correct."

—Donald Ervin Knuth

1.1 Introduction

Donald Knuth writes regarding computer programs in his book *Literate Programming*:

> Programming is best regarded as the process of creating *works of literature*, which are meant to be read. Literature of the program genre is performable by machines, but that is not its main purpose. The computer program that are truly beautiful, useful, and profitable must be readable by people. So we ought to address them to people, not to machines. All of the major problems associated with computer programming - issues of reliability, portability, learnability, maintainability, and efficiency - are ameliorated when programs and dialogs with users become more literate [1].

1.2 Motivation

The software industry is rapidly changing in every possible aspect. New technologies flow in every day; programming languages, frameworks, patterns, architectures are some. The software development culture is also changing with the Software Development Life Cycle (SDLC). New

trends like DevOps and Microservices architecture are considered more like a culture/discipline. However, though the technologies change; basic constructs and concepts remain consistent over a length of time.

Programming languages are built to instruct a computer to perform computations. The syntax may vary from language to language; the concept of instructing the computer to perform computations doesn't change. Developers are supposed to write instructions for the computer which are also based on logical constructs - again which doesn't change.

If we take a program code, it is a structural instruction to the computer. It has a predefined logical flow, from statement to statement - either in the order of writing or in the order of execution. Hence for computers what ultimately matters are the instructions, which for the developer is the logical flow of the program. It is hard for the programmers to capture the logical flow of program code at a glance by only reading the code. A visualization of the logical flow would be better. This is highly applicable to distributed software development, where one programmer is not solely responsible for upgrading/modifying the program code he/she originally writes. This is more likely to occur when the developer who tries to modify the existing code has not much experience in that particular programming language - a novice. Moreover, developers find hard to understand what the program code is supposed to do, when they are recruited into an ongoing project. Such considerations motivate to find a way out for the developers and programmers.

1.3 The Problem

Visualization of logical flow is proved to be a good way of understanding the software program. Hence people used to construct visualizations of the logical flow and then write the software based on those visualizations. Flowchart is one such visualization of logical flow which is applicable for a vast majority of disciplines. It is pretty much the standard. The syntax of it is highly simplistic, which makes the language unique; it is so powerful that you can ideally conceptualize any logical flow with it.

As stated above, flowcharts can be used to construct logical flow of software programs. Flowchart makes it easier for the developer to understand, debug and check the validity of the code for the developer. There are numerous software packages to generate code from flowchart diagrams. But there aren't ways to *generate flowcharts from source code*. What if we model a solution to convert source code to flowcharts; which could facilitate the software developer in understanding, debugging and validating the code?

Though flowchart is a diagramming tool, it can be represented in a directed-graph like syntax (See details on: Section 3.3). As argued above, the proposed solution is to model a novel methodology in converting source code to flowcharts. The problem at hand is kind of a translation/transformation problem. A programming-language translation problem is typically solved using a *compiler*. As the problem at hand is translating program source code to flowchart, we propose the solution to imitating the compiler architecture in compiler theory.

Currently, there are online prototype tools for generating flowcharts from the source code. But their implementation and algorithms aren't available. There is also no research on generating flowchart from program code, but there are journal papers available to generate code from

flowcharts (the reverse of this research). "Recently, there are some reports about the automatic generation of code from flowchart. However, these researches all have certain deficiencies, and the core algorithm and technologies are not public, so the accuracy and validity are hard to be convinced. More researches, such as 'AthTek Code to Flowchart', 'Code to Chart', 'Auto Flowchart' etc., are just it's reverse engineering, that is automatic generation of flowchart from code [2]." This research is not a reverse engineering of the products available or the research done but an innovative, novel methodology to solve the problem. This project will typically be aiming to construct a translator, which will complying with the generic architecture of a compiler, named $\_toFlowchart$, converting source code to flowcharts. Moreover the translator doesn't reflect the underlying emphasis on the architecture, consequently the solution built, hence the term compiler is used throughout the thesis. The argument is supported with references in Section 3.1.

1.4 Objectives

Main objective of the research is to evaluate the feasibility of constructing a compiler to translate programming language text to flowcharts. We can structurally list the objectives as following:

- Construction of the translator in the form of a compiler, which takes program code as input and produces flowchart as outputs.
  - Will contain 3 parts – the parser, semantic analyzer and code generator.
  - The source code of the translator will be made open-source for future researchers and developers.
  - In the process of constructing the translator, mappings of Programming Language (PL) constructs to flowchart will have been compiled.
- Building the Ontology for the process of translating program code to flowcharts.
  - In the original project proposal it was proposed to generate a process ontology on the knowledge gained by translating source-code to flowcharts, as an objective of the research. Since the knowledge was very specific and of very minute generalizability, the idea of generating the ontology was dropped.
  - Because of the inability to generate a process ontology, the knowledge of the translation was derived into a mapping of the abstractions in the process of translating generic PL constructs to flowchart snippets.

1.5 Scope

Constructing a compiler itself is a hard job. A typical compiler consists of many modules. The basic modules used to build a compiler is used to build the proposed compiler as well. Here as this research focuses on the proof of concept and given the limited duration, it was decided to use only the essential modules and build the compiler to prove the feasible nature of implementation of the concept. The modules used are:

- Parser
- Semantic analyzer (which is the core)
- Code generator (flowcharts generator)

Program code can be written in many languages and there are many programming paradigms too. As at present, it is impossible to generalize the compiler (parser) to support multi-paradigm languages hence it is decided to support only object-oriented languages. An object-oriented language is chosen because it enforces fair amount of structure in the code on the developer. It is also hard to construct the compiler to support multiple languages within the given time frame of the research – so the compiler is constructed for program code written in PHP.

PHP is chosen (amongst other languages) because it supports all object-oriented features and the language I'm most familiar with. It is true that PHP is not fully object-oriented like Java or C++, but given the time frame of the project, it is implausible learning up a new language, understanding the constructs of that particular language and building up a new translator to convert that language code into flowcharts. Moreover, Java or C++ ecosystem is huge, with various tools for building and compiling and so forth, on the contrary PHP is simple but offers all object-oriented features. The compiler requires only the structure of the source-code written to be of object orientated; not the internal representation.

As of the time constraints, it is impossible for us to go through all possible PL constructs of PHP, we are only looking at a selected subset of PL constructs. The subset of PL constructs that is being built to prove the concept are:

- If-Else condition
- While loop
- Statements (eg: variable initialization)
- Function calls

We have omitted some intricate/special language features in constructing the compiler – as this is only a proof of concept and hence not worth investing time on the specifics of a particular language feature which cannot be generalized. Thought the final system (compiler) is functional, it isn't a full-fledged compiler rather it is the bare minimum to prove the concept.

As of the prospective project proposal, it was proposed to construct an ontology on the 'process of translating' programming language code into flowcharts (a process ontology). As of the time constraint and the direction in which the research headed, the ontology was felt both impossible and irrelevant because of failing to derive necessary abstractions over the process of translation. Alternatively the mappings from generic programming language constructs to flowcharts are compiled and given in the Chapter 3.

1.6 Research Contribution

This research is expected to contribute towards this area of knowledge in numerous ways.

The compiler/converter constructed uses a novel methodology in solving the problem, this methodology can act as a new frontier for future researches in translating programming language source-code to flowcharts - or to diagramming tools in general.

The compiler/converter is constructed in a highly generalized form, in such a way that the corresponding flowchart is constructed when the grammar of the programming language and source-code written in the programming language are provided. This architecture is unique which can be used for translation of any language to corresponding flowcharts, given that the grammatical rules of input languages are provided. This compiler itself can be an extended enormously to support a variety of source languages and output diagrams/languages like Java, C++ and class diagrams respectively. This architecture is originally inspired from compiler research in Graphics Processing Unit (GPU).

Although the proposed generation of ontologies were dropped out, we've built abstractions over the translation of basic PL constructs to flowcharts. These are mappings from basic PL constructs to flowchart snippets.

1.7 Thesis Synopsis

Through this section we shall have an overview about how the thesis is organized.

1.7.1 Chapter 2: Background

Starting with a very basic introduction about flowcharts and the problem at hand, we'll be discussing about the work already done which are directly or indirectly related to the research at hand. Here a software tool and four literature articles are reviewed. Finally a summary on how these work of software craftsmanship and literature are relating to this research is compiled.

1.7.2 Chapter 3: Methodology

The crucial chapter of the thesis contains in-detail discussions about the $\_toFlowchart$ prototype compiler. The concept, design, design alternatives, the algorithm and the system architecture are discussed in-detail. The compiler is further divided into 3 sub-compilers, each of which is responsible to translate a text to another, collectively translating PHP to Dot language representations. Responsibilities and details about each sub-compilers are discussed here. Moreover, an Extended Backus Naur Form (EBNF) specification for the intermediate representation is produced in this chapter.

The next sub-part of the chapter provides a compilation of abstractions of typical programming language constructs to flowchart mappings. Each typical programming language constructs' code snippets is followed by their corresponding flowchart representations.

1.7.3 Chapter 4: $\_toFlowchart$ Implementation

The implementation details about the prototype compiler is discussed in detail here. Firstly, the high level architecture is recapped. Then each component of the prototype compiler is described for its choice of technology, provided necessary arguments for the choice. Finally, the usage of the compiler being constructed is described, followed by a note for running this on Docker environment.

1.7.4 Chapter 5: Results

The indicator to evaluate is identified in the introduction, then the scope of the compiler constructed is recapped, followed by the evaluation strategy. The prototype compiler is evaluated for a series of cases, following the evaluation strategy specified earlier. Finally a summary on the overall results is provided using a table.

1.7.5 Chapter 6: Conclusion and Future Work

After evaluating the results obtained, we can conclude we are successful in exploring the feasibility of constructing a prototype compiler to translate source-code to flowcharts. Moreover, the improvements that can be done on the work done through this research as future improvements are suggested.

Chapter 2

Background

"Those who can imagine anything, can create the impossible."
—Alan Mathison Turing

2.1 Introduction

Flowcharts are immensely used across various disciplines catering to various purposes. In our context, control-flow of source-code needs to be visually represented using the flowchart. There are very few researches done on this domain but most of them have not comprehensively studied translating source-code to flowcharts. So it was difficult to find previous studies aiming to solve the very same problem.

The works of software craftsmanship and works of literature chosen are directly or indirectly contributing to the research at hand. Valid points from each literature, which could potentially contribute to the study are highlighted. Some constructive comments on some literature are also provided.

2.2 AutoDia Review

AutoDia [3] is a software product crafted to generate Unified Modeling Language (UML) diagrams from source code. This is a Perl based open-source project, hence it was easy to evaluate what the program does even though the documentation was not complete.

- Main objective is to convert a source code to UML class diagram. Does so for multiple source/input languages.

- This achieves the goal by output-specific (UML class diagram) parsing of the source file. Which does not give the freedom for expansion through various output formats. This searches for specific programming language constructs, thereby creating a separate parser and implementation; which are very specific or targeted for UML class diagrams.
- Moreover the drawing algorithm are inseparable from the parser and the implementation classes themselves.

As far as the research and experimentation done, I have conceived a mental model about the approach it takes for the process of translation. The approach is, at the very time of parsing the corresponding target codes are generated. This highly relies upon the source language and the target (UML diagrams) implementations, and it supports multiple languages by building a parser for each language which is also responsible for generating the final output.

As specified earlier, the software product is specific about the output it produces and even more specific about the input languages; it uses its own set methodologies leaving the developer not much flexibility for extension. Moreover as we are trying to produce flowcharts, the software product doesn't allow any such extension so far.

2.3 Automatic Conversion of Structured Flowchart into Problem Analysis Diagram for Generation of Codes

The paper tackles a similar problem like the one we are trying to solve. The authors identify the basic structures of a flowchart diagram, then propose an algorithm to convert the flowchart into a program analysis diagram; which is then used as an input to the recursive algorithm which converts into specific language code.

> Flowchart is independent of any programming language. The structured flowchart can be further divided into five kinds of structures: sequence, selection, more selection, pre-check loop and post-check loop. Any complex flow chart can be built by the combination or the nest of the five basic control structures.
>
> PAD is the acronym for Problem Analysis Diagram. It is made by Japan Hitachi, evolved by flowchart. It has now been approved by ISO. Its advantage is clear, intuitive, and the order and hierarchy of program can be a good show. We can say that if the flowchart is a one- dimensional, PAD is then two-dimensional. A lot of people use PAD for system modelling at present in China and other countries [2].

The inspiration behind the paper is Model Driven Architecture (MDA) in software development processes. The question I can't seem to answer from the paper is why is there the need to convert flowcharts into Program Analysis Diagram (PAD). Flowchart is pretty much the standard in visualizing the control flow. PAD adds more visually concise syntax to the diagram, it will make no added advantage in the process of compiling to source-code. The semantics remains the same. If its the case that PAD is used to model the control flow of the program and convert that to flowcharts which ultimately converts into source code; it would have made sense. But here PAD is ultimately used to compile into program code.

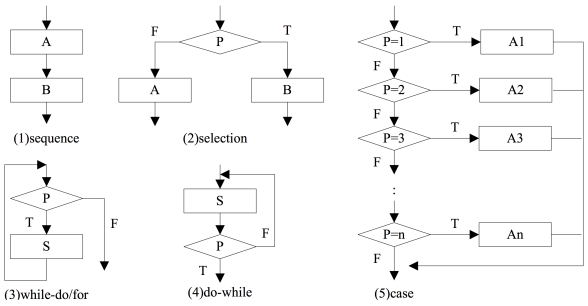

Figure 2.1: Basic structures of flow chart [2].

How the authors have come up with the conversions of basic programming language constructs; like if-else, sequence, switch-case; are quite remarkable. “In order to make flowchart model more clear and intuitive and unambiguously, in addition to the order structure, the remaining four structures all use a judge node, when the executions exit their structures, the page reference primitive (‘o’) must be used. It is called ‘on page reference’ in Visio, and in this paper, it is called convergence [2].” The usage of ‘on page reference’ or the convergence symbol is worth mentioning, which indeed acts like the termination of a program code block - similar to ‘$\{\ldots\}$’ in a for-loop or if-else block.

2.4 Flowchart Techniques for Structured Programming

This paper is very old, but it states some very valid points of flowchart diagrams. Hence it is included into this review. The paper apparently is written with the motivation of structured programming without goto statements. Hence this tends to talk about the complexity of GOTO programs, but which is not of much importance to this research.

> Certain control structures in programming languages, such as iteration, have no direct translation to flowchart language and must be built from simpler control structures, thereby losing the forest in the trees. On the other hand, the power the unrestricted GOTO affords presents problems in logical analysis of programs and program verification, optimization, and debugging. The translation from flowchart to computer program is a one to many relationships whose output ranges over programs only some of which are legible, concise, and efficient.
>
> Top-down programming as defined by Mills is the technique of analyzing an idea to form simpler ideas, and recursively applying the technique. These ideas may take the form of programs, subroutines, macros, lines of code, or other modular forms. Dijkstra’s structured programming organizes program components into levels which he calls pearls, and strings them together into a necklace [4].

The author then goes on to constructing flowchart representation based on structured programming language constructs. The advantages he claims are plausible clauses - some of which are, the scope of iteration is well defined and visible; the scope of local and global variables is immediately obvious; recursion has a trivial representation. But I felt the representation he’d come up with is adding unnecessary complexity. The flowchart in its own stands out for its simplicity.

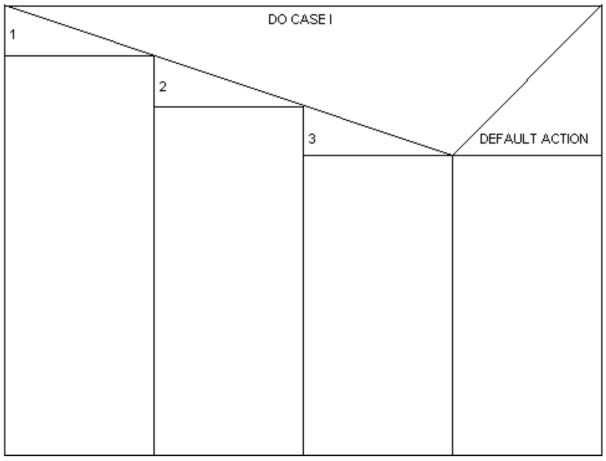

Figure 2.2: Representation of switch case block [4].

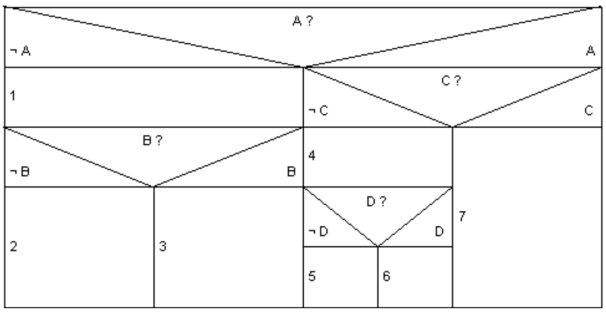

Figure 2.3: Representation of flowcharts [4].

2.5 Syntactic Knowledge for On-Line Flowchart

This paper speaks about the recognition of handwritten flowcharts, but the knowledge can be used in generating of flowchart as well. This paper more or less tries to give an in-depth structural specification to flowcharts. Backus Naur Form (BNF) of a flowchart is synthesized by the author too.

"Some statistical approaches have been proposed in the literature to label and segment the flowcharts. However, as they are very well structured documents, we propose to introduce some structural and syntactic knowledge on flowcharts to improve their recognition [5]."

The author highlights one of the main point here regarding the flowchart, the order of reading and analyzing the document must be adapted from the content of the document - main orientation of the flowchart may be left to right or top to bottom or the reverse of either; it should be read in the flow of arrows or the direction of arrows.

Then the author discusses identifying handwritten strokes and the like, which is not relevant to the research at hand.

The author moves on to Syntactic knowledge on flowcharts which is the crucial part of the paper and very much related to the research at hand. First introduces the symbols available in flowcharts:

The flowcharts are used to describe algorithms or processes. They are made of

> different symbols such as circles, rectangles. . . Arrows are used to represent the control flow. Some text is present inside the symbols or close to the arrows. The terminator and the starter can be described as circular shapes, and more particularly oval or circles. The process, the data and the decision are described as specific quadrilaterals: rectangle, parallelogram and diamond. The arrows are made of a succession of line segments, possibly ended by a pointer [5].

The author has synthesized the grammatical rules of flowcharts into a language called Enhanced Position Formalism (EPF) which is then used to generate the relevant parser using the Description and MOdification Segmentation (DMOS). I believe this is a good start at laying out the syntactic rules of flowcharts. The author has done it well. Below is the excerpt in which he explains the syntactic structure of flowcharts using the grammar rules. This can act as a good input in the research to generate the BNF or the grammatical rules in the necessary format of flowcharts which are very relevant to this research.

> In this work, we propose some syntactic rules that enable the grammatical analysis of flowcharts. As a flowchart always begins by a connection or a terminator, we propose two ways to StartDiagram:

```
StartDiagram :: terminator, arrow, RestOfDiagram.
StartDiagram :: connection, arrow, RestOfDiagram.
```

> These two rules call the analysis of the following symbols of the diagram, using RestOfDiagram. Three kinds of symbols can continue a diagram: process, data, decision. They are followed by one or two arrows (for decision). So, these three rules recursively call RestOfDiagram.

```
RestOfDiagram :: process, arrow, RestOfDiagram.
RestOfDiagram :: data, arrow, RestOfDiagram.
RestOfDiagram :: decision,
                 arrow1, RestOfDiagram1,
                 arrow2, RestOfDiagram2.
```

> The RestOfDiagram can also be the end of the diagram if we meet a terminator, a connection, or an element that has been seen before in the analysis, in the case of a loop.

```
RestOfDiagram :: terminator.
RestOfDiagram :: connection.
RestOfDiagram :: seenBeforeElement.
```

> These rules that we have proposed simply describe the syntax of our flowcharts. We have realized a global description that ensures a global consistency in the recognition of the whole flowchart [5].

2.6 Transpiler and it's Advantages

Though this paper contrasts Transpilers - source-to-source compilation - vs Compilers - source to machine executable code/object code compilation, the phases of the transpilation process are outlined. The phases are:

1. Lexical Analysis - "In this phase, each character is read and grouped into meaningful sequences called lexemes or tokens [6]."
2. Syntactic Analysis - "This is the phase where the lexemes are parsed and major constructs of the language are recognized. These constructs are then represented in the form of a tree structure called a syntax tree [6]."
3. Semantic Analysis - "Using the syntax tree the semantic analyzer checks for semantic consistency. For example, to check whether all the variables in an equation are of the same type [6]."
4. Intermediate Code Generation - "For every construct defined, there is a routine which is executed when the respective construct is recognized during the syntactic analysis [7]."
5. Machine Independent Optimization - "The aim of this phase is to optimize the intermediate code so that the resultant target code that is generated will be better [6]."
6. Translation - "In this phase intermediate code is translated into the target code with the help of translation schemes [8]."

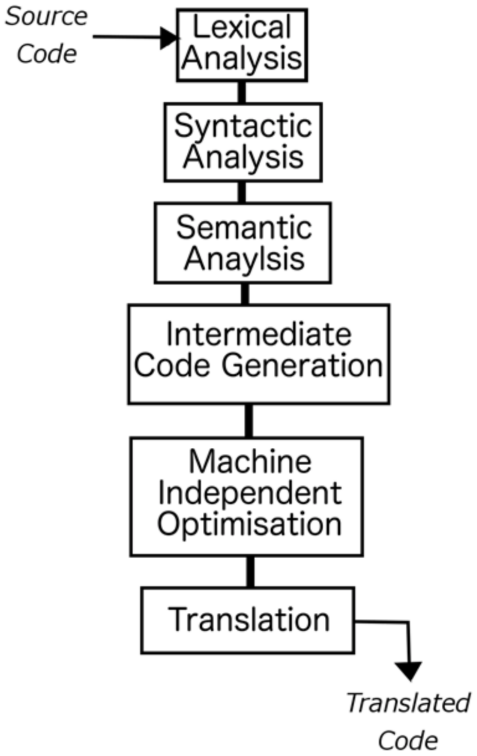

Figure 2.4: Phases of transpilation [8].

Then the author goes on describing a concept called 'Pluggable Target Compiler'. This is explained by elaborating how Java works; "One of the most famous cross-platform programming

languages is Java. It achieves the cross-platform ability by using an executor called the Java Virtual Machine or the JVM. JVM takes bytecode as input, converts it to native code and executes it immediately. JVM has been implemented in almost all major operating systems [8]."

Then the criticism of Java being slow over natively compiled languages like C is posited, and claims that was mitigated with version 7 of Java. But one of the criticisms still lacking even in version 8 is aloofness of Application Program Interface (API) from the native system [8]. This produces difficulty of using the Native APIs of the client Operating System (O/S).

"To solve such issues a transpiler can be made in which the code generator is separate. This code generator can be plugged into the transpiler and can generate native code for the respective operating systems thus allowing easy access to the native APIs [8]."

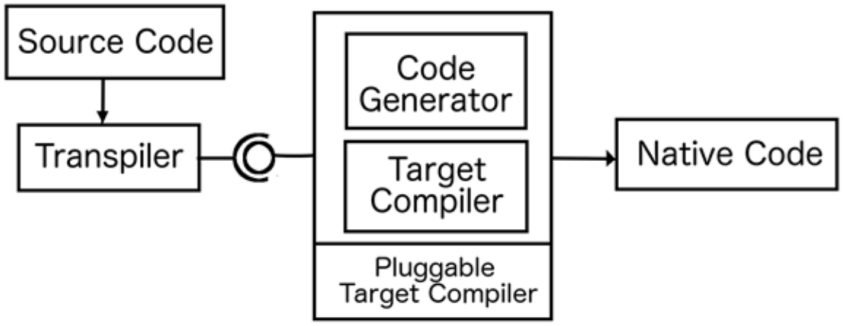

Figure 2.5: Block representation of Pluggable Target Compiler [8].

The code generator and target compiler combined together is called the Pluggable Target Compiler [8].

Then the author goes on explaining further how this compiler can be used:

> Consider two systems, one with Windows 7 and the other with Apple Mac OS X. Windows 7 supports C++ for native code generation and Mac supports Objective-C. Therefore it is needed to create two pluggable target compilers with:
>
> 1. Input: XML/JSON Intermediate
>   Code Output: Objective-C source file
>   Target Programming Language: Objective-C
>   Written in: Objective-C
> 2. Input: XML/JSON Intermediate
>   Code Output: C++ source file
>   Target Programming Language: C++
>   Written in: C++
>
> In these target compilers, rules have to be written to convert the intermediate code into equivalent target language code [8].

The paper is simple and elaborate; the structures and methods of a typical transpiler are put forward nicely.

2.7 Summary

We've reviewed both a software tool and set of literary articles. Through the software tool reviewed under Section 2.2, we can confirm that the methodology taken to solve the problem is not very efficient in terms of extensibility and novelty. Moreover, there were no much other software tools to review which did the job and were available to evaluate with known methodology.

We've also reviewed 4 literary articles, each of which have contributed in learning the background of the research area and sometimes directly affecting decision making process of the research at hand. By reviewing Automatic Conversion of Structured Flowcharts into Program Analysis Diagram for Generation of Codes [2], we've gained the knowledge of flowcharts and their basic structures of flowcharts representing typical programming language constructs. Though the inspiration behind the paper is MDA, which is not of very use to the research at hand, the introduction of the judge node, as specified on [2] can be used as a connective node from a flowchart to another, when there's a need to split the flowchart diagram into more than one. Likewise, the Flowchart Techniques for Structured Programming [4], reviewed under Section 2.4 provides an insight for the $\_toFlowchart$ algorithm. The top-down programming technique for analyzing an idea to form simpler ideas and recursively applying the technique [4] is the motivation behind dissecting the parsing and analyzing steps to shallow analyzer and deep analyzer discussed under Section 3.7.2.

In Section 2.5, we've further expanded knowledge on flowcharts. The EBNF representation for the intermediate language compiled at Section 3.7.1, took inputs from the very BNF rules provided for flowcharts in the Syntactic Knowledge for On-Line Flowchart [5].

Last but not the least, Transpiler and it's Advantages [8], discussed at Section 2.6 provided necessary knowledge in the compiler research area, and making the path ahead clear in selecting the appropriate architecture and methodology for the construction of the $\_toFlowchart$ compiler (which is of the type transpilers). The 'Pluggable Target Compiler' [8] was the initial spark in conceiving the idea of abstracting the 1st sub-compiler (refer Section 3.3), such that the extensibility of $\_toFlowchart$ compiler could be achieved.

Chapter 3

Methodology

"I was never aware of any other option but to question everything."
—Avram Noam Chomsky

3.1 Introduction

We are building a software prototype to translate code written in a programming language like PHP to flowcharts. The solution is a compiler, in generic terms, is a software tool which translates written code/text in one language to another [9]. There are various classifications of compilers; interpreters, transpilers, just-in-time compilers and so on [10]. In the literature, the term compiler is used to denote software programs which are used to convert source-code from a higher-level language to a lower-level language which can be executed directly by the machine, like byte-code or machine-code, without modifying the semantics of the source [6], [9]. By definition a transpiler is a compiler which translates source-code from one higher-level language to another [8], [10]. There is a discrepancy in the software universe for using the term compiler for a transpiler. As in [10], we can technically term a transpiler as a compiler because it belongs to the very class and 'compiler' is a broader term which is generally used for software packages, which translates from one programming language to another.

The generic compiler design follows the typical architecture as depicted in Figure 3.1.

Source-code to flowchart conversion hasn't been done with methodologies that are made available to us. Hence it is hard for us to look into a similar solution for the problem at hand.

The solution is modelled analogous to generic compiler design. The design of the solution which mimics the compiler architecture and the algorithm of the solution are discussed in Section 3.7. The solution being built is given the name $\_toFlowchart$, it tends to reflect the usage of the program, i.e. $\_toFlowchart$ can ideally be used to translate source-code from any language - given the language parser - to flowcharts.

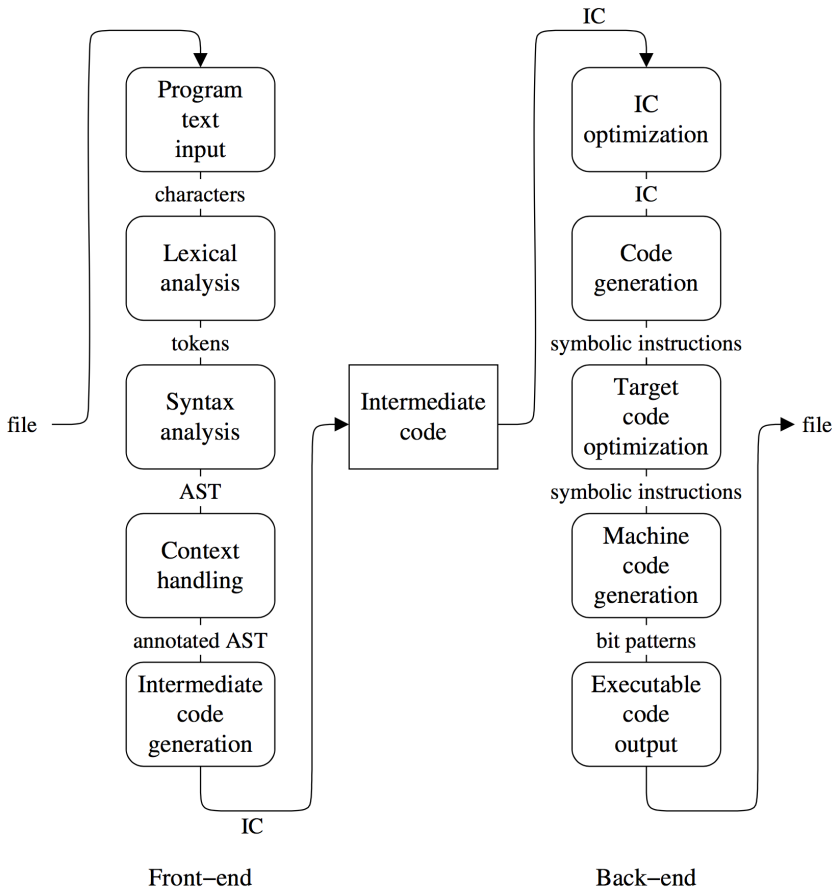

Figure 3.1: Structure of a typical compiler [11].

It should also be noted that we are building abstractions over the process of translation/conversion of source-code to flowcharts. These abstractions are useful in generalizing the conversion process, hence these abstractions are compiled as mappings in-between generic programming language constructs to flowcharts in Section 3.9.

3.2 Research Design

There are no prior solutions provided with known methodologies for translating source-code to flowcharts. The only methods available are reverse-engineering of flowcharts to source-code [2]. Hence we are experimenting a novel approach in the process of constructing such software program, i.e: the typical compiler design. Through this research we aim at exploring the possibilities of utilizing compiler based approach for source-code to flowchart translations. Hence the research conducted is a mixed method of experimental and exploratory research designs.

3.3 *_toFlowchart* Concept Overview

The proposed *_toFlowchart* is a transpiler, which will translate the source-code - given the grammar and parser definitions - to flowcharts, without changing the semantics/meaning of the original source-code. The following T-diagram (Figure 3.2) shows the highest-level design of the system.

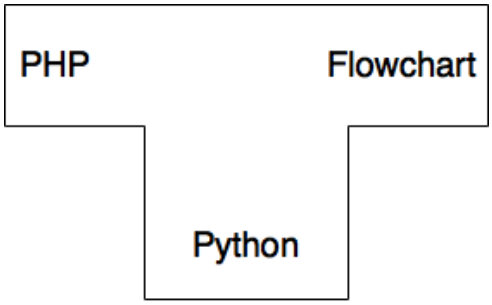

Figure 3.2: High-level design of the compiler using T-diagram.

T-diagram is used to represent the compiler design in abstract representations. Left hand side of the T indicates the source language, right hand side of the T indicates the target language and the bottom edge of the T indicates the language that was used to write the compiler itself - the host language [6], [10], [11]. In the compiler design shown in Figure 3.2; PHP is the source language, flowchart is the target language and Python is the language in which the compiler is implemented in.

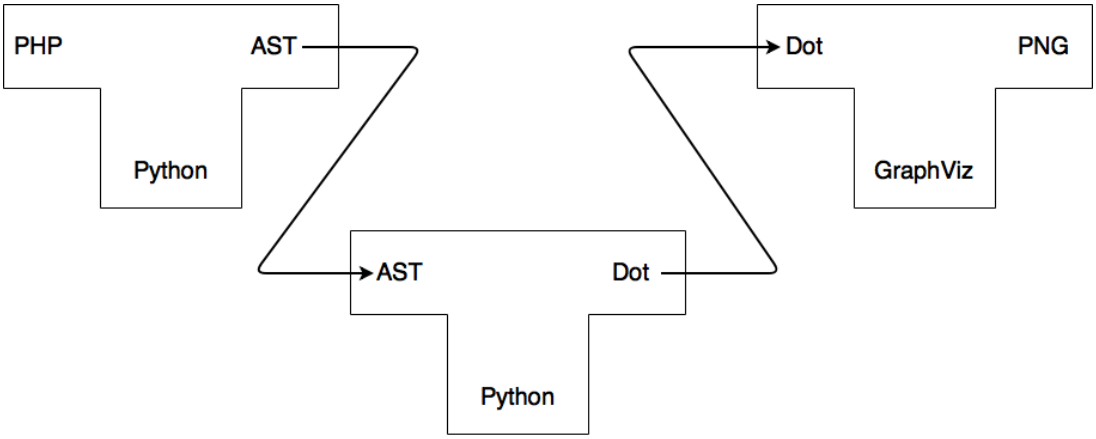

Figure 3.3: Design of the compiler using 3 T-diagrams, where output of one acts as input to the consequent compiler.

The compiler shall ultimately translate source-code written in PHP to flowcharts. This can be dissected into 3 sub-compilers where output from one compiler acts as input to the consequent one, as depicted in Figure 3.3. First compiler would tokenize, lexically analyze and parse PHP source code and translate it into Abstract Syntax Tree (AST). The second compiler, which is the core logic of the *_toFlowchart*, is responsible for translating the AST, passed from the previous compiler, to Dot language. Dot language is a specification specified by the graphical drawing toolkit Graph Visualization Software (GraphViz) [12] which is used to synthesize flowchart drawings (further information about the drawing toolkit is discussed later in this chapter). The

last compiler takes dot language specification as input and convert them into the corresponding flowchart diagram.

This architecture gives the freedom to plug a grammar and parser definition for the first compiler - PHP to AST - making the compiler feasible to support another source language. For example, one can write a grammar and parser definition to translate Python code to the specified AST format. And the output AST can be plugged into the $\_toFlowchart$ compiler - AST to Dot, extending the overall compiler to translate source-code written in Python to flowcharts.

The design of the system is intricate as we are trying to solve the problem by constructing a compiler, which could literally be generalized to any transpiler or compiler with known programming language grammars and parsers.

3.4 Scope of $\_toFlowchart$ prototype

The architecture and the algorithms are abstracted to support any language as the source language, given the corresponding grammar and parser definition. Although the abstraction is available, it is very difficult to construct additional grammar and parser for another language. Hence this prototype software will only accept PHP as the source language. This won't cater any other language as the source-language.

It is impossible for us to go through all possible PL constructs of the source language in the course of this research. Hence we shall only look at a selected subset of PL constructs. The subset of PL constructs that is being built to prove the concept are:

- If-else condition
- While loop
- Statements (eg: variable initialization)
- Function calls

As specified in the Section 1.5, the prototype expects the source-code to be object oriented. It expects for a class definition.

In-addition to these generic boundaries specified here, we have some assumptions which are elaborated in the section to follow.

3.5 Design Assumptions

We assume there exists a flowchart representation for every programming language construct. As specified earlier, it's impossible to make a full featured product, hence only the subset of PL constructs mentioned above is catered.

It is assumed that the pluggable feature of the compiler only applies to the source language; not the target language. Plugging in the grammar and parser of the compiler to the target language will be very intricate because code-generation is not easy as parsing. Given the grammar and parser definitions complying with the AST definitions given under the Section 3.8, we can convert source-code from the corresponding language to flowcharts.

It is also assumed that the source-code provided complies with the following conditions:

- Only a single file/class is evaluated.
- Only the first method/function occurring in the class is converted into the corresponding flowchart.
- Function calls within the same file or from external classes are mapped to a process nodes in the resultant flowchart; no in-depth validation/evaluation is done.
- Code segments are properly blocked. We can run a pre-processor in the parser to modify non-blocked statements into the properly blocked statements; the pre-processor wasn't implemented as part of the research.
  e.g: The snippet like the following is accepted

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

Listing 1: Properly blocked if-else statement.

e.g: The snippet like the following is not accepted although it is syntactically valid.

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

Listing 2: Shorthand if-else statement.

3.6 Alternate Designs

There are no direct implementations of the proposed solution yet. But we surely can take inspirations from the existing related systems. One such paper which we would like to mention is,

Transpiler and it's Advantages [8]. Though this paper mainly talks about the advantages of using Transpilers in-contrast with using compilers producing object code and running on Virtual Machine (VM) - Java Virtual Machine (JVM) for example- this points out what makes a transpiler unique from other compilers, and how is it usually constructed. This paper is discussed in detail under the literature review Section 2.6.

It also points out an architecture called 'Pluggable Target Compiler', which is used as an inspiration for the design of the system at hand. It takes Intermediate code (the output generated by Intermediate code generator of a typical compiler) of the source language and produces native code (executable code/ assembly code) as the output. The output from the transpiler is taken into the pluggable target compiler - which is design specific, hence pluggable - and generate the target code accordingly [8]. The architecture of the 'Pluggable Target Compiler' is depicted in Figure 3.4.

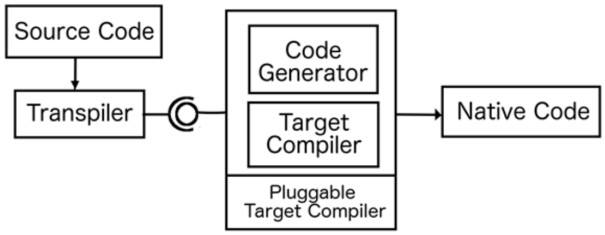

Figure 3.4: Block representation of Pluggable Target Compiler [8].

3.7 $\_toFlowchart$ Algorithm

The compiler, as depicted in Figure 3.3, is broken into 3 sub-compilers.

1. PHP to AST compiler
2. AST to Dot compiler
3. Dot to Flowchart compiler

3.7.1 PHP to AST compiler

The sub-compiler can be considered as the front-end of the whole compilation process. This sub-compiler is the one responsible to tokenize, lexicalize and parse the source-code written in PHP. As the output of the compilation/translation process, this generate an AST corresponding to the source-code file provided.

The parser used is similar to Yet Another Compiler-Compiler (Yacc). It uses LR parsing, the algorithm used is Look-Ahead LR (LALR). This produces the corresponding AST for the source-code provided.

BNF is a specification for defining grammar of languages. "BNF is very suitable for expressing nesting and recursion, but less convenient for expressing repetition and optionality, although it

can of course express repetition through recursion [11].” EBNF is an extended specification of BNF, which addresses the classical issues of BNF.

AST is key in the process of translation as it acts as the input for the subsequent sub-compiler. The whole abstraction layer of the source language trims down to complying to a single AST specification. Compliance to a single AST specification enables $\_toFlowchart$ prototype to be able to support numerous source languages. The grammar specification of the AST is provided as EBNF specification in Listing 3.

For our research, AST acts both as an intermediate representation and output of this sub-compiler.

```
ast = '(', root_node ,')';
root_node = ('"Class"' | "'Class'"), ',', '{', { key_value }, '}';

key_value = ( '"' | "'" ), letter, { letter | symbol | number }, ( '"' | "'" ), ':',
      (concrete_values | node), ',';

concrete_values = number |
      ( '"' | "'" ), { letter | symbol | number }, ( '"' | "'" ) |
      True |
      False;
node = tuple | dict | list;

tuple = '(', ( '"' | "'" ), node_type, ( '"' | "'" ), value, { ',', value }, ')';
list = '[', {concrete_values} ,']';
dict = '{', { key_value } ,'}';

node_type = 'If' |
      'Block' |
      'Method' |
      'BinaryOp' |
      'TernaryOp' |
      'Echo' |
      'Else';
```

Listing 3: Abstract grammar defining the AST specifications in EBNF.

3.7.2 AST to Dot Compiler

At the core of the $\_toFlowchart$ program is the conversion of AST to Dot language representations. This uses 2 phases in parsing and analyzing the AST:

1. Shallow Analyzer
2. Deep Analyzer

Shallow analyzer, inspired from the shallow parser in compilers, analyzes only the first level programming language constructs in the AST. It does not analyze the AST in depth. The key to shallow analysis is it doesn't recursively traverse through the AST. Moreover this also annotates what shape of the flowchart the code-block will be resulted in. This helps in generating the flowchart as it maintains states of the overall structure of the source-code as it is. For example whilst analyzing, when an if-else code block is encountered it is no further interested in what is happening within the conditionally true block or otherwise rather it is added to the stack. E.g.

```
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 4: Example code-snippet for analyzer.

In the above example, Listing 4, the shallow analyzer only identifies and mark (the code-block with the corresponding flowchart shape) the line 2, the whole *if-else* block starting in line 3 and the *echo* statement in line 11. It doesn't evaluate what is within the *if-else* block. Note that all these lines, which are evaluated, occur at the 1st level of the AST. Hence the *if* snippet within the conditionally true block (line 4) contained by the first *if* statement in line 3 is not be evaluated by the shallow analyzer.

The output of the shallow analyzer is partially annotated tree of the AST. This is then passed as input to the deep analyzer.

On the other hand, the deep analyzer evaluates, in depth and recursively, what is within each blocks. The output of the shallow analyzer is passed to the deep analyzer which in turn analyzes each shallow analyzed code-block in depth. In the example Listing 4, every-line of the code-block is evaluated by the deep analyzer. So this will evaluate what was missed by the shallow analyzer, i.e. the if-else block and the blocks/statements within that block. The output of the deep analyzer is the Dot language specification of the source-code under evaluation.

3.7.3 Dot to Flowchart compiler

The final sub-compiler takes the Dot language specification of the corresponding source-code under evaluation produced by the previous sub-compiler, AST to Dot compiler, and convert them into flowchart diagrams. This is achieved through a package called GraphViz, the details of the implementation are provided under the Section 4.4.3

3.8 System Architecture

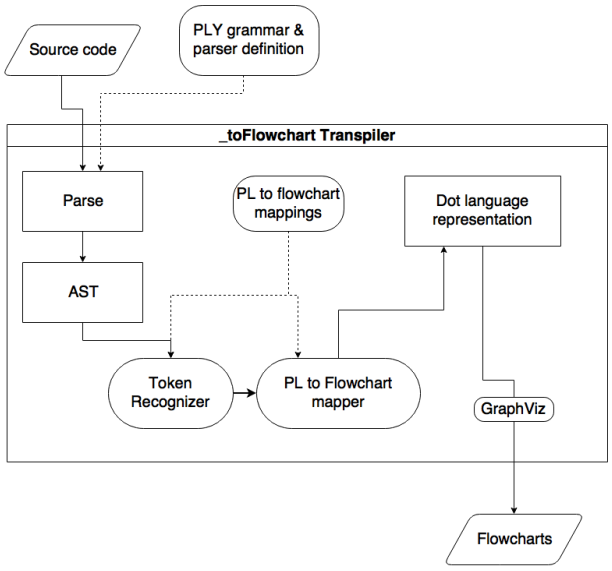

Figure 3.5: Functional breakdown of the system.

The architecture of the system is shown in Figure 3.5. The system takes source code implemented in PHP (the source language) and the PLY grammar and parser of PHP as inputs. The source-code is parsed using the PLY parser to generate the corresponding AST representation. AST is then consumed by the *token recognizer*, to which *PL to flowchart mappings* are passed as arguments, consequently passing the identified tokens to *PL to flowchart mapper*. This will generate the consequent dot language representation of the mappings specified previously. The generated Dot language specification is then used by the GraphViz toolkit to produce the corresponding flowcharts in image format.

This section provides an overview of the system including description of the functional components and sub-components of the system.

3.8.1 PLY grammar definition

As for the generalization of the system, a standard in grammar definitions of programming languages had to be chosen. As Lex and Yacc are popular and concise representation of programming language grammars, they were chosen. Python implementation of Lex and Yacc was chosen over vanilla versions, the choice is discussed in Chapter 4 with justifications. We are already aware of the choice of PHP as the source language, the grammar definition of PHP was available in PLY format. This is taken as an input to the parser, which will generate the tokens using this particular grammar definition.

3.8.2 Parser

This is the parser, responsible for generating tokenized syntax trees for the source-code text. This will take the grammar and parser definition in PLY and parse the source-code, producing AST.

3.8.3 AST

This is the output of the first compiler as shown in Figure 3.3. This acts as the intermediary in between the first and second compiler, thereby passed into the next module as input. The syntax of the AST generated by the parser should comply with the specification as provided in Section 3.7.1.

3.8.4 Token Identifier

The system should have a way to map the source-code into flowchart representation. There are arbitrarily many ways in which this can be achieved. We have chosen a method to analyze this in 2 phases; shallow analysis and deep analysis as described on Section 3.7.1.

3.8.5 PL to Flowchart Mappings

As described above the flowchart representations for basic PL constructs can be abstracted. These abstractions were mapped in Section 3.9 and used in Section 3.8.6.

3.8.6 PL to Flowchart Mapper

This uses the PL to flowchart mappings from Section 3.8.5. Using them and the annotated AST from Section 3.8.4, this functional component generates the corresponding Dot language representations. This is the core module in translation process.

3.8.7 Flowchart generator

The above Dot language representation is then used as input for the GraphViz toolkit, consequently producing flowchart representations in image format.

3.9 PL Constructs to Flowchart Abstractions

Source-code can be viewed from different perspectives such as the order in which it is written, the order of execution, the order of object calls in object-oriented languages and so on and so forth. We shall choose a perspective such that we can ensure the flowcharts generated will be comprehensive. The order in which the program is written is used for analyzing the source code. At the application level, the perspective chosen can be very helpful. Consider a debugging scenario, when the developer is trying to debug a potential bug especially in flow control he can refer to the flowchart generated for the particular class/file - which would be a direct illustration of the class/file - making it easier for the developer to spot out the bug and fix it eventually.

PL constructs are formed by one or more allowable lexical tokens of the programming language in accordance with the rules of that language. It's like gathering set of words and forming a meaningful sentence in a natural language which is complying with the grammar rules of that language. There are various classification of language constructs and they differ according to type of the language (procedural, functional, etc.). Hence there are numerous programming language constructs, we'll consider only a subset of language constructs used in most programming languages. The following mappings from source-code to flowcharts are formulated:

3.9.1 Control Structures

3.9.1.1 If-else Block

This is a choice or decision making language construct. This can be easily mapped to flowchart using the decision box (diamond shape). When the condition of the clause is true, the decision box would go along the 'true' path and vice versa. The code snippet below shows how the program would look and the corresponding flowchart that is expected to be generated is given.

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

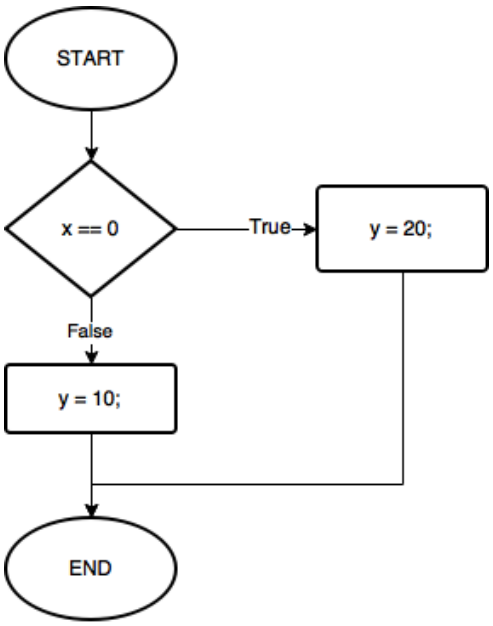

Figure 3.6: Single if-else block.

If-else block can become intricate when number of conditions are nested or when the condition is complex like the following.

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

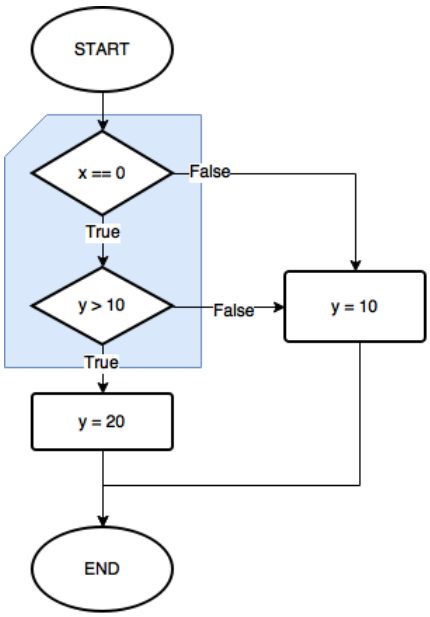

Figure 3.7: If-else condition with multiple conditional clauses.

Note that in Figure 3.7 we've utilized 2 decision boxes as our pseudocode above contains 2 conditions, of which, both should be true to make y equals to 20. Either failing the first condition or the second condition brings the execution head to the 'else' block - making y equal to 10. The combined conditional clauses are highlighted with the blue background card in the flowchart diagram in Figure 3.7.

3.9.1.2 Switch-Case Block

Switch-case is also a decision-making construct. It is used when a variable needs to be evaluated with multiple possibilities. Consider the scenario of when the variable 'contenttype' needs to be checked over a set of content types.

```
<?php
switch ($contenttype) {
    case 'news':
        echo "News flash!";
        break;
    case 'interview':
        echo "Interview with Mr. B";
        break;
    case 'presentation':
        echo "An awesome Presentation";
        break;
}
```

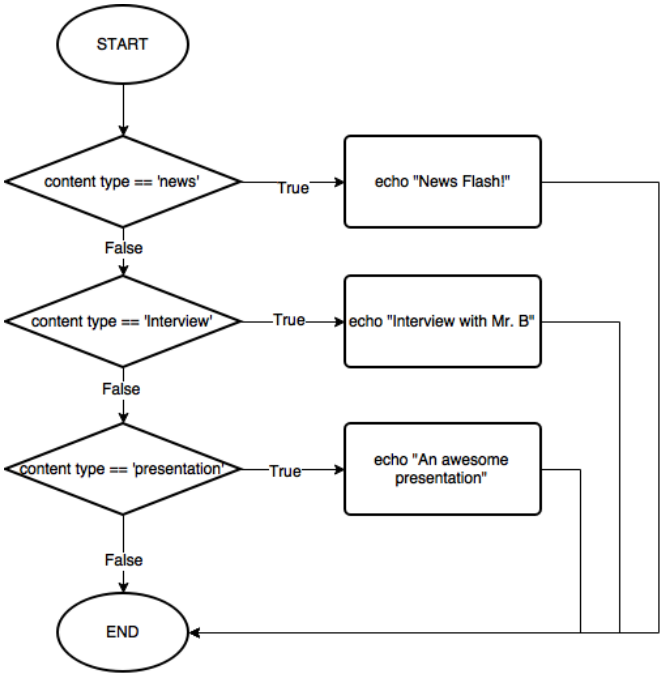

Figure 3.8: Typical switch-case representation in flowchart.

3.9.1.3 Loops

Loop is a sequence of instruction that is continually repeated until a certain condition is reached. Though we can't generalize every kind of loop, we can use the following types of loops

- For Loop

  For construct loops through a block of code a specific number of times. Following are the pseudocode and the corresponding flowchart of 'For loop'.

```php
<?php
for ($x = 0; $x < 10; $x++) {
    $y[$x] = $z+$x;
    $z = $z + 1;
}
```

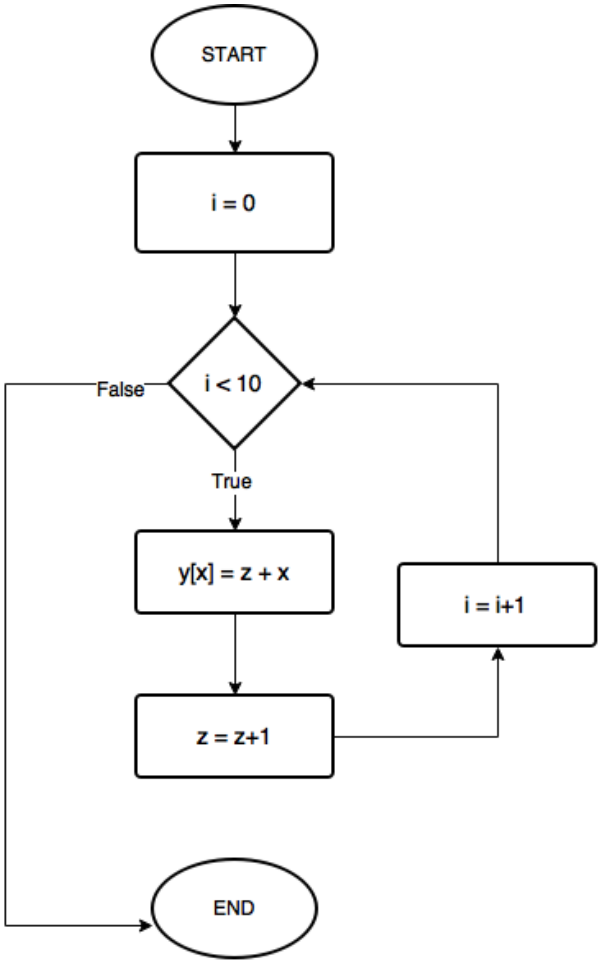

Figure 3.9: Standard for-loop construct in flowchart representation.

- While Loop

  The while statement will execute a block of code if and as long as the condition is true.

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

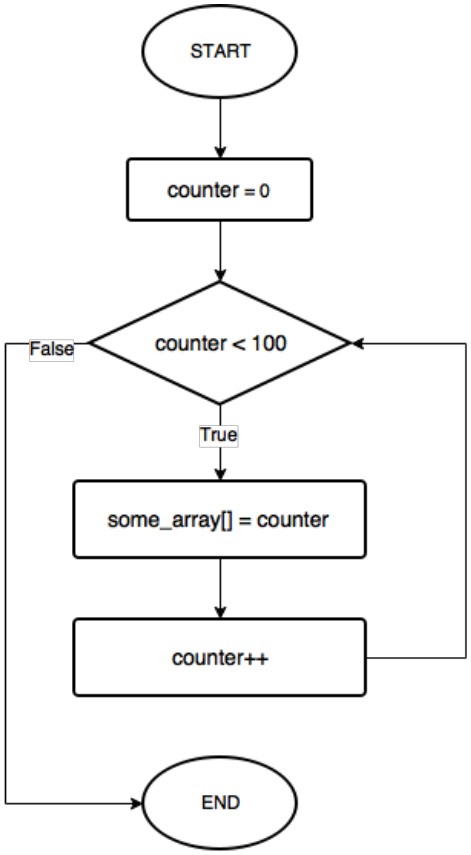

Figure 3.10: Standard while-loop construct in flowchart representation.

- Do-While Loop

  Do-While loop will execute the block of code at least once, it will then check the while condition, and repeat executing the block of code until the condition stays true.

```php
<?php
$i = 0;
$num = 50;
do {
    $num--;
    $i++;
}
while( $i < 10 );
```

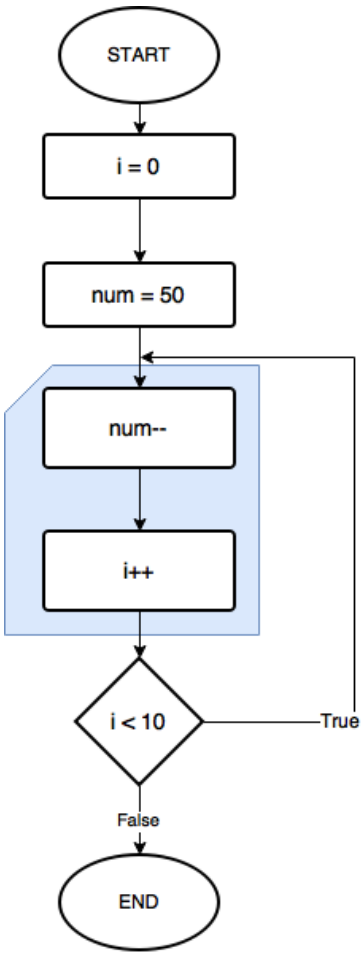

Figure 3.11: Standard do-while loop construct in flowchart representation.

- For-Each Loop

  The for-each construct is used to loop through iteratables (i.e arrays). For each pass, the current value of the array being evaluated is assigned to a temporary variable within the loop block. And the array pointer is moved by one for the next pass.

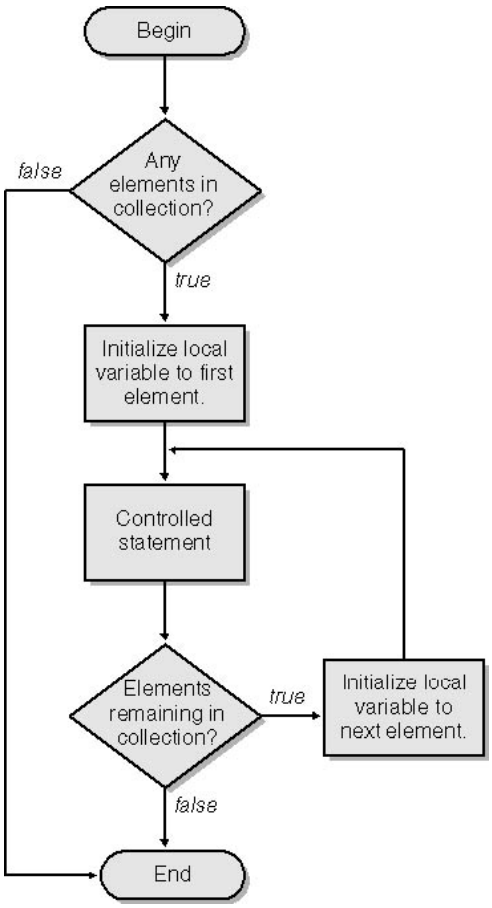

Figure 3.12: Standard for-each loop construct in flowchart representation [13].

3.9.2 Functions

Functions and procedures are set of statements written to achieve a specific goal/task. They are grouped within a function block. Functions may or may not return a value, functions and procedures respectively.

3.9.2.1 Function Declaration

Declaration of functions is easy to construct, as this breaks down to atomic statements. We should make use of a mechanism to group and label the set of statements into one - depicting the scope/boundaries of the function. Following is a typical example.

3.9.2.2 Function Calls

When we have declared the functions clearly - grouped and labelled them - it would be easier for us to depict the function calls elsewhere using flowcharts Predefined Process symbol with the label as the name of the function that is being invoked/called. Moreover, a bit of meta-data

on what type of value is returned within the Predefined Process symbol will add another level of readability of the flowchart generated.

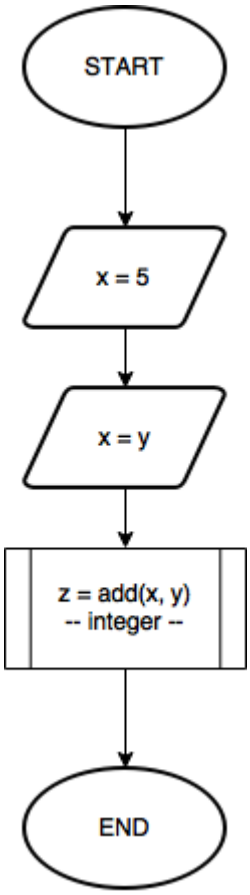

Figure 3.13: Function call.

3.9.3 Compositions

Composing number of programming constructs may turn out tricky. Intelligent labelling and grouping related blocks (like functions, if-else blocks, etc..) may resolve this to a level.

3.9.4 Classes

Classes may consist properties and methods. Property definitions may not be of very significant to the flowchart representation.

Classes would be a type of composition representation as classes are supposed to collectively represent an entity - their properties and functions.

Chapter 4

$\_toFlowchart$ Implementation

"I am enough of an artist to draw freely upon my imagination. Imagination is more important than knowledge. Knowledge is limited. Imagination encircles the world."

—Albert Einstein

4.1 Introduction

In this chapter, we will look at what decisions were taken whilst implementing the solution, what technologies were used, and what challenges were faced and how those were resolved.

4.2 High Level Architecture

A typical compiler architecture can be dissected into 3 basic abstract components - the front-end, the intermediate representation and the back-end. Front-end is usually responsible for tokenizing and parsing the input text and producing the corresponding abstract syntax tree (AST) which is used to create an intermediate representation. The intermediate representation is then used by the back-end, as its input, which is used to generate code for the target language. Figure 4.1 shows generic components of a compiler. It is not necessary for every compiler to consist of every module of a generic compiler as shown in the figure, the major components - lexical analysis, syntax analysis, intermediate code generation, intermediate code and code generation - would suffice.

In the context of the system at hand, the generic compiler architecture is modified. The front-end consists of modules like lexical tokenizer, syntax analyzer and intermediate code generator; producing AST as the output. AST acts as the intermediate representation in this modified architecture. The back-end receives AST as the input, process through series of modules to produce

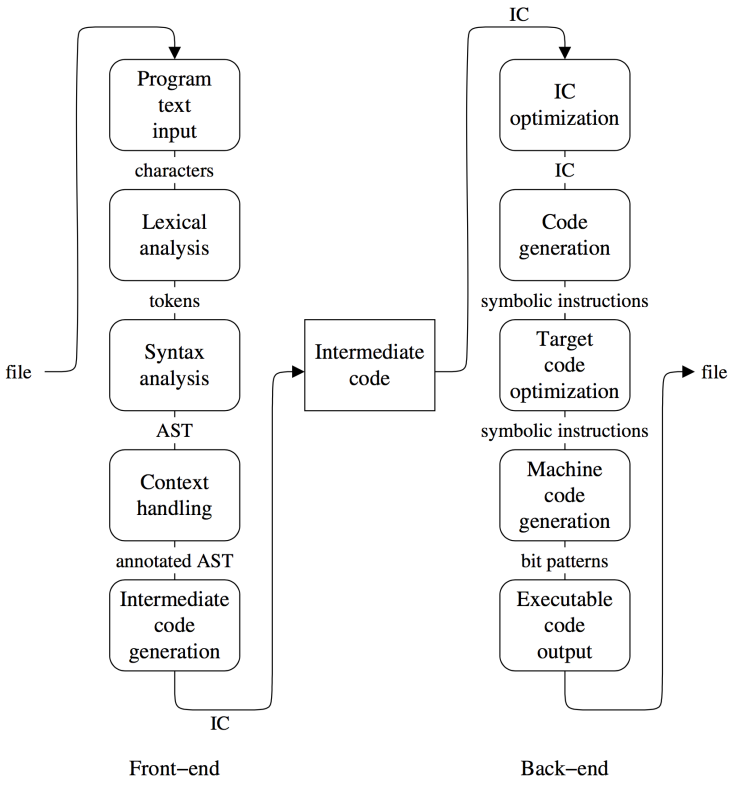

Figure 4.1: Typical architecture of a compiler [6].

the target language code through code generation. We will look into each of the components, in detail, later in this chapter.

4.3 Implementation Language

Firstly, the implementation language of the compiler was decided. In evaluating a set of candidate languages - to decide upon an implementation language - main considerations were the simplicity of the language itself, amount of support for Regular Expression (regex), ease of finding libraries and toolsets for compiler modules.

Earlier, simple tests were done in PHP language, it turned out to be harder in terms of simplicity and in finding toolsets. Then Python was evaluated, running tests similar to those done for PHP, turned out positive, and the syntax of the language was very simple and ample amount of libraries and toolsets were available. Hence Python was chosen as the implementation language.

4.4 Unfolding the Components

Compiler construction uses various tools in constructing each of its components. We shall unfold components which were introduced in the Section 4.2, providing an in-depth overview.

4.4.1 Front-End

There are famous tools for each front-end module of a compiler - Lex is a famous lexical tokenizer and Yacc is a famous language parser/intermediate code-generator and so on so forth.

Writing a lexical tokenizer and parser ourselves would have been an unwise decision, as it is already a matured piece of software. Hence pre-made front-end modules like Lex and Yacc were used. Lex and Yacc are matured software packages which have been around since 1975. They are available by default in Linux based systems and are also a POSIX standard. Lex and Yacc have their own custom syntax which aren't very flexible in terms of extension. Moreover Lex and Yacc are both based on C language, which we are not familiar with like Python (implementation language) or PHP (source/input language).

An implementation of Lex and Yacc written in Python for Python named PLY is available. It provides more flexibility for extension. Hence it was chosen.

PHP is a fully functional programming language and it has a large set of syntax definitions. Writing lex definitions for a programming language is a huge task. It should also be rigorously tested for any anomalies. The Zend interpreter is the most common interpreter used to run PHP [14]. Zend interpreter has it's own lex definition, which produces tokenized syntax tree. It also has Yacc definitions which in turn convert the tokenized lexical syntax tree - obtained from the output of the lexer - into an AST annotated with respective Zend API function for the node. The fundamental purpose of these lex and Yacc definitions are to convert the source code into object code to run on top of Zend interpreter. Hence it was hard to use the existing lex and Yacc definitions written for Zend interpreter, given that these definitions were written in vanilla lex and Yacc; whereas we were trying to explore using PLY versions of lex and Yacc. If we have taken this path forward, a mapping from these lex and Yacc definitions to PLY should have been constructed. This would have been a cumbersome task, consuming ample time, thereby delaying ourselves reaching the final goal of this research. Hence a research was done on existing libraries which tries to achieve the same or the similar. Out of some, a package has been written for PLY named PHPLY - PHP Lex and Yacc - was chosen as the most suitable and fitting one for the case at hand. Using PLY, lex and Yacc definitions for PHP language has been written named PHP Hypertext Preprocessor Python Lex YACC (phply), imitating what was already written for PHP but making sure tokens produced are not dependent on the Zend API. We chose phply as it inclined with our end goal, moreover it inclined with our implementation language.

We will now look at how the following PHP code is being converted to AST at each component of the compiler in subsequent sections. This is a typical class with a single function, *helloConditional()*, named *BasicClass*. The function has a conditional if-else block.

```php
<?php
class BasicClass
{
  public function helloConditional() {
    if (1 == 2) {
      echo("printing hello");
    } else {
      echo("HI");
    }
```

```
  }
}
```

4.4.2 Intermediate Representation

The intermediate representation is AST as stated earlier. This is a graph data structure. It contains nodes, they contain another set of nodes or properties. These nodes represent an identified code block, identified by the parser.

The following is corresponding AST represented in set of Python, tuples, dictionaries and lists for the PHP code block in Section 4.4.1. The tuple's first element is the type of the node and second element is a dictionary of it's properties and child nodes. This is the standard followed throughout in generating AST from the source-code.

```
('Class', {
    'name': 'BasicClass',
    'nodes': [('Method', {
        'modifiers': ['public'],
        'name': 'helloConditional',
        'nodes': [('If', {
            'else_': ('Else', {'node': ('Block', {'nodes': [('Echo',
                      {'nodes': ['HI']})]})}),
            'elseifs': [],
            'expr': ('BinaryOp', {'left': 1, 'op': '==', 'right': 2}),
            'node': ('Block', {'nodes': [('Echo',
                     {'nodes': ['printing hello']})]}),
            })],
        'params': [],
        })],
    }
)
```

4.4.3 Back-End

The target language is the flowchart. But flowchart is not really a textual language in the sense; rather a visual language. As we are trying to translate from one language to another, we need a textual language to output (target language) in effect representing what's in the flowchart. There are several diagramming software which use their own Domain Specific Language (DSL) to represent visualizations. Most of these DSL's are tailor-made for the diagramming software packages; it gets complicated when we try to articulate flowcharts using these DSLs. Hence a need for a diagramming library was felt.

We evaluated few diagramming libraries. Dia is a tool used for drawing various kinds of design diagrams especially in computer science discipline. It supports UML and computer architecture related diagrams (e.g: drawing turing/state machines). This software has been written in

C/C++, it also had a plugin system which exposed some of the core APIs to a Python interface, providing extensibility. A few experiments on extending the package was performed and it was felt cumbersome. Hence considering this as a candidate solution, alternate solutions were evaluated.

Next a graph based diagramming tool named GraphViz [12] was evaluated. The algorithm and how the software works were described in the A technique for drawing directed graphs [15] paper. The documentation was provided and [15] paper were used as references in evaluating this tool. Main intention of this tool is to draw and provide APIs for drawing *graph based* diagrams. The library facilitates us to draw shapes of various kinds like circles, rectangles, parallelograms and many more; these shapes are considered as nodes of a graph. The edges connecting the nodes of the graph can also be of various shaped lines, i.e: dotted lines, solid lines, arrows and many more. GraphViz also had a wrapper around vanilla GraphViz, written in python, named *pygraphviz*.

If we consider flowcharts, they are kind of directed graph structure. Each shape can be considered as a node and the connecting arrows can be considered as an edge of a directed graph. With this assumption and the facilities provided by GraphViz software package, we've choosen this as the visualizing tool.

GraphViz uses a language specification named Dot language to plot graph based diagrams [12]. The dot language is simple in form, the grammar of the language in EBNF is shown in Listing 5.

The compiler that is been constructed, converted AST to dot language. Using this specification, the GraphViz library APIs were invoked to produce the resultant flowchart into an image format.

```
1  graph       :       [ strict ] (graph | digraph) [ ID ] '{' stmt_list '}'
2  stmt_list   :       [ stmt [ ';' ] stmt_list ]
3  stmt        :       node_stmt
4              |       edge_stmt
5              |       attr_stmt
6              |       ID '=' ID
7              |       subgraph
8  attr_stmt   :       (graph | node | edge) attr_list
9  attr_list   :       '[' [ a_list ] ']' [ attr_list ]
10 a_list      :       ID '=' ID [ (';' | ',') ] [ a_list ]
11 edge_stmt   :       (node_id | subgraph) edgeRHS [ attr_list ]
12 edgeRHS     :       edgeop (node_id | subgraph) [ edgeRHS ]
13 node_stmt   :       node_id [ attr_list ]
14 node_id     :       ID [ port ]
15 port        :       ':' ID [ ':' compass_pt
16             |       ':' compass_pt
17 subgraph    :       [ subgraph [ ID ] ] '{' stmt_list '}'
18 compass_pt  :       (n | ne | e | se | s | sw | w | nw | c | _)
```

Listing 5: Abstract grammar defining the DOT language [16].

4.5 Usage

As we have built a proof-of-concept, we didn't bother about providing a user interface for users, i.e: software developers. Hence the compiler can only be used as a command line tool. The following command can be used to run the compiler:

```
python bin/toflow.py inputfile.php -o output.png
```

4.5.1 Docker image for running environment

As the compiler is built using Python language, the running environment should have Python and the dependencies installed. As this would be intricate to setup, we've built a docker image in which all the required software dependencies are installed and configured. The process of setting up and running the docker container is explained at Appendix B.

Chapter 5

Results

"I have noticed that even those who assert that everything is predestined and that we can change nothing about it still look both ways before they cross the street."

—Stephen William Hawking

5.1 Introduction

What we've built is a proof-of-concept and the methodology we've taken to solve the problem is novel, we shall evaluate the work done using the following indicator.

- Correctness of the generated flowchart

The evaluation of the product was mainly based on the correctness of the output (flowchart) it produced, as it is crucial to the solution provided. Hence *correctness of the output*, the compiler produced, was considered as the primary indicator of evaluation.

5.2 Scope of the Compiler Constructed

As the compiler constructed was for a limited scope of language constructs we shall only test programming language constructs as specified earlier in the Section 1.5 and Chapter 3. The compiler built was supporting the following programming language constructs:

- The assignment/statement construct.
- The if-else construct.

- Nested if-else construct.
- The while loop construct.

The experiments were carried out for all the standard programming language constructs given above. In addition to the above, the experiments were also performed on source code consisting a mix of the above PL constructs.

5.3 Evaluation Strategy

The experiment designed to evaluate the correctness of the outputs included the following phases:

1. Designing flowchart representation of a logic flow by hand.
2. Implementing above logic flow with PHP.
3. Using the valid code segment (implemented in step 2) as the input for the compiler and generating flowchart outputs.
4. Outputs generated (in step 3) were manually compared with the initially drawn flowchart (in step 1), for correctness and validity by hand.

5.4 Evaluating the Correctness of the Flowcharts

The experiment laid out in Section 5.3 for testing the correctness of the generated flowcharts were performed across number of standard and mixed programming language constructs. The following sub-sections contain initially designed flowchart beforehand writing of the code, the corresponding source-code and the output flowchart generated by the compiler. We have also included whether the initially drawn flowchart, the program source code are complying with the generated flowchart in a tabular format at the end of this chapter under Section 5.4.5.

5.4.1 The *if-else* Construct

The initially hand-drawn flowchart is depicted in Figure 5.1, the corresponding code in PHP is followed in Listing 6, finally the generated flowchart from $\_toFlowchart$ compiler for Listing 6 is given in Figure 5.2.

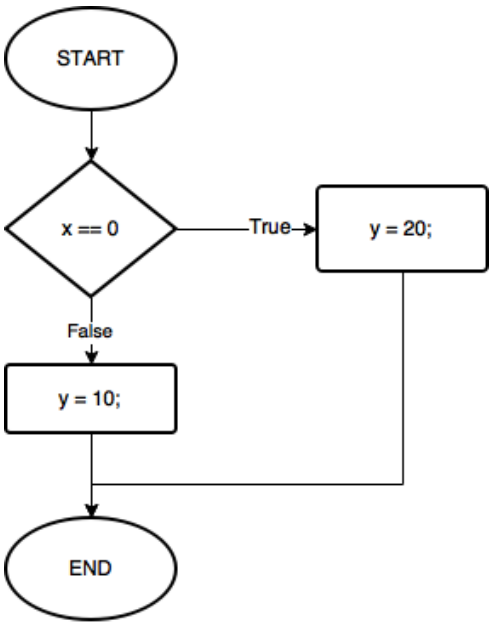

Figure 5.1: Basic if statement drawn beforehand.

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

Listing 6: PHP source-code for standard if-else condition.

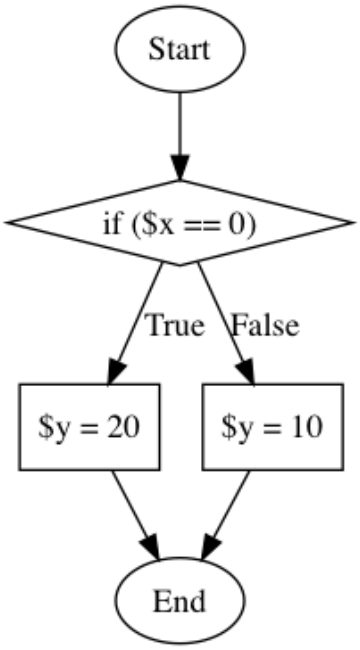

Figure 5.2: Translated flowchart from the source-code snippet given Listing 6.

We can observe that the generated flowchart (Figure 5.2) is correctly depicting each of the nodes, the arrows and the flow from start to end. Moreover, conditionally-false and conditionally-true arrows are correctly linked. Also when comparing the flowchart to hand-drawn flowchart (Figure 5.1) beforehand, they are equivalent in semantics, despite the aesthetics.

5.4.2 Nested *if-else* Construct

The nested if-else construct we've chosen to experiment has either conditionally-true-true or conditionally-true-false or conditionally-false in its logical flow. The flowchart drawn by hand is given in Figure 5.3. The code-snippet written in PHP corresponding the hand-drawn flowchart is given in Listing 7, which is followed by the flowchart generated for the code-snippet in Figure 5.4.

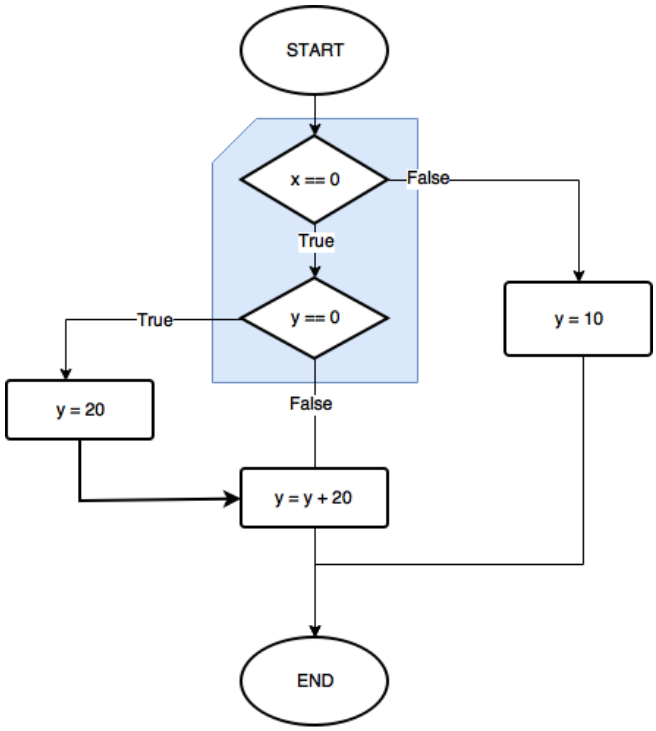

Figure 5.3: Modeled flowchart of nested if-else condition.

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

Listing 7: PHP source-code for standard nested if-else condition.

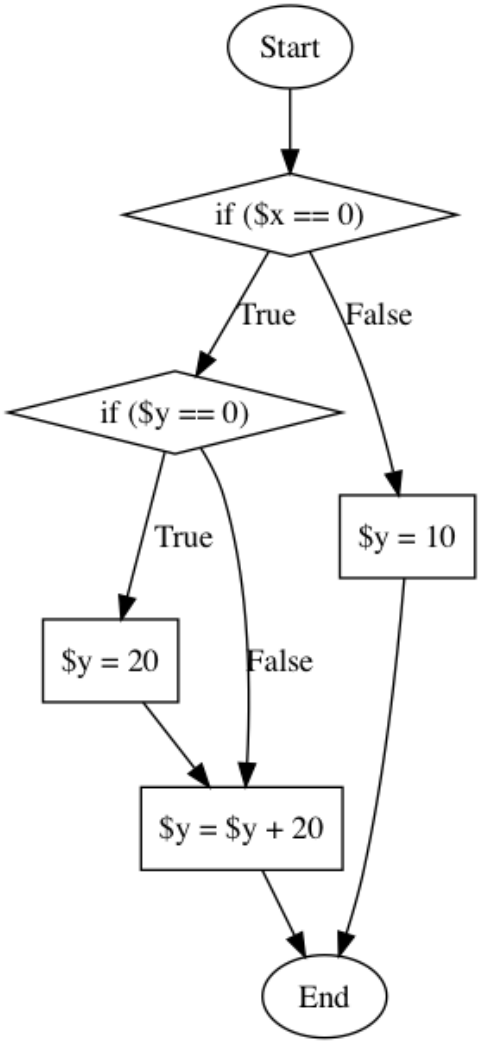

Figure 5.4: Generated flowchart for the code-snippet in Listing 7.

Likewise the previous experiment, we can observe that the generated flowchart (Figure 5.4) is semantically and syntactically equivalent to the hand-drawn flowchart (Figure 5.3) by comparison.

5.4.3 The *while* Construct

While construct can be depicted using a cyclic block in the flowchart, given a proper exit condition. Such while-construct was constructed by hand, which is shown in Figure 5.5. Corresponding code-snippet written in PHP is given in Listing 8. The generated flowchart using the code-snippet given in Listing 8 as input for $\_toFlowchart$ compiler is given in Figure 5.6. We can observe that the generated flowchart (Figure 5.6) is syntactically and semantically equivalent to the flowchart drawn by hand (Figure 5.5).

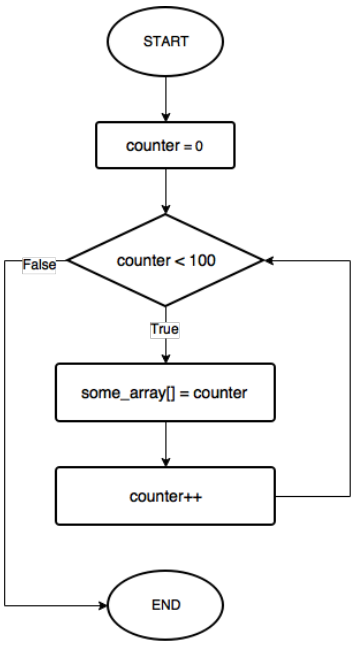

Figure 5.5: Standard while loop construct in flowchart representation.

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

Listing 8: PHP source-code for standard while loop.

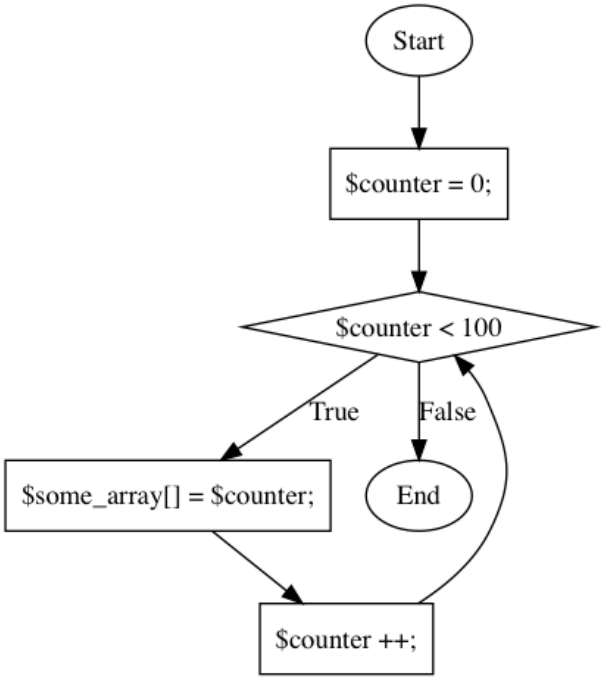

Figure 5.6: Translated flowchart from the source-code snippet given Listing 8.

5.4.4 Mixed Constructs

So far, our experiments have been based on vanilla PL constructs. Without testing single PL construct in its entirety of the experimentation process, we thought to mix a set of PL constructs and evaluate the $\_toFlowchart$ compiler. For the mixture, we've used an if-else block enclosed within a while block.

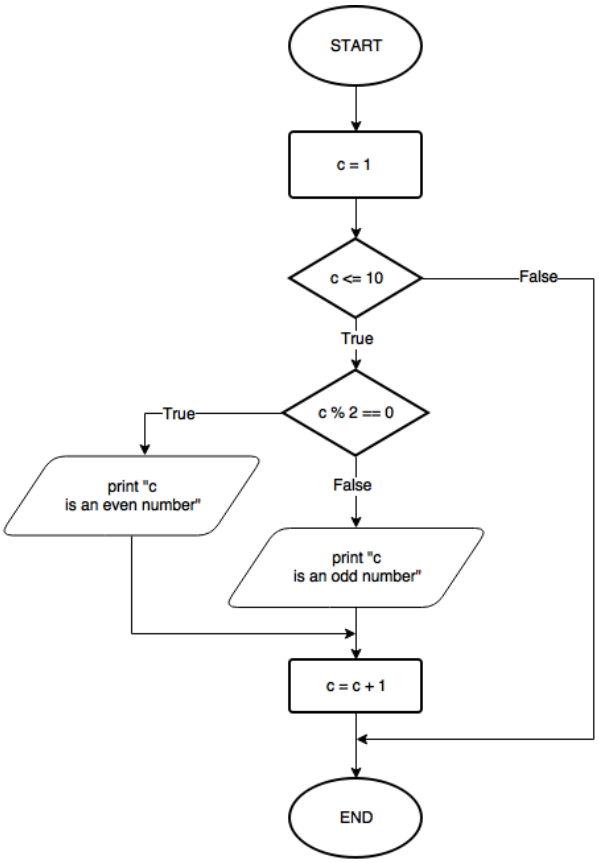

Figure 5.7: Modelled flowchart to use mixture of programming language constructs.

```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 9: PHP source-code for mixture of basic PL constructs.

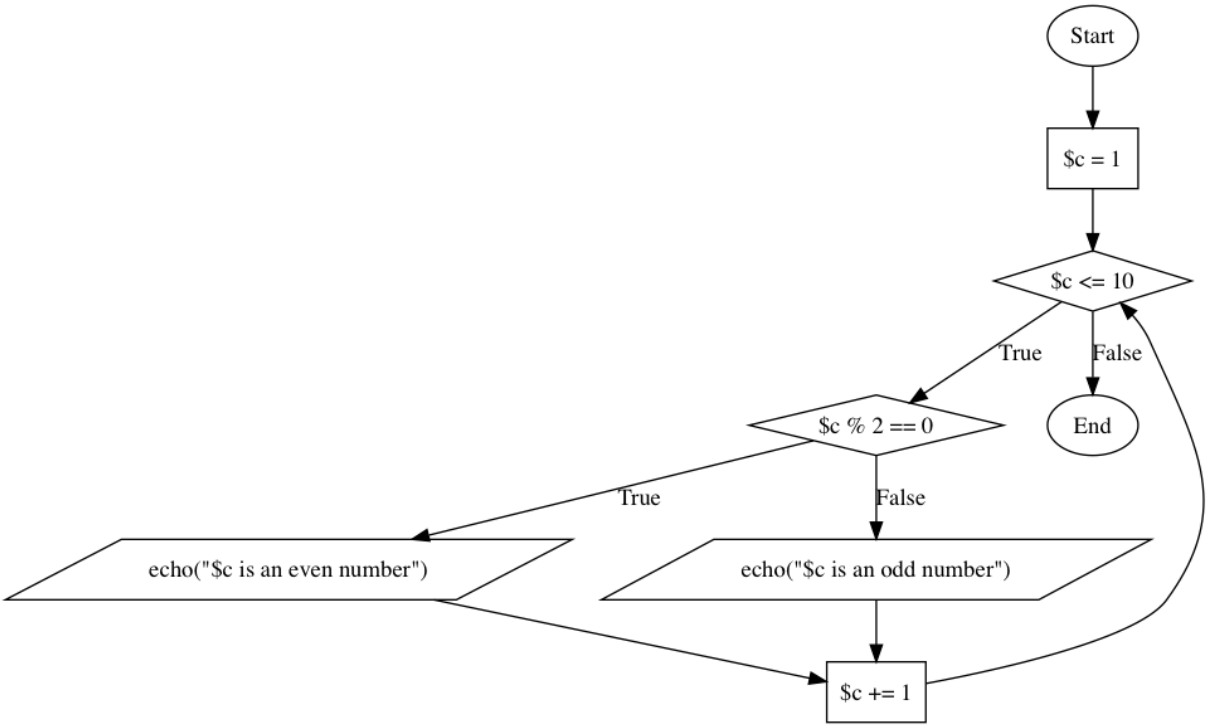

Figure 5.8: Generated flowchart for Listing 9.

The generated flowchart for the mixed PL construct, Figure 5.8, is equivalent in syntax and semantics to the flowchart drawn by hand for the algorithm given in Figure 5.7; when code given in Listing 9 is passed as input to the $\_toFlowchart$ compiler.

5.4.5 Overall Results

We've tabulized the overall results produced by each of the experiments. This contains the experimentation subject, and boolean values saying whether the flowcharts are correct, with regard to the original logic flow and source-code, and whether the flowcharts are valid, with regard to the rules of drawing flowcharts.

Flowchart for section	Is the generated flowchart correct	Is the flowchart valid
If-else construct	True	True
Nested if-else construct	True	True
While construct	True	True
Mixture of basic PL construct	True	True

Table 5.1: Overall results of experiments.

Chapter 6

Conclusion and Future Work

"I think it's very important to have a feedback loop, where you're constantly thinking about what you've done and how you could be doing it better."

—Elon Reeve Musk

6.1 Conclusion

With regard to the experimentation and the corresponding results, we can conclude that we are successful in exploring the feasibility of constructing a prototype compiler to translate source-code to flowcharts.

6.2 Research Contributions

- Novel approach in generating flowcharts from program source code.
- The abstract grammar definition of the intermediate representation (AST) was constructed using EBNF. Using this specifications, support for a new source language can be provided by means of constructing the source to AST parser.
- The methodology used to construct the compiler is generalized such that, given a source language grammar and code implemented in that language can be converted to flowcharts. This is a new frontier in designing such solutions. This method can be used for constructing other visualizations like UMLs and class diagrams.
- Generic PL constructs to flowchart mappings/abstractions are compiled and provided under Section 3.9. These can be used in future researches as basic blocks when constructing source-code based flowcharts.

6.3 Future Work

There can be various aspects in the future of the research.

- One could write a new code-generator for the same AST such that new diagrams could be generated from the source-code using the backbone of the application produced by this research.
- The program can be extended to support various other source languages by writing the appropriate lexer and parser using PLY. The output of the parser should be an AST and it should comply with the specification of the AST provided in this research.
- Optimizations for generating the dot language representation isn't done yet, this could be done as a improvement.
- The proof of concept is built upon a set of selected programming constructs. The rest of the programming constructs can be made supported by extending the code generator.
- A User Interface (UI) can be implemented for novice developers to use the product.

References

[1] D. E. Knuth, *Literate programming*. Leland Stanford Junior University, 1990.

[2] X. Wu, M. Qu, Z. Liu, and J. Li, "Automatic conversion of structured flowcharts into problem analysis diagram for generation of codes", *Journal of Software*, vol. 7, no. 5, pp. 1109–1120, 2012. DOI: doi:10.4304/jsw.7.5.1109-1120.

[3] E. R. Gansner and J. Ellson, *Autodia - automatic dia / uml generator*, http://www.aarontrevena.co.uk/opensource/autodia/index.html, [Online; accessed: 20.01.2017].

[4] I. Nassi and B. Shneiderman, "Flowchart Techniques for Structured Programming", *SIGPLAN Not.*, vol. 8, no. 8, 12–26, DOI: http://doi.acm.org/10.1145/953349.953350.

[5] L. Aurélie, M. Harold, C. Jean, and C. Bertrand, "Interest of syntactic knowledge for on-line flowchart recognition", in *Proceedings of the 9th International Conference on Graphics Recognition: New Trends and Challenges*, ser. GREC11, Berlin, Heidelberg: Springer-Verlag, 2013, pp. 89–98, ISBN: 978-3-642-36823-3. DOI: 10.1007/978-3-642-36824-0_9. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-36824-0_9.

[6] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ulman, *Compilers: Principles, techniques and tools*. Pearson Education Inc., 2013, ISBN: 9788131721018.

[7] J. J. Donovan, *Systems programming*. New York, NY, USA: McGraw-Hill, Inc., 1972, ISBN: 0070176035.

[8] R. Kulkarni, A. Chavan, and A. Hardikar, "Transpiler and it's advantages", *International Journal of Computer Science and Information Technologies*, vol. 6, no. 2, pp. 1629–1631, 2015.

[9] M. Poole, "Compilers, course notes for module CS 218, 2002", Department of Computer Science, University of Wales Swansea., Tech. Rep., 2002, [Online] Available: http://www-compsci.swan.ac.uk/~cschris/compilers.

[10] R. Toal, *Programs, Interpreters and Translators*, Loyala Marymount University, http://cs.lmu.edu/~ray/notes/introcompilers/, [Online; accessed: 22.03.2017], 2002.

[11] D. Grune, K. van Reeuwijk, H. E. Bal, C. J. Jacobs, and K. Langendoen, *Modern compiler design*. Springer-Verlag New York., 2012, ISBN: 9781493944729.

[12] E. R. Gansner and J. Ellson, *Graphviz, graph visualization software*, http://graphviz.org, [Online; accessed: 05.01.2017].

[13] M. Williams, *Microsoft Visual C# .NET*, ser. Core Reference Series. Microsoft Press, 2002, ISBN: 9780735612907. [Online]. Available: https://books.google.lk/books?id=9q5nQgAACAAJ.

[14] Z. Community, *Extension writing parti: Introduction to PHP and Zend*, https://devzone.zend.com/303/extension-writing-part-i-introduction-to-php-and-zend, [Online; accessed: 03.01.2017].

[15] E. R. Gansner, E. Koutsofios, S. C. North, and K. Vo, "A technique for drawing directed graphs", *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, vol. 19, no. 3, pp. 214–230, 1993.

[16] E. R. Gansner and J. Ellson, *Dot language specification*, https://graphviz.gitlab.io/_pages/doc/info/lang.html, [Online; accessed: 21.01.2017].

Appendix A

Open-source code

The code-base of $\_toFlowchart$ program was proposed to be open sourced in the project proposal. From the development stage through the final stage, the source code was made available in a public code repository on github.com.

The link of the github repository is: https://github.com/kisanme/programcode-to-flowcharts

The program can be downloaded from the above repository and run. To run the code there are couple of dependencies that needs be installed. Installing and setting up the environment was made easy using a containerization software named 'Docker' (Refer Appendix B).

In-order to run the program, $toflow.py$ file needs to be invoked with some parameters. First, using a terminal, navigate into the project root of the program and then run the following command:

```
$ python bin/toflow.py path-to-the-php-file.php -o output.png
```

Here the *path-to-the-php-file.php* needs to be replaced with the location of the PHP class file that needs to be converted into a flowchart. Optionally we can specify the output file name by using the $-o$ argument. By default the output flowchart image files are saved at *$project_root/outputs/* directory with the name of the PHP source file.

Appendix B

Environment

$\_toFlowchart$ is built using Python. If you have worked on Python you may know that Python 2.x and Python 3.x are both in use at the time of writing the thesis. Moreover Python has a good dependency manager called Pip but the version of Python may affect the dependencies. Hence it would become haywire to setup and run the code-base to produce the flowcharts from PHP source code.

To solve the problem of environment and dependency management, a docker image encapsulating the source code of $\_toFlowchart$ was built. Docker is a containerization software tool which performs operating system level virtualization. The docker container encapsulates the software packages those needs be installed. The definition for the container is specified in the $Dockerfile$.

```
1 FROM python:3
2
3 WORKDIR /usr/src/app
4
5 COPY requirements.txt ./
6 RUN pip install --no-cache-dir -r requirements.txt
7
8 CMD [ "python"]
```

Listing 10: Dockerfile for $\_toFlowchart$ software.

What the above file does is, from the image $python$ installs the version 3 of Python programming language along-with the compatible version of Pip. Then creates a working directory - if there isn't one - at $/usr/src/app$. Copies the $requirements.txt$ file as specified below in listing 11. The 'requirements' file will have the python dependencies/modules with the corresponding version that needs be installed. In our case, we only have 2 dependencies.

After these dependencies are installed using Pip, the command $python$ will be run. These steps will be run only when the container is being built.

To run the docker container, we can use docker-compose, an extension over the docker contain-

```
ply==3.11
pygraphviz==1.3.1
```

Listing 11: Dependencies required to use $\_toFlowchart$ program.

ers. Making it easy to build and run docker containers. This requires us to specify a file named *docker-compose.yml*.

```
version: '3'

services:
  toflow:
    build: .
    volumes:
      - .:/usr/src/app
    tty: true
```

Listing 12: Docker-compose specification for the docker container specified above.

This builds the aforementioned docker container. Whilst the docker container is running or brought up, the current directory (project root) is shared at $/usr/src/app$ within the container. This also runs the container with *tty* mode as in POSIX standards.

To run the docker container, move into the project root and run the following command.

```
$ docker-compose up -d

$ docker-compose exec toflow bash
```

Listing 13: Bringing up the container and connecting to the container respectively.

This will log us in to the container for which the required Python version is installed along-with the dependecies to execute $\_toFlowchart$ program.