Automatic Novice Program Comprehension for Semantic Bug Detection

Abejide Olu, Ade-Ibijola

School of Computer Science and Applied Mathematics

University of the Witwatersrand, Johannesburg

A thesis submitted to the Faculty of Science in fulfillment of the requirements for the degree of *Doctor of Philosophy* in Computer Science

April, 2016

Abejide Olu, Ade-Ibijola. *2016.*
Automatic Novice Program Comprehension for Semantic Bug Detection.

SUPERVISORS:
Professor Sigrid Ewert
Professor Ian Sanders

SUPPORTED BY:
National Research Foundation
Vodacom South Africa
University of the Witwatersrand, Johannesburg
JC Carstens Fund
Center of Excellence in Mathematical and Statistical Sciences

AUTOMATIC NOVICE PROGRAM COMPREHENSION FOR SEMANTIC BUG DETECTION

ABEJIDE OLU, ADE-IBIJOLA
researcher@abejide.com
www.abejide.com

A thesis submitted to the Faculty of Science,
University of the Witwatersrand, Johannesburg,
in fulfillment of the requirements
for the Degree of
Doctor of Philosophy in Computer Science

April 2016

To every person, living or dead, who has contributed *knowledge* to the *advancement* of mankind.

DECLARATION

I, Abejide Ade-Ibijola, hereby declare the contents of this doctoral thesis to be my own work. This thesis is submitted for the degree of Doctor of Philosophy in Computer Science at the University of the Witwatersrand, Johannesburg. This work has not been submitted to any other university, or for any other degree.

Abejide Ade-Ibijola
April, 2016

ABSTRACT

Automatically comprehending novice programs with the aim of giving useful feedback has been an Artificial Intelligence problem for over four decades. Solving this problem basically entails manipulating the underlying program plans; i.e. extracting and comparing the novice's plan to the expert's plan and inferring where the novice's bug is from. The bugs of interest in this domain are often semantic bugs as all syntactic bugs are handled by automatic debuggers — built in most compilers. Hence, a program that debugs like the human expert should understand the problem and know the expected solution(s) in order to detect semantic bugs. This work proposes a new approach to comprehending novice programs using: regular expressions for the recognition of plans in program text, principles from formal language theory for defining the space of program plan variations, and automata-based algorithms for the detection of semantic bugs. The new approach is tested with a repository of novice programs with known semantic bugs and specific bugs were detected. As a proof of concept, the theories presented in this work are further implemented in software prototypes. If the new idea is implemented in a robust software tool, it will find applications in comprehending first year students' programs, thereby supporting the human expert in teaching programming.

PUBLICATIONS

Some ideas in this thesis — including verbatim images, equations, and concept descriptions have been featured in the following articles. A number of these articles have been published and others have been completed, submitted, and currently undergoing peer-review as at the time of submitting this thesis.

[1] Abejide Ade-Ibijola, Sigrid Ewert, and Ian Sanders. Novice program comprehension and tutoring. *Proceedings of the M&D Symposium of the Annual Conference of the South African Institute of Computer Scientists and Information Technologists, East London, South Africa,* pages 1–2, 2013.

[2] Abejide Ade-Ibijola, Sigrid Ewert, and Ian Sanders. Abstracting and narrating novice programs using regular expressions. *Proceedings of the Annual Conference of the South African Institute of Computer Scientists and Information Technologists, Centurion, South Africa,* pages 19–28, ACM International Conference Series, doi: 10.1145/2664591.2664601, 2014.

[3] Abejide Ade-Ibijola, Sigrid Ewert, and Ian Sanders. How many ways can we skin a cat? Searching the space of novice program plan variations. *Proceedings of the 2014 Joint PRASA, RobMech and AfLaT International Joint Symposium, Cape Town, South Africa,* page 317, 2014.

[4] Abejide Ade-Ibijola, Sigrid Ewert, and Ian Sanders. Introducing Code Adviser: A DFA-driven electronic programming tutor. *In Proceedings of the 20th International Conference on the Implementation and Application of Automata (CIAA), Umeå, Sweden,* pages 307–312, Springer LNCS 9223, doi: 10.1007/978-3-319-22360-525, 2015.

[5] Abejide Ade-Ibijola, Sigrid Ewert, and Ian Sanders. Searching the space of novice program plan variations. *Submitted to the South African Computer Journal,* 2015.

[6] Abejide Ade-Ibijola, Sigrid Ewert, and Ian Sanders. Detecting semantic bugs in novice programs using DFA abstraction. *Submitted to the Journal of Frontiers of Computer Science (Springer),* 2016.

[7] Abejide Ade-Ibijola, Sigrid Ewert, and Ian Sanders. Automata-aided estimation of similarity in novice programs. *Submitted to Journal of Computer Science and Information Systems (Serbia),* 2016.

"A tree cannot make a forest" — An Old Yorùbá Proverb

ACKNOWLEDGMENTS

This PhD programme will not be successful without the contribution of very important people. Starting with my supervisors, Prof. Sigrid Ewert and Prof. Ian Sanders. I am grateful to Sigrid for giving me the opportunity to work with her, and for all the valuable insights and support, especially for: sponsoring my SAICSIT[1] conference attendance of 2013, securing my Vodacom scholarship award of 2014, her many editorial comments that have made me a better writer, and the time she spent in helping me with the refinement of the *equivalence formalism* using formal language notations. Similarly, I appreciate the support of Ian for suggesting this topic to me in the first place; all his contributions from the start to finish of this work, and his career advice.

For their support during this PhD programme, I am grateful to the following people (in random order):

- *Referees* — Prof. Bruce William Watson (Stellenbosch University), Prof. Turgay Celik, Prof. Gilbert Khadiagala, Dr. Simon Abbott, and Dr. Bernhard Zipfel.
- *WITS Computer Science and Applied Mathematics Staff* — Pravesh Ranchod, Mike Mchunu, Kgomotso Monyepote, Shashilan Singh, Dr. Benjamin Rossman, Precious Shabalala, Kebadiretse Mosiane and others.
- *Friends* — Dr. Niyo Akerele, Dr. Gideon Fareo, Jefferson Imhazenobe, Daniel Kwofie, Michael Bodunrin, Ike Innocent, and Abiodun Modupe.
- *Abeced Team* — Nicol Bell, and others.
- *Special Support* — Special thanks to: Prof. Mary Scholes for her intervention in the travel funding process that aided my participation in the CIAA[2] conference in Umeå, Sweden, and to Prof. Ebrahim Momoniat for nominating me for the IBM[3] internship programme.

1 South African Institute of Computer Scientists and Information Technologists
2 Conference on the Implementation and Application of Automata
3 International Business Machines Corporation

(My sincere apologies for anyone I have forgotten to mention.) Last but certainly not least, thanks go to my entire family for their support — in particular to the following: Prof. and Dr. (Mrs) Ibijola, Grace, Albert, Dr. Aderemi, Ololade, Abidemi, Engr. Seyi and Tolu; and others not listed here.

Sponsors.
This research was supported with the following awards/grants/scholarships.

- NRF[4] International Conference Travel Grant 2015 — NRF South Africa,
- CoEMaSS[5] International Conference Travel Grant 2015 — CoEMaSS, University of the Witwatersrand, Johannesburg,
- Doctoral Innovation Scholarship Award 2015 — NRF in conjunction with the Department of Science and Technology (DST), South Africa,
- Postgraduate Merit Award 2014 and 2015 — University of the Witwatersrand,
- Vodacom Scholarship Award 2014 — Vodacom South Africa,
- Postgraduate Student with the Highest Achievements Award 2014 — Standard Bank South Africa and WITS School of Computer Science,
- 2014 Conference Funding for SAICSIT Conference, Joint PRASA[6], RobMech[7] and AfLaT[8] International Conference, and ICSG[9] Conference — WITS (CoEMaSS) Award, and WITS Financial Aid Local Conference Travel Grant,
- Local and International Publications Award 2013 and 2014 — Microsoft South Africa and WITS School of Computer Science,
- TED[10] Fellowship — TED Innovation Community, TEDxJohannesburg, and
- JC Carstens Research Fund 2013 — University of the Witwatersrand.

4 National Research Foundation
5 Center of Excellence in Mathematical and Statistical Sciences
6 Pattern Recognition Association of South Africa
7 Robotics and Mechatronics
8 African Language Technology
9 International Conference on Serious Games
10 www.ted.com

PREFACE

In this thesis, we present a new approach to the novice program comprehension problem. This section highlights the *key contributions* of this research and the *organisation* of this thesis from a broad perspective. We have also added a list of the domains that intersect with, and the non-academic talks presented on, this work.

Key contributions.
The key contributions of this work are as follows. We have;

1. *Narrations*: presented a new tool called NOPRON[11] [Ade-Ibijola *et al.* 2014a] that aids program comprehension by narrating the lines in programs in natural language.

2. *Formalising program variation space*: presented new formalisms for describing program variations using formal language notation and presented algorithms for the iterative generation of the programs in the space of possible solutions to novice programming problems [Ade-Ibijola *et al.* 2014b 2015b].

3. *Comprehending novice programs*: used DFA[12] formalism to represent the space of program variations — paths from a start state to some accepting state in this DFA imply semantically correct programs. This DFA is a *knowledge representation* of what is expected — or likely solutions — of a novice programmer.

4. *Semantic bug detection*: devised new algorithms for finding semantic bugs in novice programs using formal language theory.

5. *Applications*: presented two more tools: one called the `Code Adviser` for advising novice programmers on how to fix semantic bugs, and the other called `Exact Code Matcher` for estimating program similarity using DFA abstractions [Ade-Ibijola *et al.* 2015a].

Thesis organisation.
This thesis is organised into six parts with varying numbers of Chapters. Part i contains the introductory chapters such as literature review, and definition of terms. The major contributions of this work are in Part ii, Part iii, and Part iv. In Part v, we discuss a number of applications of the new ideas

11 Novice Program Narrator
12 Deterministic Finite Automaton

and showcase some of the developed prototypes. The last part presents appendices of supplementary materials.

Domain of research.
Using the 2012 ACM[13] Computing Classification System, this research falls under the following categories.

- Theory of computation, semantics and reasoning, program reasoning, abstraction,
- Mathematical logic and formal languages, regular languages,
- Computers and education, computer and information science education, computer science education,

The keywords in this thesis are: program comprehension, regular languages/-expressions, novice programs, program abstraction, syntax–free approach, program variations, space search, bug detection, semantic bugs, deterministic finite automaton, and automata applications.

Non-academic Talks.
We have presented some of the ideas in this thesis at the following non–academic events.

- First and Second Heats, British Council International Fame Lab Competition, Sci-Bono Discovery Centre Johannesburg, 8th February 2014.
- Finals, German International Falling Walls Lab Competition, World Trade Centre, Sandton Johannesburg, 13th September, 2013.

13 Association for Computing Machinery

CONTENTS

I INTRODUCTION AND LITERATURE REVIEW 1

1 INTRODUCTION 3
- 1.1 Problem Definition . . . 5
  - 1.1.1 Abstracting and Narrating Novice Programs . . . 5
  - 1.1.2 Automatic Generation of Program Variations . . . 6
  - 1.1.3 Comprehension and Semantic Bug Detection . . . 7
  - 1.1.4 Automatic Tutoring . . . 7
- 1.2 Research Context . . . 7
  - 1.2.1 Aim and Objectives . . . 7
  - 1.2.2 Assumptions . . . 8
  - 1.2.3 Questions . . . 9
  - 1.2.4 Problems of Interest . . . 9
- 1.3 Key Contributions . . . 10
  - 1.3.1 Novice Program Abstraction and Comprehension via Narrations . . . 11
  - 1.3.2 Formalisation of the Space of Program Variations . . . 11
  - 1.3.3 Semantic Bug Detection using DFA Formalism and Algorithms . . . 11
  - 1.3.4 Applications . . . 12
- 1.4 Thesis Organisation . . . 12

2 PRELIMINARIES 13
- 2.1 General Definitions . . . 13
- 2.2 Program Comprehension Terms . . . 15
- 2.3 New Terms . . . 18
- 2.4 Reflections on Preliminaries . . . 19

3 RELATED WORK 21
- 3.1 Teaching Novices to Program: The challenges . . . 21
- 3.2 Cognitive Theories of Novice Program Comprehension . . . 22
  - 3.2.1 Top-down Comprehension . . . 22
  - 3.2.2 Bottom-up Comprehension . . . 23
  - 3.2.3 Knowledge-base Model . . . 23
  - 3.2.4 Opportunistic and Systematic Strategies . . . 23
  - 3.2.5 Integrated Metamodel . . . 23
- 3.3 Novice Program Abstraction . . . 24
  - 3.3.1 Abstraction as an aid for Comprehension . . . 24
- 3.4 Comprehension and Tutoring . . . 25
  - 3.4.1 Study of Variability in Programs . . . 25
  - 3.4.2 Major Successes in Comprehension and Tutoring . . . 27
- 3.5 Approaches and Methods . . . 31
- 3.6 The Gap . . . 33

3.7 Solving Problems with Graphs/DFA . . . 34
3.8 Reflections on Related Work . . . 34

II ABSTRACTING AND NARRATING NOVICE PROGRAMS 35

4 NOVICE PROGRAM ABSTRACTION 37
4.1 Narrations as Textual Description of Algorithms . . . 37
4.2 Regular Expressions for Program Abstraction . . . 39
4.3 Actualisation . . . 40
4.4 Reflections on Program Abstraction . . . 41

5 COMPREHENSION THROUGH NARRATIONS 43
5.1 The Novice Program Narrator . . . 43
5.1.1 Preprocessing . . . 44
5.1.2 Granulation . . . 44
5.1.3 Chunking . . . 44
5.1.4 Semantic Rule Lookup . . . 44
5.1.5 Narration Composition . . . 44
5.1.6 Scope of NOPRON . . . 45
5.2 Implementation and Results . . . 45
5.2.1 Average Problem . . . 45
5.2.2 Soloway's Rainfall Problem . . . 49
5.2.3 Taste-Compiler's Sum Program . . . 50
5.2.4 Other Programs in NOPRON's Domain . . . 52
5.3 Limitations . . . 53
5.4 Reflections on Narrations . . . 53

III SEARCHING THE SPACE OF NOVICE PROGRAM VARIATIONS 55

6 VARIABILITY IN NOVICE PROGRAMS 57
6.1 Variability with an Example . . . 58
6.2 Formalisation of Program Equivalence . . . 59
6.2.1 Simple Statement Equivalence . . . 61
6.2.2 Compound Statement Equivalence . . . 62
6.3 Equivalent Programs . . . 72
6.4 Reflections on Variability . . . 72

7 ITERATIVE GENERATION OF PROGRAM VARIATIONS 73
7.1 The Alternative Programs Enumeration Algorithm . . . 73
7.2 Implementation and Results . . . 77
7.2.1 APE Algorithm on the Modified Factorial Problem . . . 78
7.2.2 More Results . . . 81
7.3 Formal Aspects . . . 82
7.3.1 Complexity Analysis . . . 83
7.3.2 Proof of Finiteness of Space . . . 84
7.4 Application of Program Variations . . . 85
7.5 Limitations . . . 85
7.6 Reflections on Enumeration Algorithm . . . 87

8 VALIDATING GENERATED PROGRAMS 89

8.1 Validation with Rules of Discourse . . . 89
8.2 Validation with IO Analyzer . . . 92
8.3 Designing an IO Analyzer . . . 92
8.4 Microsoft Visual C++ IO Analyzer . . . 93
8.5 Race Condition with IO and Resolution . . . 94
8.6 Reflections on Validation . . . 95

IV COMPREHENSION AND SEMANTIC BUG DETECTION 97

9 AUTOMATA IN PROGRAM COMPREHENSION 99
9.1 Novice Solution Space as a Regular Language . . . 99
9.2 APDFA Construction . . . 100
9.2.1 Algorithms for Constructing APDFAs from Programs 100
9.3 Generating the Alphabet of $\mathcal{L}_p$ . . . 103
9.4 Test Cases and Results . . . 103
9.4.1 Next Integer Problem . . . 103
9.4.2 Modified Factorial Problem . . . 105
9.5 Reflections on APDFA Construction . . . 108

10 SEMANTIC BUG DETECTION 109
10.1 Bug Categorisation . . . 109
10.2 Matching Program Strings . . . 111
10.2.1 Generating the Alphabet for a Novice Program . . . 111
10.3 Matching Plans over Different Alphabets . . . 112
10.4 Algorithms for Bug Detection . . . 114
10.4.1 Bug-Free Novice Programs . . . 114
10.4.2 Buggy Novice Programs . . . 115
10.5 Introducing SEBUD: The SEmantic BUg Detector . . . 116
10.6 More Results . . . 118
10.6.1 Missing Code Fragments . . . 119
10.7 Reflections on Bug Detection . . . 121

V APPLICATIONS 123

11 AUTOMATIC TUTORING 125
11.1 Code Adviser: Design and Scope . . . 125
11.2 Acting Intelligent . . . 126
11.3 One Step Bug Repair . . . 126
11.4 Discussing Bug Repair . . . 126
11.5 Code Adviser in Action . . . 127
11.6 Reflections on Automatic Tutoring . . . 128

12 ESTIMATING PROGRAM SIMILARITY 129
12.1 Overview of Source Code Plagiarism . . . 129
12.2 Picking Similar Programs with SPDFAs . . . 129
12.2.1 Totally Similar Programs . . . 130
12.2.2 No Decisions . . . 130
12.3 SPDFA Construction . . . 131
12.4 Matching Exact Programs . . . 132

12.4.1 Next Integer Problem . . . 132
12.4.2 The Sum Problem with a Function . . . 133
12.5 The Exact Code Matcher . . . 135
12.6 Reflections on Program Similarity Estimation . . . 135
13 PROTOTYPING 137
13.1 Recapping Prototypes . . . 137
13.2 Bonus . . . 138
14 CONCLUSIONS AND FUTURE WORK 139
14.1 Conclusions . . . 139
14.2 Future Work . . . 140
14.2.1 Narrations . . . 140
14.2.2 Program Variations and Automata Representation . . . 141
14.2.3 Production Version of Code Adviser . . . 141
14.2.4 Tutoring by Plan Mirroring . . . 141

VI APPENDIX 143
A PLAN RECOGNIZERS 145
A.1 Regular Expressions for Program Abstraction . . . 145
B VISUAL C++ COMPILER 149
B.1 The Visual C++ Compiler Batch File . . . 149

BIBLIOGRAPHY 153

LIST OF FIGURES

Figure 2.1 Example of DFA that accepts 01. . . . 14
Figure 3.1 Automatic Source Code Summarizer [Haiduc *et al.* 2010] 24
Figure 3.2 Search Space of Possible Programs . . . 28
Figure 3.3 Conceiver vs Adil . . . 30
Figure 4.1 A new tool for novice program narration . . . 37
Figure 4.2 Structure of program abstraction . . . 40
Figure 5.1 The novice program narrator . . . 43
Figure 5.2 NOPRON in action, narrating the average of two numbers . . . 46
Figure 6.1 Generating alternative programs . . . 57
Figure 6.2 DFA for the Lecturer's, Alice's, and Bob's Solutions . . . 59
Figure 6.3 Statement categories . . . 61
Figure 6.4 Space of re-ordered implementations . . . 64
Figure 7.1 Alternative programs enumeration model . . . 74
Figure 7.2 Entity classes . . . 78
Figure 7.3 Equivalent programs enumerator . . . 79
Figure 7.4 Growth of program space . . . 82
Figure 7.5 A repository of generated programs indexed with serial numbers . . . 88
Figure 8.1 Input-Output Analyzer . . . 92
Figure 9.1 Constructing APDFAs from equivalent programs . . . 99
Figure 9.2 APDFA for the Next Integer Problem . . . 104
Figure 9.3 APDFA for 10 program strings of the Modified Factorial Problem . . . 107
Figure 10.1 Finding bugs in novice programs . . . 109
Figure 10.2 Bug Categorisation . . . 110
Figure 10.3 APDFA's Alphabet vs. Novice's Alphabet . . . 112
Figure 10.4 Selecting a problem with SEBUD . . . 118
Figure 10.5 Program verification with SEBUD . . . 118
Figure 10.6 Reporting syntax errors with SEBUD . . . 119
Figure 10.7 Detecting semantic errors with SEBUD . . . 119
Figure 10.8 Detecting missing code fragments with SEBUD . . . 120
Figure 11.1 Automatic tutoring model . . . 125
Figure 11.2 Code Adviser finding no bugs in program . . . 127
Figure 11.3 Commenting out a line . . . 128
Figure 11.4 Code Adviser suggesting a fix . . . 128
Figure 12.1 SPDFA for Alice . . . 133
Figure 12.2 SPDFA for Bob . . . 133
Figure 12.3 Alice's vs Bob's Alphabets . . . 134
Figure 12.4 The Exact Code Matcher . . . 136

Figure 14.1 The comprehension problem re-visited 139

LIST OF TABLES

Table 2.1 Logical equivalence laws . . . 15
Table 4.1 First level of abstraction . . . 39
Table 4.2 Second level of abstraction . . . 40
Table 4.3 Third level of abstraction . . . 40
Table 6.1 Abstraction of program lines . . . 60
Table 6.2 Equivalent statements . . . 62
Table 6.3 Plan for average of numbers problem . . . 63
Table 6.4 Negation and converse operators . . . 65
Table 6.5 Evaluating equivalent conditional statements with truth tables . . . 65
Table 6.6 Equivalent set from base statement . . . 66
Table 6.7 Truth values of p against q . . . 66
Table 6.8 Truth values of p `OR` q . . . 66
Table 6.9 Truth values of converse of p `OR` q . . . 67
Table 7.1 Variations of tested novice programs . . . 82
Table 7.2 Redundant compositions . . . 86
Table 9.1 Σ_{P_1} of the Next Integer Problem's APDFA. . . . 105
Table 9.2 Σ_{P_2} of the Modified Factorial Problem's APDFA. . . . 106
Table 10.1 Iterations of SEBUD when finding missing code fragment . . . 121

LISTINGS

Listing 4.1 Sum of first 10 integers . . . 38
Listing 5.1 Average of two numbers . . . 46
Listing 5.2 Average of n numbers . . . 47
Listing 5.3 Computing sum with a while loop . . . 48
Listing 5.4 Solution to rainfall problem in C++ . . . 49
Listing 5.5 Taste-Compiler: sum of first n numbers . . . 50
Listing 6.1 Lecturer's . . . 59
Listing 6.2 Alice's . . . 59
Listing 6.3 Bob's . . . 59
Listing 6.4 Syntax of `if` statement in C++ . . . 68
Listing 6.5 Template for equivalent `if` statement . . . 68
Listing 6.6 Syntax of `if-then-else` statement in C++ . . . 68
Listing 6.7 Template 1 for equivalent `if-then-else` statement . . . 68
Listing 6.8 Template 2 for equivalent `if-then-else` statement . . . 69
Listing 6.9 Template 3 for equivalent `if-then-else` statement . . . 69
Listing 6.10 Base-case of loop-iterators . . . 70
Listing 6.11 Loop-iterator L_2 . . . 70
Listing 6.12 Iterator: L_3 . . . 71
Listing 6.13 Iterator: L_4 . . . 71
Listing 6.14 Iterator: L_5 . . . 71
Listing 6.15 Iterator: L_6 . . . 71
Listing 6.16 Iterator: L_7 . . . 71
Listing 6.17 Iterator: L_8 . . . 71
Listing 6.18 Iterator: L_9 . . . 71
Listing 6.19 Iterator: L_{10} . . . 71
Listing 7.1 Sum of first `n` integers using a function . . . 76
Listing 7.2 Generated program P_1 . . . 79
Listing 7.3 Generated program P_2 . . . 79
Listing 7.4 Generated program P_3 . . . 80
Listing 7.5 Generated program P_4 . . . 80
Listing 7.6 Generated program P_5 . . . 80
Listing 7.7 Generated program P_6 . . . 80
Listing 7.8 Generated program P_7 . . . 81
Listing 7.9 Generated program P_8 . . . 81
Listing 7.10 Generated program P_9 . . . 81
Listing 7.11 Generated program P_{10} . . . 81
Listing 8.1 A chunk from the modified factorial problem . . . 91
Listing 10.1 Missing code fragment test case . . . 120
Listing 12.1 Alice's original program for the next integer problem . 133

Listing 12.2 Bob's program copied from Alice for the next integer problem 133
Listing 12.3 Alice's original program for the sum problem using a function 134
Listing 12.4 Bob's program copied from Alice for the sum problem using a function 134
Listing A.1 Regular Expressions for Plan Recognition 145
Listing B.1 Batch file for Visual C++ Compiler 149

LIST OF ALGORITHMS

4.1 Sum of first 10 integers . . . 38
4.2 Narration of the sum of first 10 integers . . . 38

5.1 Narration of average of two numbers . . . 47
5.2 Narration of average of n numbers . . . 48
5.3 Sum fragment using `while` loop . . . 49
5.4 Narration of the rainfall program . . . 51
5.5 Narration of Taste-Compiler's sum program . . . 52

7.1 Alternative Programs Enumeration (APE) . . . 75

8.1 Filtering invalid chunk permutations . . . 90
8.2 Filtering invalid programs . . . 94

9.1 Constructing APDFA from programs of the same length . . . 101
9.2 Constructing APDFA from programs of varying lengths . . . 102
9.3 Generating Σ_p, the alphabet of $\mathcal{L}_p$ . . . 103

10.1 Generating the alphabet of a novice's program . . . 111
10.2 Mapping plans from two alphabets . . . 114
10.3 Test of membership for bug-free programs . . . 115
10.4 Detecting missing code fragments . . . 115
10.5 Finding missing fragment . . . 116
10.6 Detecting plan composition errors . . . 116

12.1 Constructing SPDFA from a program . . . 131

Part I

INTRODUCTION AND LITERATURE REVIEW

Automatically understanding and debugging novice programs for the purpose of detecting semantic bugs is one of the contemporary problems in the fields of Artificial Intelligence and Computer Science Education. This problem is often referred to as *novice program comprehension* [Johnson and Soloway 1985, Johnson 1990, Spohrer 1992, Storey 2006]. The challenge of this domain is to devise intelligent computer programs that can take a given novice program and attempt to find semantic errors in it. If this can be done, a desirable application for such an intelligent system is in *Tutoring* — creating tools for guiding novice programmers to effective learning. In this part of this thesis, we present a general introduction, preliminary definitions and a review of related work in novice program comprehension.

This part has three chapters. In Chapter 1 we present an introduction and the context of this research. In Chapter 2 we set the scene with definitions of notations, and the terms used in this thesis. Chapter 3 presents related work in the area of novice program comprehension.

1 INTRODUCTION

Teaching a novice how to program is a time consuming task, sometimes requiring some degree of patience [Lahtinen *et al.* 2005]. Likewise, the learning process (from a novice's perspective) is often frustrating [Johnson 1990]. This gets more challenging as the number of enrollments in introductory programming courses increases and the ratio of novices to available human tutors gets larger every year [Lahtinen *et al.* 2005]. This challenge has resulted in a significant number of novices developing a phobia for programming, following their frustration with compilers that provide cryptic feedback while debugging programs containing logical errors [Kranch 2012]. However, this is not a compiler issue. Compilers are not designed to handle programs with semantic bugs — *syntactically correct programs that produce incorrect outputs*.

A compiler knows much about a program and if this *knowledge* is not enough to recognise semantic bugs, then it becomes necessary to add some *intelligence* to aid the inference process of comparing programs to their requirements — incorporating the *knowledge* of human experts. Soloway and Ehrlich [1984] claim that human experts are better than novices because they possess two kinds of knowledge that novices do not have — the knowledge of *programming plans* and *rules of discourse*. Hence, a program that engages with this knowledge will be regarded as intelligent and should be sufficiently able to act like a *human programming tutor* — a program that tutors programming. Therefore, an interesting question will be: what does it take to teach programming using a program? This is a formal task in Artificial Intelligence (AI) and to answer this question, it would be helpful to look critically at how the human expert would perform this task. *To be able to teach programming as a human, what do you need*? Several theories have been published on this question, suggesting that some science of cognition to model "novice thoughts", a good knowledge representation technique to store "lecturer's programming plans" for a problem and a sufficient data structure to store the "rules of discourse" for the underlying programming language are essential concerns in performing this task algorithmically.

In the early days of AI, this problem was referred to as "program understanding" [Johnson and Soloway 1985, Johnson 1990]. It is now popularly known as "program comprehension" [Storey 2006]. Comprehension is regarded as an integral part of learning how to program [Harris and Cilliers 2006]. The challenge posed by program comprehension is (finding better methods of) developing an intelligent application that attempts to

understand (or extract the *meaning* from) a program written by a person (often a novice) and matching it to a plan of the expected solution supplied by an expert, based on the requirement specification of the problem. This is clearly different from the work of a compiler since concerns and emphasis are laid on "meaning" rather than "constructs"; that is, "semantics" rather than "syntax". If the comprehension task is accomplished and knowledge of the program's logical mistakes (semantic bugs) has been acquired, deciding whether or not tutoring (and/or marking) should be done is a matter of choice.

Consequently, the interest of this work is to devise new techniques to automatically understand novice programs written in the C++ programming language and to build prototypes of software tools based on those techniques to:

1. explain programs to novices by *narrating* the steps in natural language — thereby aiding comprehension,
2. generate the space of solutions to given programming problems — automatically creating programming knowledge of what is expected of a novice programmer,
3. detect semantic bugs in novice programs using the created space of likely solutions,
4. assist — by advising/tutoring — novices in fixing the semantic bugs in their programs, and
5. use our technique to also find plagiarised programs.

Program comprehension comprises of two major aspects; cognitive theories that explain how programmers conceive their code, and research to develop advanced technological tools to aid the process of comprehension [Storey 2006]. A famous question posed by Fincher [1999, p.12a4-1] is "what are we doing when we teach programming", while she attempted to reflect on the cognitive aspects of program comprehension. Studying this cognitive process and designing a tool has been regarded as a tough task from inception, for which *there is no silver bullet* [Brooks 1983]. To date, the various theories, research methods and tools geared towards solving this problem have focused on three major components: characteristics of the programs, ability of the programmers and the software tasks involved in the process. Nevertheless, the issue of "teaching programming using software" will be incompletely discussed without the theories of teaching and learning in Computer Science Education coming into play. Several reviews — which are particular to learning programming — exist in this regard [Fincher 1999, Lahtinen *et al.* 2005]. One notable study is that of Lahtinen *et al.* [2005], in which the authors attempted to unveil the major challenges encountered by

novice programmers by conducting a considerably large survey — with a sample space of 500 students and teachers — arriving at certain suggestions on how tools may be built to aid the learning process.

Still in Computer Science Education, a controversial debate amongst researchers over the last four decades is: *how can programming best be taught?* The traditional approach — and still the most dominant — often used to teach programming is via the vehicle of language syntax and this has been faulted by many and regarded as the reason why students think of programming as "fighting the compiler" [Fincher 1999, p.12a4-1]. To address this issue, other methods of teaching the subject have been proposed, such as: the *Syntax-Free* approach (SFA) [Bornat 1987, Shackelford 1997], the *Problem Solving* approach [Barnes *et al.* 1997], the *Literacy* approach [Juliff 1997, Astrachan and Reed 1995], and the *Computation-as-Interaction* approach [Stein 1998]. All these approaches have their respective valid arguments which has made the desire to *brew* them together quite tempting — so, we have developed the following tools:

1. NOPRON[1] [Ade-Ibijola *et al.* 2014a] — a software tool that supports the syntax-free approach of teaching programming by narrating programs to novices, and
2. `Code Adviser` [Ade-Ibijola *et al.* 2015a] — another tool that supports all other three approaches by interacting with novices, finding semantic bugs and suggesting how they can be repaired.

1.1 PROBLEM DEFINITION

In this section, we define the problem solved in this thesis under the categories below.

1.1.1 *Abstracting and Narrating Novice Programs*

Many tools have been developed to assist novices in understanding programs. However, while struggling to understand the logic underlying some programs, novices also struggle to understand the constructs of the language in which they are learning to program for the first time. Is it then possible to teach novices how to program without using any specific language? Fincher [1999] has referred to the idea of teaching programming without writing programs as a paradox, and in some way "nonsensical". However, she agreed that this technique has been reported as a success by Bornat [1987] and

1 Novice Program Narrator

Shackelford [1997], both after integrating it into first year teaching, and testing it for a couple of years at two different universities — they called it the syntax-free approach to teaching the subject. There is other evidence to show that this style of teaching can be successful and tools have been created for it [Pyott and Sanders 1991]. One way of implementing the SFA is to provide the students with automatically generated algorithms. That is, *translating programs back to syntax-free algorithms — this is the problem addressed in this aspect of this thesis*.

1.1.2 *Automatic Generation of Program Variations*

This aspect is driven by the need to acquire knowledge about novice programs in order to provide automatic feedback to novice programmers about semantic bugs. In this aspect, we have answered three questions, specifically:

1. given a correct lecturer's program to a novice programming problem, how many similar/equivalent programs exist, and what are the programs?
2. can we have a generalised formalism for enumerating all possibilities in this space?

To answer these questions, we have made the following assumptions, based on established theories in program comprehension [Storey 2006, Shneiderman and Mayer 1979, Pennington 1987, Mayer 1981, Von Mayrhauser and Vans 1995] and empirical studies of novice [Soloway and Ehrlich 1984] and expert [Vessey 1985] programmers.

CORRECTNESS QUESTION. For an expert to ascertain that a novice program is semantically correct, he/she will need to know at least one correct program that solves the same problem and attempt to compare or map the novice program to the correct one [Soloway and Ehrlich 1984].

NOVICES NEED FEEDBACK ABOUT SEMANTIC ERRORS. Most novice programmers can handle syntactic errors by themselves by complying with the feedback of automatic debuggers that are associated with modern-day compilers. However, when programs are syntactically correct but have logical errors, the novices often need lecturers (or tutors) to help them find semantic bugs [Kranch 2012].

EXPERTS USE THE TOP-DOWN APPROACH. Experts (or lecturers) try to offer help by comparing a novice's program to their mental model (an idea of what the solution should be like) by first comprehending the novice program using the top-down approach. They start with the biggest chunk in novice programs and de-localise novice program fragments into lower levels of abstraction (or chunks) until the entire program makes complete sense [Brooks 1983, Soloway and Ehrlich 1984, Guindon 1990].

EXPERTS CREATE AND SEARCH A SPACE. More often than not, lecturers do not find student programs with an exact replica of their mental model of the solution. However, they try to consider all other possibilities and variations of their mental model, trying to find a match for the student's program [Johnson and Soloway 1985, Vessey 1985].

Following these assumptions, we have presented a new formalism for enumerating alternative solutions to a lecturer's program. Using the resulting enumeration, it is possible to look for students' solutions within the space of alternative solutions generated from a lecturer's program.

1.1.3 *Comprehension and Semantic Bug Detection*

This aspect addresses three questions. How do we:

1. abstract a list of *equivalent* programs, representing the space of possible solutions to a novice programming problem, to a DFA[2]? We worked with an assumption that the list is finite.
2. define an alphabet Σ over the semantic tokens (or symbols) of the list of equivalent programs?
3. find semantic bugs in such a DFA?

1.1.4 *Automatic Tutoring*

After semantic bugs have been detected, it becomes necessary to point the novice to the source of the bug. The challenge in this section is how to devise tutoring modules that advise novices about program repair steps.

1.2 RESEARCH CONTEXT

1.2.1 *Aim and Objectives*

The aim of this research work is to establish a new approach (based on formal language theory) to novice program comprehension and also build prototype tools to test the new approach. The specific objectives that make up this aim are to:

1. extensively study the field of Program Comprehension with major focus on topics such as Cognitive Mental Models in Program Comprehension, Semantic Analysis of Programs, Intelligent Tutoring Systems, and Technological Tools in Computer Science Education.

2 deterministic finite automaton

2. find better ways of aiding the comprehension process for novice programmers using textual algorithms.

3. develop a new method for the iterative generation and validation of all possible variations of programs. Validation is done using Input-Output (IO) analysis of sufficient test cases.

4. taking a complete program as a *string* and granulated program plans as semantic tokens, devise algorithms for the dynamic definition of *languages*[3] of *program strings* over some alphabet Σ of semantic tokens. A language $\mathcal{L}_p$ of correct programs is therefore a formal representation of the space of solutions to a novice problem.

5. show that $\mathcal{L}_p$ is *regular* and hence, can be represented with a DFA $\mathcal{M}$. $\mathcal{M}$ accepts all valid *program strings* of a *plan*[4] over the set of semantic tokens in the plan's alphabet Σ, where $a \in \Sigma$, is a semantic token – e.g. *increment a counter* of the plan.

6. attempt to find semantic bugs in the DFA $\mathcal{M}$.

7. as a proof of concept, develop a prototype of an ITS[5] that uses the knowledge gathered from $\mathcal{M}$ to tutor novices on how to repair their buggy programs.

1.2.2 *Assumptions*

This research is based on the assumptions that:

1. the scope is restricted to semantic errors and all buggy programs considered in this thesis are syntactically correct.

2. the focus will be on comprehending only the programs for the procedural programming paradigm, and as such, all examples/test cases in this thesis are written with the core[6] of the C++ programming language.

3. in representing a *program plan* semantically, it should be possible to present all variations of a lecturer's plan as paths on a *transition graph* while the student's plan will be a single path from the state node to the end node of the graph.

3 A language, or formal language, is a set of strings over some alphabet Σ.
4 a plan is a full or part of an algorithm for solving a problem.
5 Intelligent Tutoring System
6 The core of a programming language is the essential subset of the language needed to write simple programs in it [Satir and Brown 1995].

1.2.3 *Questions*

The questions of interest that we have answered during the course of this research are:

1. *can we get a toolkit to compile our C++ programs after generating them?* — We did and did not have to *re-invent the wheel*. The Visual C++ Compiler's DLL[7] was used to create an ad-hoc C++ compiler that was used for IO-Analysis.

2. *how many ways can a programming problem be solved?* Does this grow exponentially with respect to the size of a program? — We answered this question in Part iii. During testing, we came to the realisation that for large programs, our space enumeration algorithm actually becomes intractable. This raised very little concern because the novice programs targeted are of very small sizes.

3. *how do we enumerate (2) algorithmically? Can we feasibly tag the parts (e.g. chunks, statement blocks, functions, and line of codes) of a program and use the obtained symbols in generating all possible permutations of the plan?* — yes, this was possible and was done.

4. *is the space of solutions to a trivial novice problem a finite one?* — yes, it is. In Part iv, we showed that this space is finite by representing it with *Acyclic DFAs*[8].

5. *can we have prototypes to illustrate these concepts?* — yes, a number of tools are described in this work.

1.2.4 *Problems of Interest*

The following are the novice programming problems of interest:

1. *Next integer problem*. Read in a number, add one to it and display the result.

2. *Average problems*.
   - *Average of two numbers*. Read two numbers, sum and average them, and display the result.
   - *Average of* n *numbers*. Read n numbers (n specified by the user), sum and average these numbers, and display the result.

7 Dynamic Link Library

8 Infinite number of strings imply DFAs with cycles, while Acyclic DFAs can be constructed for any language with a finite number of strings.

3. *Soloway rainfall problem*. Read the amount of rainfall for each day. A negative value for the rainfall should be rejected since this is invalid. The program should print out the number of valid recorded days, the number of rainy days, the rainfall average over a period of valid recorded days, and the maximum amount of rain that fell on any one day. Use a sentinel value of 9999 to terminate the program.

4. *Factorial problem*. Read in an integer n and calculate $n!$, i.e the factorial of n.

5. *Modified factorial problem*. Read in an integer n and calculate $5! + n$, i.e the factorial of 5 plus integer n. This problem aims at testing the flexibility of novice programmers' thinking by not asking them to write a simple factorial program, but to write a program that adds a constant n to the factorial of 5, i.e calculates $5! + n$. We have chosen this problem as a test case because of the following reasons: For students that like memorizing programming examples, or jump at coding at the first sight of a problem without reading carefully, this problem will give a distinct-enough solution to detect if the novice programmer has completely mis-understood the problem. Thus, causing a semantic error due to *wrong problem interpretation*.

In addition to the above problems, the algorithms presented in this work can handle arrays, and therefore the following problems:

1. *Sum and average of array elements*: a program that computes the sum and average of the elements in an array.

2. *Largest/smallest item of array elements*: a program that determines the largest item (or a similar program that determines the smallest item) in an array.

3. *Matrix arithmetic*: programs that compute the addition, subtraction or multiplication of matrices.

4. *Search and sort algorithms*: programs that implement the bubble sort, selection sort, insertion sort, and the binary search algorithms.

Furthermore, we could handle any program provided its language constructs are included in the subset of the C++ language that we have covered — listed in Part ii, pages 37 — 57.

1.3 KEY CONTRIBUTIONS

The key contributions are in three major categories: novice program abstraction and comprehension via narrations, formalisation of the space of program variations, and semantic bug detection using DFA formalisms and algorithms.

1.3.1 *Novice Program Abstraction and Comprehension via Narrations*

In Part ii, we have:

1. used regular expressions in a new way that is quite different from conventional lexical analysis by recognising programming plans and not just lexemes,
2. created a new software tool that automatically translates novice programs into more detailed textual algorithms,
3. tested this tool with many novice programs and reported the results,
4. coined the word "narration" and used this word to describe highly detailed textual algorithms generated with the new tool developed in this work, and
5. argued that narrations of programs can aid comprehension.

1.3.2 *Formalisation of the Space of Program Variations*

In Part iii, we have:

1. presented a new formalism for defining program equivalence, based on the top–down program comprehension theory,
2. developed a new algorithm for the automatic generation of variations of novice programs, and
3. argued that the new algorithm can be used to generate permutations of correct solutions that, if abstracted to a DFA[9], can be used to answer questions about semantic bugs in novice programs.

1.3.3 *Semantic Bug Detection using DFA Formalism and Algorithms*

In Part iv, we have:

1. given the novice program comprehension problem an entirely new perspective by using DFAs for solution space representation,
2. developed new algorithms for;
   a) constructing APDFAs[10] from many model programs written in C++, and,

9 Deterministic Finite Automaton

10 Alternative Programs Deterministic Finite Automaton — *a type of DFA that represents the solutions space of a novice programming problem*

b) detecting specific semantic bugs in novice programs.

3. described a tool called SEBUD that reports semantic bugs using the new algorithms.

1.3.4 *Applications*

In Part v, we have presented two new applications of the DFA abstraction.

1. `Code Adviser`, a tool that finds semantic bugs and suggest to novices how they may be repaired, and
2. `Exact Code Matcher`, a tool that compares two novice programs and estimates their similarity.

1.4 THESIS ORGANISATION

This thesis is organised in Parts, each part containing a number of chapters and focused on aspects of this work as follows.

PART I presents general introduction, preliminaries, and related work.

PART II presents a new form of abstraction that we called *narration* for aiding program comprehension.

PART III presents a formalisation of the idea of equivalent programs and how we used this idea to derive new algorithms for the generation of alternative solutions to novice programs, tested and filtered these solutions using newly designed Input–Output program analyzers.

PART IV Here we abstracted the space of program variations to DFAs and defined new algorithms for finding bugs in them.

PART V presents two applications of the new comprehension approach: a new tool for tutoring novice programmers based on the semantic bugs detected, and similarity estimation of novice programs using DFAs. Conclusions and future work are also discussed in this part.

2

PRELIMINARIES

This Chapter presents preliminaries such as notation and general definitions of terms used in this thesis across the fields of formal language theory, program comprehension, Propositional logic, and mathematics.

2.1 GENERAL DEFINITIONS

In this section we present some general definitions.

Definition 2.1 (Symbol, Alphabet, and String [Martin 2003])**.** A symbol is an item or a single token. An alphabet, denoted by Σ is any finite set of symbols. A string over Σ is formed from the concatenation of zero or more symbols of Σ.

Definition 2.2 (Regular Language and Regular Expression [Martin 2003])**.** A regular expression (RE for short or REs in plural) is a string of characters that constitutes a search pattern and is used to describe a regular language $\mathcal{L}$ over some alphabet Σ. Formally, we define regular expressions according to Martin [2003] as follows:

Let R be a class of regular languages over some alphabet Σ. The corresponding regular expressions are defined as:

1. ϕ is an element of R, denoting the empty set. The corresponding regular expression is ϕ.
2. $\{\lambda\}$ is an element of R with the regular expression λ.
3. $\forall\ a \in \Sigma, \{a\}$ is an element of R, with the regular expression a.
4. Let L_1 and L_2 be elements of R, and let r_1 and r_2 be the corresponding regular expressions. Then the following operations hold:
   a) $L_1|L_2$ is an element of R with the regular expression $(r_1 + r_2)$, representing the union (or alternation) of both languages.
   b) L_1L_2 is an element of R with the regular expression (r_1r_2), representing the concatenation of both languages.
   c) ${L_1}^*$ is an element of R with the regular expression $(r_1)^*$.

Remark 2.1. Only the languages described by (1) to (4) and languages derived from them are regular languages over Σ and nothing else is.

Definition 2.3 (Deterministic Finite Automaton (or DFA) [Brzozowski and Tamm 2012]). A deterministic finite automaton is a 5-tuple $A = (Q, \Sigma, \delta, q_0, F)$, where Q is a finite, non-empty set of states, Σ is a finite set of input symbols known as the alphabet, $\delta : Q \times \Sigma \to Q$ is the transition function, q_0 is the start state and $F \subset Q$ is a finite set of final/accepting states.

Remark 2.2 (Language of a DFA). The language of a DFA M is the set of all strings accepted by the DFA, denoted by $\mathcal{L}(M)$.

Definition 2.4 (Minimality of Automata [Watson 2010]). A DFA M accepting the language L is minimal, written $\texttt{Min}(M)$ or $\texttt{Min}$, if and only if it is the smallest (measured by the number of states) DFA accepting L.

Notation 2.1 (Drawing DFAs). DFAs in this thesis are drawn according to the standard convention, with states as circles, start states with an in-edge and final/accepting states with concentric circles. Transitions are portrayed as labeled directed edges. Figure 2.1 shows an example DFA that accepts the string 01. Other sequences of 0s and 1s take the DFA to the q_d state — the dead state. Once an input takes a DFA to a dead state, the sequence of input read, or to be read, cannot result in a string in the language. The DFAs in this thesis are generated with the Automatic Graph Layout (AGL) tool [AGL].

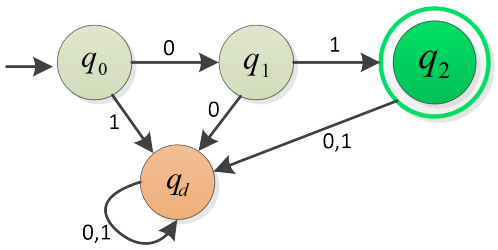


Figure 2.1: Example of DFA that accepts 01.

Definition 2.5 (Logical Equivalence [Mendelson 1997]). Given any two statements p and q, p is said to be logically equivalent to q if both p and q have the same truth values. The notation $\equiv$ is used to refer to logical equivalence and the statement, $p \equiv q$ implies that p is logically equivalent to q. Table 2.1 shows some logical equivalence laws.

Remark 2.3. More logical equivalences can be established using Truth Tables.

Definition 2.6 (Relation Algebra [Hirsch 1996, Tarski 1941]). A relation algebra $\mathcal{L}$ is an algebraic structure over some domain D, closed under the binary relations $\wedge, \vee, \neg, T, F, \cdot, \mathcal{I}, \breve{}$ and obeys the Tarski axioms. Here conjunction $\wedge$, disjunction $\vee$, and negation $\neg$ are Boolean operations, T and F are Boolean constants, converse $\breve{}$ and composition $\cdot$ are relational operations, and $\mathcal{I}$ is the identity relation.

LAW	EQUIVALENCE EXPRESSION
Identity	$p \wedge T \equiv p$ $p \vee F \equiv p$
Domination	$p \wedge F \equiv F$ $p \vee T \equiv T$
Idempotent	$p \wedge p \equiv p$ $p \vee p \equiv p$
Double Negation	$\neg(\neg p) \equiv p$
Commutative	$p \wedge q \equiv q \wedge p$ $p \vee q \equiv q \vee p$
Associative	$(p \wedge q) \wedge r \equiv p \wedge (q \wedge r)$ $(p \vee q) \vee r \equiv p \vee (q \vee r)$
Distributive	$p \wedge (q \vee r) \equiv (p \wedge q) \vee (p \wedge r)$ $p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r)$
De Morgan's	$\neg(p \wedge q) \equiv \neg p \vee \neg q$ $\neg(p \vee q) \equiv \neg p \wedge \neg q$
Absorption	$p \wedge (p \vee q) \equiv p$ $p \vee (p \wedge q) \equiv p$
Negation	$p \vee \neg p \equiv T$ $p \wedge \neg p \equiv F$

Table 2.1: Logical equivalence laws

Definition 2.7 (Inequality [Hardy *et al.* 1952])**.** This is a relation on two values, x and y, defined only if x is not the same as y.

Definition 2.8 (Cartesian product [Stacho 2007])**.** The Cartesian product (or cross product) of A and B, denoted by $A \otimes B$, is the set $\{(a, b) | a \in A$ and $b \in B\}$, satisfying the conditions: (a, b) of $A \otimes B$ are ordered pairs, and for pairs $(a, b), (c, d)$ we have $(a, b) = (c, d) \iff a = c$ and $b = d$.

Definition 2.9 (Interval [Thompson and Taylor 2008])**.** This is a set of numbers, such that every number that is between two real numbers denoting the upper and lower ranges respectively is in the set, i.e. [a, b] is an interval containing every real number x, such that $a \leqslant x \leqslant b$. Square brackets can be replaced with a parenthesis to specify that one of the endpoints is to be left out of the set.

Remark 2.4. More expressions on intervals are: $(a, b) = \{x \in \mathbb{R} \mid a < x < b\}$, $[a, b) = \{x \in \mathbb{R} \mid a \leqslant x < b\}$, $(a, b] = \{x \in \mathbb{R} \mid a < x \leqslant b\}$, and $[a, b] = \{x \in \mathbb{R} \mid a \leqslant x \leqslant b\}$. Where $\mathbb{R}$ is the set of real numbers.

2.2 PROGRAM COMPREHENSION TERMS

In this thesis, we have used a number of terms from the domain of Program Comprehension. Here we present the definitions of these terms.

Definition 2.10 (Mental Model [Storey 2006, Von Mayrhauser and Vans 1995, Walenstein 2003, Wang *et al.* 2013])**.** This is a mental representation formed in an instance to solve a problem, and often used to build knowledge at the global-structural level[1].

Definition 2.11 (Cognitive Model and Support [Von Mayrhauser and Vans 1995])**.** A cognitive model describes the processes and momentary information structures in the programmer's mind, used to formulate the mental model. Cognitive support is the reinforcement given to the cognitive tasks, such as thinking or reasoning.

Definition 2.12 (Beacons [Brooks 1983, Harris and Cilliers 2006, Crosby *et al.* 2002])**.** These are important characteristics in programs that serve as pointers to a particular programming structure. They convey some reasonable amount of knowledge about the program and are highly recognisable by expert programmers. Most theories of program comprehension are based on the identification of beacons.

Definition 2.13 (Programming Plans (or Plans) [Ebrahimi 1994])**.** A plan is an idea, with some level of abstraction, of a concept, requirement, source code, or object. A plan could be a single statement, e.g. "increment a counter" or as complex as "sort an array" [Ebrahimi and Schweikert 2006]. The term "programming plan" is used to refer to template-like solutions used to describe the steps for solving specific problems. They often consist of several sub-plans.

Definition 2.14 (Distinct and Re-ordered Plans [Ebrahimi 1994, Abd-El-Hafiz and Basili 1993])**.** A distinct plan is a unique plan that involves a different approach to solving a problem, while re-ordered plans are several valid variations (or permutations) of the same plan which is often influenced by the available constructs in a programming language

Definition 2.15 (Plan Composition [Spohrer and Soloway 1986])**.** This is the accurate arrangement of local plans (or sub-plans) to produce a correct program. The same set of plans, if wrongly ordered may produce an incorrect program. An example of this is printing an output when computation is still on.

Definition 2.16 (Rules of Discourse [Soloway and Ehrlich 1984])**.** This is a set of rules that capture the principles in programming and govern the formulation of programs from plans.

1 Mental model here is defined from a program comprehension perspective and not general teaching/learning. We have presented this definition according to the work of [Storey 2006, Von Mayrhauser and Vans 1995, Walenstein 2003, Wang *et al.* 2013].

Definition 2.17 (Program Chunk [Hansen *et al.* 2012])**.** This is a group of coherent statements, often in a structured block. Examples are functions, loops and conditional statements. Chunking is the process of recognising program chunks, and building a mental model from them.

Definition 2.18 (Novice Programming Problem [Johnson 1990])**.** This is a programming exercise given to a novice programmer.

Definition 2.19 (Novice Programmer and Novice Programs)**.** A novice programmer is a first time programming student and novice programs are the types of programs they write at the early stages of trying to learn the craft [Johnson 1990, Soloway and Spohrer 1989]. Novice programs are often written with a relatively intuitive subset of the language constructs of a programming language; that is, a novice program in a language such as C++ may contain loop structures, `if` statements, variable declarations, input and output operations, and so on, but will probably not contain advanced structures or complex operations such as recursion, modular programming using functions, classes, inheritances, and other object-oriented language constructs. A typical and generally accepted example of a novice program is Soloway's rainfall problem, which demonstrates all kinds of operations often performed by novices in computer programs [Johnson and Soloway 1985].

Remark 2.5 (Novice Programmer vs Student)**.** A novice programmer in the context of Definition2.19 is therefore a first year student in a computer science related discipline, taking computer programming for the first time. It may also refer to anyone new to computer programming at any level. The terms *novice* and *student* are used interchangeably in this thesis.

Definition 2.20 (Semantic Bug [Ebrahimi 1994, Spohrer and Soloway 1986])**.** This is a logical error in a program. Programs with semantic bugs often compile and are executed but produce incorrect output.

Examples of semantic bugs are:

1. *Plan composition errors*. Programs containing fragments that are not well arranged or mis-composed. Other types of errors that fall under this category are:
   a) *Variable scoping*. Errors resulting from mis-referencing variables having the same identifier but with different scope (such as global or local variables),
   b) *Type casting*. Truncation errors from casting real variables to integer values, e.g casting π to an integer will produce 3 instead of approximately 3.142. This affects the overall computation result.
   c) *Flipped Boolean*. Errors arising from incorrectly stating conditional statements, e.g a programmer who wants to check if `average` is less than `sum` may write: `if (sum < average)`.

d) *Operator precedence*. Inappropriate use of parentheses to enforce the precedence of operations in assignment statements, e.g calculating the average of two numbers as: `average = first + second / 2;` instead of `average = (first + second) / 2;`.

2. *Wrong problem interpretation*. This is a scenario where the programmer misinterprets the problem and uses an incorrect algorithm.

Definition 2.21 (Syntax-Free Approach (SFA) [Bornat 1987, Pyott and Sanders 1991, Shackelford 1997])**.** This approach suggests that programming should be taught with algorithms, thereby protecting novices from the early exposure to a specific language that may limit their perception of problems and hence, improving their problem solving ability.

2.3 NEW TERMS

Here we present the definition of some new terms that we have introduced.

Definition 2.22 (Equivalent Programs [Ade-Ibijola *et al.* 2015b])**.** A set of programs $\{P_1, P_2, \ldots, P_n\}$ are said to be equivalent, i.e $P_1 \equiv P_2 \equiv \ldots \equiv P_n$, if and only if two conditions are satisfied:

1. all programs P_i, $1 \leqslant i \leqslant n$ in the set have the same program plan or algorithm, and
2. for any given set of input and output test cases, all the programs produce the same output, independent of the difference in the plan composition steps of, or choice of language constructs in any of the programs.

Definition 2.23 (APDFA)**.** An APDFA or Alternative Programs Deterministic Finite Automaton is a DFA that represents the space of complete programs that lead to the solution of a programming problem.

Remark 2.6 (APDFAs are Acyclic)**.** Every APDFA constructed in this work and featured in this thesis is an *acyclic deterministic finite automaton*. Properties of, and algorithms to minimize this type of automaton in linear time can be found in Watson [2010].

Definition 2.24 (Alphabet of an APDFA)**.** The alphabet of an APDFA, denoted by Σ is the union of all the semantic tokens of its language.

Definition 2.25 (Language of an APDFA)**.** Let M be an APDFA. The language of M, denoted by $\mathcal{L}_M$, is the set of program strings accepted by M.

Definition 2.26 (SPDFA)**.** An SPDFA or Single Program Deterministic Finite Automaton is a DFA that represents a program that solves a problem. It has one start state and one final state and it accepts a single program string.

Definition 2.27 (Semantic Token)**.** A semantic token (or semantic symbol) is a line of code at its lowest granular level.

Example 2.1 (Semantic Token)**.** Examples of semantic tokens in C++ are:

1. `int sum;`
2. `void main()`
3. `i = i + 1;`

Definition 2.28 (Program String)**.** A program string w is a sequence of semantic tokens (program lines) in a program.

Remark 2.7 (Reference to New Terms)**.** The new terms defined here are used in this thesis as follows. Equivalent program is used in Part iii. APDFA, Semantic Token, and Program String are used in Part iv. SPDFA is used in Part v.

2.4 REFLECTIONS ON PRELIMINARIES

In this chapter we have presented the preliminaries to the ideas contained and terms used in this thesis. These include definition of well established terms used in the domain of program comprehension, and new terms that we have introduced. In Chapter 3, we review the literature in this field and discuss previous work that has been done in an attempt to solve the program comprehension problem.

3

RELATED WORK

How difficult can it be to teach or learn programming? Teaching programming has been regarded as a difficult task by many researchers and similarly, many students have developed a dislike for programming from taking their first course in the subject. This problem has existed and been considered a big challenge for over 40 years [Bennedsen and Caspersen 2008]. Why is it so hard to learn? Several reviews have attributed the challenges novices have in learning to issues such as: misunderstanding plan composition or plan comprehension, misinterpretation of language constructs, difference in perception and complexity of knowledge in long term memory [Ebrahimi 1994, Kranch 2012]. These findings have led to a better understanding of different problem solving techniques, and also suggested the necessary advancement in programming languages, software tools to aid comprehension and instructional methods. In this chapter we discuss related work in these areas.

3.1 TEACHING NOVICES TO PROGRAM: THE CHALLENGES

The two major categories of common novice programmer errors according to Ebrahimi [1994] are:

1. Language construct errors, and
2. Plan composition errors.

The knowledge of language constructs is paramount to understanding computer programs and novices have misconceptions about this. A few syntax and semantics related questions that often lead to confusion from a novice's perspective are:

1. When should I use a `Do`-loop instead of a `For`-loop?
2. Why should I initialise a variable?
3. Why can't I initialise anywhere, even within a loop?
4. When loops are nested, how does the control flow?
5. Why are assignment statements different from algebra? Can I write $A = 5$ as $5 = A$?

Novices are often found bringing the knowledge of mathematics and natural language into programming, which tends to lead to misconceptions.

Ebrahimi [1994] added that different programming languages often imply different programming styles, thereby generating a unique set of errors for code written in the language. It is important for a novice to understand how *plans* are composed [Johnson and Soloway 1985, Ehrlich and Soloway 1984]. In the study of human problem solving, it has been found that plans and template-like procedures are often employed to seek solutions to problems. Consequently, what the novice programmer attempts to do is formulate a plan — often in their subconscious — and implement this plan using the Language Constructs of the underlying programming language. In the quest to achieve this goal, most errors of novice programmers often come from putting the fragments of the plan together — *Plan Composition*. A major program plan may be composed by appending one sort of plan to another, interleaving plans, or branching from one plan to another when a condition is satisfied [Ebrahimi 1994]. Common plan composition errors are often related to *omission*, *misplacement*, *crooked formulation* and *misconceptions*; around variable initializations, `if` structures, loops, variable updates, and input/output operations.

Spohrer and Soloway [1986] presented an analysis of the common errors made by novices. Their conclusion was that small categories of bugs — *related to plan composition* — occurred more frequently in novice codes and constitute a higher percentage of errors and also, even though language construct misconceptions are sources of errors, this type of error does not seem to be as severe.

3.2 COGNITIVE THEORIES OF NOVICE PROGRAM COMPREHENSION

The quest to provide rich explanations about how novices understand programs and hence suggest how tools can be built to support comprehension has led to several cognitive theories. Some influential cognitive theories of program comprehension are: top-down and bottom-up comprehension, knowledge-base models, opportunistic and systematic strategies, and integrated metamodels. In this section, we discuss these theories.

3.2.1 *Top-down Comprehension*

In this approach, it is postulated that programmers attempt to understand the entire program by starting from the overall outlook of the program, looking at the highest abstraction levels (such as functions or subprograms, often referred to as chunks), and probing down the abstraction levels until they reach the lines of code, while gaining more insight about the program's functionality. The aim of this approach is to build a mental model of the application domain that is refined along granular levels of the program, from the top and down the program's abstraction levels [Brooks 1983, Storey 2006].

This approach is highly aided by *program beacons* and mostly used by experts who are familiar with the problem domain [Soloway and Ehrlich 1984].

3.2.2 *Bottom-up Comprehension*

In this approach, it is assumed that programmers attempt to understand the entire program by reading the lines and grouping the lines into higher levels of abstraction (often referred to as *chunks*). The resulting chunks are further grouped into an abstraction that is a step higher and this process continues until a complete understanding of the code's functionality is achieved. The aim of this approach is to build a mental model of the application domain [Shneiderman and Mayer 1979, Pennington 1987, Von Mayrhauser and Vans 1995, Storey 2006].

3.2.3 *Knowledge-base Model*

This is a hybrid of the Top-down and Bottom-up approaches [Letovsky 1987]. The Knowledge-base Model enforces three major components, namely: a knowledge base that keeps the programmer's internal knowledge, a mental model that describes what is known about the problem and keeps the formulation process of the solution, and an assimilation process which can either be Top-down or Bottom-up.

3.2.4 *Opportunistic and Systematic Strategies*

Littman *et al.* [1987] observed that mental models can also be built based on a study of control and data flows within a program using both static and dynamic knowledge (resulting in a systematic strategy) or by merely focusing on the task at hand and picking information on a need-to-know basis using static knowledge (resulting in an opportunistic strategy). They documented that the latter approach produces a weaker mental model of the program.

3.2.5 *Integrated Metamodel*

This model is based on a summary of the previously discussed models and it uses four major components, namely; a top-down model, a program model (used when the program is unfamiliar), a situation model (a trace of data flow and program function) and a knowledge-base model which houses the knowledge used in constructing the program and situation models [Von Mayrhauser and Vans 1995].

Some other theories have equally been published on the cognitive effect of learning computer programming, suggesting that it increases or

improves the reasoning ability of individuals and may be used as a good foundation for teaching mathematics [Pea and Kurland 1984, Feurzeig *et al.* 1981].

3.3 NOVICE PROGRAM ABSTRACTION

3.3.1 *Abstraction as an aid for Comprehension*

A good deal of work has been done on creating tools for abstracting programs to either diagrams (visualization) or text (summarization). In this section, we review the most relevant of these tools.

POLYMETRIC VIEWS OF PROGRAMS. Lanza *et al.* [2005a] presented a tool called Code Crawler (CC) in 2005. CC is a non-platform based information visualization tool targeted for object-oriented software applications. The visualizations implemented in CC are based on "polymetric views" which present source code modules as nodes and relationships between modules as edges. CC has also been proven to aid program comprehension.

MEMORY DIAGRAMS OF NOVICE PROGRAMS. Dalton and Kreahling [2010] introduced a toolkit that automatically generates memory diagrams of novice programs. Their intention was to improve the skill of novice programmers in understanding object-oriented programming.

SUMMARIZING PROGRAMS. A tool to support the program comprehension process via source code summarization was developed and presented by Haiduc *et al.* [2010]. The idea is to aid software developers in comprehending a large amount of code by simply generating a semantic summary of a large program using the automated text summarization technology. A description of how this system works is shown in Figure 3.1.

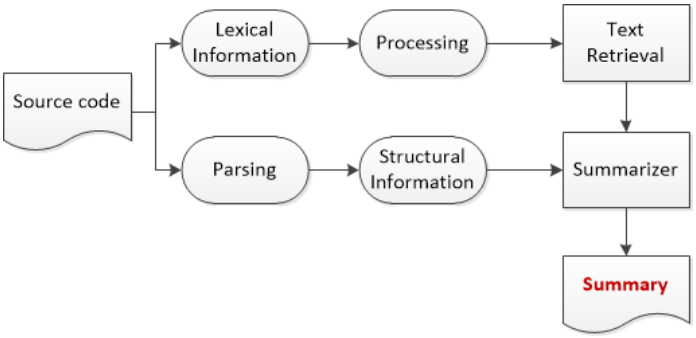


Figure 3.1: Automatic Source Code Summarizer [Haiduc *et al.* 2010]

The approach employed by this system is based on the utilization of lexical and structural information that are present in the source code.

Consequently, this system has to parse the source code in the attempt to extract reasonable information that is used in generating the code summary.

SOFTWARE CARTOGRAPHIC VISUALIZATION. To provide programmers with a graphical mental model of how their software is written, Kuhn *et al.* [2010] developed a plug-in (named CodeMap) for the Eclipse Integrated Development Environment (IDE) that automatically presents an entire solution in a cartographic visualization. This visualization is said to assist students and professional software developers in conceptualizing their programs.

VISUALIZATION OF RICH INTERNET APPLICATIONS. DynaRIA is a tool created to aid the comprehension of Rich Internet Applications (RIAs) developed with the AJAX (Asynchronous Java and XML) technology. Using defined abstractions, visualizations and a user friendly interface, DynaRIA was able to provide a rich description of AJAX-based Internet Applications [Amalfitano *et al.* 2010].

JAVA PROGRAMS VISUAL EXPLORER. SHriMP Views, by Storey [2011], Storey *et al.* [2001], Storey [2006], is an interactive visualization environment for exploring Java programs first presented in 2001. The tool is said to aid the comprehension of abstract programming concepts by offering three different kinds of view to the programmer: geometric, semantic and fisheye views. Reviews and upgrades of this tool have since been released.

There are many more examples of tools that offer some level of abstraction over programs (novice or expert programs) in an attempt to aid comprehension. In Part ii, we present a new tool that is similar to the ones presented in this section. Our tool attempts to aid comprehension but not by summarization or visualization, rather by narration using regular expressions.

3.4 COMPREHENSION AND TUTORING

3.4.1 *Study of Variability in Programs*

How many choices can be made in the process of writing a program in any chosen language? This is the question of variability [Johnson 1990, Rist 1990]. Creativity is highly involved in programming and two programmers are often likely to submit two different solutions to the same problem [Rist 1990]. In fact, novice programs are sometimes so unique that it becomes relatively easy to pick up plagiarism in them [Ahmadzadeh *et al.* 2011, Vogts 2009, Cosma and Joy 2006]. If variability can be studied and modeled, then it becomes possible to build an intelligent agent for comprehending novice programs automatically — because such an agent will be aware of every

possible *variation*, and hence, will have the *knowledge* of the *space* of *solutions*. We will discuss the notion of variability by answering two questions:

1. What varies in programs?
2. What makes novices write varying programs?

The following are reasons we have identified according to the findings of Rist [1990 1991], Robillard [1999], and Jeffries *et al.* [1981].

What varies in programs? As earlier stated, two programs designed to solve the same problem using one algorithm in a specific programming language may differ distinctly due to:

1. *Language construct*. A `for`-loop statement can have a `do`-loop alternative that iterates the same number of times. Likewise, there may be several ways of incrementing a counter in a language. For instance in C++, one may write `counter=counter+1;` or `++counter;`.
2. *Semantic composition*. Logical statements can be expressed in a finite set of distinct ways with the same meaning. For example: `if (a>b)` is the converse of `if (b<a)`.
3. *Ordering of operations*. All variables may first be declared prior to assigning values to them or they may be declared on an "as-need-arises" basis.[1].
4. *Granularity level*. Operations may be condensed or well-granulated. For example, the line `cout<<sum/n;` in C++ may also be written in two lines of code; the first line calculating the average by dividing `sum` by `n`, and the second line displaying the value of average using the `cout` statement.
5. *Program text*. The length of a program or the language in which it is written may increase the space of program variation.

What makes novices write varying programs? The known factors that contribute to the varying programs are:

1. *Impact of programming discipline.* Design style and documentation discipline vary from one novice to another.
2. *Programming paradigms.* Object-oriented programming (OOP) drives a different style of program design. Ebrahimi and Schweikert [2006] documented that OOP even compounds the problems of novice programmers.

1 One program plan can produce several solutions that differ in their ordering of the plan pieces [Rist 1991, pp. 10].

3. *Influence of individual programmer differences.* Expert programmers see the bigger picture and think of problem domain and algorithms while novices focus more on line-by-line code generation.

4. *Knowledge of language features.* The breadth of knowledge of novices varies depending on which features are available to them in programming languages. This leads to them making different choices when writing programs.

5. *Problem solved by programs.* Programs performing different types of computations vary in style of generation. Some programs simply have one unique way of writing them while other programs may have numerous algorithms to solve the same problem.

3.4.2 *Major Successes in Comprehension and Tutoring*

Over the decades, many attempts have been made to provide enough intelligence for a program to be able to comprehend other programs and hence tutor novices with the acquired knowledge. In this section we describe notable works in this area; going from those mostly related to our work and stepping wider to more systems of lesser similarity but within the same context.

3.4.2.1 *PROUST*

One remarkable success is the work of Johnson and Soloway [1985] in which they developed a program called PROUST that attempts to comprehend novice programs and give feedback to the novice on program correctness. What PROUST does is to take a program and a textual description of the problem (as inputs) and attempt to map code to requirement. To do this, PROUST had to reconstruct the design and implementation steps that the programmer went through.

How did PROUST solve this problem?
The method employed by PROUST can be categorised under the Knowledge-base approach. PROUST manipulated: a knowledge base of programming plans (correct and buggy plans, which are associated with goal decompositions as shown in Figure 3.2), strategies (such as transformation rules), and common bugs associated with different plans (misconception rules) to achieve its goal. Consequently, two major application components were produced from PROUST; a programming expert and an Intelligent Tutoring System (ITS). The ITS module made it possible for PROUST to give a reasonable depth of explanation of program bugs to aid novice programmers in the learning process.

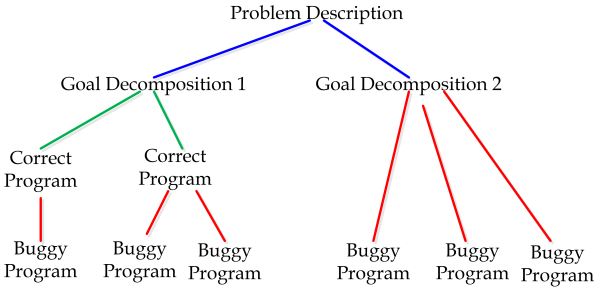


Figure 3.2: Search Space of Possible Programs

What PROUST could not do? — Limitations.
PROUST was effective for programs that are only about half-page or one-page length at most. It was also language and problem tied; designed to handle only `Pascal` programs and well-tested with only Soloway's rainfall problem. The scope of problems it could handle was evidently small because it was not designed to evolve; it contained a pool of static knowledge about the problem domain. The new approach presented in this thesis performed a similar function as PROUST but at a higher level of complexity; handling more programming problems dynamically.

3.4.2.2 *MENO-II*

Soloway *et al.* [1981] presented a system for understanding novice source codes and tutoring named MENO-II. However, Johnson and Soloway [1985] reported MENO-II as a failed project because it could not cope with the variability in actual programs.

3.4.2.3 *Programmer's Apprentice*

Although slightly different in purpose, the Programmer's Apprentice [Rich and Waters 1988] is a tool that supports the process of software engineering. The goal is to provide software engineers with intelligent assistance and to achieve this, the interaction between a real-life software engineer and their apprentice was modeled into a software application. This system however attempts to also manipulate programming plans by using a formal representation referred to as plan calculus.

3.4.2.4 *Automatic Feedback Generator*

Singh *et al.* [2013] presented a new tool that employs program sythesis technique to automatically find the minimal corrections required to fix a buggy program. Using a model solution to a programming problem and an error model (a list of likely student errors), the tool attempts to synthesize the shortest possible sequence of repairs and also give a feedback as to what the student did wrong. The tool was tested with students' programs in an introductory course at the Massachusetts Institute of Technology, USA.

3.4.2.5 *JRipples*

`JRipples` [Buckner *et al.* 2005], is a tool that was developed based on the philosophy of intelligent software assistance; similar to the Programmer's Apprentice. It therefore requires cooperation with the programmer to keep track of the application being developed. `JRipples` analyzes the source code, keeps track of the discrepancies and systematically tags the modules/classes to be revisited by the programmer. This tool therefore fixes routine errors and enables the programmer to focus on more important decisions and higher levels of code abstraction. `JRipples` was implemented as a plug-in for the Eclipse Integrated Development Environment.

3.4.2.6 *Lackwit*

`Lackwit`, a system developed by O'Callahan and Jackson [1997] uses the *type inference* approach to attempt to auto-understand programs written in C and C++. `Lackwit` could identify abstract data types, detect abstraction violations, find unused variables, functions, and fields of data structures, detect simple errors in operations on abstract data types, and locate sites of possible references to a value.

3.4.2.7 *PAT*

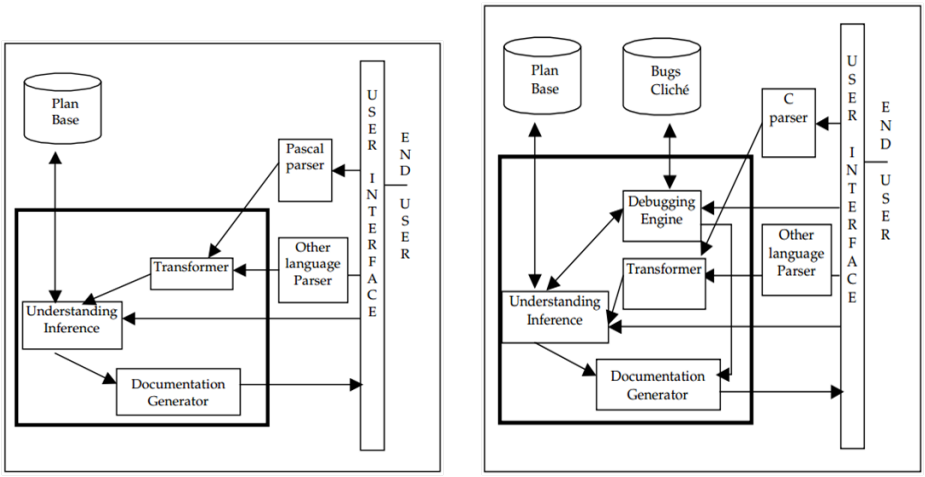
PAT [Harandi and Ning 1988], an acronym for Program Analysis Tool, is a system that was developed as an outcome of the PhD research carried out by Ning in 1988. PAT is an intelligent application that uses a knowledge base of rules of discourse and common bugs to assist programmers in debugging.

3.4.2.8 *Conceiver*

`Conceiver` [Al-Omari 1999] is an outcome of PhD research carried out at Universiti Kebangsaan Malaysia by Al-Omari about 10 years after PAT. It uses a hybrid of both bottom-up and top-down comprehension strategies to understand Pascal programs. Figure 3.3a describes the components of the `Conceiver` system.

3.4.2.9 *Adil*

`Adil`, an acronym for Automated Debugger in Learning, is an extension of the work of Al-Omari [1999]. The system is described in Figure 3.3b. It uses pattern matching techniques and some constraint satisfaction algorithms to detect semantic errors in codes with the help of a knowledge base of predefined bugs. The major addition to `Conceiver` is a bug cliché that enhances the knowledge of the system in diagnosing program faults. `Adil` handles C programs in contrast to `Conceiver`'s Pascal orientation.



(a) Conceiver System (b) Adil System

Figure 3.3: Conceiver vs. Adil [Al-Omari 1999]

3.4.2.10 *IPA*

IPA or the Intelligent Program Analyzer, proposed by Ruth [1976], is one of the earliest work in program comprehension and tutoring. IPA identifies the plan in students' programs and uses a program generation model to attempt to generate bug reports.

3.4.2.11 *LAURA*

LAURA is a program debugging system that employs a control flow graph of plans (syntax independent representations) to compare the student program and model solution heuristically. Unmatched fragments[2] are reported as likely bugs [Adam and Laurent 1980].

3.4.2.12 *TALUS*

TALUS is an automatic program debugging system that attempts to match the student's program to a collection of correct programs specified by the teacher. The major drawback is that teachers find it tedious to enumerate all possible solutions, even for a trivial problem. As the programming problem gets more sophisticated, it becomes almost impossible to manually list all correct solutions [Murray 1986].

2 These are the fragments of both programs that are not matched from the control flow graphs of both the model and student programs.

3.4.2.13 *AutoGrader*

In 2007, the `AutoGrader` framework developed at Miami University was adapted to develop an Interface for Automatic Grading of Java Programs [Helmick 2007].

3.4.2.14 *CloudCoder*

`CloudCoder` is an online system[3] for assessing programming assignments [Hovemeyer and Spacco 2013, Papancea *et al.* 2013, Hovemeyer *et al.* 2013]. `Cloudcoder` takes student program submissions and performs automatic testing on them using input-output test cases. The system reports on the passed tests and the failed ones. As at 2013, `Cloudcoder`'s submissions are in C, C++, Java, and Python.

3.4.2.15 *Other Systems*

The literature presented in this section has attempted to cover every important tool developed in an attempt to automatically assess, understand, or tutor programming. There are many more of these systems, however, we cannot discuss all of them. Other closely related systems, but not discussed above include:

1. PUDSY: a program understanding and debugging system [Lukey 1980].
2. PHENARETE: a program for understanding LISP codes [Wertz 1982 1987].
3. `Apropos`: a program for analyzing and tutoring novice programs in Prolog [Looi 1988].

3.5 APPROACHES AND METHODS

In order to perform the task of understanding programs and retrieving enough information to aid tutoring, several approaches and methods have been employed in the past. Most of these approaches are effective for discovering specific semantic errors in programs. In this section we present a summary of these methods/approaches.

BEACONS Observing program features and using the information in determining program's functionality. *Drawback for Comprehension Task*: where the documentation is very bad and the program has many bugs, it becomes difficult to acquire tangible knowledge about programs.

3 www.cloudcoder.org

A tool based on this technique was presented by Harris and Cilliers [2006].

VISUALISATION Providing visual representations and aids to enable the programmer to observe structures and objects more intuitively. *Drawback for Comprehension Task*: this is regarded as a programmer's aid rather than an attempt to understand for tutoring. Visualisation approaches do not provide information to aid comprehension. `Code Crawler` [Lanza *et al.* 2005b] and `CodeMap` [Kuhn *et al.* 2010] are examples of program visualisation tools.

INPUT-OUTPUT ANALYSIS Testing programs for correctness by entering values and observing expected outputs. *Drawback for Comprehension Task*: information provided is often not enough to explain bugs, especially in large programs because several bugs may lead to the same output. `CloudCoder` [Hovemeyer and Spacco 2013] is an example of a system that is based on IO testing. Chandra *et al.* [2011] also used a formal test-driven approach to search the space of possible program repairs as a means of debugging Java programs.

DATA FLOW ANALYSIS & OTHER COMPILER ANALYSIS Tracking values of variables and control flow in programs. *Drawback for Comprehension Task*: will not work in cases where the code is syntactically correct. Compilers only try to parse programs to get an output, not verify the logic in the code [Mohnen 2002].

PROGRAM SLICING Computing program pieces or parts — *known as program slices* — separately in an attempt to locate bugs [Krinke 2005]. Program slicing is widely used in program analysis, debugging and maintenance. *Drawback for Comprehension Task*: Similar to data flow analysis technique, slicing is effective for finding syntactic bugs but not sufficient to explain semantic bugs.

TYPE INFERENCE Matching variable types with the intention of finding bugs. *Drawback for Comprehension Task*: effective for debugging type mis-match errors and considered to be most adequate for finding syntax errors in compiler implementations. Information produced by this method is insufficient to explain semantic bugs. `Lackwit` [O'Callahan and Jackson 1997] is based on this technique.

KNOWLEDGE-BASE APPROACH Keeping a repository of plans, variations of plans and associated bugs, rules of discourse. This method was used in PROUST [Johnson and Soloway 1985] and PAT [Harandi and Ning 1988]. *Drawback for Comprehension Task*: tedious to pre-determine and expensive (almost impossible for large programs) for storing all possibilities in the search space of plan variations, thereby leading to high false acceptance/rejection rates and lesser scope of application. PROUST was only able to handle the rainfall problem.

PROGRAM SYNTHESIS This is the automated generation or construction of programs that satisfy specific non-algorithmic constraints. The constraints are often given in algebraic or logical calculus forms. *Drawback for Comprehension Task*: This technique resulted in poor factoring as many redundant in-code structures are not usually factored out. However, synthesis has been used and proven to produce realistic results for the comprehension problem. Srivastava *et al.* [2010] defined a novel approach to constraint-based program synthesis using a formal method. Solar-Lezama [2008] used a formal approach (based on specified rules) to synthesize programs from sketches using CEGIS (CounterExample-Guided Inductive Synthesis), and the automatic feedback generation presented by Singh *et al.* [2013] is also based on program synthesis.

SYMBOLIC EXECUTION OR RIGOROUS TESTING This approach is similar to the IO Testing approach with the difference that every program path/fragment/piece is tested to determine what input will cause them to execute. Each program piece or slice is executed using an interpreter that assumes an input for that piece. This input is often called a symbolic value. *Drawback for Comprehension Task*: This techique is not scalable and fails with large programs or programs with infinite loops as these type of programs generate infinite number of paths, and hence, infinite tests. Nguyen *et al.* [2013] presented an automated program repair method based on symbolic execution, constraint solving and program synthesis.

The approach introduced in this work is new and aided by the IO analysis and the knowledge-base approaches. Having said that this research is interested in semantically incorrect programs, IO Analysis does not suffice in isolation for verifying program correctness. Instead of manually enumerating the space of program variations, we have presented new formalisms to generate the space algorithmically, represented this space with DFAs, and presented new algorithms for finding bugs in these DFAs.

3.6 THE GAP

In this section, we point out outstanding problems in Novice Program Comprehension and Tutoring, given the published works in this domain. There exists a persistent need to explore and devise new/improved methods of doing the following:

1. generating a search space of all possible variants of the valid plans that offer correct solutions to a programming problem in contrast to inefficiently storing every known possibility — this does not exist prior to this work, and

2. seeking to adapt methods from formal aspects of computing (such as Automata Theory) that may be optimal for performing the task in (1) above.

3.7 SOLVING PROBLEMS WITH GRAPHS / DFA

Just as we have abstracted the program comprehension problem to DFAs (*see* Part iv), many real life problems have been abstracted to directed graphs to simplify them and also apply known graph algorithms to devise solutions to these problems. Ullman [2014] abstracted the game of tennis to a DFA that accepts a set of strings that lead to a game end to introduce undergraduate students to Automata Theory. Cycles in this DFA implied an endless game. Spohrer [1992] used a tree called the Goal and Plan Tree (or GAP tree) to define the space of correct novice programs. Krinke [2003] used control flow graphs or CFGs to depict control flow in programs with nodes representing statements and edges representing flows between them.

3.8 REFLECTIONS ON RELATED WORK

In this part of this thesis we have introduced the program comprehension problem, introduced terms and discussed the major successes in this domain. One of the contributions of this work is that we have defined a new type of program abstraction to aid program comprehension — we call this *narrations*. In Part ii, we discuss how *narrations* are generated from programs and how they are likely to aid program comprehension.

Part II

ABSTRACTING AND NARRATING NOVICE PROGRAMS

The Syntax-Free Approach to teaching programming was proposed in response to the perceived challenges often faced by novice programmers while taking their first course in programming. The idea of the SFA is to provide a level of abstraction over the language syntax and teach programming to novices as algorithms instead of lines of code. In this part, we report the development of a tool that translates novice programs into detailed textual algorithms using regular expressions. We refer to these algorithms as *narrations*. These narrations are syntax-free, can improve readability and aid the comprehension of programs. The technique described can also be employed for automatic generation of hints or tips for novice programmers during classroom or laboratory sessions.

This part contains two chapters. In Chapter 4, we describe the abstraction of novice programs to plans, and in Chapter 5, we present new algorithms and a tool for generating narrations from programs.

4

NOVICE PROGRAM ABSTRACTION

To aid program comprehension via source code abstraction, several tools have been developed in the past, offering two major types of approach: visualisation and summarisation [Storey 2006]. Visualisation tools provide a graphical description of a program, while summarisation tools provide a brief textual description of a program's functionality. The new tool described in this part does not exactly fall under either of these classifications; instead it provides a detailed description (in contrast to a summary) of the steps implied by the lines and fragments of a given program. We have chosen to refer to this as *narration*, although contextually, we mean detailed textual (algorithmic) descriptions of the source code.

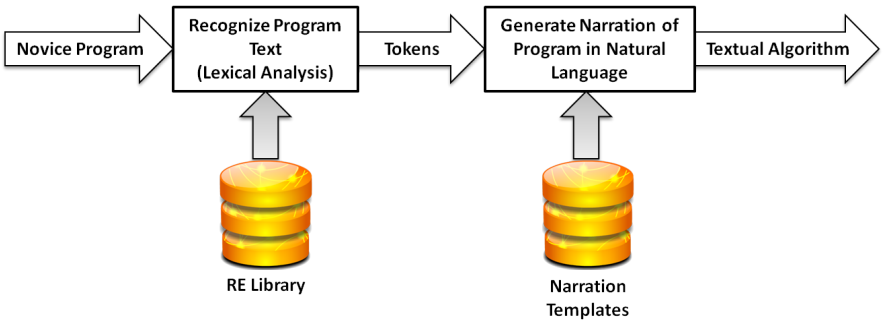


Figure 4.1: A new tool for novice program narration

In this chapter, we present a tool that takes novice programs (written in the C++ programming language) and translates them back to algorithms. A concise description of this system is shown in Figure 4.1. To do this, we have used regular expressions to recognize basic language constructs and programming plans in novice programs. Based on inferences made from the novice programs, we have also used a template-based narration algorithm to describe the program in plain text. Teaching programming with such plain texts (free of language constructs) is referred to as the Syntax-Free Approach [Fincher 1999].

4.1 NARRATIONS AS TEXTUAL DESCRIPTION OF ALGORITHMS

Narrations are, in many ways, different from program summaries or comments; they are detailed documentations of the source code, presented in a natural language. Sipser [2006] has referred to narrations as high-level descriptions of algorithms in natural language while ignoring the

implementation details. Unlike summaries, narrations are step-wise, not concise and in fact often longer (in length) than the program they represent. However, they are different from comments, because they may not contain as much semantics and "stories" as programmers may include in their comments. These features of a narration make it possible for the narration to be machine-generated. Consequently, narrations are syntax-free textual algorithms. The following is an example illustrating the difference between these representations.

Consider the code fragment written in C++ and shown in Listing 4.1:

Listing 4.1: Sum of first 10 integers

```
1 sum = 0;
2 for (int i = 1; i <= 10; i++)
3 {
4    sum = sum + i;
5 }
6 cout<<sum;
```

The following are examples of how the code fragment in Listing 4.1 may be described.

As a summary: The fragment displays the sum of all numbers between 1 and 10.

As a formal algorithm:

```
1 sum ← 0 ;
2 for integer i ← 1 to 10 do
3 |   sum ← sum + i ;
  end
4 Print sum;
```

Algorithm 4.1 : Sum of first 10 integers

As a narration:

```
Initialize a variable named sum to 0.
Start a loop that goes from 1 to 10 using a variable named i, stepping
through with 1.
In the loop, increment the variable sum by the value of i.
When the loop is done, display the value of sum.
```

Algorithm 4.2 : Narration of the sum of first 10 integers

Observe that the *summary* is a concise abstraction of the fragment in Listing 4.1, while Algorithm 4.1 (*the algorithm*) has some implementation details implied by its steps. However, in Algorithm 4.2 (*the narration*), the steps

are described textually. In this work, we have developed a tool that takes novice programs written in C++ (such as the fragment in Listing 4.1) and automatically generates a narration of the program (similar to the text in Algorithm 4.2). Some narrations generated by the new tool represent programming plans, therefore these terms — algorithm, plan, and narration — are used interchangeably in this chapter.

4.2 REGULAR EXPRESSIONS FOR PROGRAM ABSTRACTION

We advance to present a structure diagram that conceptualises how we have abstracted novice programs into higher levels using regular expressions. Figure 4.2 shows (not in detail) how we have decomposed C++ programs to levels of granularity that can be recognized by regular expressions. Table 4.1, Table 4.2 and Table 4.3 show a *few regular expressions*[1] (written in .Net RE syntax) used in recognising nodes on the structure diagram across the different levels of abstraction. Table 4.1 shows the grouping of the characters in novice programs into lexemes (such as identifiers, the assignment symbol, integers and so on). These lexemes are then mapped into tokens. Table 4.2 and Table 4.3 show further abstractions built on this level or on further levels of abstraction.

TOKEN	ABBREVIATION	RE (.NET)
Identifier	`ident`	`[A-Za-z_][A-Za-z0-9_]*`
End of line	`eol`	`\;`
White spaces	`spc`	`\s+`
Bracket open	`bra_open`	`\(`
Bracket close	`bra_close`	`\)`
Comma	`comma`	`\,`
Increment	`inc`	`\+\+`
Decrement	`dec`	`\-\-`
Relational Operators	`relop`	`\<\=|\< |\>\=|\> |\=\=|\!\=`
Arithmetic Operators	`op`	`\+|\-|\*|\/|\%`
Assignment Symbol	`ass_sym`	`\=`
Boolean Operators	`bool`	`((\&\&)|(||)|(\!))`
Float Value	`float_val`	`(\-?\d+\.\d+)`
Integer Value	`int_val`	`(\-?\d+)`
Boolean Value	`bool_val`	`(true|false)`
Import Library	`_lib`	`#include\s*\<([A-Za-z]+)\>$`

Table 4.1: First level of abstraction

1 The complete listing of regular expressions is presented in Appendix A.

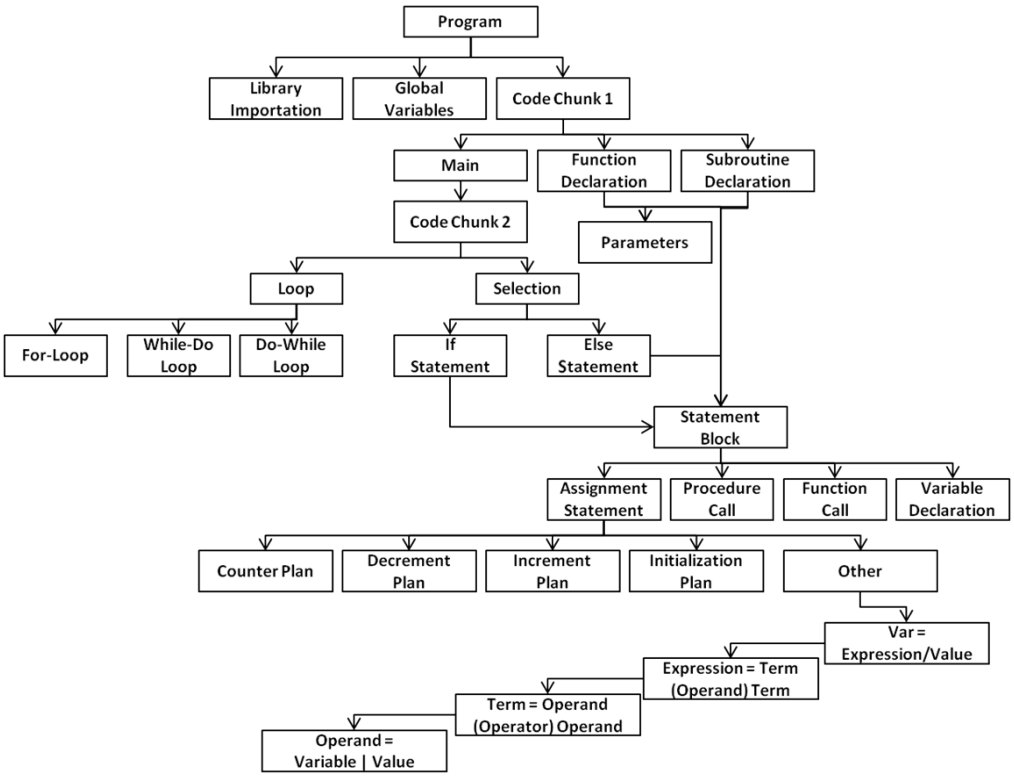


Figure 4.2: Structure of program abstraction

TOKEN	ABBREVIATION	RE (.NET)
Value	value	(int_val|float_val|bool_val)
Data type	datatype	(bool|int|float|double)
Parameter	parameter	((datatype)(spc)(ident)(spc)(comma)(spc))*((datatype)(spc)(ident))
Logical condition	condition	(bra_open)(ident)(spc)(relop)(spc)(value|ident)(bra_close)
If statement	if_stmt	(if)(spc)(condition)
While statement	while_stmt	(while)(spc)(condition)
List of variables or values separated by commas	ident_val_list	((ident|value)(spc)(comma) (spc))*(ident|value)

Table 4.2: Second level of abstraction

TOKEN	ABBREVIATION	RE (.NET)
Global variable	global_ident	(static)(spc)(datatype)(spc) (ident)(spc)(eol)
Function declaration	func_declr	(datatype)(spc)(ident)(spc)(bra_open)(spc)(parameter)*(spc)(bra_closed)
Subroutine declaration	sub_declr	(void)(spc)(ident)(spc)(bra_open)(spc)(parameter)*(spc)(bra_close)
Function call	func_call	(ident)(spc)(ass_symbol)(spc)(ident)(spc)(bra_open)(spc) (ident_val_list)?(spc)(bra_close) (eol)
Subroutine call	sub_call	(ident)(spc)(bra_open)(spc)(ident_val_list)?(spc)(bra_close)(eol)

Table 4.3: Third level of abstraction

4.3 ACTUALISATION

Regular expressions find application in many systems, for operations such as data validation, syntax highlighting, search and replace, and so on. The

modern design of some programming languages and environments (such as Microsoft's .Net framework) also offers RE libraries that are augmented with extra elements which make them able to recognise languages that cannot be stated by formal REs. Other programming languages with implementations of REs are: Perl, Java, C++ and IBM's ICU4C[2] [Heninger 2004]. In Chapter 5, we have used the RE library of the .Net framework in recognising C++ language constructs and novice plans, and built a program narration tool on this technology. The details of the .Net regular expression library can be found in [MSDN].

4.4 REFLECTIONS ON PROGRAM ABSTRACTION

Regular expressions are not always enough to recognise every possible statement in the C++ programming language. An example of such statements that cannot be recognised using REs are assignment statements that contain balanced parentheses[3]. This is not a hitch in our domain of application because we assume that all novice and expert programs in our domain are syntactically correct, hence, there is no need to recognise the whole statement. The abstraction technique therefore recognises the prefix of assignment statements with an expression such as: $\hat{}(\mathtt{ident})(\mathtt{spc})(\mathtt{ass\_sym})$ — any statement starting with an identifier followed by the assignment symbol.

In this chapter we presented a list of regular expressions for novice program abstraction as a step towards the conversion of programs to narrations. In Chapter 5, we discuss the details of this conversion process such as pre-processing, granulation, chunking, semantic rule look-up and narration composition. We also display results from the narration of popular novice programs.

2 International Components for Unicode for C, icu-project.org/apiref/icu4c/
3 The language of balanced parentheses is nonregular.

5

COMPREHENSION THROUGH NARRATIONS

In this chapter we present a new way of aiding novice programmers in comprehending programs by abstracting the source code to a natural language representation of the contained sequence of steps. This abstraction, as described in earlier chapters is called a *narration*. The source code is taken through the stages of clean up — using the regular expressions described in Chapter 4, and lexemes extraction and the lexemes are matched with a predefined repository of narration templates. The generated templates are expected to aid the readability of programs.

5.1 THE NOVICE PROGRAM NARRATOR

This section presents the design of the new tool. For ease of reference, we will refer to this system as NOPRON, an abbreviation for NOvice PROgram Narrator. As depicted in Figure 5.1, NOPRON processes novice programs in two phases. In the first phase, it uses a database of REs to preprocess (or normalize) the novice program, break it down to a certain level of granularity and build chunks of code from the contained lexemes. The second phase takes recognized tokens, matches them to a database of narration templates and generates syntax-free, textual algorithms. The first phase is further subdivided into three major operations on the source code, which are: preprocessing, granulation and chunking.

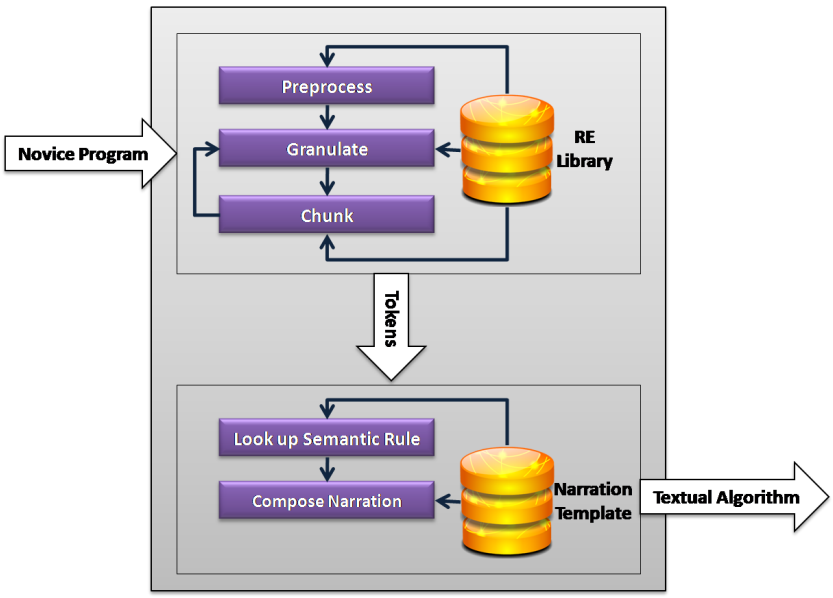


Figure 5.1: The novice program narrator

5.1.1 *Preprocessing*

In this stage, NOPRON simply removes redundant text from the source code. Redundant text is any text without which a program will function as before the removal; examples are line comments and prompts.

5.1.2 *Granulation*

After removing comments and prompts, NOPRON breaks down the source code into lines of code and splits condensed lines into atomic operations using delimiters specified in the RE library. An example of a condensed line is `float sum = 0;` this is split into two lines of code that represent two different operations, namely: `float sum;` and `sum = 0;`.

5.1.3 *Chunking*

NOPRON takes the result from the granulation module and groups the lines of code into blocks or higher levels of abstraction, using REs to recognise novice plans and multi-line structures (such as loops and selection constructs). At this stage, the statement `count = count + 1;` is no longer represented as an assignment statement, but as an *increment a counter* plan. Consecutive statements with varying lengths are represented as *statement blocks*. The second phase is divided into two major operations: semantic rule lookup and narration composition.

5.1.4 *Semantic Rule Lookup*

The chunks passed down from the previous stage are checked against a repository of semantic rules that map tokens to narration templates. If a match is found, the token is passed to the next stage for narration and if otherwise, the token is reported as an *unknown plan*. An unknown plan is usually the case when the input program contains some advanced structures of the programming language; for instance, object orientation.

5.1.5 *Narration Composition*

Here the narration template that matches a rule is applied to compose a syntax-free textual algorithm that is displayed as an output. A narration template is a predefined model with a certain degree of generalisation that is customised in real time to suit the states of the current program being narrated. A trivial example of a narration template for variable declaration is:

```
declare variableName as dataType.
```

In this example, the tokens `variableName` and `dataType` are the dynamic fields which are replaced by the specifics of the novice program being narrated. To ensure that NOPRON generates readable narrations, the composition module of the system formats the narrations into paragraphs. Operations in the source code are narrated as single line sentences and code fragments (or chunks) as paragraphs. This module also inserts a new line between paragraphs.

5.1.6 *Scope of NOPRON*

In this section we list the language constructs *understood* by NOPRON. NOPRON handles a subset (core language) of the C++ programming language version 4.8.1, released on 31st May 2013 [GCC]. The core of the C++ language covered by NOPRON includes:

STANDARD IO. Input/output operations using the `cin` and `cout` keywords.

STANDARD TEMPLATES. Iterators (`for`-loops, `while`-loops, etc), sequence containers (arrays), selection templates (`if` statements), functional templates (functions and procedures).

C STANDARD LIBRARY. Data types, strings, character classifications, file input/output, etc.

With this coverage, NOPRON could understand and narrate novice programs that are composed with constructs from the subset of C++ listed above.

5.2 IMPLEMENTATION AND RESULTS

We have implemented NOPRON as a windows application using the regular expression library made available by Microsoft Corporation in the .Net Framework [MSDN]. We tested NOPRON with a large variety of scenarios that illustrate the range of novice programs. The C++ programs used in testing were syntactically correct programs compiled with the online RiSE visual C++ compilation tool [RiSE]. Specifically, we tested NOPRON with popular novice programs such as the average problem, Soloway's rainfall problem [Ebrahimi 1994], and the sum program from Coco/R's Taste Compiler [Mössenböck 2010]. In this section we present selected example results obtained from the narration of these different programs.

5.2.1 *Average Problem*

NOPRON successfully narrated the *average of numbers* problem and its variations. Listing 5.1 shows the solution to a simple average problem, the average

of two numbers. This program was narrated by NOPRON and the output is displayed in Algorithm 5.1. A screenshot of NOPRON while performing this operation is shown in Figure 5.2.

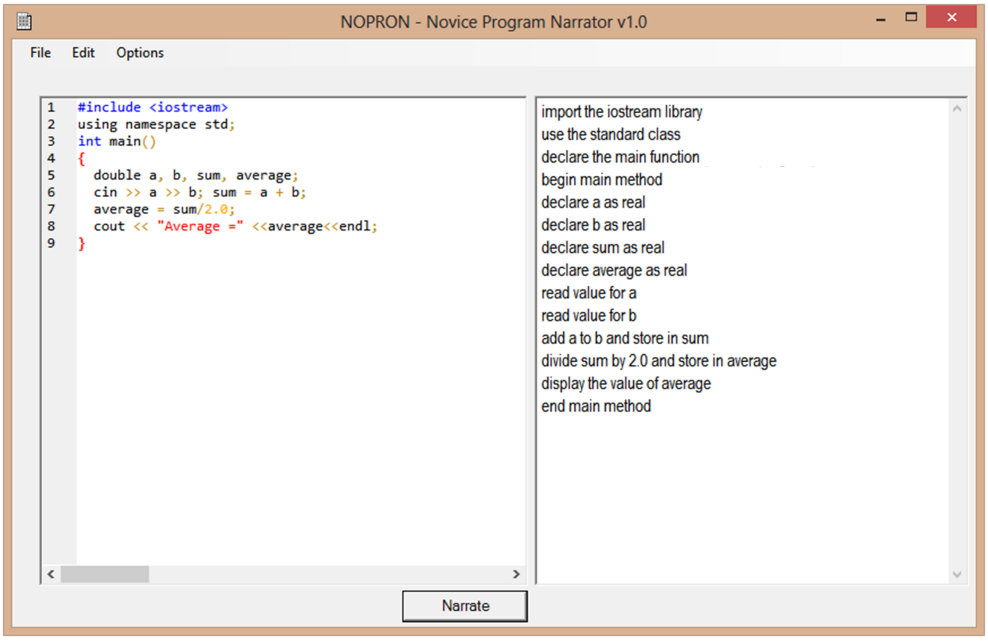


Figure 5.2: NOPRON in action, narrating the average of two numbers

Listing 5.1: Average of two numbers

```
#include <iostream>
using namespace std;
int main()
{
  double a, b, sum, average;
  cin>> a >> b; sum = a + b;
  average = sum / 2.0;
  cout<< "Average =" <<average<<endl;
}
```

```
1  import the iostream library
2  use the standard class
3  declare the main function
4  begin main method
5   declare a as real
6   declare b as real
7   declare sum as real
8   declare average as real
9  read value for a
10 read value for b
11 add a to b and store in sum
12 divide sum by 2.0 and store in average
13 display the value of average
14 end main method
```

Algorithm 5.1 : Narration of average of two numbers

Note that in Lines 5 to 8 of Algorithm 5.1, the variable declaration line (Line 5 of Listing 5.1) was narrated in four lines of text, each line representing one of the four atomic operations implied by the line from the source code. This is made possible by the granulation module of NOPRON. To demonstrate how NOPRON handles loops, we tested with a variation of the average problem: the average of n numbers (a list of numbers). A C++ program for this problem is shown in Listing 5.2 and its narration is shown in Algorithm 5.2.

Listing 5.2: Average of n numbers

```
1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5    int i, n;
6    double num, sum, average;
7    cin>> n;
8    sum = 0;
9    for (i = 0; i < n; i++){
10     cin>> num;
11     sum = sum + num;
12   }
13   average = sum / n;
14   cout<< average << endl;
15   return 0;
16 }
```

From the result shown in Algorithm 5.2, NOPRON could tell how many times a loop is repeated, while hiding the actual language construct used in the implementation of the loop.

```
1  import the iostream library
2  use the standard class
3  declare the main function
4  begin main method
5  declare i as integer
6  declare n as integer
7  declare num as real
8  declare sum as real
9  declare average as real
10 read value for n
11 initialize sum to 0
12 start a loop that goes from 0 to n-1 using the variable named i, stepping
   through with 1 (n iterations)
13 begin loop with variable i
14 read value for num
15 increment sum by num
16 end loop with variable i
17 divide sum by n and store in average
18 display the value of average
19 end main method
```

Algorithm 5.2 : Narration of average of n numbers

This abstraction is quite useful as it enables a novice to reconstruct the program in any chosen language with desirable constructs. Regardless of which control structure is to be adopted by the novice, the goal (from a syntax-free perspective) is the same, which is to implement a loop that has the same number of iterations in the target language. The resulting loop can therefore be a `for` or a `while`-loop.

Listing 5.3: Computing sum with a while loop

```
1 count = 0; sum = 0;
2 while (count < n)
3 {
4   cin>> num; sum += num;
5   count += 1;
6 }
```

Listing 5.3 describes a program fragment that implements a similar plan to Listing 5.2, but uses a different language construct — a `while` loop instead of the `for`-loop. The logic implemented with Lines 8 – 12 of Listing 5.2 is the same as the lines of Listing 5.3. Despite the difference in language constructs, NOPRON was able to narrate these fragments in plain text, successfully abstracting the syntax while retaining the semantics. Algorithm 5.3 shows the narration of the fragment in Listing 5.3.

```
1 initialize count to 0
2 initialize sum to 0
3 do the following lines repeatedly while count is less than n
4 begin while loop count
5 read value for num
6 increment sum by num
7 increment count by 1
8 end while loop count
```

Algorithm 5.3 : Sum fragment using `while` loop

5.2.2 *Soloway's Rainfall Problem*

In the early 1980s, Elliot Soloway (*see* [Ebrahimi 1994]) used a programming problem in teaching `Pascal` programming to his students at Yale University. This problem is now known as *Soloway's Rainfall Problem*. The statement of this problem has been discussed previously in Section 1.2.4 of Chapter 1. Any program that correctly solves Soloway's rainfall problem has been described as a typical novice program [Johnson and Soloway 1985, Ebrahimi 1994]. A solution to this problem, written in `Pascal`, was presented by Johnson and Soloway [1985]. We have translated this solution from `Pascal` to C++ — shown in Listing 5.4 — and narrated it using NOPRON. The resulting narration is shown in Algorithm 5.4.

Listing 5.4: Solution to rainfall problem in C++

```
#include <iostream>
using namespace std;
const int stop = 9999;
int main()
{
  double sum, rain, max, ave;
  int valid, rainy;
  sum = 0;
  valid = 0;
  rainy = 0;
  max = 0;
  cin>> rain;
  while (rain != stop) {
    if (rain < 0) {
      cout<< "invalid rainfall";
    } else {
      sum = sum + rain;
      valid = valid  + 1;
      if (rain > max) max = rain;
      if (rain > 0) rainy = rainy + 1;
      cin>> rain;
```

```
    }
  }
  cout<< valid <<" valid rainfalls were entered.";
  if (valid > 0) {
    ave = sum/valid;
    cout<< ave << "inches per day." << endl;
    cout<< "highest rainfall:" << max << " inches." << endl;
    cout<< rainy<< " rainy days." << endl;
  }
}
```

5.2.3 *Taste-Compiler's Sum Program*

We have also chosen to test NOPRON using a program from Mössenböck [2010], shown in Listing 5.5. The program, which calculates the sum of the first n integer numbers, was described as a sample program for a small compiler called *Taste*, developed with Coco/R, a compiler generator. We chose this program as a test case for NOPRON because its program text contains operations such as assignments, functions and procedure calls as well as `if` and `while` statements. Therefore a narration of this program — Algorithm 5.5 — shows how NOPRON handles these operations in novice programs.

Listing 5.5: Taste-Compiler: sum of first n numbers

```
#include <iostream>
using namespace std;
static int i;
void SumUp() {
  int sum;
  sum = 0;
  while (i > 0) {
    sum = sum + i;
    i = i - 1;
  }
  cout<< sum;
}
int Main() {
  cin>> i;
  while (i > 0) {
    SumUp();
    cin>> i;
  }
}
```

```
1  import the iostream library
2  use the standard class
3  declare a constant variable named stop as integer, with value 9999
4  declare the main function
5  begin main method
6  declare sum as real
7  declare rain as real
8  declare max as real
9  declare ave as real
10 declare valid as integer
11 declare rainy as integer
12 initialize sum to 0
13 initialize valid to 0
14 initialize rainy to 0
15 initialize max to 0
16 read value for rain
17 do the following lines repeatedly
18 while rain is not equal to stop
19 begin while loop rain
20 test if rain is less than 0. If yes,
21 then display prompt: invalid rainfall
22 if no, execute the following else block
23 begin else block
24 increment sum by rain
25 increment valid by 1
26 test if rain is greater than max.
27 If yes, then set max = rain
28 test if rain is greater than 0.
29 If yes, then increment rainy by 1
30 read value for rain
31 end of else block
32 end of while loop rain
33 display the value of valid
34 test if valid is greater than 0.
35 If yes, then execute the following if block
36 begin if block
37 divide sum by valid and store in ave
38 display the value of ave
39 display the value of max
40 display the value of rainy
41 end of if block
42 terminate the main function
43 end of main method
```

Algorithm 5.4 : Narration of the rainfall program

```
1  import the iostream library
2  use the standard class
3  declare a global variable named i as integer
4  declare a function named SumUp. This function returns nothing
5  begin function SumUp
6  declare sum as integer
7  initialize sum to 0
8  do the following lines repeatedly while i is greater than 0
9  begin while loop i
10 increment sum by i
11 decrease i by 1
12 end of while loop
13 display the value of sum
14 end of function SumUp
15 declare the main function
16 begin main method
17 read value for i
18 do the following lines repeatedly while i is greater than 0
19 begin while loop i
20 call the function named SumUp (this function takes no parameters) and
   returns nothing
21 read value for i
22 end of while loop i
23 end of main method
```

Algorithm 5.5 : Narration of Taste-Compiler's sum program

5.2.4 *Other Programs in NOPRON's Domain*

In addition to the programs and narrations presented in this section and due to NOPRON's ability to handle arrays, it was also able to narrate more complex novice programs such as:

1. *Sum and average of array elements*: a program that computes the sum and average of the elements in an array.
2. *Largest/smallest item of array elements*: a program that determines the largest item (or a similar program that determines the smallest item) in an array.
3. *Matrix arithmetic*: programs that compute the addition, subtraction or multiplication of matrices.
4. *Search and sort algorithms*: programs that implement the bubble sort, selection sort, insertion sort, and the binary search algorithms.

Furthermore, NOPRON could narrate any program provided its language constructs are included in the subset of the C++ language that we have covered.

5.3 LIMITATIONS

In this section we discuss what NOPRON cannot do. As demonstrated in the previous section, NOPRON can handle procedure declarations and calls, global variables, assignments, nested loops and expressions of different complexities. However, there are limitations to the abilities of the new system. Given that NOPRON handles only a subset of the C++ programming language, it cannot *understand* or *narrate* programs that are written with more complex structures or statements in the language such as object instances and inheritances, and other libraries. In cases where programs that contain these statements are given to NOPRON, it simply returns an output containing a correct narration of the known parts and indicates that certain lines or program fragments are unknown to it. Further possible improvements to NOPRON — such as narrating different loop types as the same text — is discussed in the Future Work section of Chapter 14 of this thesis.

5.4 REFLECTIONS ON NARRATIONS

In this chapter, we have presented the design and implementation of a tool that provides an abstraction of the syntax of novice programs written in C++, by translating the programs into syntax-free narrations. We have also described how this tool handles popular novice programs in an introductory programming course. This tool is expected to support the syntax-free style of teaching the subject in the following ways:

PROGRAM COMPREHENSION The narrations can enhance the readability and comprehension of programs. Students can read their statements in plain text and detect that their intention is not the same as what they read. A typical example of such a scenario is in the implementation of `for`-loops. A common novice semantic bug occurs with specifying the range of loops; for instance, the line `for (i=1; i<10; i++)` is often intended by novices as a range of 10 as opposed to 9; this is narrated explicitly by the new tool as *start a loop that goes from 1 to 9 using the variable named* i*, stepping through with 1 (9 iterations)* and a student can easily fix this bug by changing $<$ in the source code to $\leqslant$.

INSTRUCTION GENERATION & LANGUAGE ACQUAINTANCE Fragments of the narrations can be displayed to students as hints (electronically or in physical laboratory sessions). This is expected to assist them in conceptualising solutions (or offering a starting point) to problems. Similarly, narrations generated can be presented to students to test

their knowledge of language constructs by asking them to correctly translate narrations back to compiler-ready programs.

Automatic generation of narrations from novice programs as a comprehension aid as presented in Part ii (Chapter 4 and Chapter 5) is one of the major contributions of this work. Other contributions are:

1. formalisation of the space of program variations,
2. devising: new technique for comprehending programs using DFAs, and new algorithms for detecting semantic bugs from such DFAs.

In Part iii, we present a new formalism for defining variability in novice programs, and new algorithms for the iterative generation and validation of program variations.

Part III

SEARCHING THE SPACE OF NOVICE PROGRAM VARIATIONS

The world has many problems that cannot be solved with just one method. In computer programming, this is often the case. Arguably, the slightest programming problem one can think of can be solved in quite a number of ways using the same algorithm, each unique way being a mere variation resulting from a valid permutation of lines of the program text, choice of language construct, ordering of operation or even the use of the converse of relational conditions. These possibilities exponentially increase if we consider other different algorithms for solving the same problem. This wide variety makes it sometimes hard for a lecturer to determine, by inspection, if a student's program is semantically correct or if the program is the same as the model answer.

In this part of this thesis, we present a new formalism for enumerating finite possible variations of a lecturer's program. Using this formalism, we have developed an algorithm for generating all variations of any given novice program. To ascertain that the algorithm works, we also developed a simulation program that takes a novice program and determines the number of ways it can be written. These variations represent a *space* which, if abstracted to a DFA, can be used to answer certain questions about semantic errors in novice programs.

This part contains three chapters. Chapter 6 presents a new formalism that defines the variability in novice programs. Chapter 7 describes a new algorithm — called the Alternative Programs Enumeration (or APE) algorithm — for generating the alternative and equivalent programs based on the established formalism. The APE algorithm uses two filters to validate its generated programs. In Chapter 8, we discuss the two algorithms used by APE for filtering invalid permutations and testing generated programs.

6

VARIABILITY IN NOVICE PROGRAMS

One way to comprehend novice programs is to enumerate many possibilities — if not all — of a lecturer's solution and attempt to find a novice's program in this *space*. In this chapter, we present the derivation of a new formalism to specify the size of this space, as foundation to Chapter 7 where we will present an algorithm to automatically generate the novice programs in the space. This entire process is briefly illustrated in Figure 6.1. As we will see in Part iv, this space can be abstracted to a DFA and used to find semantic bugs in novice programs, hereby reducing the program comprehension problem to a formal graph problem — DFAs.

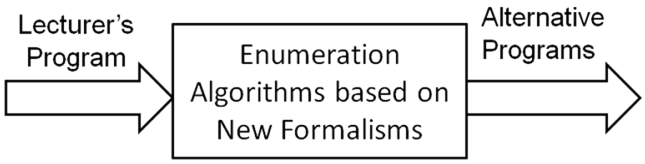


Figure 6.1: Generating alternative programs

In order to automatically comprehend novice programs, it is useful to examine how human experts (or lecturers) go about this *cognitive* process. This is the study of the human aspects involved in program comprehension or *Software Psychology* [Rugaber 1992]. Software psychology attempts to offer rich explanations about how different programmers (novices and experts) understand programs and hence suggest how tools can be built to support comprehension. The varying knowledge of rules of discourse between an expert (or lecturer) and a novice often results in the variability in programs written with the same plan. Here, we are interested in formalising the automatic *estimation* of the number of program variations, resulting from different choices made during *program composition*. These choices are *dependent* on programming styles and a programmer's knowledge of rules of discourse. If we can determine the variations (alternate and equivalent ways) of any given novice program, we can automatically determine if a novice program solves the same problem as an expert's program and hence, find semantic errors in buggy novice programs. The new formalisms for determining program equivalence presented in this chapter are based on theories from discrete mathematics and logic.

6.1 VARIABILITY WITH AN EXAMPLE

We further explain the idea of program variability with a case study of three *distinct* programs, written with the same plan. Consider the following novice programming problem:

> *Write a program in C++ that reads in an integer number* `n` *and sums up every number between* `1` *and* `n`*. If the sum is more than* `100`*, the program should display an output saying, "Sum is too large!" and halt, otherwise, the program should display the sum.*

One way of doing this, i.e. a known *plan*, is to start a program with variable declaration *formalities*, read in a value for the integer `n`, initialise a variable called `sum`, and start a loop that goes from `1` to `n` while summing up every value in the range within the loop. At the end of this loop, check if the `sum` is greater than 100 and display the required output depending on the truth value of the tested conditions.

The three programs in Listing 6.1, Listing 6.2, and Listing 6.3 show three different possibilities from the space of solutions to this problem. Two students, namely: Alice and Bob[1], whose distinct programs Listing 6.2 and Listing 6.3 are equivalent to the lecturer's program in Listing 6.1. In Listing 6.1, the lecturer has chosen to implement the loop plan with the `for`-loop construct while Alice made a choice of the `do`-loop and Bob went for the `while`-loop. Similarly, Bob omitted the prompt on Line 6 of the lecturer and Alice's programs. Bob also swapped the conditional statement used to check the size of the sum variable towards the end of the program (Lines 13–17) — instead of checking if the sum is within range, he did the reverse, checking if the sum was out of range.

These solutions can be represented by the DFA in Figure 6.2, given the set of states (representing the union of the semantic tokens of the three programs) in Table 6.1.

1 Alice and Bob are mere placeholders used as archetypal characters.

Listing 6.1: Lecturer's

```
#include <iostream>
using namespace std;
int main()
{
 int n, i, sum;
 cout<<"Enter n...";
 cin>>n;
 sum = 0;
 for(i=1; i<=n; i++)
 {
  sum += i;
 }
 if (sum <= 100){
  cout<<"sum = " <<
      sum;
 }else{
  cout<<"Sum is too
      large!";
 }
 return 0;
}
```

Listing 6.2: Alice's

```
#include <iostream>
using namespace std;
int main()
{
 int n, i, sum;
 cout<<"Enter n...";
 cin>>n;
 sum = 0;
 i = 0;
 do{
  sum = sum + i;
  ++i;
 } while (i <= n);
 if (sum < 101){
  cout<<"sum = " <<
      sum;
 }else{
  cout<<"Sum is too
      large!";
 }
return 0;
}
```

Listing 6.3: Bob's

```
#include <iostream>
using namespace std;
int main()
{
 int n, i, sum;
 cin>>n;
 sum = 0;
 i = 1;
 while (i <= n){
  sum = sum + i;
  i = i + 1;
 }
 if (sum > 100){
  cout<<"Sum is too
      large!";
 }else{
  cout<<"sum = " <<
      sum;
 }
 return 0;
}
```

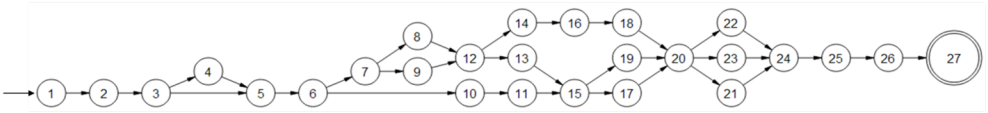


Figure 6.2: DFA for the Lecturer's, Alice's, and Bob's Solutions

Could there be other ways of writing this program that are different from the three programs in Listing 6.1, Listing 6.2, and Listing 6.3? Of course there are other paths on the DFA from the *start state* (State 1) to the *final state* (State 27) representing unique programs other than the lecturer's, Alice's or Bob's. In fact, the DFA (constructed with the union of these programs) now has 48 distinct programs (or paths) resulting from the product of its branches — even though not all of these paths will lead to correctly composed programs. Defining a formal approach for searching this space is the goal of this chapter.

6.2 FORMALISATION OF PROGRAM EQUIVALENCE

In this section we present a new formalism for defining equivalent programs. Given one program, what are the alternative and equivalent programs? Our program equivalence formalism is based on three granular levels of the input program, namely: single statement level, statement block level (group of

STATE MARKERS	SEMANTIC TOKENS
1	`#include<iostream>`
2	`int main() {`
3	`int n, i, sum;`
4	`cout<<"Enter n";`
5	`cin>> n;`
6	`sum = 0;`
7	`i = 1;`
8	`do {`
9	`while (i <= n) {`
10	`for (i = 1; i <= n; i++){`
11	`sum += i;`
12	`sum = sum + i;`
13	`i = i + 1;`
14	`++i;`
15	`}`
16	`} while (i <= n)`
17	`if (sum <= 100) {`
18	`if (sum < 101) {`
19	`if (sum > 100) {`
20	`cout<<"sum = "<< sum;`
21	`} else`^{1} `{ //for State 17`
22	`} else`^{2} `{ //for State 18`
23	`} else`^{3} `{ //for State 19`
24	`cout<<"Sum is too large!";`
25	`}`
26	`return 0;`
27	`}`

Table 6.1: Abstraction of program lines

statements, loops, `if` statement blocks etc), and the entire program (or main program).

1. For statements at the lowest granularity level, in what other valid ways can they be written?

2. For statement blocks, is there any similar construct to the used loop or a valid re-ordering of the statements enclosed in the block?

3. For the entire program, are there other permutations of function and main methods arrangement that still work fine?

These are the questions that will be answered with the new formalism. We will specify program equivalence using variations generated with: truth tables (for logical expressions), language constructs (using the rules of discourse in C++), permutations (ordering of statements that does not alter program sequence/value) and granularity (condensed and well enumerated

statements) templates. A combination of these enumeration techniques will be generated, representing the solution space of a novice program.

There are two broad categories of statements in novice programs written with procedural C++, namely; simple and compound statements [Smith, Malik 2010]. These two broad categories can be cascaded to smaller subcategories as shown in Figure 6.3. This work only considers statements composed using the core of the C++ programming language as they are assumed to be mostly used by novices in writing programs [Johnson and Soloway 1985, Ade-Ibijola *et al.* 2014a].

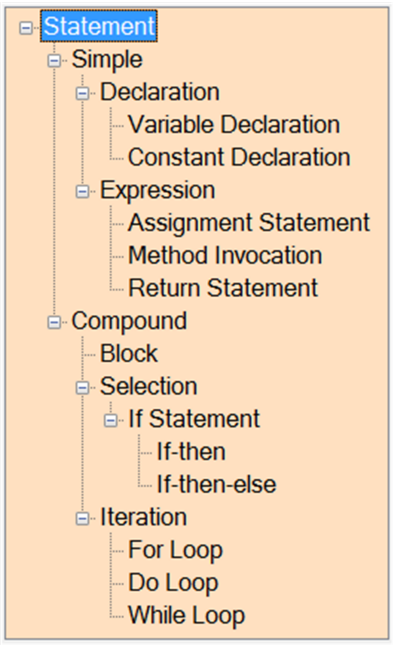


Figure 6.3: Statement categories

6.2.1 *Simple Statement Equivalence*

We proceed to define the term *Equivalent Statement* in the C++ programming language in Definition 6.1.

Definition 6.1 (Equivalent Statement)**.** Let $\{S_1, S_2, \ldots, S_n\}$ be a set of statements in an imperative programming language. We write:

$$S_1 \equiv S_2 \equiv \ldots \equiv S_n,$$

if and only if the execution of $S_1, S_2, \ldots S_n$, denoted by $\mathcal{E}(S_1), \mathcal{E}(S_2), \ldots, \mathcal{E}(S_n)$ causes the same change in the computer memory, i.e.,

$$\mathcal{E}(S_1) \equiv \mathcal{E}(S_2) \equiv \ldots \equiv \mathcal{E}(S_n).$$

S_1 is the base statement, the primarily known statement from which other equivalent statements are derived using rules of discourse (or the knowledge of language syntax).

From Definition 6.1, an example of equivalent statements to increment the value of a variable in C++ is shown in Table 6.2[2].

PROGRAM PLAN	BASE STATEMENT	EQUIVALENCE SETS (TEMPLATES)
`Increment variable named 'a' by an integer '1'`	$S_1 : a = a + 1;$	$S_2 : ++a;$ $S_3 : a+ = 1;$ $S_4 : a++;$

Table 6.2: Equivalent statements

Granulation is taken into consideration in the evaluation of equivalent statements. An example is the way variables are declared in C++. The statement: `int a = 5;` is equivalent to the dual: `int a; a = 5;`. Granulation is handled with the procedures described in Chapter 4 and Chapter 5 (*see also* Ade-Ibijola *et al.* [2014a]), used for program narration.

6.2.2 *Compound Statement Equivalence*

Beyond the atomic nature of simple statements are compound statements which include statement blocks, selection and iterative statements. In this section we formalize the equivalence of these statements. We define equivalent statement blocks in C++ in Definition 6.2.

Definition 6.2 (Equivalent Statement Block)**.** Let n be an integer, then a statement block, denoted by B is a finite sequence of consecutive statements $[S_1, S_2, \ldots, S_n | n \geqslant 2]$. Let m be an integer, then a finite set of statement blocks $\{B_1, B_2, \ldots, B_m | m \geqslant 2\}$ are equivalent if and only if the execution of the individual blocks $B_i, i = 1, \ldots, m$, denoted by $\mathcal{E}(B_1), \mathcal{E}(B_2), \ldots, \mathcal{E}(B_m)$ causes the same change in the computer memory, i.e.,

$$\mathcal{E}(B_1) \equiv \mathcal{E}(B_2) \equiv \ldots \equiv \mathcal{E}(B_m).$$

Here

2 The syntaxes described by these statements work differently in the language (C++) implementation, i.e. `++a` and `a++` differ in the sense that `++a` happens before other statements in context, while `a++` occurs after other statements in context.

$$B_1 = [S_{11}, S_{12}, \ldots, S_{1n} | n \geqslant 2],$$
$$B_2 = [S_{21}, S_{22}, \ldots, S_{2n} | n \geqslant 2], \ldots$$
$$B_m = [S_{m1}, S_{m2}, \ldots, S_{mn} | m, n \geqslant 2]$$

Definition 6.3 (Equivalence of Statement Blocks with Reordered Statements)**.** Let $\{B_1, B_2, \ldots, B_m | m \geqslant 2\}$, be a set of statement blocks, with each block containing the same set of statements as the other blocks but in another order (unique permutations), e.g. $B_1 = [S_1, S_2], B_2 = [S_2, S_1]$. We say that the statement blocks $\{B_1, B_2, \ldots, B_m\}$ are equivalent if and only if Definition 6.2 is satisfied for all $S_{ij}, 2 \leqslant i \leqslant m, 2 \leqslant j \leqslant n$ contained in blocks B_i. Here m is the number of statement blocks and n is the number of statements per block.

Example 6.1 (Equivalence of Statement Blocks with Reordered Statements)**.** We hereby give a description of reordered but equivalent statement blocks with an example of a C++ program that calculates the sum and average of n numbers.

A program written in C++ to compute the average of a set of numbers has a *plan* that should perform the logical steps shown in Table 6.3, but not necessarily in that order[3].

LOCAL PLAN SYMBOL	LOCAL PLAN DESCRIPTION
a	Import Libraries
b	Declare and Start Main
c	Declare i and n as integers
d	Declare sum and num as real
e	Initialize sum to zero
f	Read n
g	Begin loop with i from 1 to n
h	Read num
i	Increment sum with num
j	End Loop
k	Compute and Output Average
l	Halt

Table 6.3: Plan for average of numbers problem

3 For illustration purposes, Table 6.3 assumes that declaring variables of the same type is one atomic operation at the lowest granularity level (see local plan symbols 'c' and 'd'), but in reality, every variable can be declared in a self-contained program statement which means plans 'c' and 'd' can be further divided into two local plans each.

By reordering the local plan symbols 'c', 'd', 'e' and 'f' as presented, we can obtain different solutions. These correspond to the different variations of the plan that a student may employ in the quest for the solution to this problem. Figure 6.4 shows 24 different orderings (4!) of these steps and we have identified six valid permutations (statement blocks). The other 18 invalid orderings correspond to misconceived intentions of a novice who may be aware of the steps to be taken but has problems with *plan composition* – the order in which program statements in a block should be arranged. In this case, the rule of discourse is: *a variable must be declared before it is used*, and this rule is employed in eliminating invalid orderings from the solution space. More on eliminating invalid orderings is discussed in Chapter 7 and Chapter 8.

1 Correct	int i, n; float num, sum; sum = 0; cin>>n;	9 Correct	float num, sum; sum = 0; int i, n; cin>>n;	17	sum = 0; cin>>n; int i, n; float num, sum;
2 Correct	int i, n; float num, sum; cin>>n; sum = 0;	10	float num, sum; sum = 0; cin>>n; int i, n;	18	sum = 0; cin>>n; float num, sum; int i, n;
3	int i, n; sum = 0; float num, sum; cin>>n;	11	float num, sum; cin>>n; int i, n; sum = 0;	19	cin>>n; int i, n; float num, sum; sum = 0;
4	int i, n; sum = 0; cin>>n; float num, sum;	12	float num, sum; cin>>n; sum = 0; int i, n;	20	cin>>n; int i, n; sum = 0; float num, sum;
5 Correct	int i, n; cin>>n; float num, sum; sum = 0;	13	sum = 0; int i, n; float num, sum; cin>>n;	21	cin>>n; float num, sum; int i, n; sum = 0;
6	int i, n; cin>>n; sum = 0; float num, sum;	14	sum = 0; int i, n; cin>>n; float num, sum;	22	cin>>n; float num, sum; sum = 0; int i, n;
7 Correct	float num, sum; int i, n; sum = 0; cin>>n;	15	sum = 0; float num, sum; int i, n; cin>>n;	23	cin>>n; sum = 0; int i, n; float num, sum;
8 Correct	float num, sum; int i, n; cin>>n; sum = 0;	16	sum = 0; float num, sum; cin>>n; int i, n;	24	cin>>n; sum = 0; float num, sum; int i, n;

Figure 6.4: Space of re-ordered implementations

Definition 6.4 (Equivalent `if` Statements)**.** A finite set of `if` statements $\{q_1, q_2, \ldots, q_n\}$ are said to be equivalent if and only if every $q_i, 1 \leqslant i \leqslant n$ in the set has the same truth value as all other members of the set.

We will evaluate the equivalence of `if` statements by using logical equivalence laws (presented in Definition 2.5 of Chapter 2), truth tables and the converse of expressions. To do this, it is first necessary to highlight the behaviour of the six main logical operators available in the C++ programming language with respect to the Negation and Converse relations respectively.

p	¬ p	C(p)
>	<=	<
<	>=	>
==	!=	==
>=	<	<=
<=	>	>=
!=	==	!=

Table 6.4: Negation and converse operators

Table 6.4 shows the distinction between the NOT operator (¬) and the converse relation (C) in C++. For example, the negation of `if (a > b)` is "if a is not greater than b" written as `if !(a > b)` or `if (a <= b)`. Novice programmers can make different valid choices of semantic composition using relational operator equivalence as shown in Table 6.4.

Example 6.2 (Equivalent `if` Statements)**.** Let q be a logical condition included in an `if` statement in a C++ program of the form:

```
if (logical_condition) {
```

and let C(q) represent the converse of the logical condition q. Then we evaluate the alternative versions of q as presented in Table 6.5.

q	¬ q	C(q)	¬ C(q)	¬(¬ q)	¬(¬ C(q))
T	F	T	F	T	T
F	T	F	T	F	F
*		*		*	*

Table 6.5: Evaluating equivalent conditional statements with truth tables

The equivalent columns in the truth table are marked with an asterisk '*'. In this example, if q = `if (a > b)` is the base conditional statement, then the equivalent set will include the values shown in Table 6.6. From Table 6.6, S_2 is C(q), S_3 is ¬(¬ q), and S_4 is ¬(¬ C(q)). Conditional statements in C++ with larger compartments (having more than one logical condition) such as `if (a > b) && (c != d)` are expected to have more elements in their equivalent sets respectively. Example 6.3 shows a generic conditional statement with two compartments (logical conditions p and q).

Example 6.3 (Composite Logical Conditions)**.** Let r be a composite logical condition included in an `if` statement in a given C++ program, consisting of two sub-conditions (or compartments) p and q, connected with a logical operator of the form:

```
if(<condition1><logical_operator><condition2>){
```

Program Plan	Base Statement	Equivalence Set (templates)
Test if a is greater than b	S_1: if (a > b)	S_2: if (b < a) S_3: if !(!(a > b) S_4: if !(!(b < a)

Table 6.6: Equivalent set from base statement

Here p corresponds to <condition$_1$> and q corresponds to <condition$_2$>. Let C(p) and C(q) represent the converse of the logical expressions p and q respectively. Then r is given as:

r ← p <logical_operator> q

Here <logical_operator> ∈ {AND, OR}. Then we enumerate the alternative versions of r using the truth table presented in Table 6.7. Table 6.7 represents the evaluation of the truth values of p against q using the converse and negation operators respectively. Then r is given in Table 6.8 and Table 6.9.

p	q	C(p)	C(q)	¬ p	¬ q	¬ C(p)	¬ C(q)
T	T	T	T	F	F	F	F
T	F	T	F	F	T	F	T
F	T	F	T	T	F	T	F
F	F	F	F	T	T	T	T

Table 6.7: Truth values of p against q

Table 6.8 shows the truth values of r if the logical operation contained in r is disjunction (OR operator), derived from the columns of Table 6.7. To generate more alternative logical statements for p and q, we proceed to consider the converse of the logical expressions. If the converse of p and q are evaluated instead of p and q, we have Table 6.9.

p ∨ q	¬ p ∨ q	p ∨ ¬ q	¬ p ∨ ¬ q
T	T	T	F
T	F	T	T
T	T	F	T
F	T	T	T
A_1			

Table 6.8: Truth values of p OR q

By inspection, we observe the equality of the truth values of the Table 6.8 and Table 6.9. This implies an alternative *space*.

Let,

C(p)∨C(q)	¬C(p)∨C(q)	C(p)∨¬C(q)	¬C(p)∨¬C(q)
T	T	T	F
T	F	T	T
T	T	F	T
F	T	T	T
A_2			

Table 6.9: Truth values of converse of p OR q

```
r = if (a > b) || (c != d)
```

Then r is identical to A_1 with p corresponding to (a > b) and q corresponding to (c != d). Since $A_1 = A_2$, this implies that:

$$p \vee q \equiv C(p) \vee C(q)$$

Therefore we state that r can also be written as:

```
if (b < a) || (d != c).
```

Other equivalent expressions include:

$$C(p) \vee q \equiv p \vee C(q),$$

which corresponds to:

```
if (b < a) || (c != d), and
  if (a > b) || (d != c).
```

Conditional statement r therefore has four valid equivalent statements when considering only the converse of the logical expressions contained. This space can be increased by applying the commutative laws (i.e. $p \vee q \equiv q \vee p$ and $p \wedge q \equiv q \wedge p$) to obtain four more variations:

1. `if (c != d) || (a > b)`,
2. `if (d != c) || (b < a)`,
3. `if (c != d) || (b < a)`, and
4. `if (d != c) || (a > b)`.

At this stage, r has a total of eight variations. Further variations can be obtained if the conditional statement r contains the NOT operator (negation) using the De Morgan's laws:

$$\neg(p \wedge q) \equiv \neg p \vee \neg q \text{ and } \neg (p \vee q) \equiv \neg p \wedge \neg q.$$

Remark 6.1 (Generalisation of Definition 6.4)**.** We proceed to extend the idea of equivalent `if` statements with respect to the converse relation, the Commutative, De Morgan's, Absorption and Distributive laws. Let `C(p)` be the converse of the logical expression `p` and `C(q)` be the converse of the logical expression `q`. Then $p \equiv C(p)$, $q \equiv C(q)$, and for all logical expressions L involving p and q, denoted by $L(p, q)$,

$$L(p, q) \equiv L(C(p), C(q)).$$

Given more logical expressions, $p_i | i \geqslant 3$,

$$L(p_1, p_2, \ldots, p_n) \equiv L(C(p_1), C(p_2), \ldots, C(p_n)).$$

This is an extension of the converse relation over logical statements. Other generalisations of Definition 6.4 are implied by the Logical equivalence laws [Mendelson 1997] described earlier in Chapter 2.

In general, given an `if` statement in C++ of the form in Listing 6.4, a template for generating equivalent `if` statements is given in Listing 6.5.

Listing 6.4: Syntax of `if` statement in C++

```
1 if (condition) {
2    A;
3 }
```

Listing 6.5: Template for equivalent `if` statement

```
1 if (Alt(condition)) {
2    A;
3 }
```

Here A is a simple statement on its lowest granular level or a statement block (as defined in Definition 6.2) and the `Alt` function enumerates the alternative set, consisting of equivalent logical expressions as defined in Definition 6.4.

Definition 6.5 (Equivalent `if-then-else` Statements)**.** We will describe the equivalence of `if-then-else` statements using templates. An `if-then-else` statement in C++ is of the form shown in Listing 6.6. Listing 6.7, Listing 6.8, and Listing 6.9 show three different templates for generating equivalent `if-then-else` blocks.

Listing 6.6: Syntax of `if-then-else` statement in C++

```
1 if (condition) {
2    A;
3 } else {
4    B;
5 }
```

Listing 6.7: Template 1 for equivalent `if-then-else` statement

```
1 if !(condition) {
2    B;
3 } else {
4    A;
5 }
```

Listing 6.8: Template 2 for equivalent `if-then-else` statement

```
if (Alt(condition)) {
   A;
} else {
   B;
}
```

Listing 6.9: Template 3 for equivalent `if-then-else` statement

```
if !(Alt(condition)) {
   B;
} else {
   A;
}
```

Similar to Definition 6.5, the `Alt` function enumerates the alternative set, derived by exploring logical options. The `Alt(condition)` in the above templates is given as a set of alternative conditions $C_i, i \geqslant 2$. Each template with more than one equivalent condition(s) will have the same template implemented n times, given any set $\{C_n, n \geqslant 2\}$. For example, if $n = 3$, this implies that there is one condition with two other equivalent conditions and this results in three distinct implementations of the template.

Definition 6.6 (Loop Iterator)**.** A loop iterator L can be defined as a 4-tuple:

$$L = \{type, interval, step, block\_pos\}.$$

Here *type* is the type of loop construct, an element in $\{\hat{f}, w, d\}$, representing `for`, `while`, or `do` loop constructs respectively; *interval* is the numerical range of the iterator represented as an *open* or *closed*, *lower* or *upper* interval; *step* is the increment, and *block_pos* is the position of the statement block with respect to the incrementing statement, i.e. the indicator whether statements contained in the loop should be executed before the iterator is incremented or after. Block positions are given as a member of the set $\{\eta, \alpha, \beta\}$ where η, α, and β correspond to *not applicable*, *statement block before incrementer* and *statement block after incrementer* respectively. A block position is η (not applicable) if the loop type is a `for`-loop construct, because the iterator carries the incrementing statement and no variable in the loop controls the iteration length.

Similarly, *interval* is given as follows:

- $[initial, final]$ for closed lower and upper numeric boundaries, e.g. `for (i=1; i<=10; i++)`, or
- $[initial, final)$ for closed lower and open upper numeric boundaries, e.g. `for (i=1; i<10; i++).`

Remark 6.2 (No iterator with open lower boundaries)**.** We assume that there are no open lower boundaries in any of the intervals because this is the case with loops in imperative programming languages, C++ inclusive. Similarly, but for scope sake, we assume that loops are incremented with integer values.

Definition 6.7 (Equivalent Loop Iterators)**.** Two loop iterators, L_1 and L_2, are equivalent if and only if they execute exactly the same number of times, regardless of program input. This is dependent on the type of loop, the termination condition and the variable that keeps track of the number of iterations. We will formalise this using the interval notation described in Definition 2.9 of Chapter 2.

We proceed to define the base case for iterators as follows: Let L_1 be a `for`-loop iterator, starting from an arbitrary initial integer value `i` and proceeding to some final integer value `f`, with an increment `k`. Then L_1 is our base case defined as:

$$L_1 = \{\hat{f}, [i, f], k, \eta\}.$$

L_1 formalizes the C++ loop construct of the form described in Listing 6.10 — in this case, $k = 1$.

Listing 6.10: Base-case of loop-iterators

```
1 for (int i = initial; i <= final; i++){
2    statement_block;
3 }
```

We can derive other equivalent loop iterators from L_1 as follows:

$$L_2 = \{\hat{f}, [i, f + 1), k, \eta\}.$$

L_2 formalises the loop construct of the form described in Listing 6.11.

Listing 6.11: Loop-iterator L_2

```
1 for (int i = initial; i < final + 1; i++){
2    statement_block;
3 }
```

Also, in L_2, $k = 1$. Other equivalent loop iterators L_i in this space are:

- $L_3 = \{w, [i, f], k, \alpha\}$,
- $L_4 = \{w, [i, f + 1), k, \alpha\}$,
- $L_5 = \{w, [i - 1, f), k, \beta\}$,
- $L_6 = \{w, [i - 1, f - 1], k, \beta\}$,
- $L_7 = \{d, [i, f], k, \alpha\}$,
- $L_8 = \{d, [i, f + 1), k, \alpha\}$,
- $L_9 = \{d, [i - 1, f), k, \beta\}$,
- $L_{10} = \{d, [i - 1, f - 1], k, \beta\}$.

The templates for $L_i, i \leqslant 3 \leqslant 10$, are given by the code fragments shown from Listing 6.12 to Listing 6.19.

Listing 6.12: Iterator: L_3

```
int i;
i = initial;
while (i <= final){
   statement_block;
   i = i + 1;
}
```

Listing 6.13: Iterator: L_4

```
int i;
i = initial;
while (i < final + 1) {
   statement_block;
   i = i + 1;
}
```

Listing 6.14: Iterator: L_5

```
int i;
i = initial - 1;
while (i < final) {
   i = i + 1;
   statement_block;
}
```

Listing 6.15: Iterator: L_6

```
int i;
i = initial - 1;
while (i <= final - 1) {
   i = i + 1;
   statement_block;
}
```

Listing 6.16: Iterator: L_7

```
int i;
i = initial;
do {
   statement_block;
   i = i + 1;
} while (i <= final);
```

Listing 6.17: Iterator: L_8

```
int i;
i = initial;
do {
   statement_block;
   i = i + 1;
} while (i < final + 1);
```

Listing 6.18: Iterator: L_9

```
int i;
i = initial - 1;
do {
   i = i + 1;
   statement_block;
} while (i < final);
```

Listing 6.19: Iterator: L_{10}

```
int i;
i = initial - 1;
do {
   i = i + 1;
   statement_block;
} while (i <= final - 1);
```

We generalise the equivalence of these loop iterators by writing:

$$L_1 \equiv L_2 \equiv \ldots \equiv L_{10}.$$

Therefore, for every loop iterator L in a novice's program, there are at least nine other ways the loop may be implemented in C++ and this space is formally denoted by the iterators L_i. Given any $L_j, 1 \leqslant j \leqslant 10$, as the base case in a lecturer's program, the set $\{L_i, i \in [1, 10], i \neq j\}$, is the equivalent set of iterators.

6.3 EQUIVALENT PROGRAMS

In this section we present a formalism for generating the equivalent sets of alternative programs to a given lecturer's program, using formal language theory notations and based on the previously stated definitions. Let P_0 be a lecturer's program, in particular, a solution in C++. Then, the following definitions hold.

Definition 6.8 (Language of Alternative Programs)**.** The *language* of alternative programs to P_0, representing all valid program strings to a novice programming problem is given as: $\mathcal{L}_p = \{P_1, P_2, P_3, \ldots, P_n\}$.

Definition 6.9 (Language of All Solutions)**.** Let Σ_s be an alphabet of semantic tokens, then the *language* of all solutions to a novice programming problem (which includes the lecturer's solution), over Σ_s is given as:

$$\mathcal{L}_s = \{P_0\} \cup \mathcal{L}_p.$$
$$\mathcal{L}_s = \{P_0\} \cup \{P_1, P_2, P_3, \ldots, P_n\}.$$

Since a program contains a sequence of chunks $C_0C_1C_2 \ldots C_n$, we re-write $\mathcal{L}_s$ as the concatenation of the language of alternative chunks as follows:

$$\mathcal{L}_s = \mathcal{L}_{C_0} \cdot \mathcal{L}_{C_1} \cdot \mathcal{L}_{C_2} \cdot \ldots \cdot \mathcal{L}_{C_n},$$

Here

$$\mathcal{L}_{C_i} \mid 0 \leqslant i \leqslant n, n \in \mathbb{N}_+ = \{w \in \Sigma^+, w \simeq C_i\}$$

n is the number of chunks in P_0, and chunks C_i contain sequences of statements $\{S_{i_1}, S_{i_2}, S_{i_3}, \ldots, S_{i_{n_i}}\}$.

Definition 6.10 (Equivalent Programs Redefined)**.** Any two program strings in $\mathcal{L}_s$ are equivalent and satisfy the two conditions given in Definition 2.22.

6.4 REFLECTIONS ON VARIABILITY

In this chapter, we have formalised the enumeration of the *space* of alternative programs using formal language notations. This space represents novice programs' possibilities — *a knowledge of what the solutions can be*. In the next chapter, Chapter 7, we present an algorithm that enumerates these programs.

7 ITERATIVE GENERATION OF PROGRAM VARIATIONS

In this chapter, we present an algorithm — based on the new formalisms presented in Chapter 6 — that exploits the *top-down theory of program comprehension* by grouping novice program fragments into chunks that represent three abstraction levels. The lowest level consists of the single line statements in the program (e.g. assignment statements), the middle level are the statement blocks (e.g. loops, `if` statements) and the top level consists of the functions (main method and declared methods). The new algorithm automatically generates valid variations of program chunks. Variations of each chunk are collectively defined as the equivalence set of the chunk. The algorithm further creates a space of alternative solutions to the entire program by computing the concatenation of the language of the chunks that make up the program. The space of solutions generated by the new algorithm represents the alternative ways a novice may uniquely write a program using the same algorithm as the lecturer but making different choices of *language constructs* and *plan composition* while exercising his/her limited or extensive knowledge of the rules of discourse.

7.1 THE ALTERNATIVE PROGRAMS ENUMERATION ALGORITHM

In this section we present an algorithm for enumerating equivalent novice programs, based on the previously specified formalism. The idea is to take a lecturer's program and preprocess it, then look up the equivalent sets of its language constructs, and formulate alternative implementations of the same program from the product of its different alternative parts (statements and chunks). This process can be further broken down into four major operations:

1. preprocessing, granulation and source text recognition,
2. alternate program generation,
3. enumeration of valid reordered plans, and
4. elimination of incorrect programs.

PREPROCESSING, GRANULATION, AND RECOGNITION. Preprocessing is the cleaning up of source code by removing redundant text such as comments. Granulation is the breaking down of language constructs

to the lowest atomic operations. Source text recognition is using regular expressions to recognize statements of the program text. To perform these operations on programs, we have adapted the preprocessor, granulator and text recognition modules used by NOPRON for these operations.

ALTERNATE PROGRAMS GENERATION. This is the generation of alternative programs using the new program equivalence formalism.

ENUMERATION OF VALID REORDERED PLANS. Here we used a permutation algorithm that takes the `statement_block` objects of length n (from the alternate programs generated) and returns a finite set of length $n!$ of reordering.

ELIMINATION OF INCORRECT PROGRAMS. This involves filtering the set of alternative programs using two *elimination functions*: `FilterOutInvalidBlocks()` and `FilterWithIO()` (*See* Lines 17 and 22 of Algorithm 7.1). More on the elimination functions is presented in Chapter 8.

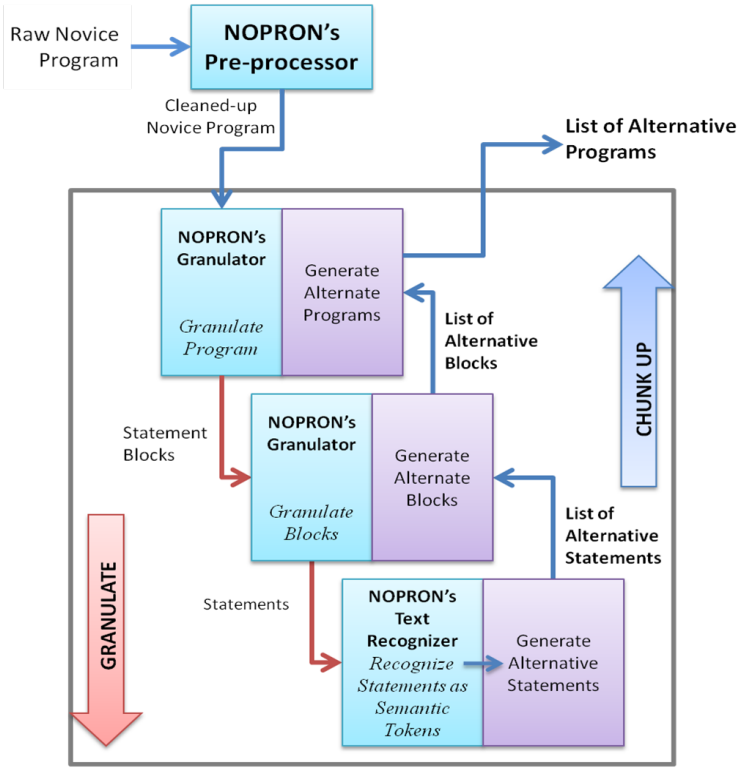


Figure 7.1: Alternative programs enumeration model

The new algorithm (Alternative Programs Enumerator or APE Algorithm) has two phases as shown in Figure 7.1; the first phase pre-processes,

granulates and recognises the semantic tokens in the program text, while the second phase generates alternate statements, which are chunked up into alternative blocks, and finally alternative programs. This process is expressed in Algorithm 7.1.

```
   Data : input_program, a lecturer's program text
   Result : altPrograms, a list of alternative programs
 1 <ProgramObject> aProgram ← preprocess(input_program);
 2 List<ProgramObject> altPrograms ← New List();
 3 List<ChunkObject> chunks ← granulate(aProgram);
 4 foreach chunk in chunks do
 5     List<StmtObject> statements ← granulate(chunk);
 6     List<ChunkObject> altChunks ← New List();
 7     List<ChunkObject> altReorderedChunks ← New List();
 8     foreach statement in statements do
 9         List<StmtObject> alternateStmts ← New List();
10         alternateStmts.add(getAlternativeStatement(statement));
           /* make a new chunk of alternative statements */
11         <ChunkObject> aChunk ← New Chunk();
12         aChunk.linesOfChunk ← altStatements;
13         aChunk.preChunkLines ← prelines;
           /* Comment: retain the misc lines */ altChunks.Add(aChunk);
       end
15     altChunks ← cartesianProduct(altChunks);
       /* altChunks now holds the product of all possibilities */
       /* before adding this chunk, enumerate the valid permutations of lines in
       it */
16     altReorderedChunks ← getPermutation(altChunks);
17      altReorderedChunks ← filterOutInvalidBlocks
       (altReorderedChunks);
18     <ProgramObject> newProgram ← New();
19     newProgram.chunks ← altReorderedChunks;
20     altPrograms.add(newProgram);
   end
   /* generate the product of program functions */
21 altPrograms ← getPermutationOfFunctions(altPrograms);
   /* reduce programs to IO valid ones */
22  altPrograms ← FilterWithIO(altPrograms);
   Result : altPrograms
```

Algorithm 7.1 : Alternative Programs Enumeration (APE)

Example 7.1 (Enumerating a program's space)**.** Consider an arbitrary lecturer's program that sums up the first `n` integers using a function as presented in Listing 7.1. We hereby estimate the space of this program's variations using Algorithm 7.1.

Listing 7.1: Sum of first n integers using a function

```
1  // Sum of first n integers using a function
2  #include<iostream>
3  using namespace std;
4    int n; // this is a global variable
5    void sumUp(){
6      int sum;
7      sum = 0;
8      while (n > 0){
9        sum = sum + n;
10       n = n - 1;
11     }
12     cout<<sum;
13   }
14   int main(){
15     cin>>n;
16     while (n > 0){
17       sumUp();
18     }
19     return 0;
20   }
```

From Listing 7.1,

- Line 1 will be removed by the preprocessing module because it is a comment/remark,
- Lines 2 (input/output library inclusion) and 3 (using standard library) will remain,
- Line 4 is a global variable declaration with only one item in its equivalent set (there is only one way it can be declared),
- Lines 5 to 13 represent a function named `sumUp` and similarly, Lines 14 to 20 make up the `main` function. These two functions can be rearranged in 2! ways (in C++ it does not matter which comes first).
- Lines 6 and 7 can only have one valid permutation,
- the logical expression on Line 8 has four elements in its equivalence set (namely: `n > 0`, `(n-1) >= 0`, `0 < n`, and `0 <= (n-1)` which can be evaluated using the truth table in Definition 6.4,
- the `while`-loop from Lines 8 to 11 has an equivalent set containing 10 elements (Definition 6.7),
- Lines 9 and 10 can be written in four ways each, but cannot be reordered differently. The iterator is identical to L_4 (*See* page 71) to imply that the `statement_block` (Line 9) must come before the increment or decrement (Line 10),

- the `main` method on Line 14 can be declared in two ways (to return `void` by removing line 19 and re-writing Line 14 as `void main()`, or secondly, to leaving it the way it is).
- Line 15 has only one possibility,
- the `while`-loop in Lines 16 through 18 is similar to that in Lines 8 to 11 and as such, the logical expression has four elements in its equivalent set with 10 elements in its loop equivalence set, and finally,
- Line 19 has just one possibility.

Exploring this space, we have:

$\Sigma P_i, i > 0 = 2! \times 1! \times 4 \times 10 \times 1 \times 1! \times 4 \times 4 \times 1 \times 2 \times 1 \times 4 \times 10 = 102400$ possibilities.

As shown later in Section 7.2, this space grows exponentially with respect to the number of lines contained in the program. However, this example can be regarded as a worst case because several novice programs have fewer lines and of course, there will be an exponential decrease as the number of lines reduces.

7.2 IMPLEMENTATION AND RESULTS

We have implemented Algorithm 7.1 in a software application using .Net Framework libraries and reusable classes from NOPRON. In the design, we used entity classes (*see* Figure 7.2) to hold objects such as `program`, `chunk`, `line_of_code`, and `block_of_code`. The application takes a lecturer's program and enumerates the alternative programs that are equivalent to the program. Figure 7.3 shows the application at runtime as it takes a program that solves the *Next Integer Problem* in a text editor to the left, all programs found with the enumeration algorithm in another text editor in the middle and a list of valid equivalent programs (after filtering using I/O analysis and rules of discourse[1]) in a scrollable text editor to the right. The application also shows a brief summary of the total programs found, and the number of correct and wrong ones. The Code Editor interface of the application (with C++ syntax highlighting and formatting) was developed using the Scintilla.NET Application Programming Interface (API) [Scintilla].

1 Validation of generated programs is discussed in Chapter 8

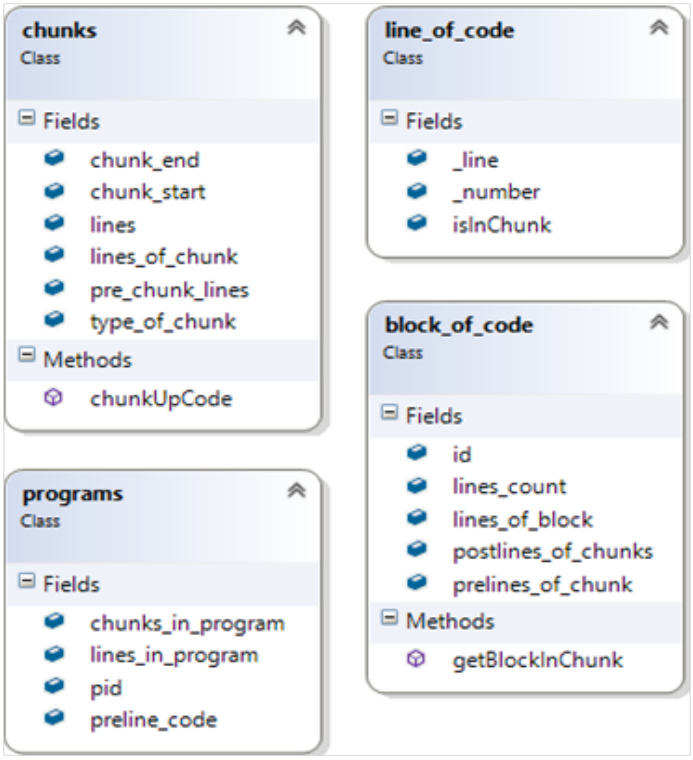


Figure 7.2: Entity classes

7.2.1 *APE Algorithm on the Modified Factorial Problem*

The APE Algorithm found 2400 valid *program strings* to the Modified Factorial Problem. In this section, we display 10 of these solutions from Listing 7.2 to Listing 7.11. Observe that these programs contain distinct types of loop iterator implementations.

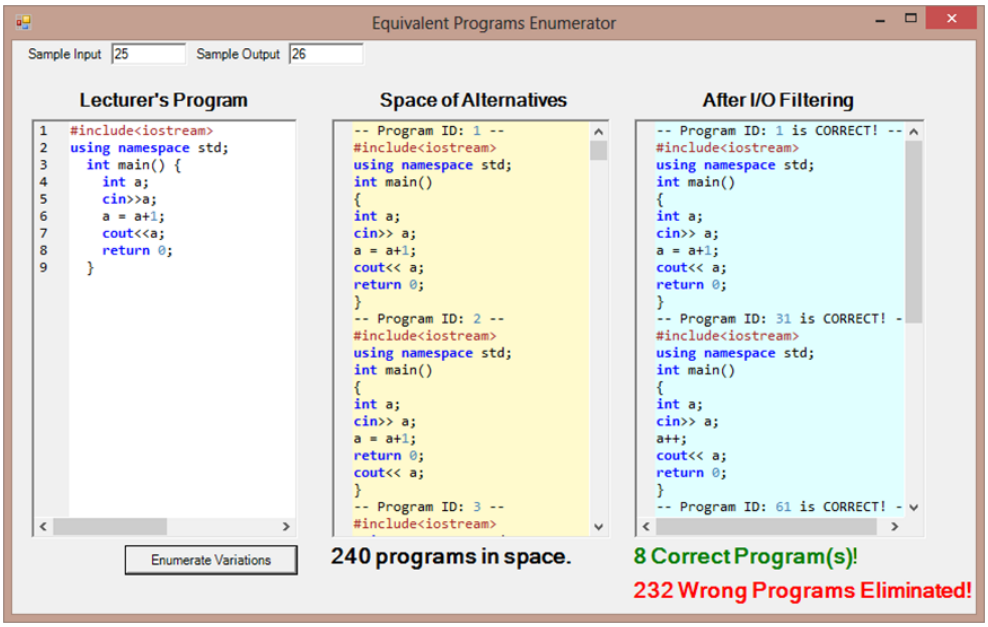


Figure 7.3: Equivalent programs enumerator

Listing 7.2: Generated program P_1

```
#include<iostream>
using namespace std;
int main(){
  int n;
  int i;
  int nfac;
  cin>> n;
  nfac = 1;
  for (i = 1; i <= 5; i++){
    nfac = nfac * i;
  }
  nfac = nfac + n;
  cout<< nfac;
  return 0;
}
```

Listing 7.3: Generated program P_2

```
#include<iostream>
using namespace std;
int main(){
  int n;
  int i;
  int nfac;
  cin>> n;
  nfac = 1;
  for (i = 1; i < 6; i++){
    nfac = nfac * i;
  }
  nfac = nfac + n;
  cout<< nfac;
  return 0;
}
```

Listing 7.4: Generated program P_3

```
#include<iostream>
using namespace std;
int main(){
  int n;
  int i;
  int nfac;
  cin>> n;
  nfac = 1;
  i = 1;
  while (i <= 5){
    nfac = nfac * i;
    i = i + 1;
  }
  nfac = nfac + n;
  cout<< nfac;
  return 0;
}
```

Listing 7.5: Generated program P_4

```
#include<iostream>
using namespace std;
int main(){
  int n;
  int i;
  int nfac;
  cin>> n;
  nfac = 1;
  i = 1;
  while (i < 6){
    nfac = nfac * i;
    i = i + 1;
  }
  nfac = nfac + n;
  cout<< nfac;
  return 0;
}
```

Listing 7.6: Generated program P_5

```
#include<iostream>
using namespace std;
int main(){
  int n;
  int i;
  int nfac;
  cin>> n;
  nfac = 1;
  i = 0;
  while (i < 5){
    i = i + 1;
    nfac = nfac * i;
  }
  nfac = nfac + n;
  cout<< nfac;
  return 0;
}
```

Listing 7.7: Generated program P_6

```
#include<iostream>
using namespace std;
int main(){
  int n;
  int i;
  int nfac;
  cin>> n;
  nfac = 1;
  i = 0;
  while (i <= 4){
    i = i + 1;
    nfac = nfac * i;
  }
  nfac = nfac + n;
  cout<< nfac;
  return 0;
}
```

Listing 7.8: Generated program P_7

```
#include<iostream>
using namespace std;
int main(){
  int n;
  int i;
  int nfac;
  cin>> n;
  nfac = 1;
  i = 1;
  do{
    nfac = nfac * i;
    i = i + 1;
  } while (i <= 5);
  nfac = nfac + n;
  cout<< nfac;
  return 0;
}
```

Listing 7.9: Generated program P_8

```
#include<iostream>
using namespace std;
int main(){
  int n;
  int i;
  int nfac;
  cin>> n;
  nfac = 1;
  i = 1;
  do{
    nfac = nfac * i;
    i = i + 1;
  } while (i < 6);
  nfac = nfac + n;
  cout<< nfac;
  return 0;
}
```

Listing 7.10: Generated program P_9

```
#include<iostream>
using namespace std;
int main(){
  int n;
  int i;
  int nfac;
  cin>> n;
  nfac = 1;
  i = 0;
  do{
    i = i + 1;
    nfac = nfac * i;
  } while (i < 5);
  nfac = nfac + n;
  cout<< nfac;
  return 0;
}
```

Listing 7.11: Generated program P_{10}

```
#include<iostream>
using namespace std;
int main(){
  int n;
  int i;
  int nfac;
  cin>> n;
  nfac = 1;
  i = 0;
  do{
    i = i + 1;
    nfac = nfac * i;
  } while (i <= 4);
  nfac = nfac + n;
  cout<< nfac;
  return 0;
}
```

7.2.2 *More Results*

Furthermore, the enumeration algorithm was tested using other novice programs of considerably small sizes (not more than 20 lines) as shown in Table 7.1. It is important to note that this space only represents the variations of a single program plan, algorithm or method for solving the problem in question. Some novice problems may have more than one way of solving

them. This will result in a larger space of variations.

PROBLEM	LINES IN LECTURER'S PROGRAM	VALID VARIATIONS
Next integer	9	8
Sum of five numbers	12	240
Largest of two numbers	13	96
Next integers of two numbers	13	2240
Sum of n numbers	14	2400
Average of n numbers	16	14400
Recursive sum of n integers	19	102400

Table 7.1: Variations of tested novice programs

Plotting Table 7.1 in a graph as shown in Figure 7.4, we can infer that the growth of the programs in space is arithmetic and reasonably acceptable for programs with lines not more than 16. The space gets exponentially larger as the graph gets steeper towards the end of the program line axis (x-axis) when the number of lines increases in the lecturer's program. This inference however, depends highly on the type of programming problem being solved. The complexity of the enumeration algorithm is presented in detail in Section 7.3 of this chapter.

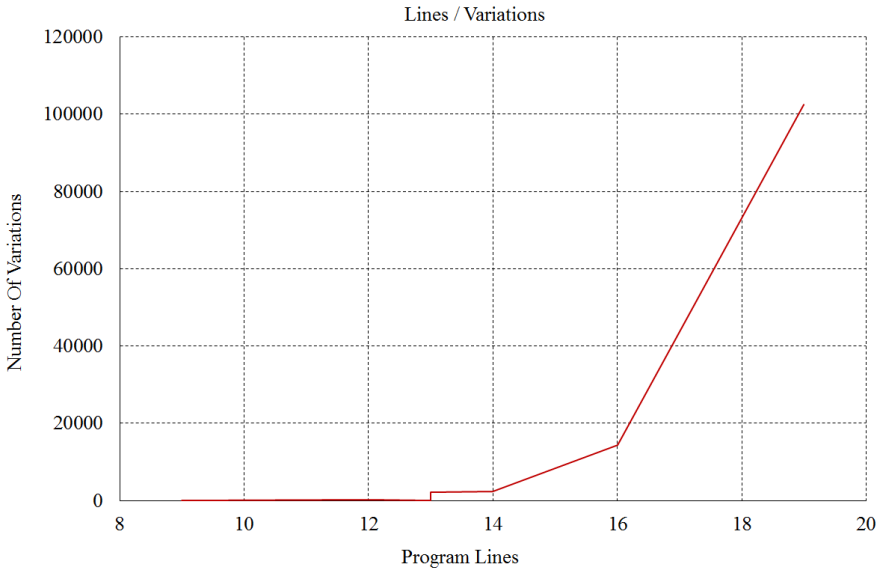


Figure 7.4: Growth of program space

7.3 FORMAL ASPECTS

There are two formal aspects of this idea of counting program variations, resulting in two questions:

1. How long will it take the algorithm to run?
2. Is the space finite?

We present the algorithm complexity analysis and proof of finiteness in this section.

7.3.1 *Complexity Analysis*

An important question we ask whenever we have an algorithm is: *How much time does it take to run as a function of the input size, n?* This is the question of algorithmic complexity [Dasgupta *et al.* 2007, pp. 12]. Time complexity is very crucial in finding and evaluating a good search algorithm [Russell *et al.* 1995]. We hereby provide a theoretical analysis of the time complexity, $T(n)$ of Algorithm 7.1 as a function of the number of computer steps n required.

Given that the algorithm contains two initial loops, we have: one loop (counting number of statements, n_s, per chunk) will run in $T(n_s)$. This loop is nested in the other (counting the number of chunks, n_c, per program) that will run in $T(n_c)$. In the outer loop, the Cartesian product (of the number of chunks) function will take $T({n_c}^2)$. Likewise, the permutation function of statement lines per chunk will take $T(n_c \cdot n_s!)$ for all n_c chunks. Here k is the constant step of k-operations performed to remove invalid enumerations using the `filterOutInValidBlocks()` function within the permutation loops.
This implies a total time complexity of:

$$T(n) = T(n_c \cdot [T(n_s + {n_c}^2)]) + T(n_c \cdot n_s!) + k)$$

Furthermore, the reordering of the number of functions (denoted by n_f) per lecturer's program is carried out in $T(n_f!)$ steps before the algorithm returns the final list of alternative programs. Adding this, we have:

$$T(n) = T(n_c \cdot [T(n_s + {n_c}^2)]) + T(n_c \cdot n_s!) + k + T(n_f!)$$

Simplifying further by taking k as insignificant because its order is in a constant time, we have:

$$T(n) = {n_c}^3 + {n_c}^2 \cdot n_s! + n_c \cdot n_s + n_f! \tag{7.1}$$

Since ${n_c}^3 \leqslant {n_c}^2 \cdot n_s!$, this gives a time complexity of:

$$T(n) = O({n_c}^2 \cdot n_s!).$$

This is in Factorial time which confirms that our algorithm is, in fact, computationally intractable [Van Leeuwen 1994]. We can do better and be more precise by introducing some *real world* constraints from our application domain to reduce the running time of this algorithm. We know for sure that:

- if we consider a smaller sample space of novice programs, then we can limit consecutive statements per chunk to 5, i.e. $2 \leqslant n_s \leqslant 5$, and
- we can also limit our space to programs with at most 3 functions, i.e. $1 \leqslant n_f \leqslant 3$.

Simplifying Equation 7.1, we have the worst-case $(n_f = 3, n_s = 5)$ running time as:

$$\begin{aligned} T(n) &= {n_c}^3 + 5! \cdot {n_c}^2 + 5 \cdot n_c + 3! \\ &= {n_c}^3 + 120{n_c}^2 + 5n_c + 6 \\ &= O({n_c}^3) \end{aligned} \tag{7.2}$$

This is now in cubic time with respect to the number of chunks in a lecturer's program. This can be executed in deterministic polynomial time.

In general, we present the complexity as:

$$T(n) = \begin{cases} O({n_c}^3) & \text{if } n_s \leqslant 5, n_f \leqslant 3 \\ O({n_c}^2 \cdot n_s!) & \text{if } n_s > 5, n_s \geqslant n_f \end{cases} \tag{7.3}$$

This algorithm will not work for large input sizes, i.e. programs with many lines, chunks and functions. This is of little concern to us because our interest is solely in novice programs of typically small sizes which are computationally realisable with the new algorithm.

7.3.2 *Proof of Finiteness of Space*

It is useful to know if the space of novice program variations is finite and hence, if countable. This is for obvious reasons. It will be fruitless to attempt to search an infinite space because the algorithm will never terminate, even for small input sizes, regardless of its complexity. In this section we discuss proofs of finiteness of this space. Since our space is an enumeration of a set of alternative program implementations, permutation of statements and Cartesian product of alternative statements and chunks, it suffices to prove the finiteness of this space by showing that the:

1. set of alternative language constructs, semantic arrangements and programming choices is finite — *for correctly composed programs*,
2. Cartesian product of finite sets is finite, and
3. permutation of finite elements of any given set is finite.

Argument 1 needs no formal proof. We can show that the set of language constructs for C++ is finite by referring to the manufacturer's documentation of the programming language [Smith, ISOCPP, Malik 2010] and the rules of discourse of program composition [Soloway and Ehrlich 1984]. Therefore, if

the number of ways statements can be composed correctly in C++ is finite, then the number of possible valid compositions is. Argument 2 is true and has been proven, i.e. the Cartesian product of two or more finite sets is finite. The proof is given in Schwartz *et al.* [2011, pp. 275], and Pinter [2014, pp. 146]. Argument 3 is also true for any integer n. It suffices to show that the permutation of n elements of a finite set is finite by showing that there are only $n!$ elements in the resulting set.

7.4 APPLICATION OF PROGRAM VARIATIONS

The automatically generated variations can be abstracted to DFAs. The DFA is a new formal abstraction for the program comprehension problem. Given a lecturer's model program, the DFA of alternative solutions constructed from the lecturer's model program, and using decision algorithms, we can determine and/or answer the following questions about semantic errors in novice programs:

1. *Is a student's solution semantically correct with respect to the lecturer's model?* This is a question of *if the student's solution "string" is in the lecturer's "language"*, a membership problem in regular languages and one that can be solved by simulating the DFA on the input string [Martin 2003, Ullman 2014, Hopcroft *et al.* 2006].

2. *Is there a missing code fragment in a student's code?* This is a question of *if substrings of the student's "string" are in the lecturer's "language"*. The strings of the lecturer's language is given by all paths from the start state to some accepting state on the lecturer's DFA.

3. *What is the shortest correct version of the lecturer's program?* This is a question of *the shortest string in the language of the DFA*.

4. *Is the space of the lecturer's alternative solutions finite?* This is a question of whether or not *there are cycles in the DFA*. If there are cycles, the language is said to be infinite — it accepts too many programs to be counted; else, it is finite and contains a countable number of programs.

Answering the above questions will lead to the discovery of semantic bugs (such as plan composition errors, missing code fragments, flipped Boolean, etc.) in novice programs.

7.5 LIMITATIONS

Here we present the limitations of generating program variations.

1. *Granular Variations.* The currently used preprocessing module cleans the output statements and retains only the variable or value that is to be displayed. For example, given the output statement:

`cout<<"sum="<<sum<<endl;` the preprocessor reduces the statement to: `cout<<sum;` thereby removing the prompt and the `endl` command. This works very well for many novice programs that do not require the novice to display literal strings as output. To further expand this in terms of scope, one will need an intelligent string processor that can distinguish between important prompts and unnecessary ones, retaining a prompt such as "process completed" (which is presumably an output expected by the user) and removing a prompt like "sum =".

2. *Existence of other equivalent solutions.* These are students' solutions outside the lecturer's plan, composed with different algorithms. This work has not considered this space. However, it is sufficient to apply this same algorithm to other lecturer's programs and this will increase the this space by variably twice its size.

3. *Picking up students "going round in circles".* There exists a wider range of program variations that we have not considered in this work. We call this *Students going round in circles*. These are scenarios where students do highly unexpected or even unreasonable coding that somewhat appears to be correct. Examples are statements like the ones shown in Table 7.2.

	STATEMENTS	PREFERRED ALTERNATIVES
1	`a = a + 1 + 1;`	`a = a + 2;`
2	`if (a > 4) && (true)`	`if (a > 4)`
3	`if (b < 1) || (false)`	`if (b < 1)`
4	`if (any_condition) || (true)`	`if (true)`
5	`if (any_condition) && (false)`	`if (false)`
6	`if (c != 1) || (c != 1)`	`if (c != 1)`
7	`if (k == n) && (k == n)`	`if (k == n)`
8	`if !(!(a > 6))`	`if (a > 6)`

Table 7.2: Redundant compositions

Items 2 and 3 conform to the identity laws (i.e., $p \wedge T \equiv p$, $p \vee F \equiv p$), items 4 and 5 are the same as the domination laws (i.e. $p \vee T \equiv T$, $p \wedge F \equiv F$), 6 and 7 is the same as idempotent laws ($p \vee p \equiv p$, $p \wedge p \equiv p$) and 8 is the double negation law ($\neg(\neg p) \equiv p$). These logical expressions are algebraically valid but do not add any value programmatically. This work has not considered scenarios where students write lines of code similar to these ones — *this is a large space*. This presents more possibilities and truly confirms how big the space of problem solving in programming is. However, this does not deprive this work of its relevance because the presented formalism and algorithm can be extended to take care of these cases.

4. *Intractability.* The algorithm presented in this work will not work for very large input sizes, at least not on an average-capacity, present-day computer. This is a concern when ambitiously thinking of extending this method to handle large programs for automatic software maintenance or in similar fields where automatic program comprehension is useful. However, as shown in Section 7.2, we can handle as many novice programs as possible in our domain (novice program comprehension) provided the program length is considerably small. This algorithm's inability to handle large inputs also re-establishes the real-world problem faced by humans when attempting to comprehend large programs written by other programmers.

7.6 REFLECTIONS ON ENUMERATION ALGORITHM

In this chapter we have presented an algorithm for the iterative generation of valid alternative programs to a Lecturer's program. We have also implemented this algorithm, tested it with some novice programs and obtained valid alternative programs. The complexity analysis of this algorithm reveals that it can be intractable when dealing with large programs because of some Factorial operations embedded in it — which raises a few questions that we have answered as follows.

1. *Is this algorithm implementable in a software tool?* Yes it is. The prototype presented in this chapter named the *Equivalent Programs Enumerator* (*See* Figure 7.3) is a proof of this. Moreover, the generation of equivalent programs is a once-off operation, i.e. a software tool that advises novice programmers (based on the knowledge of many correct programs to a problem) will only need a static repository of the alternative programs. Figure 7.5 shows a screenshot example of such a repository with generated programs stored in files with `.cpp` extension, and a preview of the selected program is highlighted in red on the right side of the screenshot. A tutoring tool can then be designed to iterate through this repository while programs from it are loaded to the knowledge base of such a tool and used for inferences on bugs. Therefore, the algorithm will only need to run when the lecturer enters his/her program at the initial time.

2. *Can we have a prototype of a tutoring system based on the APE Algorithm?* Yes, and later chapters of this thesis describe such a tool called `Code Adviser`, a prototype that shows this is a realisable technique.

3. *Can this algorithm handle large programs in future?* The advancements in High Performance Computing [Springel 2016] give greater hope to this

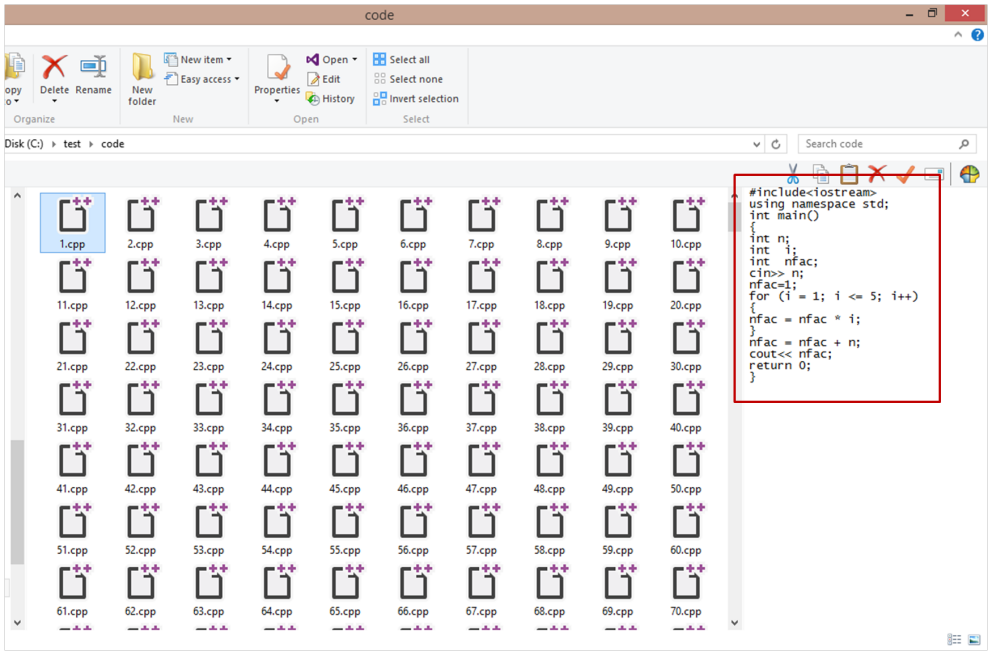


Figure 7.5: A repository of generated programs indexed with serial numbers

algorithm. Using fast computers running GPUs[2], it should be possible to extend the application of this algorithm to larger programs with over fifty to hundreds of lines and possibly millions of valid variations.

As indicated on Lines 17 and 22 of Algorithm 7.1, the APE algorithm uses two `filter` functions to eliminate the incorrect program permutations. In Chapter 8 we discuss these functions.

2 Graphical Processing Units

8 VALIDATING GENERATED PROGRAMS

Generating alternative and equivalent programs using the APE Algorithm has been presented in the last chapter of this thesis. APE computes the permutation of lines of code within chunks as a way to check if there exist other *valid* re-orderings of the Lecturer's program lines. To do this, APE uses two elimination functions to filter the invalid permutations. The first function is `FilterOutInvalidBlocks()`, which takes a list of generated chunks and removes the chunks that do not satisfy given *rules of discourse* (such as *variables should be declared before used*). The second function is `FilterWithIO()`, which takes a list of generated programs and returns a subset that compiles and produces correct outputs. In this chapter, we describe the algorithms used by these functions to validate generated programs.

8.1 VALIDATION WITH RULES OF DISCOURSE

Rules of programming discourse, or simply *rules of discourse* have been studied and applied in different forms over the years [Soloway and Ehrlich 1984, Hansen *et al.* 2012, Panichella *et al.* 2012, Haiduc *et al.* 2012, Morgan *et al.* 2015]. The following are some examples of rules of discourse:

1. `if` statements are for one time executions, and `for` statements are for multiple executions,
2. a variable initialised with an assignment statement must be updated somewhere in the code by an assignment statement,
3. it should be possible for a conditional statement to be true,
4. code fragments that are not in any execution paths should not be included,
5. variable names should communicate their purpose,
6. a variable should be declared before it is used, and
7. the order of a plan's sequencing should be the order of program composition.

Rules 6 and 7 are useful in filtering incorrect program permutations. Rule 7 has been embedded in the *chunking* procedure of NOPRON (*see* Section 5.1.3 of Chapter 5), therefore permutations of lines of codes within different chunks are not generated with the APE Algorithm (Algorithm 7.1).

We have adopted Rule 6 in defining the `FilterOutInvalidChunks()` function described in Algorithm 8.1.

```
   Data : chunks, a list of chunks
   Result : subChunks, a list of valid chunks
1  foreach chunk in chunks do
2      VariableList ← getAllVariablesInChunk(chunk);
3      IsValidChunk ← true;
4      foreach variable in VariableList do
5          if IndexOf(WhereUsed(variable, chunk)) <
           IndexOf(WhereDefined(variable, chunk)) then
               /* set validity to false to remove this chunk */
6              IsValidChunk ← false;
               /* save time, break out of loop because this chunk is already
               invalid */
7              Exit For;
           end
       end
       /* gone through every variable in this chunk */
8      if IsValidChunk is true then
           /* add this to the list of valid chunks */
9          subChunks.add(chunk);
       end
   end
   Result : subChunks
```

Algorithm 8.1 : Filtering invalid chunk permutations

In Algorithm 8.1, a generated list of chunks is checked one chunk at a time to see if Rule 6 is satisfied. The variables of each chunk are first listed using the `getAllVariablesInChunk()` function on Line 2 and the chunk is assumed to be valid until proven otherwise with the loop (iterating through all variables of the chunk) that starts on Line 4. The conditional `if` statement in this loop checks if any variable has been used before being declared, and if yes, it sets this chunk as an invalid one. At the end of the loop, another condition checks if the chunk is still valid, and if yes, it is added to the new list of valid chunks.

Example 8.1 (Filtering Invalid Permutations)**.** Here we give an example of how Algorithm 8.1 works with a chunk from a solution to the modified factorial problem, containing five statements, shown in Listing 8.1. Lines 1, 2 and 3 of Listing 8.1 are declaring variables `n`, `i`, and `nfac`. Lines 4 and 5 are using variable `n` and `nfac` for input and assignment respectively. There are a total of 120 (5!) arrangements of these lines, however, valid permutations should be such that Line 1 always appears before Line 4, and Line 3 always appears before Line 5. Using the filter in Listing 8.1, these permutations are reduced to 30 valid chunks shown from C_1 to C_{30}.

Listing 8.1: A chunk from the modified factorial problem

```
1 int n;
2 int i;
3 int nfac;
4 cin>>n;
5 nfac = 1;
```

C_1

```
1 int n;
2 int i;
3 int nfac;
4 cin>>n;
5 nfac = 1;
```

C_2

```
1 int n;
2 int i;
3 int nfac;
4 nfac = 1;
5 cin>>n;
```

C_3

```
1 int n;
2 int i;
3 cin>>n;
4 int nfac;
5 nfac = 1;
```

C_4

```
1 int n;
2 int nfac;
3 int i;
4 cin>>n;
5 nfac = 1;
```

C_5

```
1 int n;
2 int nfac;
3 int i;
4 nfac = 1;
5 cin>>n;
```

C_6

```
1 int n;
2 int nfac;
3 cin>>n;
4 int i;
5 nfac = 1;
```

C_7

```
1 int n;
2 int nfac;
3 cin>>n;
4 nfac = 1;
5 int i;
```

C_8

```
1 int n;
2 int nfac;
3 nfac = 1;
4 int i;
5 cin>>n;
```

C_9

```
1 int n;
2 int nfac;
3 nfac = 1;
4 cin>>n;
5 int i;
```

C_{10}

```
1 int n;
2 cin>>n;
3 int i;
4 int nfac;
5 nfac = 1;
```

C_{11}

```
1 int n;
2 cin>>n;
3 int nfac;
4 int i;
5 nfac = 1;
```

C_{12}

```
1 int n;
2 cin>>n;
3 int nfac;
4 nfac = 1;
5 int i;
```

C_{13}

```
1 int i;
2 int n;
3 int nfac;
4 cin>>n;
5 nfac = 1;
```

C_{14}

```
1 int i;
2 int n;
3 int nfac;
4 nfac = 1;
5 cin>>n;
```

C_{15}

```
1 int i;
2 int n;
3 cin>>n;
4 int nfac;
5 nfac = 1;
```

C_{16}

```
1 int i;
2 int nfac;
3 int n;
4 cin>>n;
5 nfac = 1;
```

C_{17}

```
1 int i;
2 int nfac;
3 int n;
4 nfac = 1;
5 cin>>n;
```

C_{18}

```
1 int i;
2 int nfac;
3 nfac = 1;
4 int n;
5 cin>>n;
```

C_{19}

```
1 int nfac;
2 int n;
3 int i;
4 cin>>n;
5 nfac = 1;
```

C_{20}

```
1 int nfac;
2 int n;
3 int i;
4 nfac = 1;
5 cin>>n;
```

C_{21}

```
1 int nfac;
2 int n;
3 cin>>n;
4 int i;
5 nfac = 1;
```

C_{22}

```
1 int nfac;
2 int n;
3 cin>>n;
4 nfac = 1;
5 int i;
```

C_{23}

```
1 int nfac;
2 int n;
3 nfac = 1;
4 int i;
5 cin>>n;
```

C_{24}

```
1 int nfac;
2 int n;
3 nfac = 1;
4 cin>>n;
5 int i;
```

C_{25}

```
1 int nfac;
2 int i;
3 int n;
4 cin>>n;
5 nfac = 1;
```

C_{26}

```
1 int nfac;
2 int i;
3 int n;
4 nfac = 1;
5 cin>>n;
```

C_{27}

```
1 int nfac;
2 int i;
3 nfac = 1;
4 int n;
5 cin>>n;
```

C_{28}

```
1 int nfac;
2 nfac = 1;
3 int n;
4 int i;
5 cin>>n;
```

C_{29}

```
1 int nfac;
2 nfac = 1;
3 int n;
4 cin>>n;
5 int i;
```

C_{30}

```
1 int nfac;
2 nfac = 1;
3 int i;
4 int n;
5 cin>>n;
```

Chunks C_1 to C_{30} represent the 30 valid ways novice programmers may arrange or sequence the semantic tokens of their code.

8.2 VALIDATION WITH IO ANALYZER

An IO Analyzer is useful in this research in two ways:

1. to validate the iteratively generated programs,
2. in developing a tutoring tool that detects semantic bugs in novice programs, an IO Analyzer will be useful in checking if the novice's program is, at least, *syntactically* correct, and
3. if the novice's program passes an IO test, then the task is narrowed down to checking if the novice's algorithm is either correct or wrong — a program that passes an IO test may be *crooked*[1].

We will discuss the categories of bugs (with respect to IO Analysis) further in Chapter 9.

8.3 DESIGNING AN IO ANALYZER

In this section, we present a description of the new IO Analyzer that was developed to validate the programs generated with the APE Algorithm. Figure 8.1 shows a block diagram of how the IO Analyzer works. It takes a program and one or more test cases[2] and determines if the program is correct or wrong based on its IO behaviour.

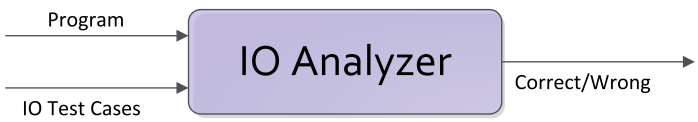


Figure 8.1: Input-Output Analyzer

1 A program designed to pass IO tests, while ignoring the right logic or algorithm. This is common if the test case is known to the programmer.
2 A *test case* is a sample input with a sample output.

There are two ways we can go about the design of a C++ IO Analyser:

1. use a compiler generator, such as COCO/R[3], or ANTLR[4].
2. use the API[5] or DLL[6] of an available compiler, such as Microsoft's Visual C++ Developer Command Prompt for VS2013.

The first approach is *re-inventing the wheel* because we will have to write out the entire grammar of the C++ language in Backus-Naur Form (or BNF) and this is tedious. The second approach allows us to reuse the available Microsoft technology, and this is what we have implemented.

8.4 MICROSOFT VISUAL C++ IO ANALYZER

The Microsoft Visual C++ compiler, if installed on a computer running Windows operating system, should be located in the directory:

```
C:\Program Files (x86)\Microsoft Visual Studio
        12.0\Common7\Tools\Shortcuts
```

This is a batch file that has the shortcut name: `Developer Command Prompt for VS2013` pointing to the file at:

```
C:\Program Files (x86)\Microsoft Visual Studio
       12.0\Common7\Tools\VsDevCmd.bat
```

The content of the `VsDevCmd.bat` is listed in Appendix B. We have created a new batch file that is executed by the new IO Analyzer. The batch file loads this Visual C++ compiler, and runs it against a program file. The program file fetches its input from, and stores its output in the file system. The new IO Analyzer uses Algorithm 8.2 to step-through each program to be tested, passing each test case to it. Similar to the way Algorithm 8.1 works, Algorithm 8.2 iterates through the list of input programs to be tested, and checks if each program satisfies the IO requirement of the given set of test case. If yes, the program instance is added to the new list of valid programs.

The following are the steps carried out by the new IO Analyzer, it:

1. stores the program to be tested in a file,
2. stores the test cases in another file,

3 http://www.ssw.uni-linz.ac.at/Research/Projects/Coco/
4 http://www.antlr.org/
5 Application Programming Interface
6 Dynamic Link Library

3. grants administrative privileges to a system process that will perform the operation, and

4. using Algorithm 8.2; calls the new batch file for the operation on Line 3.

When the batch file is done, an output or IO Error message must have been stored in the output file. The new IO Analyzer proceeds by fetching this output and decides if the program has passed the IO test.

```
   Data : programs, a list of programs;
   testcases, a list of [input, output]
   Result : validPrograms, a list of valid programs
1  foreach program in programs do
       /* first assume that this program is valid */
       isValidProgram ← true;
2      foreach tcase in testcases do
3          if Output(program, tcase.input) != tcase.output then
4              isValidProgram ← false;
               /* save time, break out of loop because this program is already
               invalid, there's no need to go through other test cases */
5              Exit For;
           end
       end
       /* gone through every test case */
6      if IsValidprogram == true then
           /* add this to the list of valid programs */
7          validPrograms.add(program);
       end
   end
   Result : validPrograms
```

Algorithm 8.2 : Filtering invalid programs

8.5 RACE CONDITION WITH IO AND RESOLUTION

During experimentations, there was a concurrency problem with the output file. This is because many programs were being tested iteratively, their output stored in and read from the same system file at run time. The storage process of the Operating System however, did not appear to be as quick as our IO application — thereby leading to *race condition*[7]. To resolve this, we enforced the sequence of operations by making our IO analyzer wait for the

7 problem with operation sequencing.

system process to be completed and exit, before it attempts to use the output file.

8.6 REFLECTIONS ON VALIDATION

Blindly combining the lines of codes is not enough to generate the list of alternative solutions to a given program. In this Chapter, we have presented two filter algorithms for testing the permutations and validating the generated programs. The two algorithms work at different granularity levels; chunk level and program object level. The algorithms described in this Chapter have been used to validate the programs generated with the APE algorithms. In Part iv we discuss how we have abstracted the space of equivalent programs generated to DFAs and devised algorithms to find semantic bugs using those DFAs.

Part IV

COMPREHENSION AND SEMANTIC BUG DETECTION

Automatic detection of semantic bugs in novice programs is instrumental in building electronic programming tutoring systems that can suggest to students how to repair their buggy programs. Given a model solution to a novice programming problem, APE is an algorithm that can be used to iteratively enumerate the alternative programs to the model solution. Ideally, these alternative programs will have varying language constructs, and plan composition steps but equivalent semantics and input–output behaviour. This space of alternative programs is finite and using the class of *regular languages* as a formal abstraction, it can be represented with a DFA.

In this part, we have adopted the APE algorithm to enumerate alternative programs to specific novice programs written in C++. Taking well-granulated program lines as semantic *symbols* or *tokens*, we have also abstracted this space (of alternative programs) to DFAs and referred to this as *Alternative Programs Deterministic Finite Automaton* (or APDFA). Using Finite Automata decision properties (e.g. membership testing), we have presented new algorithms that take: an APDFA and a novice program, and attempts to find the novice program in the APDFA. If found, the novice program has no semantic bugs; but if not found, then we make inferences as to what the bugs are. This technique is implementable in a software tool and such tool can be used to support the teaching of first year programming courses in higher institutions of learning.

This part has two chapters. Chapter 9 presents new algorithms for abstracting program strings to APDFAs, and Chapter 10 presents new algorithms for finding semantic bugs in such APDFAs.

9 AUTOMATA IN PROGRAM COMPREHENSION

Finite Automata (FAs) have been applied in many domains, but not in program comprehension. As far as the author is aware this work is the first of this approach in the attempt to understand novice programs. The FA formalism offers a *new* and relatively *simple to implement* abstraction to the program comprehension problem because many algorithms exist to manipulate and make inferences on finite state machines. If each program generated with the APE algorithm can be abstracted to a string, an APDFA can be constructed from the strings of solutions. In this Chapter, we present algorithms for constructing such APDFAs. The process of constructing an APDFA from a set of equivalent programs $\{P_0, P_1, \ldots P_n\}$ is shown in Figure 9.1.

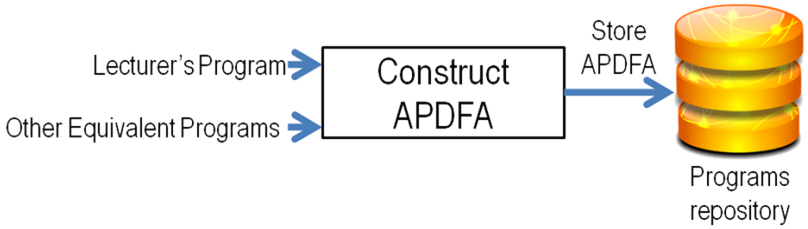


Figure 9.1: Constructing APDFAs from equivalent programs

In Figure 9.1, we take a lecturer's program representing the solution to a well defined problem and a set of equivalent programs generated with APE and we used a new algorithm to construct an APDFA. This APDFA is stored in a repository and used for matching student programs.

9.1 NOVICE SOLUTION SPACE AS A REGULAR LANGUAGE

The APDFAs constructed in this work are done using a finite set of model solutions (generated with the APE algorithm) to novice programming problems, i.e. a finite set of *program strings*. In this section we discuss regularity of the language of APDFAs.

Remark 9.1 (Language of a set of program strings is regular)**.** Let $\mathcal{L}_p$ be the language of a finite set of program strings over some alphabet of semantic tokens Σ_p, then $\mathcal{L}_p$ is regular.

We can ascertain this by showing that there exists a regular expression r_p for the language $\mathcal{L}_p$, hence it is regular.

Let $\mathcal{L}_p = \{w_1, w_2, w_3, \ldots, w_n\}$, where $w_i | 1 \leqslant i \leqslant n, i \in \mathbb{N}$ are the *program strings* contained in $\mathcal{L}_p$. Then one possible expression is:

$$w_1 \cup w_2 \cup w_3 \cup \ldots \cup w_n = \bigcup_{i=1}^{n} w_i \tag{9.1}$$

This is a regular expression, and since there is one for the language, we conclude that the language is regular and hence, an APDFA can be constructed to accept all the strings of the language.

Theorem 9.1 (Every APDFA is Acyclic)**.** *Let M be an APDFA for the language $\mathcal{L}_p$ that consists of a finite set of program strings w_i. Then M is acyclic.*

We will prove Theorem 9.1 by *contradiction*.

Proof. Let us assume that M has at least a cycle. In that case, we could generate the string $w = xyz$, where y is the cycle. And also we can generate the strings $xyyz, xyyyz, xyyyyz, xy^5z, \ldots, xy^nz, n \geqslant 0$. This is the same as xy^*z, which contains the Kleene star $\{*\}$ and implies that we will have infinitely many strings. Hence, the language of M will not be finite. This contradicts our initial assumption, and since we know that the language of M is finite, we conclude that M is acyclic. □

The acyclic nature of APDFAs is important for two major reasons:

1. it proves that infinite program strings do not exist in the Language of solutions to novice programming problems, for any finite alphabet Σ_p of semantic tokens.

2. as the number of equivalent programs (or program strings) grow, the APDFA also gets bigger for any given programming problem. APDFAs are acyclic and hence there are algorithms for minimizing such DFAs in linear time (*see* Watson [2010]) — *this saves the computer memory in large applications*.

Remark 9.2 (Exception for Theorem 9.1)**.** Within the scope of this thesis, Theorem 9.1 is always true. However, there are exceptions where a DFA may have *unreachable* cycles, this is out of the scope of this research and not the case with APDFAs.

9.2 APDFA CONSTRUCTION

9.2.1 *Algorithms for Constructing APDFAs from Programs*

In this section, we present algorithms for the construction of APDFAs from a list of program strings. The idea is to start by constructing a digraph with the first program string. Since a program string is a sequence of program lines (semantic tokens) in a given program, we are looking at two cases (or possibilities) in the construction process:

CASE I. A list of programs with exactly the same number of lines should have the same number of APDFA states, i.e for n program strings, where

$$|w_1| \equiv |w_2| \equiv |w_3| \equiv \ldots \equiv |w_n|$$

The new APDFA will have $n + 1$ states, with semantic tokens as the transitions between the states.

CASE II. If Case I does not hold, i.e. if the list of alternative programs have varying lengths, then we will construct digraphs for every program string with the assumption that all the programs start with the same semantic token.

These two possibilities have led us to the design of two construction algorithms shown in Algorithm 9.1 and Algorithm 9.2.

```
Data : A lecturer's program P_0, list of equivalent programs
       {P_1, P_2, ..., P_n}
Result : An APDFA M accepting only the lecturer's program string
         and equivalent programs
1  initialize M as new DFA object;
2  create new state object and set M's start state attribute to new state;
3  add new state to APDFA states;
4  create next state;
5  create first transition from start state to next state on first semantic
   token in P_0;
6  add transition to M transitions;
7   foreach other semantic token ∈ P_0 do
8  |  create and add new state;
9  |  set state to accepting if it is last semantic token in P_0;
10 |  create and add new transitions to new state on this semantic
   |  token;
   end
11 foreach program P_i ∈ {P_1, P_2, ..., P_n} do
12 |  iterate through the semantic tokens of other programs;
13 |  foreach semantic token ∈ P_i do
14 |  |  if transition on this semantic token does not exist in M then
15 |  |  |  add this transition to M;
   |  |  else
16 |  |  |  do nothing;
   |  |  end
   |  end
   end
17 return M
```

Algorithm 9.1 : Constructing APDFA from programs of the same length

In Algorithm 9.1, a DFA with a single path is first created from the lecturer's program string with the `for`-loop that starts on Line 7 and other programs on the list are later added to the APDFA. Algorithm 9.2 takes a slightly different construction style, knowing that the program paths may be different, this implies different accepting states for every program string.

Data : A lecturer's program P_0, list of equivalent programs $\{P_1, P_2, \ldots, P_n\}$
Result : An APDFA M accepting only the lecturer's program string and equivalent programs

```
1  initialize M as new DFA object;
2  create new state object and set M's start state attribute to new state;
3  if new state is not in APDFA then
       add new state to APDFA states
   end
4  create next state;
5  add transition to M transitions;
6  if next state is not in APDFA then
       add next state to APDFA states
   end
7  add lecturer's program to program list;
8  foreach program P_i ∈ {P_0, P_1, P_2, ..., P_n} do
       foreach semantic token ∈ P_i do
9          if this is the first semantic token in P_i then
10             create first transition from start state to next state on first
               semantic token in P_i;
11             add this transition to M;
           else if this is the last semantic token in P_i then
12             create last transition from start state to next state on last
               semantic token in P_i;
13             set next state to accepting state;
14             add this transition to M;
           else
15             this is neither a start nor an accepting state;
16             create and add new transition to next state on this
               semantic token;
17             add this transition to M;
           end
       end
18     reset next state to a new state;
   end
19 return M
```

Algorithm 9.2 : Constructing APDFA from programs of varying lengths

In Algorithm 9.2, new states are created for every semantic token in each program in the list such that each program string leads to a different accepting state. The cost of this is that the APDFA will have many states, however, using a DFA minimization algorithm, indistinguishable states can be merged [Martin 2003]. We have used Algorithm 9.1 and Algorithm 9.2 to construct APDFAs from list of equivalent programs.

9.3 GENERATING THE ALPHABET OF $\mathcal{L}_p$

In this section we present an algorithm for the automatic generation of the alphabet Σ_p of a given $\mathcal{L}_p$. Σ_p is the union of the semantic tokens across all program strings in $\mathcal{L}_p$.

Data : List of equivalent programs $\{P_0, P_1, P_2, \ldots, P_n\}$
Result : An Alphabet, Σ_p
1 $\Sigma_p \leftarrow$ *New* List() ;
2 **foreach** *Program* $P_i \in \{P_0, P_1, P_2, \ldots, P_n\}$ **do**
3 **foreach** *Semantic token,* $a_i \in P_i$ **do**
4 **if** $a_i \notin \Sigma_p$ **then**
 Σ_p.Add(a_i);
 end
  end
 end
5 **return** Σ_p

Algorithm 9.3 : Generating Σ_p, the alphabet of $\mathcal{L}_p$

We have used Algorithm 9.3 to generate the alphabet of any given $\mathcal{L}_p$ for the programming problems covered in this work. Some of the results are discussed in the next section of this chapter.

9.4 TEST CASES AND RESULTS

Here we present the result of two programming problems and their automatically generated APDFAs using the two different algorithms presented earlier in this chapter.

9.4.1 *Next Integer Problem*

Let us consider the Next Integer Problem. There are at least four valid solutions to this problem. These solutions are shown in Programs P_0, P_1, P_2, and P_3.

P_0

```
1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5    int n;
6    cin>>n;
7    n = n + 1;
8    cout<<n;
9    return 0;
10 }
```

P_1

```
1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5    int n;
6    cin>>n;
7    ++n;
8    cout<<n;
9    return 0;
10 }
```

P_2

```
1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5    int n;
6    cin>>n;
7    n += 1;
8    cout<<n;
9    return 0;
10 }
```

P_3

```
1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5    int n;
6    cin>>n;
7    n++;
8    cout<<n;
9    return 0;
10 }
```

Observe that the difference between these programs is in Line 7, where a novice programmer can make four distinct choices of language construct in an attempt to increment the counter n. Applying Algorithm 9.1 on this lists of programs, we get the APDFA in Figure 9.2.

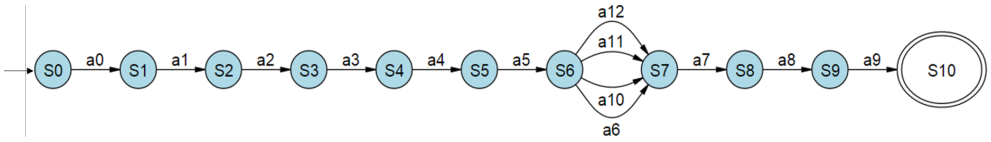


Figure 9.2: APDFA for the Next Integer Problem

The alphabet Σ_{P_1} of this APDFA, generated with Algorithm 9.3, is shown in Table 9.1.

State S_{10} is the accepting state of this APDFA, and it accepts only four program strings. Using Σ_{P_1}, the four strings accepted by this APDFA are:

1. $P_1 = [a_0 \cdot a_1 \cdot a_2 \cdot \ldots \cdot a_9]$,
2. $P_2 = [a_0 \cdot a_1 \cdot a_2 \cdot \ldots \cdot a_5] \cdot [a_{10}] \cdot [a_7 \cdot \ldots \cdot a_9]$,
3. $P_3 = [a_0 \cdot a_1 \cdot a_2 \cdot \ldots \cdot a_5] \cdot [a_{11}] \cdot [a_7 \cdot \ldots \cdot a_9]$, and

STATE MARKERS	SEMANTIC TOKEN
a_0	`#include <iostream>`
a_1	`using namespace std;`
a_2	`int main()`
a_3	`{`
a_4	`int n;`
a_5	`cin>> n;`
a_6	`n = n + 1;`
a_7	`cout<<n;`
a_8	`return 0;`
a_9	`}`
a_{10}	`n++;`
a_{11}	`n += 1;`
a_{12}	`++n;`

Table 9.1: Σ_{P_1} of the Next Integer Problem's APDFA.

4. $P_4 = [a_0 \cdot a_1 \cdot a_2 \cdot \ldots \cdot a_5] \cdot [a_{12}] \cdot [a_7 \cdot \ldots \cdot a_9]$.

We can write a regular expression to accept these four strings as follows:

$$r_p = P_1|P_2|P_3|P_4.$$

9.4.2 *Modified Factorial Problem*

We have presented this problem earlier in this thesis. Unlike the Next Integer Problem, there are many distinct solutions to this problem, and also, these solutions have varying program lengths — so we apply Algorithm 9.2 in constructing the problem's APDFA. For presentation purpose[1], we have used a list of 10 equivalent programs in constructing the APDFA[2]. The APDFA for this problem is shown in Figure 9.3 with its alphabet shown in Table 9.2.

In this case, there are 10 accepting states, each for a program string (see Figure 9.3). The program strings accepted by this APDFA are:

1. $P_1 = [a_0 \cdot a_1 \cdot a_2 \cdot a_3 \cdot a_4 \cdot a_5 \cdot a_6 \cdot a_7 \cdot a_8 \cdot a_9 \cdot a_3 \cdot a_{10} \cdot a_{11} \cdot a_{12} \cdot a_{13} \cdot a_{14} \cdot a_{11}]$,
2. $P_2 = [a_0 \cdot a_1 \cdot a_2 \cdot a_3 \cdot a_4 \cdot a_5 \cdot a_6 \cdot a_7 \cdot a_8 \cdot a_{15} \cdot a_3 \cdot a_{10} \cdot a_{11} \cdot a_{12} \cdot a_{13} \cdot a_{14} \cdot a_{11}]$,

1 There are more programs in this space, and a larger APDFA (for more program strings) will be too big to display on this page.
2 *See* Listing 7.2 to Listing 7.11 in Chapter 7 for the listing of these programs

STATE MARKERS	SEMANTIC TOKEN
a_0	`#include <iostream>`
a_1	`using namespace std;`
a_2	`int main()`
a_3	`{`
a_4	`int n;`
a_5	`int i;`
a_6	`int nfac;`
a_7	`cin>> n;`
a_8	`nfac = 1;`
a_9	`for (i = 1; i <- 5; i++)`
a_{10}	`nfac = nfac * i;`
a_{11}	`}`
a_{12}	`nfac = nfac + n;`
a_{13}	`cout<<nfac;`
a_{14}	`return 0;`
a_{15}	`for (i = 1; i < 6; i++)`
a_{16}	`i = 1;`
a_{17}	`while (i <= 5)`
a_{18}	`i = i + 1;`
a_{19}	`while (i < 6)`
a_{20}	`i = 0;`
a_{21}	`while (i < 5)`
a_{22}	`while (i <= 4)`
a_{23}	`do`
a_{24}	`while (i <= 5);`
a_{25}	`while (i < 6);`
a_{26}	`while (i < 5);`
a_{27}	`while (i <= 4);`

Table 9.2: Σ_{P_2} of the Modified Factorial Problem's APDFA.

3. $P_3 = [a_0 \cdot a_1 \cdot a_2 \cdot a_3 \cdot a_4 \cdot a_5 \cdot a_6 \cdot a_7 \cdot a_8 \cdot a_{16} \cdot a_{17} \cdot a_3 \cdot a_{10} \cdot a_{18} \cdot a_{11} \cdot a_{12} \cdot a_{13} \cdot a_{14} \cdot a_{11}]$,
4. $P_4 = [a_0 \cdot a_1 \cdot a_2 \cdot a_3 \cdot a_4 \cdot a_5 \cdot a_6 \cdot a_7 \cdot a_8 \cdot a_{16} \cdot a_{19} \cdot a_3 \cdot a_{10} \cdot a_{18} \cdot a_{11} \cdot a_{12} \cdot a_{13} \cdot a_{14} \cdot a_{11}]$,
5. $P_5 = [a_0 \cdot a_1 \cdot a_2 \cdot a_3 \cdot a_4 \cdot a_5 \cdot a_6 \cdot a_7 \cdot a_8 \cdot a_{20} \cdot a_{21} \cdot a_3 \cdot a_{18} \cdot a_{10} \cdot a_{11} \cdot a_{12} \cdot a_{13} \cdot a_{14} \cdot a_{11}]$,
6. $P_6 = [a_0 \cdot a_1 \cdot a_2 \cdot a_3 \cdot a_4 \cdot a_5 \cdot a_6 \cdot a_7 \cdot a_8 \cdot a_{20} \cdot a_{22} \cdot a_3 \cdot a_{18} \cdot a_{10} \cdot a_{11} \cdot a_{12} \cdot a_{13} \cdot a_{14} \cdot a_{11}]$,
7. $P_7 = [a_0 \cdot a_1 \cdot a_2 \cdot a_3 \cdot a_4 \cdot a_5 \cdot a_6 \cdot a_7 \cdot a_8 \cdot a_{16} \cdot a_{23} \cdot a_3 \cdot a_{10} \cdot a_{18} \cdot a_{11} \cdot a_{24} \cdot a_{12} \cdot a_{13} \cdot a_{14} \cdot a_{11}]$,
8. $P_8 = [a_0 \cdot a_1 \cdot a_2 \cdot a_3 \cdot a_4 \cdot a_5 \cdot a_6 \cdot a_7 \cdot a_8 \cdot a_{16} \cdot a_{23} \cdot a_3 \cdot a_{10} \cdot a_{18} \cdot a_{11} \cdot a_{25} \cdot a_{12} \cdot a_{13} \cdot a_{14} \cdot a_{11}]$,

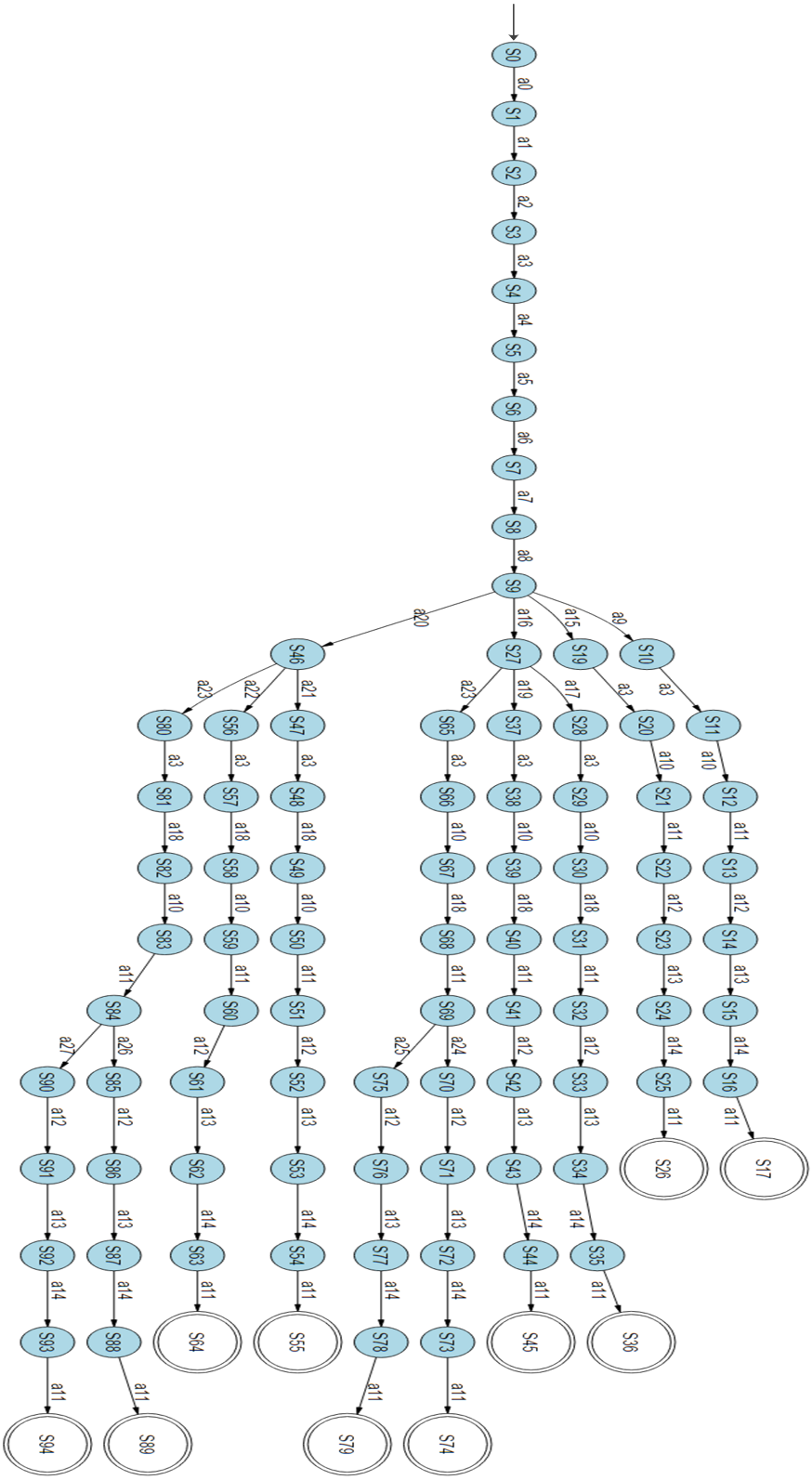


Figure 9.3: APDFA for 10 program strings of the Modified Factorial Problem

9. $P_9 = [a_0 \cdot a_1 \cdot a_2 \cdot a_3 \cdot a_4 \cdot a_5 \cdot a_6 \cdot a_7 \cdot a_8 \cdot a_{20} \cdot a_{23} \cdot a_3 \cdot a_{18} \cdot a_{10} \cdot a_{11} \cdot a_{26} \cdot a_{12} \cdot a_{13} \cdot a_{14} \cdot a_{11}]$, and
10. $P_{10} = [a_0 \cdot a_1 \cdot a_2 \cdot a_3 \cdot a_4 \cdot a_5 \cdot a_6 \cdot a_7 \cdot a_8 \cdot a_{20} \cdot a_{23} \cdot a_3 \cdot a_{18} \cdot a_{10} \cdot a_{11} \cdot a_{27} \cdot a_{12} \cdot a_{13} \cdot a_{14} \cdot a_{11}]$.

Similarly, the regular expression that accepts all 10 strings is:

$$r_p = P_1|P_2|P_3|P_4|P_5|P_6|P_7|P_8|P_9|P_{10}$$

This representation, if implemented in a tool, can be updated with more program strings and can serve as a knowledge base of solutions for making inference on semantic bugs in novice programs.

9.5 REFLECTIONS ON APDFA CONSTRUCTION

In this chapter, we have presented a new type of abstraction for the program comprehension problem — APDFAs, program strings, and alphabets of semantic tokens. For the task of comprehending novice programs, a repository of APDFAs can serve as a *knowledge base*, thereby reducing the *comprehension problem* to the *string matching problem*. However, as demonstrated with the modified factorial problem, 10 solutions produced an APDFA of 95 states. This means that APDFAs can grow to be very large, depending on the problem and also the choices available to the novice programmer. If we consider the entire space of 2400 solutions to the same problem[3], we will have (a rough estimate based on 10 to 95 benchmark) 22800 states. Storing APDFAs in the computer memory then becomes a concern for large applications. Hence, there are two interesting solutions:

1. one may choose to minimize the APDFAs before storing them using quick algorithms such as the Hopcroft's Minimization algorithm [D'Antoni and Veanes 2014], or the MADFA[4] Construction Algorithm [Watson 2010].
2. one may store the strings — or the regular expression that represents the strings — instead of the graph structure, thereby gaining some compression ratio. The graph structure contains redundant information about states and transitions.

In the next Chapter, we discuss new algorithms for detecting semantic bugs in novice programs using APDFAs.

3 *See* Section 7.2.1 of Chapter 7 where we discussed this enumeration.
4 Minimal Acyclic Deterministic Finite Automata

10 SEMANTIC BUG DETECTION

Semantic bug detection in programs is a task that is traditionally carried out by human expert programmers. They often step through source code and attempt to match the lines to a *plan* — a mental model of what the solution should be. If mis-matches are detected, attempts are made to repair the bugs. Our desire is to carry out this task automatically, in order to aid novice programmers in debugging when human experts are not available to help. So far in this thesis, we have presented new algorithms for searching the space of plan variations, and we have abstracted this space, per problem, to an APDFA — a special type of DFA. In this Chapter, we present algorithms for finding semantic bugs in these APDFAs.

The process of finding semantic bugs in APDFAs is described with a block diagram in Figure 10.1.

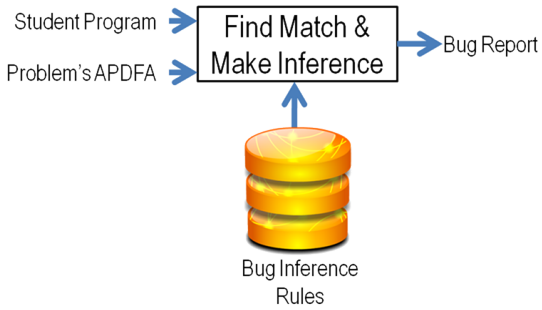


Figure 10.1: Finding bugs in novice programs

In Figure 10.1, we pass a student's program to a problem and the problem's APDFA to an algorithm that attempts to find the student's program in the problem's APDFA. If a full path exists from the start state of the APDFA to some accepting state, we conclude that the student's program is semantically correct. Else, we apply a set of rules to find bugs such as missing code fragment and plan composition errors.

10.1 BUG CATEGORISATION

Given a novice program, it is helpful to determine if it is buggy or not, and if buggy, is it semantically buggy or still has some syntax errors? We therefore present a decision tree for categorising novice programs based on plan and output correctness.

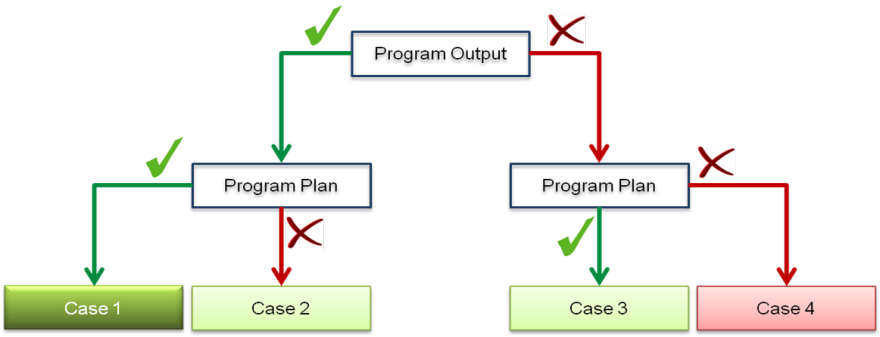


Figure 10.2: Bug Categorisation

In Figure 10.2, we present four possible scenarios. Given a novice program, we conclude that:

CASE I: BUG-FREE. It is bug-free if it passes the IO test and its program string is accepted by the problem's APDFA.

CASE II: RIGHT OUTPUT, WRONG PLAN. This plan is unknown to the system, if the program string is not accepted by the problem's APDFA but somehow the program passes the IO test. This can happen for two reasons:

1. the APDFA's language does not cover every solution, and therefore, other valid program strings exist that represent equivalent programs — using different algorithms to solve the same problem. For example, let us consider the factorial problem with two different algorithms; one having a loop that goes `n` times, and the other implemented with a recursive function. An APDFA may represent the variations of the first algorithm and the novice may have submitted a solution based on the second algorithm. Therefore, a software tool that uses this decision tree should not outrightly disqualify student solutions if they pass the IO test. Rather, it should tell the student that the algorithm is unknown. This solution can then be suggested as a new plan to the human expert. If approved by the expert, the tool can learn by appending the student's program plan to the repository of plans for the problem — hence, a larger APDFA.
2. the student may have written a *crooked* program.

CASE III: WRONG OUTPUT, RIGHT PLAN. An impossible scenario, given the approach employed in this work. Our approach is centered around plan verification — program strings are instances (or implementations) of a plan. Therefore, if a plan is verified as correct, it can be accepted by a problem's APDFA and of course, its output must be correct. This is because the language of every APDFA is a set of valid program strings.

CASE IV: WRONG OUTPUT, WRONG PLAN. Semantic bug is apparent. This category is the main focus of this work. In the next section of this Chapter, we will present algorithms for detecting bugs in this category, such as: plan composition, missing code fragments or wrong problem interpretation, etc.

10.2 MATCHING PROGRAM STRINGS

The algorithms for bug detection presented later in this chapter will need to compare two programs (the novice's and a model program) written with different identifiers, literals, etc. To do this, the novice's program and the set of lecturer's programs (generated with the APE algorithm) are first converted to program strings. The novice's program string is then compared to the language of lecturer's strings to deduce what type of bugs exist in the novice's program — if any. However, both string classes are not defined over the same *alphabet* — the implementation details of the novice program are *local* and often not the same as the model solutions. Therefore, the APDFA's alphabet will be different from the novice's alphabet. In this section we present an algorithm for generating a different alphabet for the novice's program.

10.2.1 *Generating the Alphabet for a Novice Program*

Here we present an algorithm for generating the alphabet Σ_{P_s} for a novice's program P_s in Algorithm 10.1. This is the union of the semantic tokens in P_s. The algorithm steps through the well-granulated novice's program and makes each unique semantic token a symbol of the program alphabet.

```
   Data : A Novice Program P_s
   Result : An Alphabet, Σ_{P_s}
1  Σ_{P_s} ← New List() ;
2  foreach Semantic token, b_i ∈ P_s do
3  |  if b_i ∉ Σ_{P_s} then
   |  |  Σ_{P_s}.Add(b_i);
   |  end
   end
4  return Σ_{P_s}
```

Algorithm 10.1 : Generating the alphabet of a novice's program

Remark 10.1 (Notation for Distinguishing Alphabets)**.** For the purpose of bug detection, it is important to know what alphabet a program string is composed of. For this reason, we have used different symbols for the lecturer's string and novice's string. The symbols of the lecturer's alphabet are denoted with a_i, while those of the novice are denoted with b_i. Figure 10.3 shows a screenshot of an example of these alphabets during testing.

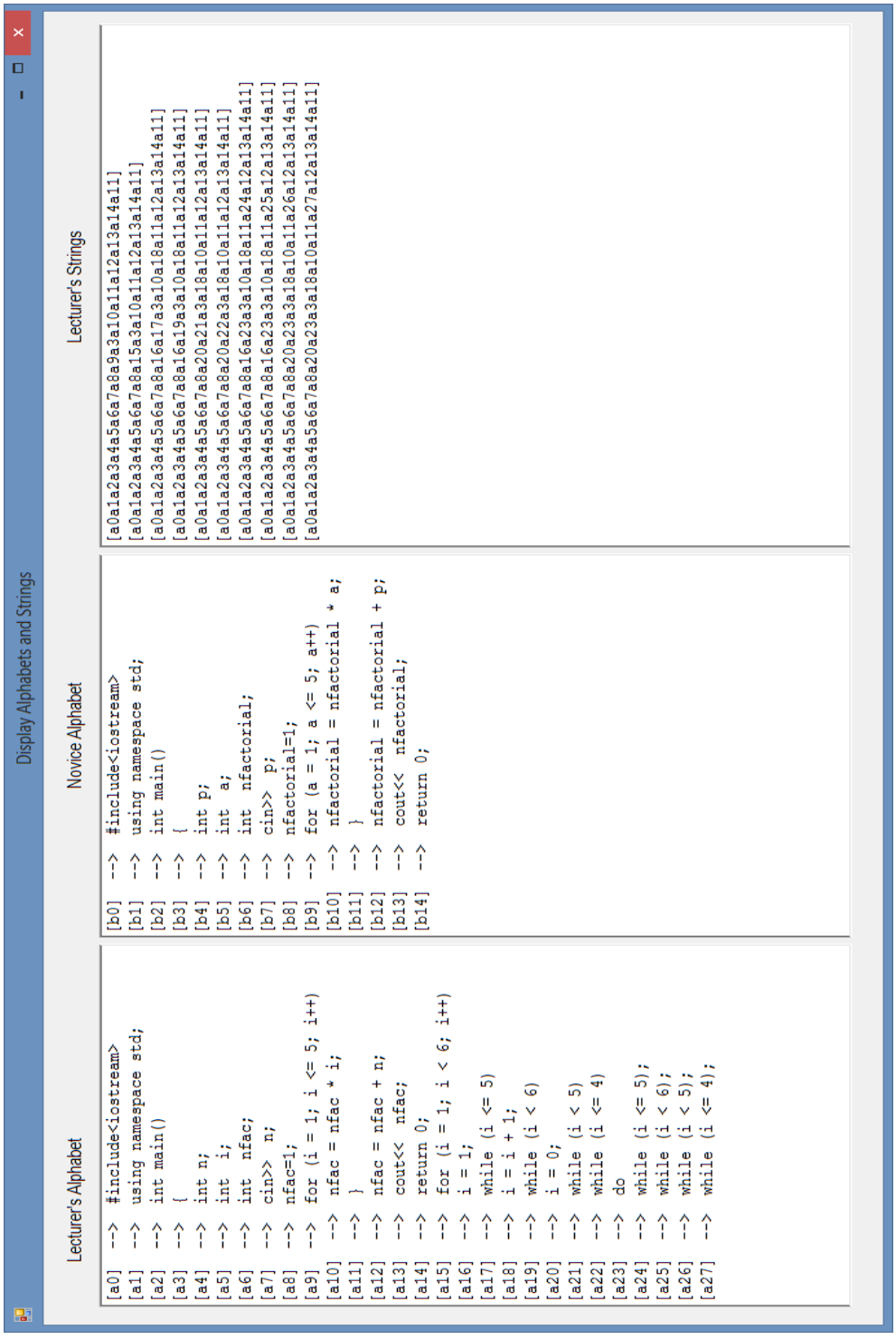


Figure 10.3: APDFA's Alphabet vs. Novice's Alphabet

10.3 MATCHING PLANS OVER DIFFERENT ALPHABETS

Given two program strings $a_0a_1a_2\ldots a_n$ and $b_0b_1b_2\ldots b_n$ where a_i's are symbols from the lecturer's alphabet and b_i's are symbols from the novice's

alphabet, when is $a_i = b_i$? This is the problem of alphabet mapping. Here we present an algorithm for mapping alphabets. First, we introduce the notion of *plan similarity*.

Definition 10.1 (Plan Similarity)**.** Two distinct plans a_i and b_i are similar if:

1. both plans contain the exact same sequence of characters, or
2. the following three conditions are true:
   a) the number of tokens in a_i is the same as in b_i, and
   b) for $j \in [n], p_j = q_j$, where each p_j and q_j are lexemes in a_i and b_i such that $a_i = p_1 p_2 \dots p_n$ and $b_i = q_1 q_2 \dots q_n$. That is, every token in all positions in each plan are of the same literal category, e.g. both are identifiers, same keywords, or numeric values.
   c) if a numeric literal appears in the same position in both plans, the floating value of both literals must be the same, i.e. $9 = 9.0$, but $9.1 \neq 9.0$.

Conditions 2a, 2b and 2c must be satisfied for plans to be similar, except when Condition 1 is true.

Example 10.1 (Similar plans)**.** The following are examples of similar plans:

1. `a = a + 1;` *is similar to* `k = k + 1.0;`
2. `c = a + b;` *is similar to* `z = x + y;`
3. `cout<<a;` *is similar to* `cout<<b;` and
4. `nfac = 1;` *is similar to* `nfac = 1;`.

Example 10.2 (Dissimilar plans)**.** Instances of dissimilar plans are:

1. `cin>> p;` *is not similar to* `cout<<p;`
2. `x = x + 1;` *is not similar to* `x += 1;`
3. `cout<<a+b;` *is not similar to* `cout<<b+1.0;` and
4. `#include <iostream>` *is not similar to* `#include <string>`.

We proceed to define an algorithm for mapping two plans (i.e. symbols from two alphabets). In Algorithm 10.2, we take a novice's plan[1] and a lecturer's plan, and we decide if they are the same. The algorithm first checks if they

1 same as a semantic token

are similar and then checks if all the literals in both plans have the same appearance in the novice's and lecturer's programs.

Data : Lecturer's plan in position i, a_i
Data : Novice's plan in position i, b_i
Data : Lecturer's program string, P_s
Data : Novice's program string, N_s
Result : Decision true, if $a_i = b_i$; false if otherwise.

```
1 if IsSimilar(a_i, b_i) then
    /* The plans are similar, now check all occurrences */
    b_tokens ← getLiteralsInPlan(b_i);
    a_tokens ← getLiteralsInPlan(a_i);
2   foreach pair of token (a_t, b_t) ∈ {a_tokens, b_tokens} do
3     if Appearances(a_t ∈ P_s) = Appearances(b_t ∈ N_s) then
        Decision ← true;
      else
        Decision ← false;
        Exit For;
      end
    end
  else
    Decision ← false;
  end
4 return Decision
```

Algorithm 10.2 : Mapping plans from two alphabets

The mapping algorithm presented here enables us to compare the novice's and lecturer's program strings.

10.4 ALGORITHMS FOR BUG DETECTION

In this section we present algorithms for finding semantic bugs in novice programs. Here, there are two broad novice input possibilities: bug-free and buggy novice programs.

10.4.1 *Bug-Free Novice Programs*

It is important to detect if a novice program is bug-free, or semantically correct with respect to an APDFA's language. This is a *membership problem*[2].

2 Membership problem: given a string and any formal representation of a language, is the string in the language?

Here we present an algorithm for this.

Data : A problem's APDFA M, Student's program P_s
Result : Decision D: *Correct* if P_s is accepted by M, *Buggy* if otherwise
1 get final states of M: F ← `getFinalStates(M);`
2 get start state of M: S_0 ← `getStartState(M);`
3 **if** $\delta(S_0, P_s) \in F$ **then**
  D ← *Correct*;
else
  D ← *Buggy*;
end
4 **return** D

Algorithm 10.3 : Test of membership for bug-free programs

Algorithm 10.3 checks if there is a transition from the start state (S_0) of an APDFA to some accepting state (F) on the sequence of semantic tokens of a student's program strings (P_s). If this is true, it concludes that the program is bug-free, or buggy if otherwise.

10.4.2 *Buggy Novice Programs*

Here we present algorithms for detecting semantic bugs such as: missing code fragment and plan composition errors.

Data : A problem's APDFA M, Student's program P_s
Result : Decision D: *'Has Missing Fragments'* if P_s has missing code fragments, *'No Missing Fragments'* if otherwise
/ get the shortest program string in the language of* M **/*;
P_{min} ← `getShortestProgram(M);`
1 **if** $|P_s| < |P_{min}|$ **then**
  D ← *'Has Missing Fragments'*;
end
2 **return** D

Algorithm 10.4 : Detecting missing code fragments

Algorithm 10.4 checks if a student's program is shorter in length than the shortest possible solution in the APDFA; if *yes*, it implies that, at least a piece of P_s is definitely missing. However, a novice program may not satisfy the condition in Line 1 of Algorithm 10.4 and still have a missing fragment — if it contains other incorrect fragments that *balance* its number of lines. We also need to know what the missing fragment is, if there is any. Algorithm 10.5 finds the missing code fragment by checking if the novice's program string contains any two substrings w_1 and w_2 such that for some nonempty string z, $w_1 z w_2$ is in the language of the problem's APDFA. If this is true, then

we infer that z is missing in the novice's program. The expressions zw_1w_2 (missing fragment at the start of a novice program leads to syntax errors) and w_1w_2z (missing fragment at the end of a novice program leads to syntax errors) are not considered here because we assume they cannot lead to semantic errors — such programs will not pass the compilation stage. An example is when the `include` statement is omitted at the beginning or the last brace (or end brace '}') is omitted at the end of a program. In both cases, the Parser[3] will report syntax errors — *this work focuses only on semantic errors*.

Data : Language of an APDFA $\mathcal{L}_M$, student's program P_s
Result : Missing fragment z
1 **if** $P_s = w_1w_2$ and $w_1zw_2 \in \mathcal{L}_M$, for some $|z| \neq 0$ **then**
  'z *is missing*';
end
2 **return** z

Algorithm 10.5 : Finding missing fragment

We proceed to provide an algorithm for detecting plan composition errors in Algorithm 10.6. A mis-composed plan is detected by computing the difference between the right fragment and the wrong one. The algorithm checks if a novice's program P_s is identical to any program string in the language of solutions $\mathcal{L}_M$ with just one mis-matching substring. If yes, we conclude that this section of the code is mis-composed.

Data : Language of an APDFA $\mathcal{L}_M$, Student's program P_s
Result : Mis-composed fragment x
1 **if** $P_s = w_1xw_2$ and $w_1zw_2 \in \mathcal{L}_M$, for some $|x|, |z| \neq 0, x \neq z$ **then**
  'z *is mis-composed as* x';
end
2 **return** x

Algorithm 10.6 : Detecting plan composition errors

Algorithm 10.6 can be used to detect flipped boolean, variable scoping, type casting, wrongly ordered plans, and operator precedence errors. All these errors are related to plan composition.

10.5 INTRODUCING SEBUD: THE SEMANTIC BUG DETECTOR

In this section we describe a tool that attempts to find semantic bugs in novice programs using the algorithms described earlier in the chapter. For ease of reference, we will refer to this tool as SEBUD, an acronym for SE-

3 Syntax Analyzer

mantic BUg Detector. SEBUD takes a novice program and the problem description, and determines if it is buggy or not. It also attempts to categorise buggy programs and offer information about where the bug is. To do this, SEBUD has to:

1. preprocess and granulate the novice program using pre-processing and granulation modules from NOPRON (*see* Section 5.1.1 and Section 5.1.2 of Chapter 5),
2. extract the plan in the novice's program by generating an alphabet (using Algorithm 9.3 described in Chapter 9) for the novice's program and matching the alphabet to the alphabet of the lecturer's solutions, and
3. use the decision tree in Figure 10.2 to determine the category of the novice's program with respect to its IO and plan correctness.

Hence, SEBUD could tell if a novice's program is:

1. syntactically buggy — here, SEBUD does not proceed further because this is the *compiler's territory*,
2. syntactically correct, but fails IO tests — in this case, SEBUD suspects semantic errors and tries to match the novice's string to the lecturer's language of program strings,
3. syntactically correct, passes IO tests, but its string is not in the lecturer's language — SEBUD concludes that the program is either crooked, or its algorithm is unknown.

Figure 10.4 shows SEBUD at runtime. At startup, SEBUD loads the DFA and program string objects (the language of solutions) for the programming problems within its scope. A drop down menu is provided to specify the programming problem that is solved by the novice's program and this is used to point to the language to be used in the matching process.

Figure 10.5 shows a test result from trying SEBUD on the modified factorial problem. The conclusion of SEBUD for a correct solution is shown along with the novice and lecturer's program strings that were generated and compared.

Due to SEBUD's ability to map two different alphabets (novice's and the lecturer's alphabet), it could determine that the novice's string is the same as the lecturer's string even when the identifiers, and program text are different — which, of course, is often the case. The algorithm used by SEBUD to map alphabets is discussed later in Chapter 12. Figure 10.6 and Figure 10.7 show SEBUD detecting syntax and semantics errors respectively.

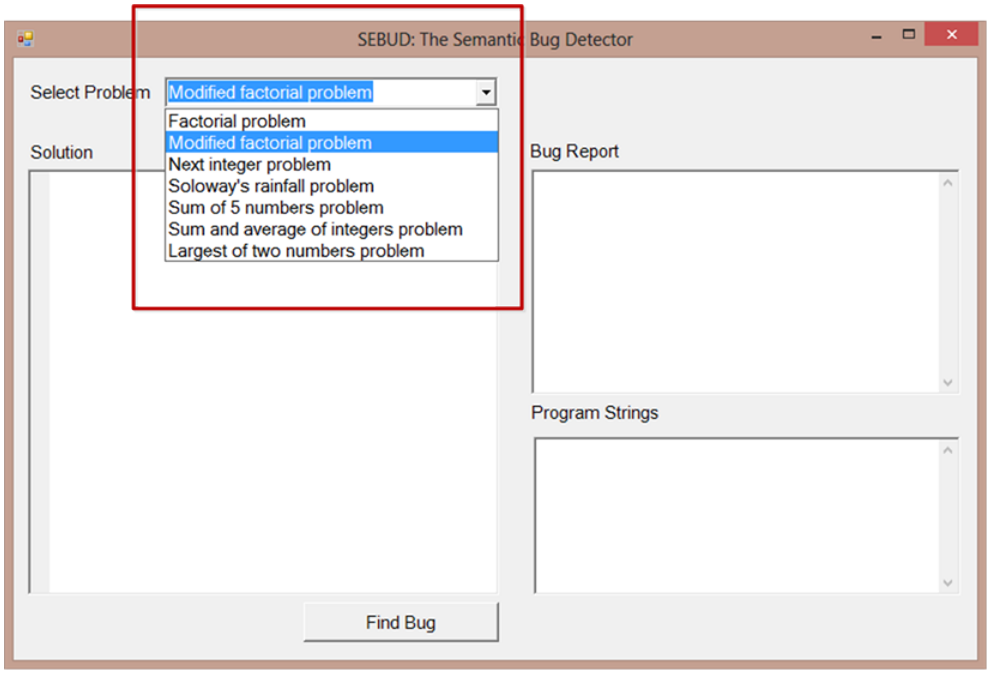


Figure 10.4: Selecting a problem with SEBUD

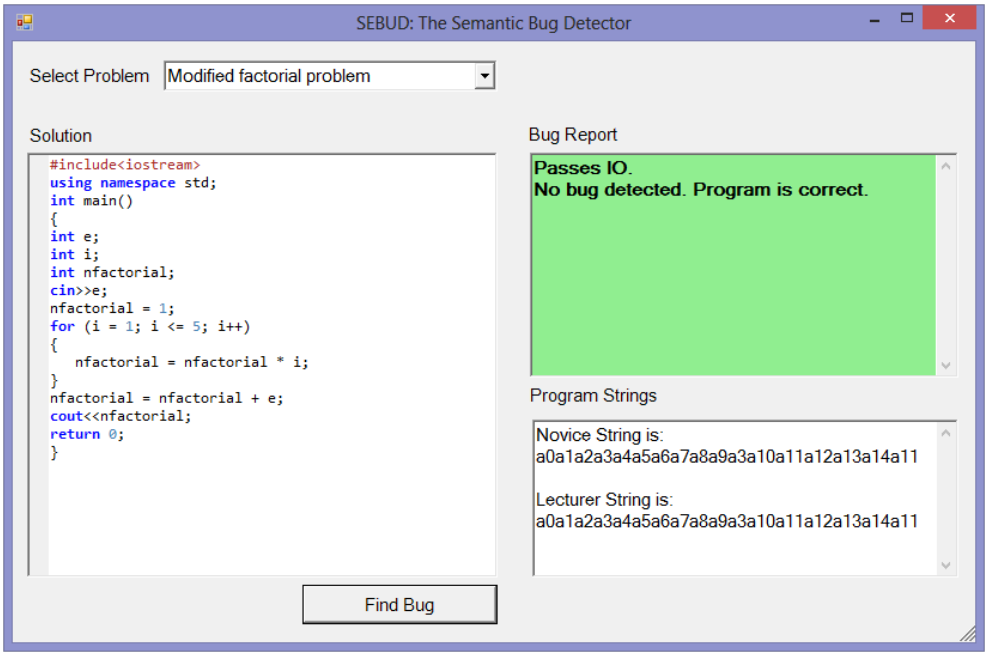


Figure 10.5: Program verification with SEBUD

10.6 MORE RESULTS

Here we present the program strings that were manipulated by SEBUD at runtime and how it was able to decide and find missing code fragments, and mis-composed plans.

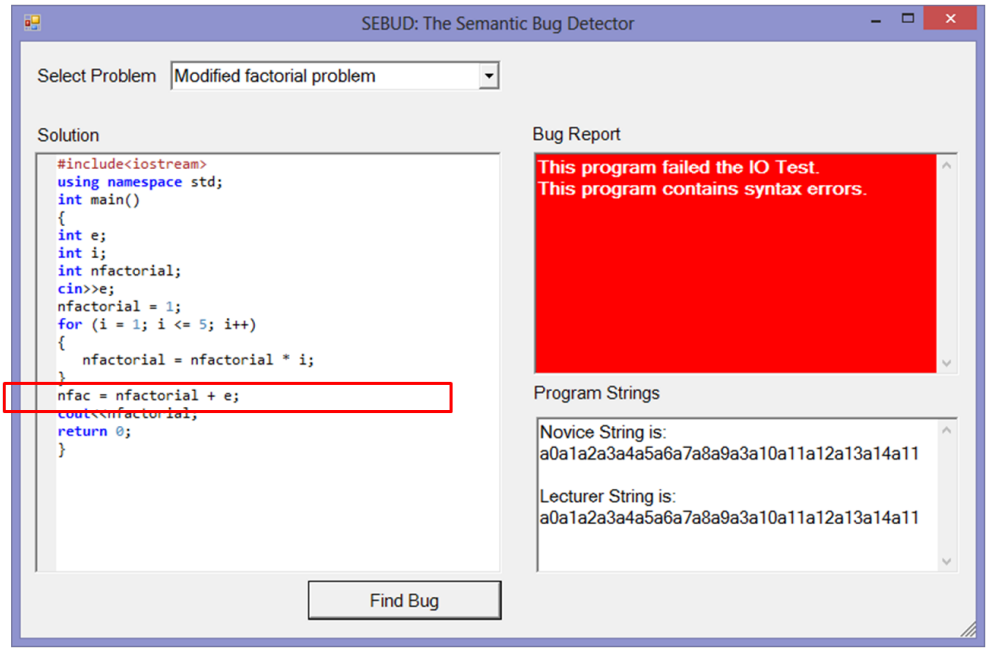


Figure 10.6: Reporting syntax errors with SEBUD

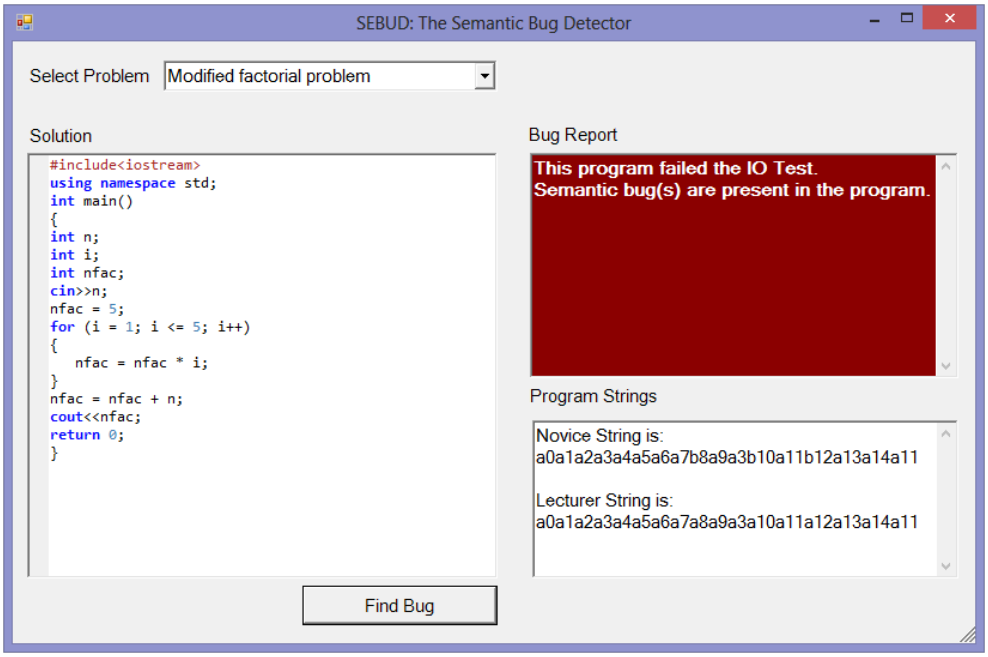


Figure 10.7: Detecting semantic errors with SEBUD

10.6.1 *Missing Code Fragments*

SEBUD was able to detect and fetch a missing code fragment that caused a semantic bug in the program shown in Listing 10.1. This bug is caused by

a novice who has commented out Line 14 — the line that was supposed to increment the factorial value by n.

Listing 10.1: Missing code fragment test case

```
1  #include<iostream>
2  using namespace std;
3  int main()
4  {
5    int n;
6    int i;
7    int nfac;
8    cin>>n;
9    nfac = 1;
10   for (i = 1; i <= 5; i++)
11   {
12     nfac = nfac * i;
13   }
14   // nfac = nfac + n;
15   cout<< nfac;
16   return 0;
17 }
```

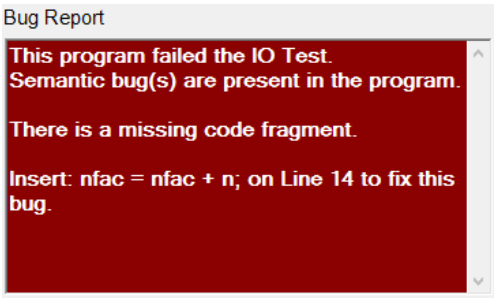


Figure 10.8: Detecting missing code fragments with SEBUD

This bug, though caused by a single missing statement, has changed the result of the program as it now computes 5! instead of $5! + n$. To be able to detect this bug, SEBUD used Algorithm 10.5 to step through the lecturer's string, making different choices of w_1, z, and w_2 from the lecturer's string, and checking each time, if $w_1 w_2 \equiv P_s$. After 13 iterations (when $z = a_{12}$) as shown in Table 10.1, SEBUD found the bug and fetches the line of code that is corresponding to the semantic symbol a_{12} as a possible fix. The bug report generated by SEBUD is shown in Figure 10.8.

Remark 10.2 (SEBUD is not a Tutor)**.** SEBUD does not have a module that can explain to a novice programmer how to fix the detected bug, even though it appears as though it explained in Figure 10.8. SEBUD only contains simple report generation templates that says what the type of bug is; and if semantic, what the missing or mis-composed fragment is. In Chapter 11, we will present an electronic tutor that has a *pedagogue module* with

many narration templates to guide the novice through the bug repair process.

	w_1	z	w_2
1	a_0	a_1	$a_2\,a_3\,a_4\,a_5\,a_6\,a_7\,a_8\,a_9\,a_3\,a_{10}\,a_{11}\,a_{12}\,a_{13}\,a_{14}\,a_{11}$
2	$a_0\,a_1$	a_2	$a_3\,a_4\,a_5\,a_6\,a_7\,a_8\,a_9\,a_3\,a_{10}\,a_{11}\,a_{12}\,a_{13}\,a_{14}\,a_{11}$
3	$a_0\,a_1\,a_2$	a_3	$a_4\,a_5\,a_6\,a_7\,a_8\,a_9\,a_3\,a_{10}\,a_{11}\,a_{12}\,a_{13}\,a_{14}\,a_{11}$
4	$a_0\,a_1\,a_2\,a_3$	a_4	$a_5\,a_6\,a_7\,a_8\,a_9\,a_3\,a_{10}\,a_{11}\,a_{12}\,a_{13}\,a_{14}\,a_{11}$
5	$a_0\,a_1\,a_2\,a_3\,a_4$	a_5	$a_6\,a_7\,a_8\,a_9\,a_3\,a_{10}\,a_{11}\,a_{12}\,a_{13}\,a_{14}\,a_{11}$
6	$a_0\,a_1\,a_2\,a_3\,a_4\,a_5$	a_6	$a_7\,a_8\,a_9\,a_3\,a_{10}\,a_{11}\,a_{12}\,a_{13}\,a_{14}\,a_{11}$
7	$a_0\,a_1\,a_2\,a_3\,a_4\,a_5\,a_6$	a_7	$a_8\,a_9\,a_3\,a_{10}\,a_{11}\,a_{12}\,a_{13}\,a_{14}\,a_{11}$
8	$a_0\,a_1\,a_2\,a_3\,a_4\,a_5\,a_6\,a_7$	a_8	$a_9\,a_3\,a_{10}\,a_{11}\,a_{12}\,a_{13}\,a_{14}\,a_{11}$
9	$a_0\,a_1\,a_2\,a_3\,a_4\,a_5\,a_6\,a_7\,a_8$	a_9	$a_3\,a_{10}\,a_{11}\,a_{12}\,a_{13}\,a_{14}\,a_{11}$
10	$a_0\,a_1\,a_2\,a_3\,a_4\,a_5\,a_6\,a_7\,a_8\,a_9$	a_3	$a_{10}\,a_{11}\,a_{12}\,a_{13}\,a_{14}\,a_{11}$
11	$a_0\,a_1\,a_2\,a_3\,a_4\,a_5\,a_6\,a_7\,a_8\,a_9\,a_3$	a_{10}	$a_{11}\,a_{12}\,a_{13}\,a_{14}\,a_{11}$
12	$a_0\,a_1\,a_2\,a_3\,a_4\,a_5\,a_6\,a_7\,a_8\,a_9\,a_3\,a_{10}$	a_{11}	$a_{12}\,a_{13}\,a_{14}\,a_{11}$
13	$a_0\,a_1\,a_2\,a_3\,a_4\,a_5\,a_6\,a_7\,a_8\,a_9\,a_3\,a_{10}\,a_{11}$	a_{12}	$a_{13}\,a_{14}\,a_{11}$
14	$a_0\,a_1\,a_2\,a_3\,a_4\,a_5\,a_6\,a_7\,a_8\,a_9\,a_3\,a_{10}\,a_{11}\,a_{12}$	a_{13}	$a_{14}\,a_{11}$
15	$a_0\,a_1\,a_2\,a_3\,a_4\,a_5\,a_6\,a_7\,a_8\,a_9\,a_3\,a_{10}\,a_{11}\,a_{12}\,a_{13}$	a_{14}	a_{11}

Table 10.1: Iterations of SEBUD when finding missing code fragment

To detect a block of mis-composed or missing program statements (consecutive statements in novice's program), SEBUD increases the length of z and iterates in a similar way to what is shown in Table 10.1.

10.7 REFLECTIONS ON BUG DETECTION

In this chapter, we have presented:

1. new algorithms for finding specific semantic bugs in novice programs using APDFA abstraction, and
2. SEBUD, a prototype software tool that uses the new algorithms to find and report semantic bugs in novice programs.

This technique can be implemented on a large scale for bug detection in novice programs — all we have done with SEBUD is a mere proof of concept. In Chapter 11, we will discuss how we have adapted this approach in the development of a software tool that can tutor novice programmers and hence be used for supporting the teaching of programming.

Part V

APPLICATIONS

There are many applications of the theories proposed in this thesis. This is not surprising because the field of novice program comprehension has grown rapidly in the last two decades with advancements in sub-topics such as automatic program marking, automatic bug detection, automatic tutoring, program visualisation, program summarisation, source code plagiarism detection, cognitive models of program comprehension, and so on. It is exciting that we can find a new approach from the field of formal language theory. Automata theory offers many dimensions to the way a *language* can be manipulated and there are a number of possible inferences on the new *program string* abstraction. In this part, we present a few applications that we have pursued. In Chapter 11, we present the design of an electronic programming tutor that suggests how semantic bugs can be repaired. Chapter 12 presents new algorithms for mapping the alphabets of two novice programs and estimating the similarity of the programs, if its 100% similar, we suggest that one of the programs may be plagiarised. Chapter 13 recaps the prototypes presented in this thesis for narration, bug detection, plagiarism detection and tutoring. In Chapter 14, we conclude this thesis and discuss the future directions.

11

AUTOMATIC TUTORING

Computer-aided teaching of programming still remains a difficult task for one simple reason — *dealing with the large variability in programs*. This is the problem we have defined formalisms and algorithms for solving from the beginning to Chapter 10 of this thesis. Towards teaching programming using software, we have established the following:

1. a way of enumerating all the possible variations of a program by concatenating the *regular languages* of program segment alternatives,
2. a way of building APDFAs from many program strings that represent the space of solutions, and
3. a tool (named SEBUD) that demonstrates how achievable these ideas are.

In this chapter, we present another tool called `Code Adviser` that attempts to find semantic bugs (using SEBUD's modules) and advises a novice programmer on how they may be repaired.

11.1 CODE ADVISER: DESIGN AND SCOPE

In this section we describe the design and scope of the `Code Adviser` using Figure 11.1.

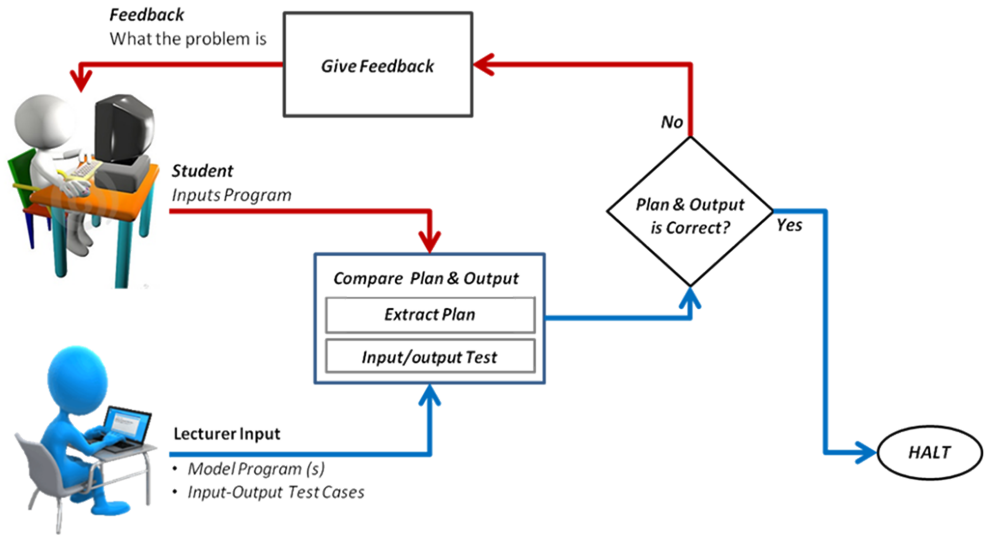


Figure 11.1: Automatic tutoring model

A lecturer enters a model program for a programming problem with a number of test cases. `Code Adviser` takes these inputs, cleans up and granulates the model program using the methods described in Chapter 4. It then proceeds to build APDFAs for each problem with the algorithms described in Chapter 9. Next, it attempts to compare a buggy novice program entered by a student to the finite list of program strings accepted by the problem's APDFA using the algorithms presented in Chapter 10. This is possible with the support of the IO Analyser presented in Chapter 8. Depending on the student's plan and output correctness, `Code Adviser` reports on discovered bugs and suggests repairs or declares the student's program as correct.

11.2 ACTING INTELLIGENT

To seemingly act intelligent, `Code Adviser` had to manipulate both the novice's and lecturer's program strings to give specific feedback that is not only helpful to the novice, but also demonstrates the wealth of knowledge it has about the novice's program. `Code Adviser` achieved this by:

1. reporting the percentage of similarity between the novice's and lecturer's programs by calculating the Levenshtein distance between the program strings of both programs,
2. discussing with the novice in *first person*, with statements such as *Ask me ...*, *I think you should ...*, and *Your program is ...*,
3. explaining bugs in the novice's terms, i.e. with the variables in the novice's programs and not the ones in the model solution, and
4. pointing to the buggy lines.

11.3 ONE STEP BUG REPAIR

`Code Adviser` does a *top-down one-step repair*[1]. It starts with the first mismatched program statement in the novice's program and prompts the novice to correct it. If the bug is well corrected, it informs the novice and terminates. Else, it points the novice to the next bug.

11.4 DISCUSSING BUG REPAIR

When `Code Adviser` discovers a bug, it *explains* how to repair the bug to the novice programmer using NOPRON's *narration composition module* discussed

1 repairing bugs, one bug at a time, starting from the first line in a program (the initial symbol) to the last line.

in Chapter 5. For example, if the bug repair is to add a new line that displays the value of the variable called sum, `Code Adviser` says:

I think you should insert an additional line after Line <line number here>.
The new line should display the value of `sum`*.*
Try this and seek my advice again.

More examples of how the `Code Adviser` handles the bug discussion are presented in Section 11.5.

11.5 CODE ADVISER IN ACTION

In this section, we demonstrate how `Code Adviser` reports bugs and discusses bug repairs. In Figure 11.2, `Code Adviser` uses a grid view control to show each program line with a feedback and a general note on the program's correctness. In this case the program is correct and the novice gets a corresponding message.

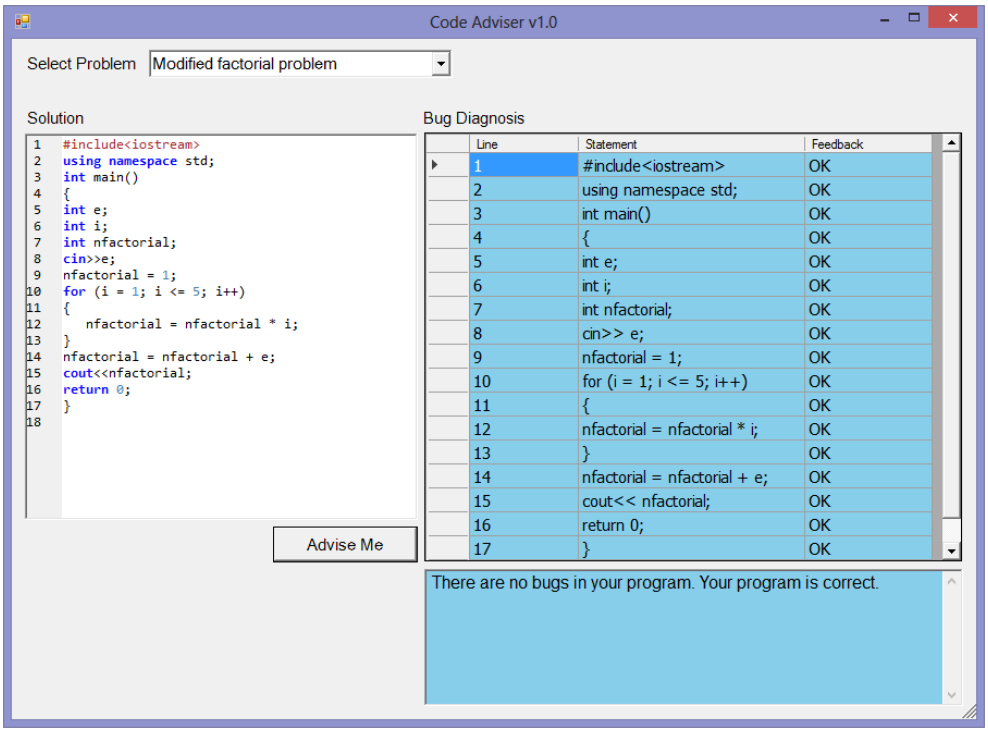


Figure 11.2: Code Adviser finding no bugs in program

On the contrary, in Figure 11.4 we show how `Code Adviser` reports a semantically buggy program, resulting from a line that was commented out as shown in Figure 11.3. `Code Adviser` points to the line and uses NOPRON's narration module to explain what needs to be done in syntax-free sentences. `Code Adviser` also offers a grid view control that further emphasizes the repair progress made by the student, showing the verified lines in the program, the suspected buggy line and other lines that are yet to be verified.

```
13  }
14  //nfactorial = nfactorial + e;
15  cout<<nfactorial;
```

Figure 11.3: Commenting out a line

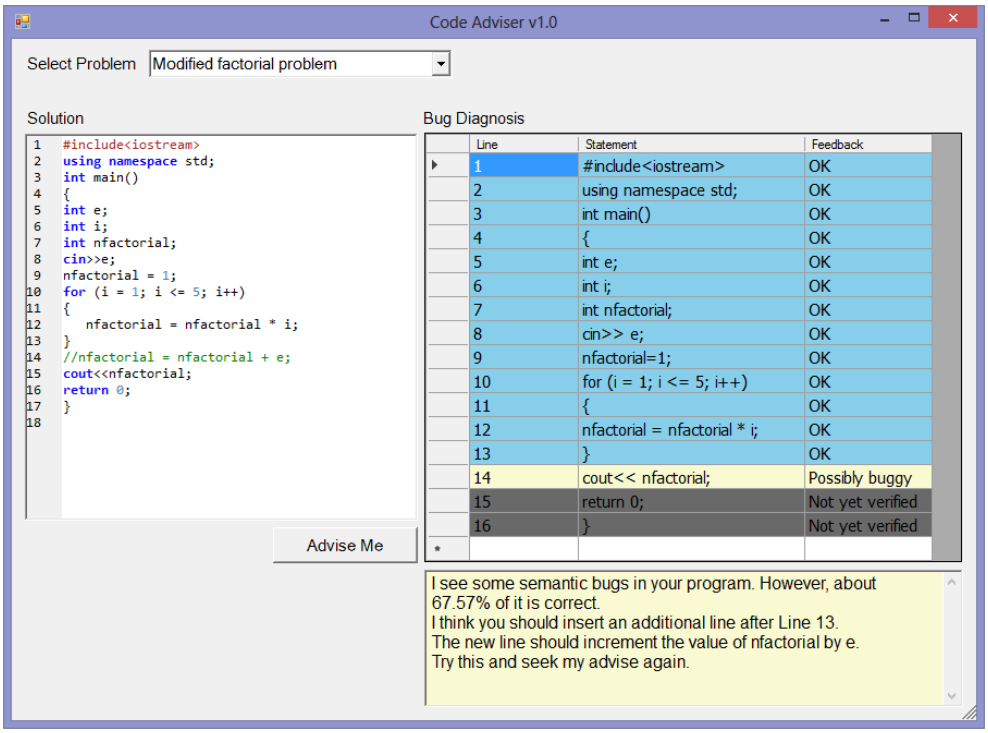


Figure 11.4: Code Adviser suggesting a fix

11.6 REFLECTIONS ON AUTOMATIC TUTORING

This chapter has presented one of the possibilities of the new bug detection technique proposed in this thesis. `Code Adviser` is *not a robust tool*, it is a proof of concept that we have used to demonstrate how a tool can be used to tutor novice programmers based on APDFAs of alternative solutions and SEBUD's detection modules. In the next chapter, we will examine another application of APDFAs, this time in plagiarism detection.

12 ESTIMATING PROGRAM SIMILARITY

Plagiarism sounds extreme, but when *two* novice programmers write *two* identical programs — it is very unlikely and raises suspicion. Source code plagiarism is very common among undergraduate computer science students and a lot of research has been carried out on how it can be detected, penalized, controlled or even stopped [Hage *et al.* 2011, Koss and Ford 2013, Đurić and Gašević 2012]. During this research, we realised that our formal DFA abstraction can be used to detect similar novice programs; hence, we have decided to comment about this. In this chapter, we present a new technique for detecting plagiarism in source code using DFAs.

12.1 OVERVIEW OF SOURCE CODE PLAGIARISM

Source code plagiarism is the textual alteration of computer programs originally owned by another person [Đurić and Gašević 2012]. This is a problem that is on the rise in undergraduate computer science courses that involve submitting programming assignments [Maurer *et al.* 2006]. There are two broad categories of source code plagiarism: Lexical and Structural alterations [Kustanto and Liem 2009]. Lexical alterations involve changing the program text slightly and usually do not require much knowledge of the semantics, while structural alterations involve reordering the code fragments or even using a different control structure.

The approach proposed in this chapter focuses on detecting lexical alterations. This includes detecting:

1. reformatting in the program being copied,
2. editing its comments,
3. adjusting its output (by adding or removing prompts), and
4. adjusting the program's granularity — breaking down lines into simple operations or chunking them up to condensed operations.

12.2 PICKING SIMILAR PROGRAMS WITH SPDFAS

In Chapter 9, we presented algorithms for constructing APDFAs from many equivalent programs, here, we are proposing a similar process — constructing Single Program Deterministic Finite Automata or SPDFAs. Given two

SPDFAs representing two novice's programs, the task of checking if the programs are similar is then reduced to the task of matching their respective program strings.

12.2.1 *Totally Similar Programs*

For the examples in this chapter, we will not use the term *plagiarised programs*, instead we will refer to the *apparent* plagiarism as *similarity*. Hence, we will estimate similarity in percentage. First let us establish what we mean by *totally similar programs*.

Definition 12.1 (Totally Similar Programs)**.** Let P_1 and P_2 be two novice programs. We say that P_1 is totally similar to P_2, if for some alphabets Σ_{p_1} and Σ_{p_2}, there exist two regular languages $\mathcal{L}p_1$ and $\mathcal{L}p_2$, each containing a single program string w_1 and w_2, such that:

$$w_1 = w_2 \mid \Sigma_{p_1} \rightarrow \Sigma_{p_2}, \text{ is an injective (one-to-one) mapping of the symbols in } \Sigma_{p_1} \text{ to the symbols in } \Sigma_{p_2}.$$

12.2.2 *No Decisions*

With Definition 12.1, we can easily decide if the program strings of two novice programs are totally similar, i.e. their program strings will be 100% the same. The reverse of this scenario is if the program strings only contain *matching substrings*. The following are the ways we may classify this:

1. We can use a string similarity estimation algorithm (such as Levenshtein distance) to provide a percentage of similarity between the program strings, and hence the programs. The challenge with this approach is that during alphabet matching (*see* Algorithm 10.2), symbols in program A that are not found in program B are not matched and replaced with symbols from the alphabet of B throughout the program string of B.

   Example 12.1 (Unfair Percentage)**.** Consider the two program fragments in P_1 and P_2 below. Let both programs be over arbitrary alphabets with symbols a and b respectively. Then the program string of P_1 is: $a_0a_1a_2a_3a_4a_5a_6$, while that of P_2 is: $b_0b_1b_2b_3b_4b_5b_6$. Using the mapping algorithm in Algorithm 10.2 to find P_2 in the alphabet of P_1, we will still arrive at $b_0b_1b_2b_3b_4b_5b_6$, which gives us a 0% match. This is because our algorithm will not see a as x, b as y, or c as z because of Line 6 in both programs. The plans on this line are dissimilar, contains all three variables and hence, none of the symbols from both alphabets can be matched.

P_1

```
1 int a;
2 int b;
3 int c;
4 cin>>a;
5 cin>>b;
6 c = a + b;
7 cout<<c;
```

P_2

```
1 int x;
2 int y;
3 int z;
4 cin>>x;
5 cin>>y;
6 z = x + y + 1;
7 cout<<z;
```

This does not produce a *fair* percentage. By inspection, we would have concluded that the programs contain 6 out of 7 matching lines and hence, should have about 86% percentage similarity. This, of course, reveals that our program string abstraction is very poor at detecting partial similarity, and we are left with the second classification option.

2. No decision. We have chosen not to classify program strings that do not match completely. This is because the percentage generated does not always reflect the situation of the programs.

12.3 SPDFA CONSTRUCTION

In this section we present an algorithm for constructing SPDFAs from a given novice program. Algorithm 12.1 shows the construction process. We start by creating a start state, and then new states afterwards and add each semantic token in the novice's program as transitions between the states. If a newly created state corresponds to the last token of the novice's program, this state is set as the accepting state of the SPDFA.

Data : A novice's program P_s
Result : An SPDFA M accepting only the novice's program string
1 initialize M as new DFA object;
2 create new state object and set M's start state attribute to new state;
3 add new state to SPDFA states;
4 create next state;
5 create first transition from start state to next state on first semantic token in P_s;
6 add transition to M transitions;
7 **foreach** *other semantic token* $\in P_s$ **do**
8 create and add new state;
9 set state to accepting if it is last semantic token in P_s;
10 create and add new transitions with new state on this semantic token;
end
11 **return** M

Algorithm 12.1 : Constructing SPDFA from a program

The alphabet generation algorithm of SPDFAs remains the same as earlier presented for novice programs in Algorithm 10.1 of Chapter 10.

12.4 MATCHING EXACT PROGRAMS

To ascertain that two programs are totally similar, we first have to construct SPDFAs for each program and check with either of the following methods:

SPDFA EQUIVALENCE. The two SPDFAs are equivalent if they accept the same language[1].

MAPPING PLANS. Using Algorithm 10.2 to check if the plans of both programs produce the same program string.

Here we present examples of the estimation process for totally similar programs using two programming problems:

1. next integer problem, and
2. the average of n numbers (with recursion) problem.

12.4.1 *Next Integer Problem*

We consider a scenario where a student programmer (Bob) had copied the code of another (Alice). In Listing 12.2, Bob tries to alter his program by changing variable names, and adding comments to Alice's version in Listing 12.1. However, before SPDFA construction, the programs are first granulated, and stripped of comments, prompts and all unnecessary texts with algorithms discussed for preprocessing and granulation in Chapter 5. The SPDFAs produced from these programs are displayed in Figure 12.1 and Figure 12.2 respectively. The different alphabets of the SPDFAs' languages are shown in Table 9.1 and Table 9.2.

Alice's SPDFA accepts the string $b_0 b_1 \dots b_9$ while Bob's SPDFA accepts $a_0 a_1 \dots a_9$. Using Algorithm 10.2, we note that:

$$a_i = b_i \mid 0 \leqslant i \leqslant 9.$$

That is, $a_0 a_1 \dots a_9 = b_0 b_1 \dots b_9$. These SPDFAs accept the same language of a single program string, hence, we conclude that the programs are *totally similar*.

1 Two DFAs are equivalent if they accept the same regular language [Martin 2003].

Listing 12.1: Alice's original program for the next integer problem

```
#include<iostream>
using namespace std;
int main() {
  int a;
  cin>>a;
  a = a + 1;
  cout<<a;
  return 0;
}
```

Listing 12.2: Bob's program copied from Alice for the next integer problem

```
//Author: Bob, John Doe
//Date: 25-August-2015
//Assignment for Next Integer
    Problem
#include<iostream>
using namespace std;
//Starting the main function
int main()
{
  int x; //Declaring x
  cin>>x; //Read x
  x = x + 1; //increment x
  cout<<x; //Display x
  return 0;
}
```

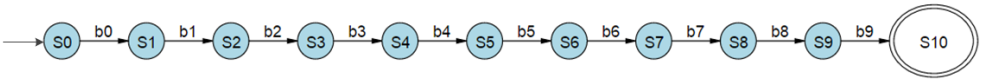


Figure 12.1: SPDFA for Alice

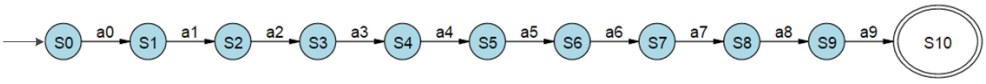


Figure 12.2: SPDFA for Bob

12.4.2 *The Sum Problem with a Function*

In this section we present how the similarity technique works with a modular program — sum of n numbers that calls the `SumUp()` method. The process is similar to the next integer problem example. In this case, Alice's and Bob's programs are shown in Listing 12.3 and Listing 12.4. As before, Bob is the one copying Alice in this example but he tries to change not only the comments and variables of the programs but also the formatting.

Symbol	Semantic Token
a_0	`#include <iostream>`
a_1	`using namespace std;`
a_2	`int main()`
a_3	`{`
a_4	`int a;`
a_5	`cin>> a;`
a_6	`a = a + 1;`
a_7	`cout<<a;`
a_8	`return 0;`
a_9	`}`

(a) Alice's Alphabet

Symbol	Semantic Token
b_0	`#include <iostream>`
b_1	`using namespace std;`
b_2	`int main()`
b_3	`{`
b_4	`int x;`
b_5	`cin>> x;`
b_6	`x = x + 1;`
b_7	`cout<<x;`
b_8	`return 0;`
b_9	`}`

(b) Bob's Alphabet

Figure 12.3: Alice vs Bob's Alphabets

Listing 12.3: Alice's original program for the sum problem using a function

```
1  //This is a program that
       calculates the sum of n
       integers
2  //using a function called
       SumUp()
3  #include<iostream>
4  using namespace std;
5  int i;
6  void SumUp() {
7    int sum;
8    sum = 0;
9    while (i > 0) { sum = sum + i;
         i = i - 1; }
10   cout<< sum;
11 }
12 void main() {
13   cin>> i;
14   while (i > 0) {
15     SumUp();
16   }
17 }
```

Listing 12.4: Bob's program copied from Alice for the sum problem using a function

```
1  //Author: Bob, John Doe
2  //Average Assignment
3  #include<iostream>
4  using namespace std;
5  int n;
6  void AddUp()
7  {
8    int s;
9    s = 0;
10   while (n > 0)
11   {
12     s = s + n;
13     n = n - 1;
14   }
15   cout<<s;
16 }
17 void main()
18 {
19   cin>> n;
20   while (n > 0)
21   {
22     AddUp();
23   }
24 }
```

Our technique generated the following two program strings from the programs' SPDFAs:

1. $a_0a_1a_2a_3a_4a_5a_6a_7a_4a_8a_9a_{10}a_{11}a_{10}a_{12}a_4a_{13}a_{14}a_4a_{15}a_{10}a_{10}$, and

2. $b_0b_1b_2b_3b_4b_5b_6b_7b_4b_8b_9b_{10}b_{11}b_{10}b_{12}b_4b_{13}b_{14}b_4b_{15}b_{10}b_{10}$.

These strings are found to be the same using the alphabet mapping technique and hence, we conclude that the programs are *totally similar*.

12.5 THE EXACT CODE MATCHER

In this section we describe a simple tool, called the `Exact Code Matcher` that matches two seemingly different programs using the SPDFA matching technique discussed earlier in this chapter.

In Figure 12.4, `Exact Code Matcher` takes two programs looking different, granulates them and tells if they are 100% similar or not. As part of the report generated, the `Exact Code Matcher` displays the SPDFAs of the two programs (as AGL scripts[2]) and their corresponding alphabets. It also shows a feedback of its assessment which is either 100% or no decision.

12.6 REFLECTIONS ON PROGRAM SIMILARITY ESTIMATION

Our new technique of abstracting programs to strings happened to be very useful in detecting that programs that are apparently dissimilar (because of the comments, identifier choices, and other program texts) may be plagiarised. This technique has its pitfall because it cannot report on partially copied programs, but it is efficient because the complexities of the algorithms involved are in linear time (both the algorithms for building SPDFAs and comparing strings). The technique can be useful for checking small programming assignments in large class sizes, i.e. $n \times n$ program similarity verification, where it is expected that the n^{th} program will be 100% similar to itself.

Finally, the exciting aspect of this chapter is seeing DFAs being applied in the domain of source code plagiarism detection. Advancement and modification of this technique hold better possibilities to the results displayed here.

2 AGL is an online tool for drawing directed graphs. http://rise4fun.com/Agl

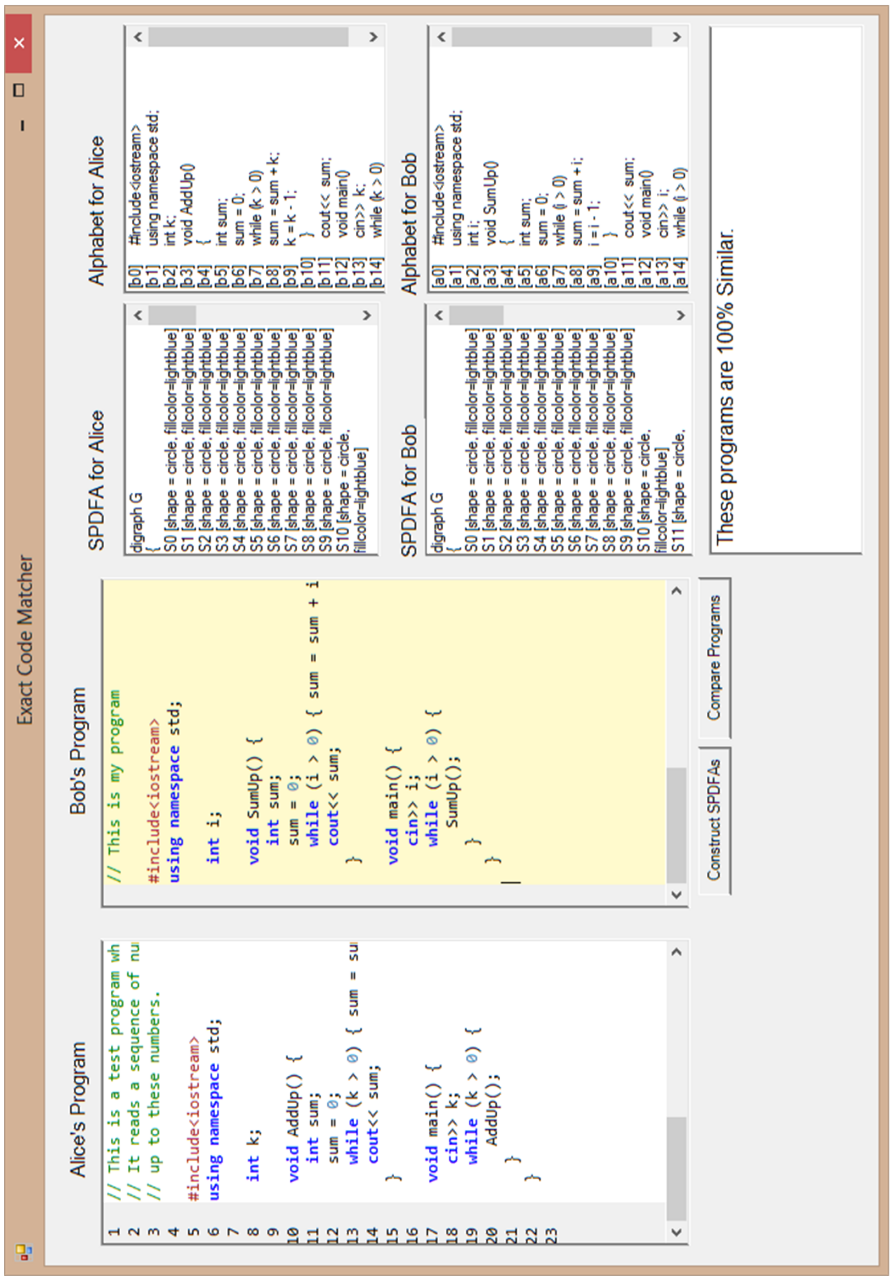


Figure 12.4: The Exact Code Matcher

13 PROTOTYPING

Earlier in this thesis, we have presented prototypes of software tools that demonstrate how realisable our new theories and algorithms are. One common reason why most of these prototypes are not robust enough to be deployed or distributed to end users is that it is tedious to recognise the entire language syntax for the C++ programming language. As such, proofs of the theories are presented with the prototypes. In this Chapter, we recap these prototypes; what they demonstrate, their limitations and expansion prospects.

13.1 RECAPPING PROTOTYPES

The following are the prototypes we have presented in this thesis:

NOPRON. In Chapter 5, we presented a prototype called NOPRON, an acronym for Novice PROgram Narrator. This idea has appeared in Ade-Ibijola *et al.* [2014a]. NOPRON aids program comprehension by abstracting source code to plain text. Possible improvements on NOPRON include upgrading it to handle more language constructs and across many programming languages.

APE. The Alternative Programs Enumerator or APE was presented in Chapter 7. APE takes a program and an IO test case and generates all the possible variations of the program using an algorithm. The variations are used as a knowledge-base (in the form of DFAs) for matching novice programs in later chapters. This idea has appeared in Ade-Ibijola *et al.* [2014b 2015b]. APE works well with relatively small novice programs. Improvements are possible with the advancements in High Performance Computing.

IO ANALYZER. The IO Analyzer, presented in Chapter 8, was used solely to verify generated programs using fixed test cases. This is not dynamic enough. An ideal robust tool should be able to generate random test cases given certain constraints — this is a possible improvement.

SEBUD. The Semantic Bug Detector was presented in Chapter 10. It makes inferences on program strings and reports known bugs. This in itself is not a complete tool because the *tutoring* or *pedagogue* module needs to be incorporated for explaining bugs to novice programmers — this was presented in the `Code Adviser` tool, discussed in Chapter 11.

CODE ADVISER. `Code Adviser` is presented in Chapter 11. This application attempts to explain to novice programmers, how semantic bugs may be repaired. It turned out to be a good prototype and opens possibilities for appending many more rules for inferencing on program strings and explaining bugs. Code Adviser has appeared in Ade-Ibijola *et al.* [2015a].

EXACT CODE MATCHER. Detection of source code plagiarism is one of the applications of the DFA abstraction of programs presented in this thesis. The `Exact Code Matcher`, presented in Chapter 12, does well in detecting lexically altered programs. However, estimating partly plagiarised programs is desirable and leaves us with room for improvement.

13.2 BONUS

Here we present two tools that were developed during the course of this work to provide some time-saving features such as generating AGL scripts and testing regular expressions. The following are the tools:

1. *Regex Parser*: for checking if a *string* is a member of a *language* using regular expressions,
2. *AGL Scripts Generator*: for automatic generation of scripts that can be used to generate DFAs or other directed graphs on the AGL website.

Both tools are available for free download at: `www.abejide.com/downloads`.

14

CONCLUSIONS AND FUTURE WORK

The need to automatically comprehend computer programs is rapidly increasing. As we have demonstrated in this thesis, comprehending novice programs is useful in many ways, from automated code marking, semantic bug detection, and code tutoring, to plagiarism detection. Likewise, comprehending expert programs is useful in software maintenance, documentation, and quality assurance. Revisiting the problem with the illustration in Figure 14.1, a comparison of a compiler to an ideal intelligent program tutor is presented.

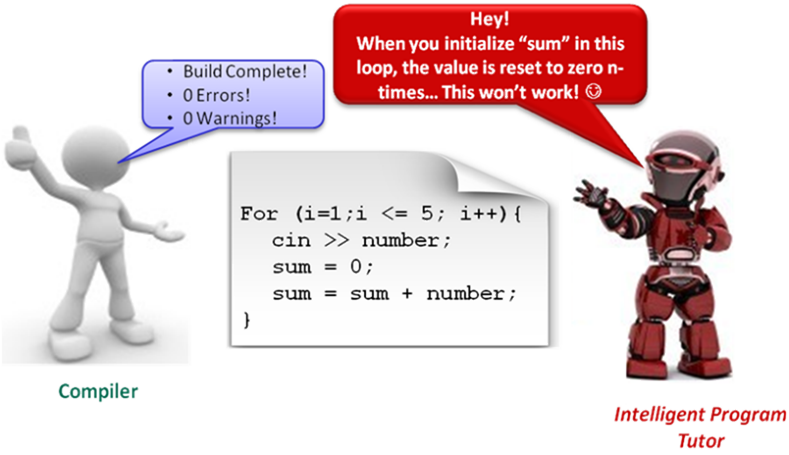


Figure 14.1: The comprehension problem re-visited

The difference is centered around the ability of the tutor to *understand* the problem being solved by the program, and hence, help find and explain semantic bugs. In this chapter we conclude this work, and list a number of future directions.

14.1 CONCLUSIONS

This thesis has addressed the comprehension problem by presenting techniques for:

1. *Abstraction*: regular expressions for abstracting novice programs,
2. *Aiding Comprehension*: a new way of enhancing the readability of programs using narrations,

3. *Automatic Enumeration of Program Variations*: new formalisms for defining program variations and an algorithm that generates all variations for a given program,
4. *Finite Automata for Knowledge Representation*: a new formal abstraction, using DFAs for representing the space of possible solutions,
5. *Semantic Bug Detection*: new algorithms for finding semantic bugs in APDFAs — DFAs for *alternative programs*,
6. *Applications*: description of how the new approach can be used in automatic tutoring and even plagiarism detection, and
7. *Prototypes*: tools that demonstrate how implementable the idea is.

However, one fact that we have to embrace is that there is no *silver bullet* for the program comprehension problem — there's always a cost for every new technique or approach. In this case, the cost is the computation time of the APE Algorithm that enumerates all possible variations on a novice program. In short, automatic comprehension of programs is nightmarishly difficult, solely because of the large space of possibilities involved. We do not have much concern about the performance of our algorithm because it is designed to handle small programs and we can also tolerate some significant waiting time while the computer generates these possibilities in a reasonable time.

14.2 FUTURE WORK

There are a number of desirable research directions leading from this work. With the foundation established here, it will be interesting to investigate the following aspects.

14.2.1 *Narrations*

AUTOMATIC INSTRUCTION GENERATION Can we employ this technique in generating laboratory instructions for novice programmers in a programming course? A *laboratory instruction* is a document that describes the procedure for solving problems, often given to undergraduate students in laboratory exercises or sessions. The format of this document varies widely across institutions and Computer Science courses. However in programming courses at the University of the Witwatersrand, Johannesburg; this document is often referred to as a *lab sheet* and it contains instructions about programming problems. The technique described in this paper can be adopted to automatically create this document from a template program entered by a lecturer.

EQUIVALENT LOOP NARRATION Can we have `for`, `do` and `while` loops narrated as the same? If this can be achieved, it will abstract the syntax and a novice will be forced to make a suitable choice when implementing the textual algorithm.

BETTER CHOICES OF NATURAL LANGUAGE Can we come up with words in English that are not also used as keywords in high level programming languages such as C++? Words such as *repeat* are considered "somewhat technical" and replacing them with *do again* or *keep doing* may aid complete novices from backgrounds outside of science to better understand the generated textual algorithms.

SURVEY It will also be interesting to see how much NOPRON helps novice programmers. Sampling the opinions of a class of students through a survey will give an insight to this.

14.2.2 *Program Variations and Automata Representation*

We will also see how:

1. the APE algorithm performs on high performance computers running GPUs (Graphics Processing Units) such as the GTX Titan Black 700 series, having more than 2000 CUDA cores. A faster performance may further justify the application of this technique to larger programs, and
2. we can reduce APDFAs using state minimization algorithms.

14.2.3 *Production Version of Code Adviser*

The next level with `Code Adviser` is to enhance its inference modules, store APDFAs for more programming problems in its database and create a production version of it. With a more robust version, we may also conduct a survey to gain insight about how helpful `Code Adviser` really is to students. On a similar note, a beta version of `Code Adviser` may contain the lecturer's *short note of purpose* for each line of the lecturer's solution. This will help `Code Adviser` to generate more intelligent repair suggestions for the novice programmer and answer the *why*-question of the novice, e.g. *why should I remove this line?*

14.2.4 *Tutoring by Plan Mirroring*

The plans extracted from the lecturer's program can be used, not only to validate the novice's program as presented in this thesis, but also to guide novice programmers from the start to finish of program composition. We refer to this as plan mirroring. As future work, we hope to design a tool that takes a lecturer's solution to a trivial problem and automatically explains to

the novice, what the next step in program composition is. The tool will also validate the last composition of the novice using the underlying lecturer's program as the answer model. To do this, the tool will have to handle:

1. Abstraction: extract the program plan from the lecturer's program and store it in a custom data structure using methods that we have established in Chapter 4,
2. Translation: generate a natural language explanation of the plan to the novice, so that the novice knows what to write next using the templates we have presented in Chapter 5,
3. Validation: compare the novice's line implementation to the lecturer's line implementation, considering other implementation variations, such as using different but equivalent operator types, and
4. Feedback: give a remark, correct or wrong, to the novice for every line in the program.

Part VI

APPENDIX

This part contain additional materials, organised into two categories as follows:

1. the library of regular expressions used for plan recognition (*see* Appendix A), and
2. the content of Microsoft's visual C++ compiler batch file (*see* Appendix B).

A

PLAN RECOGNIZERS

A.1 REGULAR EXPRESSIONS FOR PROGRAM ABSTRACTION

The following is the content of the library of regular expressions written in .Net — using the `System.Text.RegularExpressions` class — to recognise plans in C++ programs. The notations are relatively intuitive with ampersand (&) representing concatenation.

Listing A.1: Regular Expressions for Plan Recognition

```
// Building blocks
letter    = "([A-Za-z])"
letters    = letter & "+"
digit    = "([0-9])"
ident    = "([A-Za-z_][A-Za-z0-9_]*)"
eol    = "(\;)"
spc    = "(\s*)"
n_spc    = "(\s+)"
z_spc    = "(\s?)"
bra_open    = "\("
comma    = "(\,)"
inc_    = "(\+\+)"
dec_    = "(\-\-)"
inc_or_dec    = "(" & inc_ & "|" & dec_ & ")"
datatype    = "(int|float|double|bool)"
bra_closed    = "\)"
relop    = "(\<\=|\<|\>\=|\>|\=\=|\!\=)"
arth_op    = "(\+|\-|\*|\/|\%)"
special_ch    = "(\=|\,|\:|\;)"
ass_symbol    = "(\=)"
bool_op    = "((\&\&)|(\|\|)|(\!))"
real_value    = "(\-?\d+\.\d+)" '"(\-?[0-9]+\.[0-9]+)"
int_value    = "(\-?\d+)"
bool_value    = "(true|false)"
std_value    = "(" & int_value & "|" & real_value & "|" & bool_value &
    ")"
var_or_val    = "(" & ident & "|" & std_value & ")"
term    = "\(?" & spc & var_or_val & spc & arth_op & spc & var_or_val &
    spc & "\)?"
expr    = "\(?" & spc & term & spc & arth_op & spc & term & spc & "\)?"
parameters    = "(" & datatype & spc & ident & spc & comma & spc & ")*"
    & "(" & datatype & n_spc & ident & ")"
condition    = "\(" & ident & spc & relop & spc & "(" & std_value & "|"
    & ident & ")" & "\)"
```

```
condition_tester    = "\(?" & "(" & std_value & "|" & ident & ")" & spc
    & relop & spc & "(" & std_value & "|" & ident & ")" & "\)?"
complex_condition    = "\(" & condition & "(" & spc & bool_op & spc &
    condition & ")*" & spc & bool_op & spc & condition & spc & "\)" '
    varname & spc & relop & spc & std_value & "\)$"
output_item    = "(" & var_or_val & "|" & term & "|" & expr & ")"

// Array Declarations
square_bra_open    = "\\["
curly_bra_open    = "\{"
square_bra_closed    = "\]"
curly_bra_closed    = "\}"
array_decl1    = datatype & n_spc & ident & spc & "(" & square_bra_open
    & spc & "(" & std_value & ")*" & spc & square_bra_closed & ")+" &
    spc & eol
array_decl2    = datatype & n_spc & ident & spc & "(" & square_bra_open
    & "(" + std_value + ")*" & square_bra_closed + ")+" + spc +
    ass_symbol & spc & curly_bra_open & spc & "(" &
    list_of_var_comma_delimited & ")*" & spc & curly_bra_closed & spc &
    eol
array_decl    = array_decl1 & "|" & array_decl2

// Semantic tokens - line plans
_lib    = "#include\s*\<([A-Za-z]+)\>$"
_lib_iostream    = "#include\s*\<(iostream)\>$"
_lib_fstream    = "#include\s*\<(fstream)\>$"
_hdr    = "(using)\s+(namespace)\s+([A-Za-z]+)\;$"
_hdr_std    = "(using)\s+(namespace)\s+(std)\;$"
static_line    = "(static)" & n_spc & datatype & n_spc & ident & spc &
    eol
func_decr_with_parameters    = datatype & n_spc & ident & spc & bra_open
     & spc & "(" & parameters & ")*" & spc & bra_closed
sub_decr_with_parameters    = "(void)" & n_spc & ident & spc & bra_open
    & spc & "(" & parameters & ")*" & spc & bra_closed
subroutine_    = "(void)" & n_spc & ident & spc & bra_open & spc &
    bra_closed
functn_decr    = datatype & n_spc & ident & spc & bra_open & spc &
    bra_closed
func_or_sub    = "(" & func_decr_with_parameters & "|" &
    sub_decr_with_parameters & "|" & subroutine_ & "|" & functn_decr & "
    )"

// Assignments
ass_stmt1    = ident & spc & ass_symbol & spc & std_value & eol   //var
    = value
ass_stmt2    = ident & spc & ass_symbol & spc & ident & eol  //var = var
ass_stmt3    = ident & spc & ass_symbol & spc & term & eol //var = term
ass_stmt4    = ident & spc & ass_symbol & spc & expr & eol //var =
    expression
ass_stmt    = "(" & ass_stmt1 & "|" & ass_stmt2 & "|" & ass_stmt3 & "|"
    & ass_stmt4 & ")"
```

```
63 ass_stmt_gen    = "^(" & ident & spc & ass_symbol & spc & ")" //var =
       unknown expression, recognising the prefix
64
65 //Loops
66 for_loop    = "(for)" & spc & bra_open & spc & ident & spc & ass_symbol
       & spc & int_value & spc & eol & spc & ident & spc & relop & spc &
       int_value & spc & eol & spc & ident & inc_or_dec & spc & bra_closed
       & spc & "$"
67 for_loop2    = "(for)" & spc & bra_open & spc & "(" & datatype & ")?" &
       spc & ident & spc & ass_symbol & spc & int_value & spc & eol & spc &
        ident & spc & relop & spc & "(" & var_or_val & ")" & spc & eol &
       spc & ident & inc_or_dec & spc & bra_closed & spc & "$"
68
69 //Declarations
70 single_var_declr    = "^(" & datatype & n_spc & ident & eol & ")$"
71 single_parameter_declr    = datatype & n_spc & ident
72 multiple_var_declr    = datatype & n_spc & "(" & ident & comma & spc & "
       )+" & ident & eol & "$"
73 var_declr_and_use    = datatype & n_spc & ident & spc & ass_symbol & spc
        & std_value & eol
74 var_decl    = "(" & single_var_declr & "|" & multiple_var_declr & ")"
75
76 //Conditional Statements
77 if_stmt    = "(if)" & spc & condition
78 if_cond_then_stmt    = if_stmt & spc & ass_stmt
79 else_stmt    = "(else)" & spc & ass_stmt
80 else_if_stmt    = "(else if)" & spc & condition
81 else_if_cond_then_stmt    = else_if_stmt & spc & ass_stmt
82 while_loop    = "(while)" & spc & condition
83 do_stmt    = "(do)$"
84
85 //Other plans: initialisations, incrementations, etc
86 return_0    = "(return 0)" & spc & eol
87 init    = ident & spc & ass_symbol & spc & "(0)" & eol
88 gen_init    = ident & spc & ass_symbol & spc & "(" & int_value & ")" &
       spc & eol
89 incr1    = "(" & ident & spc & ass_symbol & spc & ident & spc & "(\+)" &
        spc & "(1)" & eol & ")" //a = a + 1;
90 decr1    = "(" & ident & spc & ass_symbol & spc & ident & spc & "(\-)" &
        spc & "(1)" & eol & ")" //a = a - 1;
91 incr2    = "^" & ident & spc & inc_ & eol & "$"
92 decr2    = "^" & ident & spc & dec_ & eol & "$"
93 incr3    = "(" & ident & spc & "(\+\=)" & spc & "(1)" & eol & ")"
94 decr3    = "(" & ident & spc & "(\-\=)" & spc & "(1)" & eol & ")"
95 incr4    = inc_ & spc & ident & eol
96 decr4    = dec_ & spc & ident & eol
97 incr5    = "(" & ident & spc & ass_symbol & spc & ident & spc & "(\+)" &
        spc & ident & eol & ")"
98 decr5    = "(" & ident & spc & ass_symbol & spc & ident & spc & "(\-)" &
        spc & ident & eol & ")"
99 incr6    = "(" & ident & spc & "(\+\=)" & spc & ident & eol & ")"
```

```
decr6    = "(" & ident & spc & "(\-\=)" & spc & ident & eol & ")"
incr7    = "(" & ident & spc & ass_symbol & spc & ident & spc & "(\+)" &
     spc & int_value & eol & ")"
decr7    = "(" & ident & spc & ass_symbol & spc & ident & spc & "(\-)" &
     spc & int_value & eol & ")"
incr8    = "(" & ident & spc & "(\+\=)" & spc & int_value & eol & ")"
decr8    = "(" & ident & spc & "(\-\=)" & spc & int_value & eol & ")"
inc_by_one_plan    = "(" & incr1 & ")|(" & incr2 & ")|(" & incr3 & ")|("
     & incr4 & ")"
decr_by_one_plan    = "(" & decr1 & ")|(" & decr2 & ")|(" & decr3 & ")|(
    " & decr4 & ")"
inc_by_int_plan    = "(" & incr7 & ")|(" & incr8 & ")"
decr_by_int_plan    = "(" & decr7 & ")|(" & decr8 & ")"
inc_by_n_plan    = "(" & incr5 & ")|(" & incr6 & ")"
decr_by_n_plan    = "(" & decr5 & ")|(" & decr6 & ")"
const_    = "(const)" & spc & datatype & spc & ident & spc & ass_symbol
    & spc & std_value & eol

//IO plans, methods, routines, etc
input    = "(cin)" & spc & "(\>\>)" & spc & ident & spc & "((\>\>)" &
    spc & ident & spc & ")*" & eol
output    = "(cout)" & spc & "(\<\<)" & spc & output_item & spc & "
    ((\<\<)" & spc & output_item & spc & ")*" & "((\<\<)(endl))?" & eol
cin_line    = "(cin)" & spc & "(\>\>)" & spc & ident & spc & eol
cout_line    = "(cout)" & spc & "(\<\<)" & spc & ident & spc & eol
main_method    = "(void|int)" & n_spc & "(main)" & spc & bra_open & spc
    & bra_closed
global_variable    = "(static)" & n_spc & datatype & n_spc & ident & spc
     & eol
list_of_var_comma_delimited    = "(" & ident & spc & comma & spc & ")*"
    & "(" & ident & ")"
subroutine_call    = ident & spc & bra_open & spc & "(" &
    list_of_var_comma_delimited & ")?" & spc & bra_closed & spc & eol
function_call    = ident & spc & ass_symbol & spc & ident & spc &
    bra_open & spc & "(" & list_of_var_comma_delimited & ")?" & spc &
    bra_closed & spc & eol

//Multi-line plans
stmt_block    = "(" & ass_stmt & spc & ")+"
if_block    = if_stmt & spc & "\{" & spc & stmt_block & spc & "\}"
do_while_block    = do_stmt & spc & "\{" & spc & stmt_block & spc & "\}"
     & spc & while_loop
```

B

VISUAL C++ COMPILER

B.1 THE VISUAL C++ COMPILER BATCH FILE

The following are the commands in `VsDevCmd.bat`.

Listing B.1: Batch file for Visual C++ Compiler

```
@call :GetVSCommonToolsDir
@if "%VS120COMNTOOLS%"=="" goto error_no_VS120COMNTOOLSDIR

@call "%VS120COMNTOOLS%VCVarsQueryRegistry.bat" 32bit No64bit

@if "%VSINSTALLDIR%"=="" goto error_no_VSINSTALLDIR
@if "%FrameworkDir32%"=="" goto error_no_FrameworkDIR32
@if "%FrameworkVersion32%"=="" goto error_no_FrameworkVer32
@if "%Framework40Version%"=="" goto error_no_Framework40Version

@set FrameworkDir=%FrameworkDir32%
@set FrameworkVersion=%FrameworkVersion32%

@if not "%WindowsSDK_ExecutablePath_x86%" == "" (
@set "PATH=%WindowsSDK_ExecutablePath_x86%;%PATH%"
)

@if not "%WindowsSdkDir%" == "" (
@set "PATH=%WindowsSdkDir%bin\x86;%PATH%"
@set "INCLUDE=%WindowsSdkDir%include\shared;%WindowsSdkDir%include\um;%
    WindowsSdkDir%include\winrt;%INCLUDE%"
@set "LIB=%WindowsSdkDir%lib\winv6.3\um\x86;%LIB%"
@set "LIBPATH=%WindowsSdkDir%References\CommonConfiguration\Neutral;%
    ExtensionSDKDir%\Microsoft.VCLibs\12.0\References\
    CommonConfiguration\neutral;%LIBPATH%"
)

@rem
@rem Root of Visual Studio IDE installed files.
@rem
@set DevEnvDir=%VSINSTALLDIR%Common7\IDE\

@rem PATH
@rem ----
@if exist "%VSINSTALLDIR%Team Tools\Performance Tools" (
@set "PATH=%VSINSTALLDIR%Team Tools\Performance Tools;%PATH%"
)
```

```
35 @if exist "%ProgramFiles%\HTML Help Workshop" set PATH=%ProgramFiles%\
       HTML Help Workshop;%PATH%
36 @if exist "%ProgramFiles(x86)%\HTML Help Workshop" set PATH=%
       ProgramFiles(x86)%\HTML Help Workshop;%PATH%
37 @if exist "%VCINSTALLDIR%VCPackages" set PATH=%VCINSTALLDIR%VCPackages;%
       PATH%
38
39 @if exist "%FrameworkDir%%Framework40Version%" set PATH=%FrameworkDir%%
       Framework40Version%;%PATH%
40 @if exist "%FrameworkDir%%FrameworkVersion%" set PATH=%FrameworkDir%%
       FrameworkVersion%;%PATH%
41 @if exist "%VSINSTALLDIR%Common7\Tools" set PATH=%VSINSTALLDIR%Common7\
       Tools;%PATH%
42 @if exist "%VCINSTALLDIR%BIN" set PATH=%VCINSTALLDIR%BIN;%PATH%
43 @set PATH=%DevEnvDir%;%PATH%
44
45 @rem Add path to MSBuild Binaries
46 @if exist "%ProgramFiles%\MSBuild\12.0\bin" set PATH=%ProgramFiles%\
       MSBuild\12.0\bin;%PATH%
47 @if exist "%ProgramFiles(x86)%\MSBuild\12.0\bin" set PATH=%ProgramFiles(
       x86)%\MSBuild\12.0\bin;%PATH%
48
49 @rem Add path to TypeScript Compiler
50 @if exist "%ProgramFiles%\Microsoft SDKs\TypeScript\1.0" set PATH=%
       ProgramFiles%\Microsoft SDKs\TypeScript\1.0;%PATH%
51 @if exist "%ProgramFiles(x86)%\Microsoft SDKs\TypeScript\1.0" set PATH=%
       ProgramFiles(x86)%\Microsoft SDKs\TypeScript\1.0;%PATH%
52
53 @if exist "%VSINSTALLDIR%VSTSDB\Deploy" (
54 @set "PATH=%VSINSTALLDIR%VSTSDB\Deploy;%PATH%"
55 )
56
57 @if not "%FSHARPINSTALLDIR%" == "" (
58 @set "PATH=%FSHARPINSTALLDIR%;%PATH%"
59 )
60
61 @if exist "%DevEnvDir%CommonExtensions\Microsoft\TestWindow" (
62 @set "PATH=%DevEnvDir%CommonExtensions\Microsoft\TestWindow;%PATH%"
63 )
64
65 @rem INCLUDE
66 @if exist "%VCINSTALLDIR%ATLMFC\INCLUDE" set INCLUDE=%VCINSTALLDIR%
       ATLMFC\INCLUDE;%INCLUDE%
67 @if exist "%VCINSTALLDIR%INCLUDE" set INCLUDE=%VCINSTALLDIR%INCLUDE;%
       INCLUDE%
68
69 @rem LIB
70 @rem ---
71 @if exist "%VCINSTALLDIR%ATLMFC\LIB" set LIB=%VCINSTALLDIR%ATLMFC\LIB;%
       LIB%
72 @if exist "%VCINSTALLDIR%LIB" set LIB=%VCINSTALLDIR%LIB;%LIB%
```

```
73
74 @rem LIBPATH
75 @if exist "%VCINSTALLDIR%ATLMFC\LIB" set LIBPATH=%VCINSTALLDIR%ATLMFC\
       LIB;%LIBPATH%
76 @if exist "%VCINSTALLDIR%LIB" set LIBPATH=%VCINSTALLDIR%LIB;%LIBPATH%
77 @if exist "%FrameworkDir%%Framework40Version%" set LIBPATH=%FrameworkDir
       %%Framework40Version%;%LIBPATH%
78 @if exist "%FrameworkDir%%FrameworkVersion%" set LIBPATH=%FrameworkDir%%
       FrameworkVersion%;%LIBPATH%
79
80 @rem VisualStudioVersion
81 @set VisualStudioVersion=12.0
82
83 @goto end
84
85 :GetVSCommonToolsDir
86 @set VS120COMNTOOLS=
87 @call :GetVSCommonToolsDirHelper32 HKLM > nul 2>&1
88 @if errorlevel 1 call :GetVSCommonToolsDirHelper32 HKCU > nul 2>&1
89 @if errorlevel 1 call :GetVSCommonToolsDirHelper64  HKLM > nul 2>&1
90 @if errorlevel 1 call :GetVSCommonToolsDirHelper64  HKCU > nul 2>&1
91 @exit /B 0
92
93 :GetVSCommonToolsDirHelper32
94 @for /F "tokens=1,2*" %%i in ('reg query "%1\SOFTWARE\Microsoft\
       VisualStudio\SxS\VS7" /v "12.0"') DO (
95 @if "%%i"=="12.0" (
96 @SET "VS120COMNTOOLS=%%k"
97 )
98 )
99 @if "%VS120COMNTOOLS%"=="" exit /B 1
100 @SET "VS120COMNTOOLS=%VS120COMNTOOLS%Common7\Tools\"
101 @exit /B 0
102
103 :GetVSCommonToolsDirHelper64
104 @for /F "tokens=1,2*" %%i in ('reg query "%1\SOFTWARE\Wow6432Node\
       Microsoft\VisualStudio\SxS\VS7" /v "12.0"') DO (
105 @if "%%i"=="12.0" (
106 @SET "VS120COMNTOOLS=%%k"
107 )
108 )
109 @if "%VS120COMNTOOLS%"=="" exit /B 1
110 @SET "VS120COMNTOOLS=%VS120COMNTOOLS%Common7\Tools\"
111 @exit /B 0
112
113 :error_no_VS120COMNTOOLSDIR
114 @echo ERROR: Cannot determine the location of the VS Common Tools folder
       .
115 @goto end
116
117 :error_no_VSINSTALLDIR
```

```
118 @echo ERROR: Cannot determine the location of the VS installation.
119 @goto end
120
121 :error_no_FrameworkDIR32
122 @echo ERROR: Cannot determine the location of the .NET Framework 32bit
        installation.
123 @goto end
124
125 :error_no_FrameworkVer32
126 @echo ERROR: Cannot determine the version of the .NET Framework 32bit
        installation.
127 @goto end
128
129 :error_no_Framework40Version
130 @echo ERROR: Cannot determine the .NET Framework 4.0 version.
131 @goto end
132
133 :end
```

BIBLIOGRAPHY

[Abd-El-Hafiz and Basili 1993] S Abd-El-Hafiz and Victor R Basili. Documenting programs using a library of tree structured plans. In *Proceedings of the Conference on Software Maintenance*, pages 152–161. IEEE, 1993.

[Adam and Laurent 1980] Anne Adam and Jean-Pierre Laurent. Laura, a system to debug student programs. *Artificial Intelligence*, 15(1):75–122, 1980.

[Ade-Ibijola *et al.* 2014a] Abejide Ade-Ibijola, Sigrid Ewert, and Ian Sanders. Abstracting and narrating novice programs using regular expressions. In *Proceedings of the Annual Conference of the South African Institute for Computer Scientists and Information Technologists*, pages 19–28. ACM, 2014.

[Ade-Ibijola *et al.* 2014b] Abejide Ade-Ibijola, Sigrid Ewert, and Ian Sanders. How many ways can we skin a cat? searching the space of novice program plan variations. In *Proceedings of the Joint PRASA, RobMech and AfLaT International Symposium*, page 317. IEEE, 2014.

[Ade-Ibijola *et al.* 2015a] Abejide Ade-Ibijola, Sigrid Ewert, and Ian Sanders. Introducing Code Adviser: A DFA-driven electronic programming tutor. In *Proceedings of the 20th International Conference on the Implementation and Application of Automata*, pages 307–312. Springer, 2015.

[Ade-Ibijola *et al.* 2015b] Abejide Ade-Ibijola, Sigrid Ewert, and Ian Sanders. Searching the space of novice program plan variations. *Completed and to be submitted to the South African Computer Journal*, 2015.

[AGL] AGL. *Automatic Graph Layout*. http://rise4fun.com/agl. Accessed: 2015-04-10.

[Ahmadzadeh *et al.* 2011] Marzieh Ahmadzadeh, Elham Mahmoudabadi, and Farzad Khodadadi. Pattern of plagiarism in novice students' generated programs: An experimental approach. *Journal of Information Technology Education*, 10, 2011.

[Al-Omari 1999] Hani Moh'D Ahmad Al-Omari. *CONCEIVER: A program understanding system*. PhD thesis, University Kebangsaan Malaysia, 1999.

[Amalfitano *et al.* 2010] D. Amalfitano, A.R. Fasolino, A. Polcaro, and P. Tramontana. DynaRIA: A tool for AJAX web application comprehension.

IEEE 18th International Conference on Program Comprehension, pages 46–47, 2010.

[Astrachan and Reed 1995] Owen Astrachan and David Reed. AAA and CS-1: The applied apprenticeship approach to CS 1. In *ACM SIGCSE Bulletin*, volume 27, pages 1–5. ACM, 1995.

[Barnes *et al.* 1997] David J Barnes, Sally Fincher, and Simon Thompson. Introductory problem solving in computer science. In *5th Annual Conference on the Teaching of Computing*, pages 36–39, 1997.

[Bennedsen and Caspersen 2008] Jens Bennedsen and Michael E Caspersen. Optimists have more fun, but do they learn better? on the influence of emotional and social factors on learning introductory computer science. *Computer Science Education*, 18(1):1–16, 2008.

[Bornat 1987] Richard Bornat. *Programming from First Principles*. Prentice Hall International Series in Computer Science. Prentice-Hall International, Hertfordshire, United Kingdom, 1987.

[Brooks 1983] Ruven Brooks. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, 18(6):543–554, 1983.

[Brzozowski and Tamm 2012] Janusz Brzozowski and Hellis Tamm. Quotient complexities of atoms of regular languages. In *Developments in Language Theory*, volume 7410 of *16th International Conference on Developments in Language Theory, Lecture Notes in Computer Science*, pages 50–61. Springer Berlin Heidelberg, 2012.

[Buckner *et al.* 2005] Jonathan Buckner, Joseph Buchta, Maksym Petrenko, and Vaclav Rajlich. Jripples: A tool for program comprehension during incremental change. In *International Workshop on Program Comprehension*, volume 5, pages 149–152, 2005.

[Chandra *et al.* 2011] Satish Chandra, Emina Torlak, Shaon Barman, and Rastislav Bodik. Angelic debugging. In *33rd International Conference on Software Engineering*, pages 121–130. IEEE, 2011.

[Cosma and Joy 2006] Georgina Cosma and MS Joy. Source-code plagiarism: A UK academic perspective. *Research Report No. 422, Department of Computer Science, University of Warwick*, 2006.

[Crosby *et al.* 2002] Martha E Crosby, Jean Scholtz, and Susan Wiedenbeck. The roles beacons play in comprehension for novice and expert programmers. In *14th Workshop of the Psychology of Programming Interest Group*, pages 58–73, 2002.

[Dalton and Kreahling 2010] A.R. Dalton and W. Kreahling. Automated construction of memory diagrams for program comprehension. In *Proceedings of the 48th Annual Southeast Regional Conference*. ACM, 2010.

[D'Antoni and Veanes 2014] Loris D'Antoni and Margus Veanes. Minimization of symbolic automata. In *ACM SIGPLAN Notices*, volume 49, pages 541–553. ACM, 2014.

[Dasgupta *et al.* 2007] Sanjoy Dasgupta, Christos Papadimitriou, and Umesh Vazirani. *Algorithms*. McGraw-Hill Higher Education, New York, 2007.

[Ebrahimi and Schweikert 2006] A. Ebrahimi and C. Schweikert. Empirical study of novice programming with plans and objects. *ACM SIGCSE Bulletin*, 38(4):52–54, 2006.

[Ebrahimi 1994] A. Ebrahimi. Novice programmer errors: Language constructs and plan composition. *International Journal of Human-Computer Studies*, 41(4):457–480, 1994.

[Ehrlich and Soloway 1984] Kate Ehrlich and Elliot Soloway. An empirical investigation of the tacit plan knowledge in programming. In *Human Factors in Computer Systems*, pages 113–133. Ablex Norwood, NJ, 1984.

[Feurzeig *et al.* 1981] W. Feurzeig, P. Horwitz, and R.S. Nickerson. *Microcomputers in Education*. Report No. 4798, National Institute of Education (U.S.) and Ministerio para el Desarrollo de la Inteligencia, Venezuela. Bolt, Beranek, and Newman, Incorporated, 1981.

[Fincher 1999] S. Fincher. What are we doing when we teach programming? *29th Annual Frontiers in Education Conference*, 1:12A4–1, 1999.

[GCC] GCC. *GCC Releases: A timeline of GNU Compiler Collection*. http://www.gnu.org/software/gcc/releases.html. Accessed: 2014-03-20.

[Guindon 1990] Raymonde Guindon. Knowledge exploited by experts during software system design. *International Journal of Man-Machine Studies*, 33(3):279–304, 1990.

[Hage *et al.* 2011] Jurriaan Hage, Peter Rademaker, and Nikè van Vugt. Plagiarism detection for Java: a tool comparison. In *Computer Science Education Research Conference*, pages 33–46. Open Universiteit, Heerlen, 2011.

[Haiduc *et al.* 2010] S. Haiduc, J. Aponte, and A. Marcus. Supporting program comprehension with source code summarization. *ACM/IEEE 32nd International Conference on Software Engineering*, 2:223–226, 2010.

[Haiduc *et al.* 2012] Sonia Haiduc, Gabriele Bavota, Rocco Oliveto, Andrian Marcus, and Andrea De Lucia. Evaluating the specificity of text retrieval queries to support software engineering tasks. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 1273–1276. IEEE, 2012.

[Hansen *et al.* 2012] Michael E Hansen, Andrew Lumsdaine, and Robert L Goldstone. Cognitive architectures: a way forward for the psychology of programming. In *Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pages 27–38. ACM, 2012.

[Harandi and Ning 1988] MT Harandi and JQ Ning. Pat: A knowledge-based program analysis tool. In *Proceedings of the Conference on Software Maintenance*, pages 312–318. IEEE, 1988.

[Hardy *et al.* 1952] Godfrey Harold Hardy, John Edensor Littlewood, and George Pólya. *Inequalities*. Cambridge University press, 1952.

[Harris and Cilliers 2006] Nathan Harris and Charmain Cilliers. A program beacon recognition tool. In *7th International Conference on Information Technology Based Higher Education and Training*, pages 216–225. IEEE, 2006.

[Helmick 2007] Michael T Helmick. Interface-based programming assignments and automatic grading of Java programs. *ACM SIGCSE Bulletin*, 39(3):63–67, 2007.

[Heninger 2004] A. Heninger. Analyzing unicode text with regular expressions. *Proceedings of the 26th Internationalization and Unicode Conference, San Jose, CA*, pages 1–18, 2004.

[Hirsch 1996] Robin Hirsch. Relation algebras of intervals. *Artificial Intelligence*, 83(2):267–295, 1996.

[Hopcroft *et al.* 2006] John E Hopcroft, Rajeev Motwani, and Jeffrey D Ullman. *Automata theory, languages, and computation*. Addison Wesley, USA, third edition, 2006.

[Hovemeyer and Spacco 2013] David Hovemeyer and Jaime Spacco. Cloudcoder: a web-based programming exercise system. *Journal of Computing Sciences in Colleges*, 28(3):30–30, 2013.

[Hovemeyer *et al.* 2013] David Hovemeyer, Matthew Hertz, Paul Denny, Jaime Spacco, Andrei Papancea, John Stamper, and Kelly Rivers. Cloudcoder: building a community for creating, assigning, evaluating and sharing programming exercises. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, pages 742–742. ACM, 2013.

[ISOCPP] ISOCPP. *The C++ programming language: The Standard.* https://isocpp.org/std/the-standard. Accessed: 2015-01-12.

[Jeffries *et al.* 1981] Robin Jeffries, Althea A Turner, Peter G Polson, and Michael E Atwood. The processes involved in designing software. *Cognitive Skills and Their Acquisition*, 255:283, 1981.

[Johnson and Soloway 1985] W.L. Johnson and E. Soloway. PROUST: Knowledge-based program understanding. *IEEE Transactions on Software Engineering*, 3:267–275, 1985.

[Johnson 1990] W Lewis Johnson. Understanding and debugging novice programs. *Artificial Intelligence*, 42(1):51–97, 1990.

[Juliff 1997] Peter Juliff. Marketing programming to nonprogrammers. In *Informatics in Higher Education*, pages 73–82. Chapman and Hall Limited, 1997.

[Koss and Ford 2013] Ian Koss and Richard Ford. Authorship is continuous: Managing code plagiarism. *IEEE Security & Privacy*, (2):72–74, 2013.

[Kranch 2012] Douglas A Kranch. Teaching the novice programmer: A study of instructional sequences and perception. *Education and Information Technologies*, 17(3):291–313, 2012.

[Krinke 2003] Jens Krinke. *Advanced slicing of sequential and concurrent programs*. PhD Thesis, Universitat Passau, April 2003.

[Krinke 2005] Jens Krinke. Program slicing. *Handbook of Software Engineering and Knowledge Engineering*, 3:307–332, 2005.

[Kuhn *et al.* 2010] A. Kuhn, D. Erni, and O. Nierstrasz. Towards improving the mental model of software developers through cartographic visualization. *arXiv preprint arXiv:1001.2386*, 2010.

[Kustanto and Liem 2009] C. Kustanto and I. Liem. Automatic source code plagiarism detection. In *10th ACIS International Conference on Software Engineering, Artificial Intelligences, Networking and Parallel/Distributed Computing*, pages 481–486, May 2009.

[Lahtinen *et al.* 2005] Essi Lahtinen, Kirsti Ala-Mutka, and Hannu-Matti Järvinen. *A study of the difficulties of novice programmers*, volume 37, pages 14–18. ACM, 2005.

[Lanza *et al.* 2005a] M. Lanza, S. Ducasse, H. Gall, and M. Pinzger. Code Crawler–an information visualization tool for program comprehension. In *Proceedings of the 27th IEEE International Conference on Software Engineering*, pages 672–673, 2005.

[Lanza *et al.* 2005b] Michele Lanza, Stéphane Ducasse, Harald Gall, and Martin Pinzger. Codecrawler-an information visualization tool for program comprehension. In *Proceedings of the 27th International Conference on Software Engineering*, pages 672–673. IEEE, 2005.

[Letovsky 1987] Stanley Letovsky. Cognitive processes in program comprehension. *Journal of Systems and Software*, 7(4):325–339, 1987.

[Littman *et al.* 1987] David C Littman, Jeannine Pinto, Stanley Letovsky, and Elliot Soloway. Mental models and software maintenance. *Journal of Systems and Software*, 7(4):341–355, 1987.

[Looi 1988] Chee-Kit Looi. Analysing novice programs in a prolog intelligent teaching system. In *European Conference on Artificial Intelligence*, pages 314–319, 1988.

[Lukey 1980] F J Lukey. Understanding and debugging programs. *International Journal of Man-Machine Studies*, 12(2):189–202, 1980.

[Malik 2010] DS Malik. *C++ programming: Program design including data structures*. Cengage Learning, 2010.

[Martin 2003] J.C. Martin. *Introduction to Languages and the Theory of Computation, Third Edition*. McGraw-Hill, New York, 2003.

[Maurer *et al.* 2006] Hermann Maurer, Frank Kappe, and Bilal Zaka. Plagiarism - a survey. *Journal of Universal Computer Science*, 12(8):1050–1084, 2006.

[Mayer 1981] Richard E Mayer. The psychology of how novices learn computer programming. *ACM Computing Surveys*, 13(1):121–141, 1981.

[Mendelson 1997] Elliott Mendelson. *Introduction to mathematical logic*. CRC press, 1997.

[Mohnen 2002] Markus Mohnen. A graph-free approach to data-flow analysis. In R. Nigel Horspool, editor, *Compiler Construction*, volume 2304 of *Lecture Notes in Computer Science*, pages 46–61. Springer Berlin Heidelberg, 2002.

[Morgan *et al.* 2015] Andrew Morgan, Bonita Sharif, and Martha E Crosby. Understanding a novice programmer's progression of reading and summarizing source code. *Eye Movements in Programming Education II: Analyzing the Novice's Gaze*, page 13, 2015.

[Mössenböck 2010] H. Mössenböck. An example of a Taste program. *The Compiler Generator Coco/R User Manual*, pages 30–32, 2010.

[MSDN] MSDN. *Regular Expression Language–Quick Reference: Microsoft Developer Network Documentation on the .Net framework, version 4.5, regular*

expressions. http://msdn.microsoft.com/en-us/library/az24scfc.asp. Accessed: 2013-12-19.

[Murray 1986] W R Murray. *Automatic Program Debugging for Intelligent Tutoring Systems*. PhD Thesis, Texas University, Austin USA, November 1986.

[Nguyen *et al.* 2013] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. SemFix: Program repair via semantic analysis. In *Proceedings of the International Conference on Software Engineering*, pages 772–781. IEEE Press, 2013.

[O'Callahan and Jackson 1997] Robert O'Callahan and Daniel Jackson. Lackwit: A program understanding tool based on type inference. In *Proceedings of the 19th International Conference on Software Engineering*, pages 338–348. Citeseer, 1997.

[Panichella *et al.* 2012] Sebastiano Panichella, Jairo Aponte, Massimiliano Di Penta, Andrian Marcus, and Gerardo Canfora. Mining source code descriptions from developer communications. In *20th International Conference on Program Comprehension*, pages 63–72. IEEE, 2012.

[Papancea *et al.* 2013] Andrei Papancea, Jaime Spacco, and David Hovemeyer. An open platform for managing short programming exercises. In *Proceedings of the 9th Annual International Conference on International Computing Education Research*, pages 47–52. ACM, 2013.

[Pea and Kurland 1984] Roy D Pea and D Midian Kurland. On the cognitive effects of learning computer programming. *New Ideas in Psychology*, 2(2):137–168, 1984.

[Pennington 1987] Nancy Pennington. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology*, 19(3):295–341, 1987.

[Pinter 2014] Charles C Pinter. *A Book of Set Theory*. Courier Dover Publications, USA, 2014.

[Pyott and Sanders 1991] S. Pyott and I. Sanders. ALEX: an aid to teaching algorithms. *ACM SIGCSE Bulletin*, 23(3):36–44, 1991.

[Rich and Waters 1988] Charles Rich and Richard C Waters. The programmer's apprentice. *Computer*, 21(11):10–25, 1988.

[RiSE] RiSE. *Rise4fun tools: Online visual C++ compiler and regular expression evaluator*. http://rise4fun.com. Accessed: 2013-12-23.

[Rist 1990] Robert S Rist. Variability in program design: the interaction of process with knowledge. *International Journal of Man-Machine Studies*, 33(3):305–322, 1990.

[Rist 1991] Robert S Rist. Knowledge creation and retrieval in program design: A comparison of novice and intermediate student programmers. *Human-Computer Interaction*, 6(1):1–46, 1991.

[Robillard 1999] Pierre N Robillard. The role of knowledge in software development. *Communications of the ACM*, 42(1):87–92, 1999.

[Rugaber 1992] Spencer Rugaber. Program comprehension for reverse engineering. In *AAAI Workshop on AI and Automated Program Understanding*, pages 106–110. Citeseer, 1992.

[Russell *et al.* 1995] Stuart Russell, Peter Norvig, and Artificial Intelligence. A modern approach. *Artificial Intelligence. Prentice-Hall, Eaglewood Cliffs*, 25, 1995.

[Ruth 1976] Gregory R Ruth. Intelligent program analysis. *Artificial Intelligence*, 7(1):65–85, 1976.

[Satir and Brown 1995] G. Satir and D. Brown. *C++: the core language*. O'Reilly Media, Inc., 1995.

[Schwartz *et al.* 2011] Jacob T Schwartz, Domenico Cantone, Eugenio G Omodeo, and Martin Davis. *Computational logic and set theory: Applying Formalized Logic to Analysis*. Springer, 2011.

[Scintilla] Scintilla. *Online Documentation of ScintillaNET Code Editor API*. http://scintillanet.codeplex.com/documentation. Accessed: 2015-01-10.

[Shackelford 1997] R.L. Shackelford. *Introduction to Computing and Algorithms*. Addison-Wesley Longman Publishing Company, USA, 1997.

[Shneiderman and Mayer 1979] Ben Shneiderman and Richard Mayer. Syntactic/semantic interactions in programmer behavior: A model and experimental results. *International Journal of Computer & Information Sciences*, 8(3):219–238, 1979.

[Singh *et al.* 2013] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. Automated feedback generation for introductory programming assignments. In *ACM SIGPLAN Notices*, volume 48, pages 15–26. ACM, 2013.

[Sipser 2006] M. Sipser. *Introduction to the Theory of Computation*. Cengage Learning, 2006.

[Smith] R Smith. *Working Draft, Standard for Programming Language C++*. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4296.pdf. Accessed: 2015-01-12.

[Solar-Lezama 2008] Armando Solar-Lezama. *Program Synthesis by Sketching*. ProQuest, 2008.

[Soloway and Ehrlich 1984] Elliot Soloway and Kate Ehrlich. Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering*, (5):595–609, 1984.

[Soloway and Spohrer 1989] E. Soloway and J.C. Spohrer. *Studying the Novice Programmer*. Lawrence Erlbaum Associates, 1989.

[Soloway *et al.* 1981] Elliot M Soloway, Beverly Woolf, Eric Rubin, and Paul Barth. Meno-II: An intelligent tutoring system for novice programmers. In *Proceedings of the 7th International Joint Conference on Artificial Intelligence*, volume 2, pages 975–977. Morgan Kaufmann Publishers Inc., 1981.

[Spohrer and Soloway 1986] James C Spohrer and Elliot Soloway. Novice mistakes: Are the folk wisdoms correct? *Communications of the ACM*, 29(7):624–632, 1986.

[Spohrer 1992] J.C. Spohrer. *MARCEL: Simulating the novice programmer*. Ablex Publishing Corporation, New Jersey, USA, 1992.

[Springel 2016] Volker Springel. High performance computing and numerical modelling. In *Star Formation in Galaxy Evolution: Connecting Numerical Models to Reality*, pages 251–358. Springer, 2016.

[Srivastava *et al.* 2010] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S Foster. From program verification to program synthesis. In *ACM SIGPLAN Notices*, volume 45, pages 313–326. ACM, 2010.

[Stacho 2007] J. Stacho. *Lecture Notes on Discrete and Combinatorial Mathematics, University of Toronto*. http://www.cs.toronto.edu/ stacho/-macm101.pdf, 2007. Accessed: 2014-12-10.

[Stein 1998] Lynn Andrea Stein. What we've swept under the rug: Radically rethinking CS1. *Computer Science Education*, 8(2):118–129, 1998.

[Storey *et al.* 2001] Margaret-Anne Storey, C. Best, and J. Michand. SHriMP views: An interactive environment for exploring Java programs. *Proceedings of the 9th International Workshop on Program Comprehension*, pages 111–112, 2001.

[Storey 2006] Margaret-Anne Storey. Theories, tools and research methods in program comprehension: past, present and future. *Software Quality Journal*, 14(3):187–208, 2006.

[Storey 2011] Margaret-Anne Storey. An interactive visualization environment for exploring Java programs: SHriMP views revisited. *IEEE 19th International Conference on Program Comprehension*, pages xviii–xviii, 2011.

[Tarski 1941] Alfred Tarski. On the calculus of relations. *The Journal of Symbolic Logic*, 6(03):73–89, 1941.

[Thompson and Taylor 2008] Ambler Thompson and Barry N Taylor. *Guide for the Use of the International System of Units*. 2008.

[Ullman 2014] Jeffrey Ullman. *Chapter 2: Finite Automata*. Undergraduate Lecture notes in Computer Science, Stanford University, 2014.

[Đurić and Gašević 2012] Zoran Đurić and Dragan Gašević. A source code similarity system for plagiarism detection. *The Computer Journal*, page bxs018, 2012.

[Van Leeuwen 1994] Jan Van Leeuwen. *Handbook of theoretical computer science: Algorithms and Complexity*, volume 1. MIT Press, 1994.

[Vessey 1985] Iris Vessey. Expertise in debugging computer programs: A process analysis. *International Journal of Man-Machine Studies*, 23(5):459–494, 1985.

[Vogts 2009] Dieter Vogts. Plagiarising of source code by novice programmers a cry for help? In *Proceedings of the Annual Conference of the South African Institute of Computer Scientists and Information Technologists*, pages 141–149. ACM, 2009.

[Von Mayrhauser and Vans 1995] Anneliese Von Mayrhauser and A Marie Vans. Program comprehension during software maintenance and evolution. *Computer*, 28(8):44–55, 1995.

[Walenstein 2003] Andrew Walenstein. Observing and measuring cognitive support: Steps toward systematic tool evaluation and engineering. In *11th IEEE International Workshop on Program Comprehension*, pages 185–194. IEEE, 2003.

[Wang *et al.* 2013] Tzu-Hua Wang, Mei-Hung Chiu, Jing-Wen Lin, and Chin-Cheng Chou. Diagnosing students' mental models via the web-based mental models diagnosis system. *British Journal of Educational Technology*, 44(2):E49–E51, 2013.

[Watson 2010] Bruce William Watson. *Constructing minimal acyclic deterministic finite automata*. PhD Thesis, University of Pretoria, November 2010.

[Wertz 1982] Harald Wertz. Stereotyped program debugging: an aid for novice programmers. *International Journal of Man-Machine Studies*, 16(4):379–392, 1982.

[Wertz 1987] Harald Wertz. *Automatic correction and improvement of programs*. Halsted Press, USA, 1987.

COLOPHON

This work is based on doctoral research supported by the National Research Foundation (NRF) of the Republic of South Africa. Any opinion, findings and conclusions or recommendations expressed in this thesis are those of the author and therefore the NRF does not accept liability in regard thereto.

This thesis was typeset using the typographical look-and-feel `classicthesis` developed by André Miede, and made available for free at: `http://code.google.com/p/classicthesis/`. The author of this thesis has made many modifications to the `classicthesis` template to arrive at this final version.

Automatic Novice Program Comprehension for Semantic Bug Detection

Abejide Olu, Ade-Ibijola | 2016
researcher@abejide.com
www.abejide.com