EXPLORATIONS IN THE COGNITIVE SCIENCE OF COMPUTER PROGRAMMING

By

Jinrong Li

A Thesis Submitted to the Graduate

Faculty of Rensselaer Polytechnic Institute

in Partial Fulfillment of the

Requirements for the Degree of

DOCTOR OF PHILOSOPHY

Major Subject: COGNITIVE SCIENCE

Approved by the
Examining Committee:

____
Professor Selmer Bringsjord , Thesis Adviser

____
Professor Mike Kalsher, Member

____
Professor Sergei Nirenburg, Member

____
Doctor Nick Cassimatis , Member

Rensselaer Polytechnic Institute
Troy, New York

December 2014
(For Graduation December 2014)

المنارة للاستشارات

www.manaraa.com

ProQuest Number: 10029802

All rights reserved

INFORMATION TO ALL USERS
The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript
and there are missing pages, these will be noted. Also, if material had to be removed,
a note will indicate the deletion.

ProQuest 10029802

Published by ProQuest LLC (2016). Copyright of the Dissertation is held by the Author.

All rights reserved.
This work is protected against unauthorized copying under Title 17, United States Code
Microform Edition © ProQuest LLC.

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 - 1346

المنارة للاستشارات

www.manaraa.com

CONTENTS

LIST OF TABLES . . . iii

LIST OF FIGURES . . . iv

ACKNOWLEDGMENT . . . v

ABSTRACT . . . viii

1. INTRODUCTION . . . 1
   - 1.1 Our Two Main Challenges . . . 2
     - 1.1.1 The Educational Challenge **E** . . . 2
     - 1.1.2 The Technological Challenge **T**: . . . 3
   - 1.2 Approaches to the Educational Challenge **E** . . . 3
     - 1.2.1 Constructivism . . . 3
     - 1.2.2 Cognitivism . . . 4
     - 1.2.3 Our Approach: Formal Meta-Cognitivism$^+$ . . . 4
   - 1.3 Hypotheses/Claims . . . 5
     - 1.3.1 Main Hypothesis **H1** With Respect to **E** . . . 5
   - 1.4 Approaches to the Technological Challenge **T** . . . 5
   - 1.5 Computers and Computer Programming . . . 6

2. EXPERIMENTS . . . 7
   - 2.1 Purpose . . . 7
     - 2.1.1 Chief Hypothesis . . . 7
     - 2.1.2 Other Factors . . . 7
   - 2.2 Methods . . . 7
     - 2.2.1 Programming Problems . . . 7
     - 2.2.2 Background Questions and Variables . . . 8
     - 2.2.3 Procedure . . . 8
     - 2.2.4 Test of Chief Hypothesis (**H1**) Using χ^2 Statistics . . . 9
       - 2.2.4.1 Data Analysis Methods: χ^2 Test . . . 9
       - 2.2.4.2 Contingency Tables . . . 11
       - 2.2.4.3 χ^2 Test Assumptions . . . 12
   - 2.3 Experiment 1: Language Problems . . . 12
     - 2.3.1 Programming Challenges . . . 12
     - 2.3.2 Experiment 1A: Online . . . 14

المنارة للاستشارات
www.manaraa.com

2.3.2.1 Subjects 14
2.3.2.2 Procedure 14
2.3.2.3 Results 15
2.3.3 Experiment 1B: On-Site 15
2.3.3.1 Subjects 15
2.3.3.2 Procedure 15
2.3.3.3 Results 15
2.3.4 Experiment 1C: With Logic Training 16
2.3.4.1 Subjects 16
2.3.4.2 Procedure 16
2.3.4.3 Results 17
2.4 Experiment 2: Digital Circuits Problems 17
2.4.1 Subjects 17
2.4.2 Programming Challenges 17
2.4.3 Results 18
2.5 Experiment 3: Target Defense Problems 18
2.5.1 Subjects 18
2.5.2 Programming Challenges 19
2.5.3 Results 19
2.6 Comprehensive Data Analysis of Combined Experimental Results 19
2.6.1 Success and Educational Factors 19
2.6.1.1 Logic Training 20
2.6.1.2 Number of Logic Courses 21
2.6.1.3 Programming-Language Training 22
2.6.1.4 Amount of Programming Language Training 23
2.6.2 Success and Academic/Demographic Factors 23
2.6.2.1 Academic Major 23
2.6.2.2 SAT/ACT Scores 24
2.6.2.3 Gender and Race/Ethnicity 25
2.7 Discussion 26
2.7.1 Summary and Conclusion 26
2.7.2 Limitations and Future Work 27
3. ASSESSMENT OF EDUCATIONAL PROGRAMMING ENVIRONMENTS . . . 28
3.1 History and Overview of Educational Programming Languages and Environments 28
3.1.1 Logo 28
3.1.1.1 What is Logo? 28

المنارة للاستشارات
www.manaraa.com

3.1.1.2 The Dream of Logo's Educational Power 29
3.1.2 A *Brief* Assessment of Logo 29
3.1.3 Logo's Relatives 31
3.1.3.1 StarLogo, LEGOsheets, Lego Mindstorms 31
3.1.4 Recent Applications in the Same Vein 31
3.1.4.1 Alice Projects 32
3.2 Conclusion 32
4. COGNITION AND AUTOMATIC PROGRAMMING 33
4.1 Overview & Assessment of Automatic Programming 33
4.2 On Cognitive Science, Hypercomputation, and Automatic Programming . . 40
4.3 "Visual Logic" and Suggestive Aspect of Experiments 43
5. A NEW EDUCATIONAL PROGRAMMING LANGUAGE: REASON 44
5.1 Elementary Logic Encapsulated 45
5.2 What is Clear Thinking? 50
5.2.1 Context-Dependent v. Context-Indepedent Reasoning 51
5.2.2 The King-Ace Problem 52
5.2.3 The Wine Drinker Problem 53
5.3 The Logo Programming Language; Logic Programming 53
5.4 The Reason Programming Language 54
5.4.1 Reason in the Context of the Four Paradigms of Computer Programming 54
5.4.2 Proofs as Programs through a Simple Denotational Proof Language 55
5.4.3 Cracking King-Ace and Wine Drinker with Reason 58
5.4.3.1 Cracking the King-Ace Problem with Reason 58
5.4.3.2 Cracking the Wine Drinker Problem with Reason 60
5.5 The Future of Reason 61
6. THE FUTURE 63
6.1 A Necessary Confession 63
6.2 Creativity, Computer Programming, and Automatic Programming 63
6.2.1 An Exception: Story Generation 65
6.3 Programming East versus West 66
6.4 Final Remarks 66
LITERATURE CITED 67
APPENDICES

المنارة للاستشارات
www.manaraa.com

A. Experiment 1 Programming Problems 76

B. Experiment 2 Programming Problems 78

C. Experiment 3 Programming Problems 83

D. List of Background Questions And Options 85

D.1 Background Information 85

D.2 Academic Information 86

D.3 Education 87

D.3.1 Logic Education 87

D.3.2 Computer Programming Language (PL) Education 88

D.3.3 Programming Language Knowledge 89

D.3.4 First Programming Language 91

E. Post Problem Solving Feedback Questions 94

F. List of Courses Input and Standard Names 96

G. List Logic and Programming Courses Entries and Groups 104

H. Additional Data 108

المنارة للاستشارات
www.manaraa.com

LIST OF TABLES

2.1 A sample of 2×2 contingency table 11

2.2 Grading scale 14

2.3 Experiment 1 Result: Success and Logic Training 17

2.4 Success and Logic Training 20

2.5 Success and Logic Training (Derived) 20

2.6 Success and Number Logic Courses 22

2.7 Success and Number Logic Courses (Derived) 22

2.8 Success and Programming Language (PL) Training 23

2.9 Success and Programming Language (PL) Training (Derived) 23

2.10 Success and Number Programming Language Courses 25

2.11 Success and Number Programming Language Courses (Derived) 25

2.12 Overall Result: Success and SAT/ACT Scores 26

المنارة للاستشارات
www.manaraa.com

LIST OF FIGURES

2.1 Success Proportion (%) and Logic Training 21

2.2 Success Proportion (%) and Logic Training (Derived) 21

2.3 Success (%) and PL Training 24

2.4 Success (%) and PL Training (Derived) 24

4.1 Automantic Programming Problem 39

5.1 Visual Countermodel in Wine Drinker Puzzle 61

B.1 *AND* Gate ($O = A \wedge B$). 78

B.2 *OR* Gate ($O = A \vee B$). 78

B.3 *NOT* Gate ($O = \neg A$). 78

B.4 An Example of Programmable Logic Array (PLA) 79

B.5 An example of implementation of two functions (Xs mark connections). 80

B.6 An example of implementation of two equations (matrix). 80

B.7 Experiment 2. Programming Problem 1 81

B.8 Experiment 2. Programming Problem 2 81

H.1 Success and Academic Major 109

H.2 Success and SAT/ACT Scores 109

H.3 Success and Race/Ethnicity 110

www.manaraa.com

ACKNOWLEDGMENT

This is both the hardest and easiest part of my dissertation. It is the hardest because it is impossible to list all who have made this possible; it could easily become the longest chapter in the dissertation. Making matters even more challenging is that even though I list only a few contributors, the brevity of what I say about them pales in comparison to what they have meant to my work and my life. On the other hand, this is at the same time the easiest section, because many of these people and events are fresh in my memory — indeed some are continuing to actively work with me toward eventually establishing a true cognitive science of computer programming.

It is with great emotions, gratitude, and honor that I write below to thank those who have inspired, taught, and guided me all those years.

First, I would like to thank Professor Selmer Bringsjord, my thesis advisor; without his unwavering help, this thesis would have been impossible. This thesis is part of an ongoing and deep collaboration, and many parts of it were borne of direct collaboration with Dr. Bringsjord. It was a joy to take his logic classes. Conversations with him are always inspirational. I initially entered study of programming from a cognitive-science point of view with very different assumptions than his. But the experimental results reported below showed me otherwise. More than once, I felt low and was on the edge of quitting because of other distractions, but Professor Bringsjord was there to extend his advice and support, enabling me to carry on. An intimately related thanks is due to Dr. Bringsjord's longtime collaborator, Dr. Konstantine Arkoudas, whose orginal work in programming langauges is truly impressive and deeply inspiring. I am most fortunate to have been directly involved in the earliest days of the Bringsjord-Arkoudas pursuit of new approaches to teaching programming, and to automatic programming.

I would like to thank my dissertation committee members, Professors Mike Kalsher, who provided extra help and advice on many occasions to make sure I was on the right track. I also thank Professor Sergei Nirenburg and Dr. Nick Cassimatis for their generous support in providing guidance in times of need, despite their busy schedules.

I would like to thank Rensselaer "wizards" Paula Monahan, Betty Osgania, and Jennifer Mumby for their help in my RPI graduate program at different times. I also thank my RAIR-Lab mates, Naveen Sundar Govindarajulu, John Licato, and Josh Taylor, who helped me with my thesis research & discussions. Naveen, in particular, was very

المنارة للاستشارات
www.manaraa.com

helpful when it came to analyzing the proposed solutions provided by subjects on difficult programming challenges. John provided invaluable assistance with svn. Josh's herculean work on putting citations and references into conformity with required formats is deeply appreciated.

I would like to thank NYU Professors Mike Landy and Larry Maloney, who introduced me to computational modeling of human perception. A special note of thanks is due to the late Dr. Hao Wang, a legendary professor of logic, who not only generously opened his home for me to stay for free for over two years, but also taught me logic is a clear and beautiful language. Although he has long passed, I have many fond memories of dinner-table conversations related to Russian puzzles and Chinese dissidents, two of whom stayed at his house, shortly after June 1989. I also like to thank Professors Dehua Ju in East China University of Science and Technology at Shanghai and Yuqing Gu at Chinese Academy of Sciences. It was under their guidance, that I started my first computer programming project experience, which installed my passion and interests in programming and automatic programming generation.

I have special memories of, and express my thanks to, Shimmy Biegeleisen for his calls to me a week after my son was born, to raise my salary twice in one day, and let me know I could work part time to take care of my new baby and to pursue my research. To Goumatie Thackurdeen, I have gratitude for her treasured surprise baby shower card, on which were the signatures of more than a dozen who, along with Shimmy, were gone on that September day in 2001. For a long time, I thought my school project was insignificant after attending so many memorial services and funerals after that day. It took a long time for me to realize that terrorists could take their lives, but the caring and high spirits of those who were taken will not die. We need to live on for our future generations, to enable them to learn better and to build a better world.

Special thanks to Dr. Elizabeth Bringjord, who showed her support and advice in so many ways over the years, from sharing her doctoral research and writing experiences early on, to acting as a role model demonstrating both a strong work ethic and the ability to take on difficult tasks yet complete them to absolute perfection.

My deepest thanks to Gary Blose, who in the past ten years has guided and helped me in so many areas. This is especially true with respect to the present thesis project, given his willingness to review my work, provide feedback, deal with my ups and downs, while patiently and strongly guiding me to overcome obstacles both large and small. Without

المنارة للاستشارات
www.manaraa.com

his help I would not have completed my graduate study.

I thank my colleague Audrey Schwartz, who covered many of my job duties while I worked on this project. Her friendly support and almost daily nagging to complete my thesis are much appreciated. Also thanks to my other SUNY colleagues who were like my extended family during my entire study at RPI.

Many thanks to my friends near by and far away: Kary and her family, Erica, Michelle, Ming Yu, Xuehong, Haihong, Alex, Katherine, Dawn, Leigh, Arpad, Chen, Guang, Deedi and many more, for their friendship across time and space. Larry & Meta, thank you for your hospitality in holidays and those books generously left on our doorstep. I wish Meta a speedy and total recovery.

Finally, thanks to my parents for their unconditional love and for their trust in me to study any field I like, and go wherever it takes me. Thanks to my brother and sister who have been taking care of my parents for over two decades while I'm far away. Finally, thanks to my darling son Edwin, who lets me believe in miracles. This dissertation is for him, and his generation. Explore new ways of learning!

المنارة للاستشارات
www.manaraa.com

ABSTRACT

Despite decades of research and development devoted to building systems for teaching computer programming to young students, and for enabling a computer to automatically create computer programs, we still face two fundamental challenges that have not yet been met:

- The Educational Challenge (**E**): How can we teach computer programming to humans (particularly young ones) in such a way that these humans thereby acquire the ability to solve complicated problems, not only in programming itself, but in other domains calling for problem-solving power?
- The Technological Challenge (**T**): How can we "teach" programming to computing machines, so that they have the ability to write computer programs on their own?

The defense focuses primarily on results regarding **E**.

There have been many attempts and approaches pursued in the attempt to meet these two challenges. In the case of such work aimed at meeting **E**, we can safely say that the vast majority of this work has been based on *constructivism*. Constructivism, which is in turn based on (what we call) *naive Piagetian theory*, holds that young students learn best in bottom-up fashion, by constructing small, simple programs, and working gradually toward deep understanding and deep problem-solving power, which they supposedly obtain as they gradually progress. *Cognitivism* (at least of the elementary sort), on the other hand, holds that human cognition is computation, where that computation is restricted to relatively simple computationally mechanisms and routines; and that pedagogy should teach students routines. This research has explored a framework beyond both constructivism and cognitivism: *formal meta-cognitivism*$^+$. Formal meta-cognitivism$^+$ is based on the belief that deep understanding at an abstract level completely independent of domain is the key to problem-solving, but the abstractions in question are built out of those available in the science of computation. Programmatically, formal meta-cognitivism$^+$ includes the view that the best way to teach computer programming is to teach facility at this abstract, logical level, and in turn teaching at that level means teaching formal logic, and the parts of computer science that relates directly to formal logic. The $^+$ is included because unlike cognitivism, our theoretical framework leaves open the possibility that computation

المنارة للاستشارات
www.manaraa.com

beyond standard, Turing-level computation is needed for suitably accurate abstractions and models.

Our main formal meta-cognitivist hypothesis is that dedicated training in abstract formal reasoning will help students solve complicated computer programming problems. An initial pilot study tended to support this hypothesis. To test the hypothesis further, we investigated, in three experiments, a set of factors (e.g., ability, educational background, logic training, programming experience) in order to gauge the impact of such factors on skill in computer programming. The results support our main hypothesis, as is explained and shown in the dissertation.

المنارة للاستشارات
www.manaraa.com

1. INTRODUCTION

Since the invention of the first programmable computer in 1930s and the birth of cognitive science in 1950s, there have been decades of research on theories and development of systems to teach humans and computing machines to program. For our discussion, we can distinguish programming into two types: what we term *simple programming* (**SP** or "coding") versus *original programming* (**OP** or "programming from scratch"). We can thus speak about programming$_s$ versus programming$_o$. In simple programming, one receives an explicit algorithms or pseudo-code as input, and produces code that when executed computes the algorithm using a programming language. **SP** is usually made easy by the following of a limited set of syntax and semantics of the programming language that sets the context of the task. Original programming, where one receives only an abstract, formal description of a function f in natural-language/formal-language content, and then must *create* solutions that include algorithms, and then produce a working computer program that computes the function, is clearly much more difficult. (We in fact point out in Chapter 4 that original programming may involve information-processing beyond the reach of a Turing machine.) On the educational side, our primary interest is: How can we better teach **OP**? **OP** is difficult for both people and computers, because our understanding of the learning mechanisms for programming in this deep way is severely limited.

With the number and variety of contemporary computing devices in the world today, and the explosion of Internet-based information systems reaching into almost every area of people's lives, the need for and the demand for computer programming has increased dramatically, and the demand will likely continue to increase, with no end in sight. This state-of-affairs gives rise to many challenges; in this study, we focus on two: an educational challenge, and a technological challenge. The bulk of our effort is targeted at the first of these two challenges.

The reader already knows, but it is important to affirm here nonetheless, that there already is a science of computer programming, and, at least by some metrics, it is quite mature. This science falls within computer science and overlaps significantly with logic and mathematics; it specifically involves the sub-fields of theoretical computer science, programming languages (or just 'PL,' the abbreviation used by those in the field), the study of algorithms, computability and complexity theory, and so on. However, there is no *cognitive* science of computer programming (= CSCP). At best, there are relevant

المنارة للاستشارات
www.manaraa.com

but exceedingly slim pockets of work (the psychology of computer programming special interest group, papers on the pedagogical effectiveness of Logo (which we shall canvass), etc.). The aim of this dissertation is to help establish the cognitive science of computer programming by systematically addressing the two challenges noted immediately above. These challenges would need to be addressed by any mature version of of the cognitive science of computer programming. It should be emphasized that this dissertation is intended to be only a *prolegomenon* to a mature CSCP. It should also be admitted out that the list of challenges from which the two addressed in this book are drawn would inevitably be a very large one. For example, here are some of the questions that should be answered in any mature CSCP:

- What are the basic mental operations in the processing of computer programming and problem solving?
- What kinds of education, experiences and strategies improve programming skills?
- What factors influence successful problem-solving, not only in programming, but beyond?
- How can we design better programming languages and environments?
- How can we build better automatic programming technology?

A truly mature CSCP would address *all* of these challenges, and more. The present project's final chapter, 6, includes in §6.3 a list of some of the culture-relevant challenges that would be on any "master" list of challenges for CSCP.

We now briefly set the two aforementioned challenges that constitute the foci for the dissertation.

1.1 Our Two Main Challenges

1.1.1 The Educational Challenge E

Given the foregoing, **E** is easy to state:

E How can we best bring about the cognitive skills that humans (particularly young ones) need to acquire in order to effectively program$_o$, where the given input function f is non-trivial and perhaps even quite complicated?

المنارة للاستشارات

www.manaraa.com

1.1.2 The Technological Challenge T:

It is equally straightforward, given the discussion above, to state the second challenge that motivates the search for a cognitive science of computer programming:

> **T** How can we create the algorithms that computing machines can use to program$_o$, where the input functions are, again, non-trivial and perhaps even quite complicated? And how can we build computing machines that not only generate computer programs that compute functions given as input, but also establish that the programs they produce are correct? That is, how can we create *self-verifying* automatic-programming programs?

There have been attempts to meet both challenges. In the case of **E**, which, again, is our main area of focus, there have been two two fundamentally different approaches taken.

1.2 Approaches to the Educational Challenge E

A number of different approaches have been taken in the attempt to meet these two challenges; but all this work has met with, at best, limited success. The approaches taken to **T** (along with the lack of progress made) are discussed in Chapter 4. But we provide here a quick, preview summary of the two main approaches taken to **E**. This pair is: *constructivism* and *cognitivism*.

1.2.1 Constructivism

Constructivism, which has its roots in the great psychologist Piaget, holds that young students learn best in bottom-up fashion, by constructing small, simple programs naÄśvely, and working gradually toward deep understanding and deep problem-solving power, which they supposedly obtain as they gradually progress through harder and harder problems. Unfortunately, studies have shown that students exposed to educational programming systems based directly on constructivism, such as Logo (Papert, 1980), did not experience cognitive transfer as a result: they do not develop problem-solving skill that can be applied in other domains (Kurland et al., 1987; Pea et al., 1987; Khasawneh, 2009), nor do they acquire skills that persist in the domain of computer programming itself.

Constructivism is not unfairly called "learning by doing." This approach holds that humans generate knowledge and meaning from an interaction between their concrete

المنارة للاستشارات
www.manaraa.com

attempts to build, and reaction to the results of these attempts. So, children in this approach supposedly best learn to program$_o$ by diving into examples, without first receiving training about the formal essence of programs and their formal idealizations (e.g., Turing machines), and without being given any background in even naive set and string theory to aid in achieving a deep understanding of programs. Often approach $\mathcal{C}$ is accompanied by a concerted effort to engage the student with entertaining elements (e.g., turtles and other cute animals) that has rather little to do with the formal essence of a program or the function that it computes.

1.2.2 Cognitivism

Cognitivism is distinguished by the view that human cognition is standard computation, and is associated with computational "architectures" of the mind, for example the prominent cognitive architecture known as 'ACT-R' (Anderson & Lebiere, 2003). (For an overview of computational cognitive modeling, see Sun (1999).) The teaching of computer programming in the cognitivist paradigm has almost single-handedly been led by John Anderson and colleagues (Anderson & Lebiere, 1998; Anderson, 1993; Anderson et al., 1995), and has focused on learning relatively simple routines for producing code that implements simple, given algorithms. An important part of the context for the research reported on in the present dissertation, is that the use of computer systems (intelligent tutoring systems, as they are called) to tutor humans in computer programming, based as they have been on simple computational cognitive models, has been limited to simple programming; that is, to programming$_s$.[1]

1.2.3 Our Approach: Formal Meta-Cognitivism$^+$

This research explores a framework beyond both constructivism and cognitivism: *formal meta-cognitivism*$^+$. Formal meta-cognitivism$^+$ is based on the belief that deep understanding at an abstract level completely independent of domain is the key to problem-solving, but the abstractions in question are built out of those available in the science of computation. Programmatically, formal meta-cognitivism$^+$ includes the view that the best way to teach computer programming is to teach facility at this abstract, logical level,

[1] An intelligent tutoring system (ITS) is a computer system that provides immediate and customized instruction or feedback to learners, usually without intervention from a human teacher. In the case of cognitivism-based work based on ACT-R, which has completely dominated ITS in the area of computer programming, all programming taught is decidedly of what we have called the $_s$ variety. An excellent, up-to-date overview of ITSs is provided by Bourdeau & Mizoguchi (2010, chapter 8), who discuss "cognitive" tutors based on ACT-R, confirms our diagnosis.

المنارة للاستشارات
www.manaraa.com

and in turn teaching at that level means teaching formal logic, and the parts of computer science that relate directly to formal logic (design and formal assessment of algorithms, e.g.). The $^+$ is included because unlikecognitivism, our theoretical framework leaves open the possibility that computation beyond standard, Turing-level computation is needed for suitably accurate abstractions and models. This possibility is discussed in Chapter 4.

In formal meta-cognitivism$^+$, the most efficacious learning is assumed to happen via deliberative reasoning, applied to progressively more difficult problems, until the learner reaches at least Stage IV in Piaget's 1958 continuum for cognitive development. At Stage IV and beyond the learner can reason at the level of full first-order logic (FOL), and can generate and assess hypotheses expressed in FOL. Formal meta-cognitivism$^+$ is based on the idea that the best way to learn to program$_o$ is to gradually come to understand essential, logico-mathematical, theoretical concepts, and on the idea that under the right conditions some humans can indeed reach Stage IV and beyond (Rinella et al., 2001; Bringsjord et al., 1998). Formal meta-cognitivism$^+$ also suggests that programming languages and environments should themselves be reflective of formal logic; this topic is dealt with in Chapter 5.

1.3 Hypotheses/Claims

1.3.1 Main Hypothesis H1 With Respect to E

Our main formal meta-cognitivist hypothesis, **H1**, is that dedicated training in abstract formal reasoning will help students solve complicated computer programming problems. An initial pilot study tended to support this hypothesis, and served to catalyze the present dissertation. To test the hypothesis further, we investigated, in three rather elaborate experiments, a set of factors (e.g., ability, educational background, logic training, programming experience) in order to gauge the impact of such factors on skill in computer programming. The results, as shall be seen in Chapter 2, support our main hypothesis. In short, it certain indeed seems that training in formal logic as an abstract framework for programming does facilitate programming skill. Therefore, such training helps to meet the educational Challenge **E**. We believe, accordingly, that formal meta-cognitivism as a pedagogical strategy and approach will allow us to meet **E**.

المنارة للاستشارات
www.manaraa.com

1.4 Approaches to the Technological Challenge T

Engineering approaches to meeting the challenge **T** that are connected to straight computation, without taking account of the kind of robust cognition that is posited as key by formal meta-cognitivism^{+} (e.g., evolutionary machine-learning approaches), it seems to us, will not allow us to solve $\mathcal{T}$. But here we do not yet have empirical results that bear out this position. Of course, it does seem quite reasonable that, on the other hand, if we can understand the nature of the deep reasoning and creativity in exceptional individuals who are adept at programming$_o$, which is what the kind of experiments reported on in Chapter 2 can in principle disclose, we believe we can profitably apply this understanding to the engineering of programs that meet challenge **T**. These matters are taken up in Chapter 4.

1.5 Computers and Computer Programming

It is beyond the scope of this dissertation to provide self-contained exposition of computers and computer programming, as they appear and have meaning in the modern world. This is true of necessity, since this world, an increasingly digital one, includes an ever-expanding (and rapidly, at that!) list of programming languages. In our experiments, as we make clear in Chapter 2, we allowed subjects to use, in their answers, pretty much any programming language they happened to be familiar and comfortable with — and indeed we even permitted subjects to simply use pseudo-code or flow charts, in order to allow them to describe algorithms in generic form. Accordingly, we do assume readers to have at least a rudimentary, general understanding of both computers and computer programs, in the form of some of some kind of generic, idealized hardware to play the role of 'computer,' and of the concept of some programming language or algorithm-specification language to play the role of 'computer programs.' Those few readers who are completely ignorant of these concepts are encouraged to consult two excellent textbooks that together provide everything needed, from the side of the logico-mathematics of computers and computer programming, to fully assimilate and appreciate the present dissertation: Lewis & Papadimitriou (1981) and Davis et al. (1994).

المنارة للاستشارات
www.manaraa.com

2. EXPERIMENTS

2.1 Purpose

The purpose of the experiments that form the heart of this dissertation are two-fold: One is to test the hypothesis that logic training increases the chance of successfully solving programming problems, especially difficult ones. The second is to investigate what educational and academic factors impact on the success of programming, in the hope of gaining some insight into meeting the Educational Challenge **E** and the Technological Challenge **T**.

2.1.1 Chief Hypothesis

Our chief hypothesis, recall, is:

H_1 In line with Meta-Cognitivism in the programming domain: Training in formal logic as an abstract framework for programming which facilitates programming skill. Therefore, such training helps to meet the educational Challenge **E**.

2.1.2 Other Factors

In addition to logic training, this study examined relationships between success in solving computer programming problems with other potential factors; for example, training and experience in programming languages (identified with number of dedicated programming-languages courses taken by subjects), nature of the first programming languages to which subjects were exposed, and, to a degree, learning methods (e.g., whether learning was by individual tutorials, or by online instruction, primarily self-study, etc.).

2.2 Methods

This research included three experiments, each with a different set of programming problems, but with the same set of educational and academic-background questions.

2.2.1 Programming Problems

There were three sets of programming problems presented to three different group of subjects. The first set of problems were composed of two language-oriented problems

المنارة للاستشارات
www.manaraa.com

(Appendix A). The second set were composed of three questions related to digital-circuit analysis (Appendix B); and the third set was made up of two target-defense problems (see Appendix C). The variety of problems was felt to be an important measure to take in order to reduce the possibility of bias created by subject familiarity with a particular kind of programming challenge.

2.2.2 Background Questions and Variables

While, as noted, there were different programming problems for different projects, all three experiments (except Experiment 1C, which had programming problems only) shared the same set of background information questions. (Details see D) As a result, the experiments shared the same type of variables:

- Dependent Variable S: Successful outcome of problem solving.
- Education variables:
  - $E1$: Had logic training, defined as taking one or more logic courses;
  - $E2$: Number of logic courses;
  - $E3$: Had Programming Language (PL) Courses; and
  - $E4$: Number of PL Courses.
- Other Variables: Academic/Demographic Background
  - $V1$: Academic Major;
  - $V2$: SAT, and equivalent (ACT Scores);
  - $V3$: Gender; and
  - $V4$: Race/Ethnicity.

2.2.3 Procedure

Subjects first answered a set of questions related to their academic background, logic education, and programming-language education and experiences. Then they were given two to three programming problems. After attempting to solve the problems, they answered a set of questions related to their perception of the difficulty of the problems they had tackled.

المنارة للاستشارات
www.manaraa.com

2.2.4 Test of Chief Hypothesis (H1) Using χ^2 Statistics

To test the hypothesis that logic training improves the probability of solving difficult programming problems, and to examine the relationship between programming-language training, programming experience, learning style, and other education-background factors with the outcome of programming problem solving, we employ χ^2 test.

The χ^2 test provides a method for testing if the relationship between categorical variables differs significantly from chance, and which categories account for any differences (e.g., see Pearson, 1900, 1904; Elderton, 1902; Fisher, 1922; Yates, 1934).

2.2.4.1 Data Analysis Methods: χ^2 Test

Let us examine the relationship between two variables: A and B, where each has multiple mutual exclusive levels of values

$$A = (A_1, A_2, ..., A_m)$$

$$B = (B_1, B_2, ..., B_n)$$

Null Hypothesis H_0: A and B are independent. That is, $P(AB) = P(A)P(B)$.

Alternative Hypothesis H_1: A and B are not independent.That is: knowing the value of A can help to predict the value of B.

Pearson introduced χ^2 statistics.

$$\chi^2 = \sum \frac{(Observed - Expected)^2}{Expected} = \sum_j \sum_i \frac{(F_O(i,j) - F_E(i,j))^2}{F_E(i,j)}$$

where $F_O(i,j)$is the observed frequency count for $A = A_i$ and $B = B_j$; $F_E(i,j)$, the expected frequency count for $A = A_i$ and $B = B_j$.

Expected Frequency is the count of an event that happens if variables A and B are unrelated, while taking consideration of the observed results. Assume there are total N independent samples; n_i is the total number of sample observations when for $A = A(i)$. n_j is the total number of $B = B_j$. Then the expected frequency of $A = A_i$ and $B = B_j$ can be computed by

$$F_E(i,j) = \frac{n_i \times n_j}{N}$$

المنارة للاستشارات
www.manaraa.com

Degrees of Freedom are related to the number of levels of variables.

$$k = (m - 1) \times (n - 1)$$

p-value shows the possibility the observed sample data could occur by chance (i.e. null hypothesis is true), associated with a given statistical distribution (e.g. χ^2) and k degrees of freedom.

Level of Significance (α) is associated with level of confidence and indicates the threshold value of p-values. It indicates how extreme observed results must be to reject the null hypothesis.

This study selects 95% level of confidence as the threshold $\alpha = 1 - .95 = .05$. If $p \leq \alpha$, we reject the null hypothesis. (The critical value corresponds to $\alpha = 0.05$.)

Cramér's V measures the strength of association between the two variables of interest. It has values that range between 0 and 1.

$$V = \sqrt{\frac{\chi^2/N}{min(m-1, n-1)}}$$

Where N is the number of observations, m is the number of columns, and n is the number of rows. V is between 0 and 1, with 0 as no association between A and B, and 1 as $A = B$. For a 2×2 contingency table, $m = n = 2$.

$$V = \frac{\chi^2}{N} = \phi$$

, which also called ϕ coefficient.

Residual in a contingency table is the difference between observed and expected values.

$$R(i, j) = F_O(i, j) - F_E(i, j)$$

Standard Residual indicates the importance of the cell to the overall χ^2 , similar to Z-score, showing standard deviations between observed and expected values.

$$R_{Std}(i, j) = \frac{F_O(i, j) F_E(i, j)}{\sqrt{F_E(i, j)}}$$

المنارة للاستشارات

www.manaraa.com

Table 2.1: ***A sample of*** 2×2 ***contingency table***

	A_1	A_2	Total
B_1	$f(1,1)$	$f(1,2)$	r_1
B_2	$f(2,1)$	$f(2,2)$	r_2
Total	c_1	c_2	N

Adjusted Residual adjusts the residual with overall sample size.

$$R_{Adj}(i,j) = \frac{F_O(i,j)F_E(i,j)}{\sqrt{F_E(i,j) \times n_i \times n_j}}$$

where n_i and n_j are the row and column total proportions, respectively.

2.2.4.2 Contingency Tables

A contingency table, also called cross tabulation or cross tab, was introduced by Pearson in 1900, and represents a matrix which displays the frequency distribution of the variables of interest (Pearson, 1900, 1904). For example, Table 2.1 is a simple 2×2 contingency table, with a sample size (e.g., number of observations) as N, for two levels of attributes of two variables $A(A_1, A_2)$ and $B(B_1, B_2)$.

The data in the cells of the contingency table are frequencies or counts of the occurrence:

$f(i,j)$ is the frequency (count) of event $A = A_j$ and $B = B_i$. $r_i = f(i,1) + f(i,2)$, the marginal total of row i, is the frequency of the event $B = B_i$. $c_j = f(1,j) + f(2,j)$, the marginal total of column j, is the frequency of the event $A = A_j$.

Given the observed events $f_O(i,j)$, the expected values can be computed. In the example in Table 2.1, the expected frequency of $A = A_1$, and $B = B_1$ can be computed by

$$f_E(1,1) = \frac{r_1 \times c_1}{N}$$

where N is the total sample size $N = r_1 + r_2 + c_1 + c_2$.

The χ^2 value of a cell is as follows:

$$\chi^2 = \frac{(F_O - F_E)^2}{F_E}$$

المنارة للاستشارات
www.manaraa.com

The Xi^2 test statistic is computed by

$$X^2 = \sum \frac{(F_O - F_E)^2}{F_E}$$

Then the Null Hypothesis (H_0) states that $A$and B are independent: Knowing the value of A does not help predict the outcome of B.

The Alternative Hypothesis (H_1): A and B are associated: Knowing the value of A can help to predict the outcome of B.

2.2.4.3 χ^2 Test Assumptions

There are certain restrictions for using χ^2 test of hypotheses. The main assumptions are given below; experiments in this study satisfy them. Thus, most of our data analysis will employ this method.

- Variables should be ordinal/nominal level.
- Variables should consist of two or more categorical, independent levels. All levels of a variable are mutual exclusive.
- In a contingency table, 80% of cells' values of expected counts should be at least 5. Otherwise there are other tests of significance, such as Fisher's exact test and maximum likelihood test (McHugh, 2013).

2.3 Experiment 1: Language Problems

Experiment 1 is to test the feasibility of the study. To determine if our selection of programming problems are in the right range, in that most of the people will have difficulty solving them, but at the same time not too difficult for people to solve them in a reasonable amount of time. We also wished to determine if our experimental questions and programming problems would be better done online or on-site.

2.3.1 Programming Challenges

Description

Two programming problems are given below, and involve working with a very simple language, $\mathcal{L}$, a fragment of English. The words used in $\mathcal{L}$ are:

{Bill, Jane, likes, chases, makes, a, the, man, woman, cat, happy, thin, quickly}

المنارة للاستشارات
www.manaraa.com

Bill, Jane, man, woman, and cat, are nouns; happy and thin are adjectives; likes, chases, and makes are verbs; a and the are determiners; and quickly is an adverb.

The following grammar defines the sentences of $\mathcal{L}$.

S	::=	NOUN VERB NOUN	R1
	|	DET NOUN VERB NOUN	R2
	|	DET ADJ* NOUN VERB DET ADJ* NOUN	R3
	|	DET ADJ* NOUN ADV VERB DET ADJ* NOUN	R4
	|	DET ADJ* NOUN VERB DET ADJ* NOUN ADV	R5

where NOUN stands for any noun, VERB stands for any verb, DET stands for any determiner, ADV stands for any adverb, ADJ stands for any adjective, ADJ* stands for zero, one, or more adjectives, and S stands for a well-formed sentence of $\mathcal{L}$.

For instance, the sequence ⟨the, thin, cat, makes, a, Bill⟩ is a sentence of $\mathcal{L}$, because cat and Bill are nouns, likes is a verb, the and a are determiners, and thin is an adjective; so the sequence has the form DET ADV* NOUN VERB DET ADJ* NOUN, which, by rule R3, is a sentence of $\mathcal{L}$. (Notice that the first ADJ* is matched with the one adjective thin, while the second ADJ* is matched with the lack of adjectives between a and Bill.)

In each of the two problems you will be asked to write a program. You may use one of the following programming languages: BASIC; C; C++; Java; Lisp; Pascal; or you may use pseudo-code to describe your program in detail. Please indicate whether you are using pseudo-code or a programming language to solve the problem, and, if using a programming language, which programming language you are using.

Problem 1

Write a program P that takes as input a (finite) sequence of words used in $\mathcal{L}$ and outputs **yes** if the sequence is a sentence of $\mathcal{L}$, and outputs **no** otherwise. For example, given the sequence ⟨Bill, likes, Jane⟩, P should output **yes** because the sequence is a sentence, according to R1. When given ⟨Bill, Jane, likes⟩, P should output **no**, because this sequence is not a sentence of $\mathcal{L}$.

Problem 2

Write a program P that takes as input a (finite) sequence of words used in $\mathcal{L}$ and outputs **yes** if the sequences is a palindrome sentence of $\mathcal{L}$, and outputs **no** otherwise. A palindrome sentence is a sentence which reads the same in both directions. For example,

المنارة للاستشارات
www.manaraa.com

Table 2.2: ***Grading scale. Succeeded: Scores*** ≥ 4***; Failed: Score*** ≤ 3

Score	Success	Description
5	1	A complete, correct and clear program/algorithm.
4	1	A complete, mostly correct algorithm/program.
3	0	An incomplete program, with some correct steps in the right direction.
2	0	A few basic steps, but incomplete/unclear.
1	0	Wrote little or some unrelated notes.
0	0	Blank

given the sequence $\langle \mathsf{Bill}, \mathsf{likes}, \mathsf{Bill} \rangle$, P should output **yes**, since the sequence is a palindrome sentence. When given $\langle \mathsf{the}, \mathsf{cat}, \mathsf{likes}, \mathsf{Jane} \rangle$, P should output **no**, since the sequence, although a sentence, is not a palindrome sentence.

2.3.2 Experiment 1A: Online

2.3.2.1 Subjects

Subjects were college students who answered a post on Experimetrix, an online experiment scheduling tool used at Rensselaer.[2] Most of these subjects were taking an introductory course, such as statistical methods, in Rensselaer's Department of Cognitive Science. Full IRB approval was obtained for these and all other subjects used in the experiments.

2.3.2.2 Procedure

Experiment 1A was conducted online, via an online survey platform called Opinio,[3] where subjects signed in at their own time and place (ObjectPlanet, 2014). The subjects first answered a set of background questions, then they were presented with two programming problems.

Subjects initially answered a set of questions related to their demographics and academic background, such as major, undergraduate grade point average (GPA), logic education, programming-language education and experiences. Then they were presented with two challenging programming problems. Subjects were permitted to answer them directly in the text box below each question, or to copy/paste from another editor. After that, they were asked to rate the problems for perceived difficulty. At any time, the subject can restart from the beginning, and/or move back to a prior screen. But they needed to answer all required questions in order to move forward on a screen.

[2]Experimetrix (2006) http://www.experimetrix.com/index.htm (Date Last Accessed October 10, 2014).
[3]Opinio (2014) http://www.objectplanet.com/opinio/ (Date Last Accessed October 10, 2014).

المنارة للاستشارات
www.manaraa.com

2.3.2.3 Results

There were 183 subjects who took part in the online Experiment 1A. Of the 183, 171 of them were valid. One of the background questions asked subjects to check a specific selection. If they selected anything else, their results were considered invalid, because most likely those subjects were not paying attention to what they read. All of the experiments included this test question, and the data analysis included only those with valid answers.

The rate of success were low: 13 out of 171 subjects, 7.6% succeeded in solving Problem 1 ($P1$); while 92% failed. 9 out of 171, 5.3% of subjects solved Problem 2 ($P2$). Overall, there only 4.1% of subjects succeeded in solving both problems.

Although it is easy to get a large number of subjects online, with the flexibility of time and location, the quality of effort among such subjects looked low. Hence, in Experiment 1B, subjects were in a classroom and answered questions and attempted the programming problems on paper.

2.3.3 Experiment 1B: On-Site

Experiment 1B used the same programming problems as those in Experiment 1A, but the experiment was conducted on-site in a classroom setting.

2.3.3.1 Subjects

Subjects were 18 volunteers signed up by the same method/system: Experimetrix. The same sample pool was here as Experiment 1A, but different individuals.

2.3.3.2 Procedure

After sign up, subjects came to campus in a classroom, answered background questions, and solved the problems on paper.

2.3.3.3 Results

We used the same grading scale as in Experiment 1A. This time the attempted (those who scored 2 and over) rates were much higher. All 18 records were valid. All 100% attempted to solve problem 1, and 17 out of 18 (94%) subjects attempted to solve Problem 2. 88.9% had scores 2 or higher. The success rate were higher than those in Experiment 1A. Eight of out 10, 44% solved Problem 1. But still there were only two out 18, 11.1% who solved Problem 2.

المنارة للاستشارات
www.manaraa.com

Comparing Experiments 1A and 1B, for subjects who solved both problems, using a χ^2 test contingency table of success rate in online group and on-site group, we report that the on-site group performed slightly better, but the differences were not statistically significant: $p = .184 > 0.05(\alpha)$.

Subjects put much more effort into solving the problems in an on-site setting than online, located remotely. The on-site generated a higher attempted rate, that is a higher rate of subjects attempted to solve the problems, instead of just by-passing them quickly as in the online case. The differences were significant ($p < 0.05$).

Several reasons could explain why there was no difference in success rate between the two settings. One is that survey mode does not play a role in a subject's success in problem-solving (of this and similar type). Some other factors need to be considered: The lack of a relationship between logic training and performance could indicate that there are too few people who had logic training. Out of the 189 subjects included in Experiments 1A and 1B, only nine subjects had logic training. Sample size was too small to warrant a conclusion.

On-site students expended much more effort than those online. But the general performance on the programming problems was not significantly different from those in Experiment 1A. One of the possible reasons for this: The randomly signed up subjects were mostly students who were taking an entry-level psychology class, so many may have lacked either logic or programming-language training, or both. To examine this, Experiment 1C was piloted.

2.3.4 Experiment 1C: With Logic Training

2.3.4.1 Subjects

Subjects were students who were taking a logic class, and volunteered to attempt to solve the same two challenge problems.

2.3.4.2 Procedure

For a group of students who were taking an introductory logic course near the end of the term, we presented them with the same two problems as in Experiments 1A and 1B. At their choice, they were asked to solve the two programming problems, and small monetary awards were given to those who solved the problems correctly. No background questions were asked.

المنارة للاستشارات
www.manaraa.com

Table 2.3: ***Experiment 1 Result: Success (S) and Logic Training (E1)***

			E1		Total
			0 No	1 Yes	
Success	0 Failed	Observed	166	44	210
		Expected	159.6	50.4	210.0
	1 Succeeded	Observed	8	11	19
		Expected	14.4	4.6	19.0
Total		Observed	174	55	229
		Expected	174.0	55.0	229.0

Note: $\chi^2(1) = 13.0, p < 0.001.$

2.3.4.3 Results

Out of 40 students, eight solved the problems; the success rate was 20%. Combining Experiment 1A, 1B, and 1C, we note that those who had logic training had significantly higher chances in solving the problems successfully (Table 2.3).

From the results in Experiment 1, the performance of students who took logic training (Experiment 1C) had a much higher rate of success. But one can argue or speculate that a reason for this was because the stimulus problem perhaps was similar to problems or puzzles the logic class might have seen during lectures. (There was no evidence of this at all. But it is a conceptual possibility.)

Experiment 2 used programming problems related to analysis and design of digital electronic circuits, which were described more in graphic-based and diagrammatic than in linguistic terms.

2.4 Experiment 2: Digital Circuits Problems

2.4.1 Subjects

Subjects here were students enrolled in an intermediate logic class. With the advantages of on-site setting and small monetary awards, the chances that a subject would try harder to solve the problems were raised. The next two experiments were be conducted on-site in a classroom setting with small payments.

2.4.2 Programming Challenges

One of the concerns of Experiment 1 was that the programming questions were presented in a mostly linguistic manner, which could conceivably be biased in favor of

المنارة للاستشارات
www.manaraa.com

subjects who had facility with language. Experiment 2 presented subjects with mostly graphic logic gates and digital-circuits diagram problems (for details, see Appendix B).

2.4.3 Results

The rate of success was relatively high, 11 out of 13; 84.6% solved Problem 1. Four out of 13, 30.8%, solved Problem 2, and three out of 13, about 23.1%, solved Problem 3.

But many subjects reported difficulty in understanding some of the questions because some of the descriptions were long and the circuit concepts were difficult to follow for students without an electrical-engineering background.

Further, both Experiment 1 and 2 had only two questions, so the success in Experiment 2 was be defined as subjects who solved at least two questions, at the combined data sets analysis below.

Since all subjects in Experiment 2 had taken a logic class, it is not meaningful to see if logic training played any role in the success rate using data from this experiment alone. These data increased the number of subjects with logic training, as well as other types of background information. Further analysis of these data in combination with data from the other experiments will be included in an overall data-analysis section following discussion of Experiment 3.

2.5 Experiment 3: Target Defense Problems

The relatively low problem success rate could be due to the fact that most of the relevant subjects did not have proper computer-programming training. The intent of Experiment 3 was to obtain a large sample of subjects with computer-language training and present them with programming problems that do not require the reading of a lengthy description or require prior knowledge of language forms or circuits.

2.5.1 Subjects

Experiment 2 had a limited sample size and all subjects had logic training. Experiment 3's subjects were from volunteers who had taken either of two advanced computer courses (Modeling and Algorithms) in Rensselaer's Department of Computer Science, which of course increased the chances of those who had computer-programming training and experience. Furthermore, these data will be combined with data sets from different groups of subjects.

المنارة للاستشارات
www.manaraa.com

2.5.2 Programming Challenges

Experiment 3 presented two problems from a computer textbook (Kleinberg & Tardos, 2005). These problems did not require prior knowledge of standard language forms, digital-circuits terms, or the need to read a long description to understand the concepts (details are found in Appendix B).

2.5.3 Results

There were 77 valid subject records. 35.1% succeeded in solving Problem 1, 18.2% solved Problem 2, and 12% were successful in solving both of the problems. Further results and discussions will be presented in the next section.

2.6 Comprehensive Data Analysis of Combined Experimental Results

The results presented below were from data in all three experiments except a portion of the records from Experiment 1C, in which — as noted above — the subjects did not answer background questions. The total number of valid records were 279 different subjects.

2.6.1 Success and Educational Factors

We first examine four educational variables, each with a derived variable:

- $E1$: Had logic training, as defined as taking one or more logic courses ($E1'$);
- $E2$: Number of logic courses ($E2'$);
- $E3$: Had Programming Language (PL) Courses ($E3'$); and
- $E4$: Number of PL Courses ($E4'$).

where $E1, E2, E3$ and $E4$ were direct input from subjects; and $E1'$ and $E3'$ were derived from subject's list of course names, combined with published course descriptions. $E2'$, number of logic courses, and $E4'$, number of PL courses, were derived from values of $E1'$ and $E3'$. Although many of the derived and self-reported variables were the same, for some records the derived values were different from the self-reported. For example, some subjects listed "Database Systems" as one of the PL courses, when PL was not part of the course curriculum.

المنارة للاستشارات
www.manaraa.com

Table 2.4: **Overall Results: Success (S) and Logic Training (E1)**

			E1 - Logic Training		Total
			0 No	1 Yes	
Success	0 Failed	Observed	177	72	249
		Expected	171.1	77.9	249.0
	1 Succeeded	Observed	14	15	29
		Expected	19.9	9.1	29.0
Total		Observed	191	87	278
		Expected	191.0	87.0	278.0

Note: $\chi^2(1) = 6.3, p = 0.012.$

Table 2.5: **Success (S) and Logic Training (E1' - Derived)**

			E1' - Logic Training (Derived)		Total
			0 No	1 Yes	
Success	0 Failed	Observed	221	29	250
		Expected	214.2	35.8	250.0
	1 Succeeded	Observed	18	11	29
		Expected	24.8	4.2	29.0
Total		Observed	239	40	279
		Expected	239.0	40.0	279.0

Note: $\chi^2(1) = 14.7, p < 0.000.$

2.6.1.1 Logic Training

Table 2.4 shows a contingency table of outcomes of problem solving (S) and the status of logic training ($E1$). When there was logic training $E1 = 1$, yes), the observed frequencies (15) of success were significantly higher than the expected (9.1), with $\chi^2(1) = 6.4, p = 0.012$.

Similarly, with respect to the derived logic training variable ($E1'$) in Table 2.5, the observed number (11) of success was significantly higher than expected (4.2), $\chi^2(1) = 14.7, p < 0.000$.

Another way to look at the data is as shown in Figure 2.1. Comparing those without logic training ($E1 = 0$, No) to those who had logic training ($E1 = 1$, Yes), the success rate increased from 6.6% to 22.9%. In Figure 2.1, for derived variable $E1'$, the success improved from 7.5% to 26.3%.

المنارة للاستشارات
www.manaraa.com

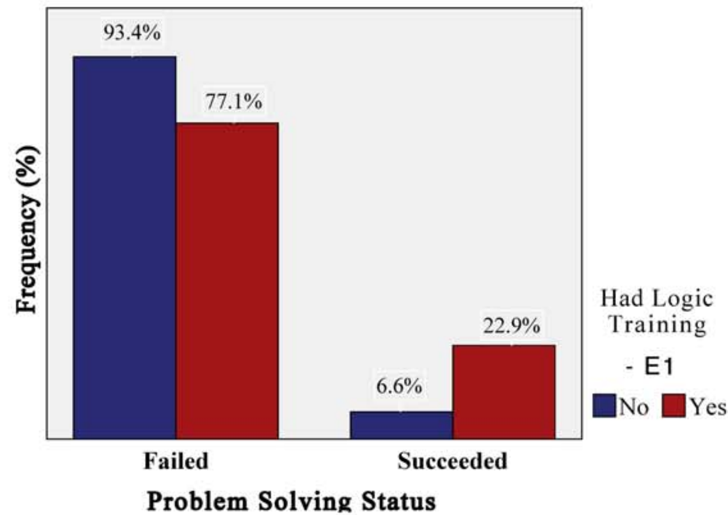


Figure 2.1: Success Proportion (%) and Logic Training (E1).

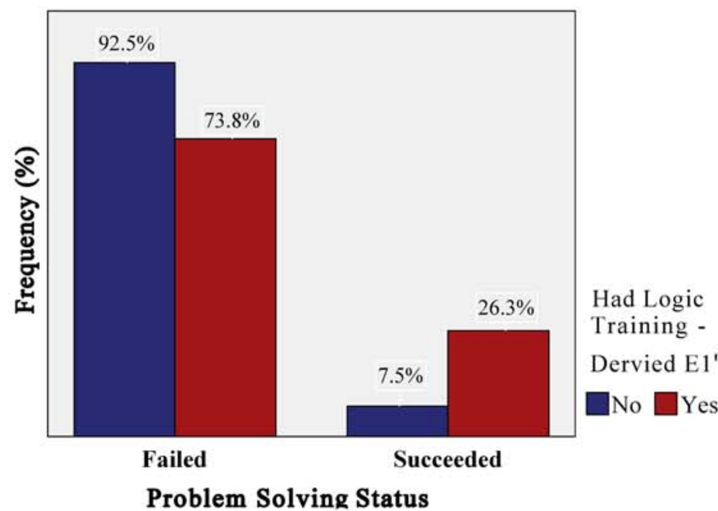


Figure 2.2: Success Proportion (%) and Logic Training (E1' - Derived).

2.6.1.2 Number of Logic Courses

To further check the relationships between success and the number of logic courses a subject took, Table 2.6 shows the results of success and number of logic courses: For those without logic training, subject success rates were below expected, with adjusted residual ($R_{adj} = -2.8$) indicating the differences were not due to chance; for subjects with one logic course, the number of successes increased, to about the expected level ($R_{adj} = 0$); for those taking two or more logic courses, the number of successes increased significantly higher than expected ($R_{adj} = 3.9$).

المنارة للاستشارات
www.manaraa.com

Table 2.6: ***Success (S) and Number of Logic Courses (E2)***

			E2 - Number of Logic Courses			Total
			0 None	1 One	2 Two or More	
Success	0 Failed	Observed	183	43	24	250
		Expected	176.5	43.0	30.5	250.0
		Adjusted Residual	2.8	.0	-3.9	
	1 Succeeded	Observed	14	5	10	29
		Expected	20.5	5.0	3.5	29.0
		Adjusted Residual	-2.8	.0	3.9	
Total		Observed	197	48	34	279
		Expected	197.0	48.0	34.0	279.0

Note: $\chi^2(2) = 15.5, p < 0.001.$

Table 2.7: ***Success (S) and Number of Logic Courses (E2' - Derived)***

			E2' - Number of Logic Courses (Derived)		
			0 None	1 One	2 Two or More
Success	0 Failed	Oberved	221	21	8
		Expected	214.2	20.6	15.2
		Adjusted Residual	3.8	.3	-5.9
	1 Succeeded	Oberved	18	2	9
		Expected	24.8	2.4	1.8
		Adjusted Residual	-3.8	-.3	5.9
Total		Oberved	239	23	17
		Expected	239.0	23.0	17.0

Note: $\chi^2(2) = 35.2, p < 0.001.$

Comparing subjects without logic training to those took two logic courses, we see that the results continue to be in line with formal meta-cognitivism$^+$: The ratio between observed and expected number of success increased from 0.67 (14/20.5), to 2.56 (=10/3.5). That is, taking two or more logic courses, subjects were about 3.7 times more likely to succeed in solving the problems. The overall success and number of logic courses had $\chi^2(1) = 15.5, p < 0.000$.

The derived number of logic courses, $E2'$ (as in Table 2.7, had the same trends: Two or more logic courses improved the chance of success. Derived $E1'$ showed slightly stronger relationship with outcomes than self-reported $E1$ (Cramér's $V = .355$ vs $V = .24$).

2.6.1.3 Programming-Language Training

From the sampling data, self-reported PL training ($E3$) had significantly higher observed level of success than expected (Table 2.8 and Figure 2.3): $\chi^2(1) = 11.7$ and

المنارة للاستشارات
www.manaraa.com

Table 2.8: **Success (S) and Programming Language PL Training (E3)**

			E3 - PL Training		Total
			No	Yes	
Success	0 Failed	Observed	108	142	250
		Expected	99.5	150.5	250.0
	1 Succeeded	Observed	3	26	29
		Expected	11.5	17.5	29.0
Total		Observed	111	168	279
		Expected	111.0	168.0	279.0

Note: $\chi^2(1) = 11.7, p = 0.001.$

Table 2.9: **Success (S) and PL Training (E3' - Derived)**

			E3' - PL Training (Derived)		Total
			0 No	1 Yes	
Success	0 Failed	Observed	180	70	250
		Expected	176.5	73.5	250.0
	1 Succeeded	Observed	17	12	29
		Expected	20.5	8.5	29.0
Total		Observed	197	82	279
		Expected	197.0	82.0	279.0

Note: $\chi^2(1) = 2.2, p = 0.13.$

$p = 0.001$. But the impact of the derived PL training $E3'$ was not significant: within the range of errors ($\chi^2(1) = 2.2, p = 0.13$; see Table 2.9 and Figure 2.4).

2.6.1.4 Amount of Programming Language Training

The pattern of impact of $E4$ and $E4'$ of success are similar to those of $E3$ and $E3'$. The impact of self-reported $E4$ improved success significantly ($\chi^2(1) = 13.2, p = 0.001$), while the improvement of $E4'$ was within range of chance $\chi^2(1) = 2.9, p = 0.235.$, even for the cases where number of PL course were two or more: Adjusted residual $R_{adj} = 1.5$ less than 1.96 to be counted significant.

2.6.2 Success and Academic/Demographic Factors

2.6.2.1 Academic Major

Although overall academic major had a role in outcome of success, $\chi^2(4) = 18.7, p = 0.001$, in detailed data for adjusted residuals of each major, not every one had significant impact. Math as a major had the best positive results, while the difference of most of other majors were within the range of chance. Meanwhile, math as major had the highest number of subjects with logic training. 66.7% reported ($E1$), while all others less than

المنارة للاستشارات
www.manaraa.com

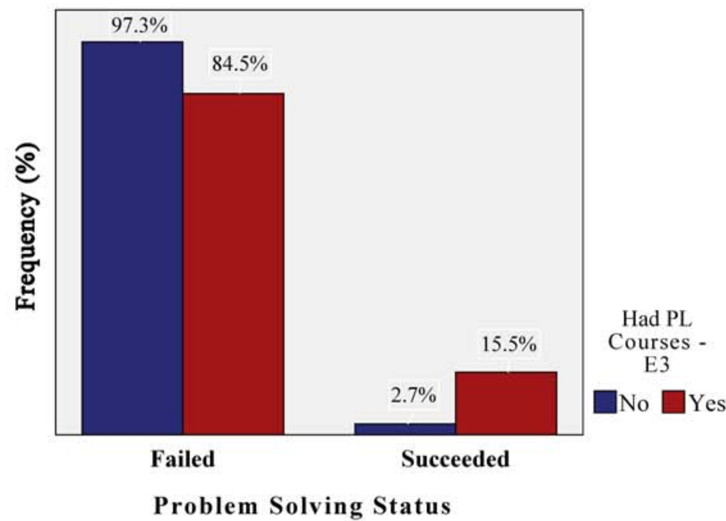


Figure 2.3: Success (%) and PL Training (E3). $\chi^2(1) = 11.7, p = 0.001$.

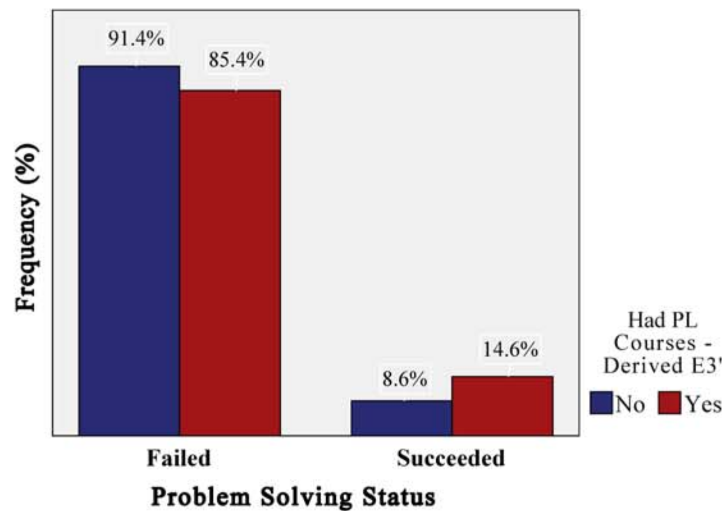


Figure 2.4: Success (%) and PL Training (E3' - Derived). $\chi^2(1) = 2.24, p = .13$.

50%, and overall average 31,3%. In the derived factor $E1'$, 46.7% of math majors had logic training; the second-highest major, computer science, had 21.5%, and overall 14.3% had logic training.

2.6.2.2 SAT/ACT Scores

ACT scores were converted to SAT; the combined SAT/ACT score was divided into three groups: a) no SAT and ACT score reported; b) score range less than 1400; and the third group of scores 1400 or higher.

المنارة للاستشارات
www.manaraa.com

Table 2.10: **Success (S) and Number of PL Courses (E4)**

			E4 - Number of PL Courses			Total
			None	One	Two or More	
Success	0 Failed	Observed	113	42	95	250
		Expected	103.9	45.7	100.4	250.0
		Adjusted Residual	3.6	-1.9	-2.1	
	1 Succeeded	Observed	3	9	17	29
		Expected	12.1	5.3	11.6	29.0
		Adjusted Residual	-3.6	1.9	2.1	
Total		Observed	116	51	112	279
		Expected	116.0	51.0	112.0	279.0

Note: $\chi^2(1) = 13.2, p = 0.001.$

Table 2.11: **Success (S) and Number of PL Courses (E4' - Derived)**

			E4' - Number of PL Courses (Derived)			Total
			0 None	1 One	2 Two or More	
Success	0 Failed	Observed	180	48	22	250
		Expected	176.5	49.3	24.2	250.0
		Adjusted Residual	1.5	-.6	-1.5	
	1 Succeeded	Observed	17	7	5	29
		Expected	20.5	5.7	2.8	29.0
		Adjusted Residual	-1.5	.6	1.5	
Total		Observed	197	55	27	279
		Expected	197.0	55.0	27.0	279.0

Note: $\chi^2(1) = 2.9, p = 0.235.$

As shown in Table 2.12, the impact of the outcome with SAT/ACT scores below 1400 did not turn out to be significant, while the impact of 1400 or over had strong relationship to success in solving the relevant problems.

2.6.2.3 Gender and Race/Ethnicity

The differences among the gender and race categories were not statistically significant, so it is unlikely there is a difference in their problem-solving outcomes. The tests for gender: $\chi^2(2) = 4.1(8unknowns), p = 0.13.$; and race: $\chi^2(5) = 3.8, p = 0.58$. Both p-values were greater than $\alpha = 0.05$. (Degrees of freedom of gender were 2 because of eight unknown records.)

المنارة للاستشارات
www.manaraa.com

Table 2.12: **Overall Result: Success (S) and SAT/ACT Scores (V2)**

			V2 - SAT_ACT Scores			
			0 Unknown	1 Less than 1400	2 1400 or over	Tot
Success	0 Failed	Observed	172	48	30	25
		Expected	161.3	47.5	41.2	250.
		Adjusted Residual	4.4	.3	-5.9	
	1 Succeeded	Observed	8	5	16	2
		Expected	18.7	5.5	4.8	29.
		Adjusted Residual	-4.4	-.3	5.9	
Total		Observed	180	53	46	27
		Expected	180.0	53.0	46.0	279.

Note: $\chi^2(2) = 36.3, p < 0.001$.

2.7 Discussion

2.7.1 Summary and Conclusion

In conclusion, the overall results dramatically in line with formal meta-cognitivism$^+$: logic training on abstract reasoning certainly seems to facilitate the learning of programming. For the four educational factors, our research shows that having logic training improves the chances of solving difficult programming problems ($E1/E1'$). Increasing the number of logic courses, in turn, significantly increases the chance of being successful in solving programming problems ($E2/E2'$). Having programming-language (PL) training also improves the chance of solving programing problems ($E3$). This could be due to logic training components that were part of reported PL course curriculum ($E3'$). On the other hand, increasing the number of PL courses taken improved the chance of success somewhat, but within the range of sampling error ($E4/E4'$).

When considering other academic/demographic factors, some contributed significantly to problem-solving success: academic major, specifically being a math major, improved success (probably because math majors had the most training in logic); and SAT/ACT scores higher than 1400 correlated with success as well. Other demographics such as gender and race showed no statistically significant differences in performance outcomes.

Based on these results, computer programming is a difficult task for most of the college students, even those whose major was computer science. One of the ways to improve programming ability would be to strengthen the logic eduction requirements. This of course speaks directly and unmistakably to the the challenge **E**.

المنارة للاستشارات
www.manaraa.com

2.7.2 Limitations and Future Work

These results are encouraging and promising, but not conclusive. A χ^2 test can provide evidence for association or independence between variables, but, as is well-known, it does not prove *causal* relationships. Ideally, one could assign weights to each variable representing the contributions of that variable to the success, but unfortunately this is impossible because the variables used in the study are not orthogonal, and the relationships may not be linear. Although initial logistic-regression analysis results did not show a clear pattern, further study of possible relationships could be interesting and useful.

In the future, we can study the types of individuals who performed well and develop ways to record and observe their problem-solving processes. Along with their educational background, we should try to determine what specific methods or strategies they used during the problem-solving process, such as whether they used diagrams, or top down vs. bottom up mental strategies, or similar cognitive techniques to improve their success in solving difficult problems. This could possibly provide insight into machine learning of automatic programming. As a matter of fact, a surprising number of those subjects who met with success did make use of visual of diagrammatic constructions, and we do speculate about this promising direction for investigation in Chapter 4.

المنارة للاستشارات
www.manaraa.com

3. ASSESSMENT OF EDUCATIONAL PROGRAMMING ENVIRONMENTS

3.1 History and Overview of Educational Programming Languages and Environments

Programming languages are of course often used to accomplish meaningful things in the "real world." For example, NASA produces many programs in order to make its projects possible: The much-discussed future mission to Mars, for example, will be possible only if many sophisticated computer programs work their magic. However, a programming language can be designed strictly for, or at least primarily for, educational purposes. And this is the kind of programming language that relates to the educational challenge, **E**, we are aiming to solve. Indeed, in Chapter 5, we describe the first version of our own invention in this space. But before getting to that, it is important to take stock of prior work in this space; hence the present chapter.

Starting with low-level programming languages, there have been numerous educational programming languages, usually only simplified instruction codes, designed to enable programmers to learn the basic architectures of processors and main functions of programming. One such example is Little Computer 3 (LC-3), an assembler-like language developed by Patt & Patel (2004). For high-level programming languages, Logo, modeled after Lisp, is by far the dominant representative of educational programming languages/environments. Because Logo's influence is, frankly, towering, and many other educational programming languages and environments are clearly derivative of Logic, and because Logo is deeply and directly rooted in constructivism, we will examine Logo in detail in the present chapter.

3.1.1 Logo

3.1.1.1 What is Logo?

Logo is an educational programming language, based on "constructive" pedagogy, intended to have "low threshold and no ceiling" (Papert, 1980). It was designed and implemented by Bobrow, Feurzeig, and Papert at MIT in late 1960s. It was intended to have a "low threshold" that would allow rank beginners and young children to learn to program (in our nomenclature, to program_s) quickly. Since Logo was designed to align with constructivism, Logo was intended to serve as a standalone center in classrooms

المنارة للاستشارات
www.manaraa.com

for learning not only programming, but thinking skills in a broader sense. As of course a Turing-complete programming language, Logo, at least theoretically, enables expert programmers to use it to solve even complex programs.

Logo programming centers around giving commands to a turtle. Children began their programming learning experience by instructing the turtle to execute a sets of command, such as SQUARE or TRIANGLE, to draw different shapes. Then new commands can be based on the defined set of commands, and the process iterates.

There were three primary goals for Logo: Lisp-like manipulation for syntactic simplicity, dynamic creation, and easy debugging (Feurzeig, 2010).

Through the years, there have different variations of the Logo implementation on different platforms and in various dialects. But the original setup was based on working with a turtle robot on the floor, and then later turtle graphics on a monitor display.

3.1.1.2 The Dream of Logo's Educational Power

Originally, Papert (1980) envisioned that "the child programs the computer and, in doing so, both acquires a sense of mastery over a piece of the most modern and powerful technology and establishes an intimate contact with some of the deepest ideas from science, from mathematics, and from the art of intellectual model building." In this book, Papert pointed out two major themes: viz., with Logo, even children can learn to use computers in a "masterful" way; and learning to use computers can change the way children learn everything.

3.1.2 A *Brief* Assessment of Logo

Since we aspire to provide the world with technology designed on the basis of our theoretical foundation (meta-cognitivism$^+$, of course), technology intended to enable students to learn to program$_o$, and to give students a very deep understanding of the nature of programming and problem-solving (we explain the steps toward such a system in Chapter 5), it is incumbent upon us to provide an assessment of Logo. In briefer and more direct words: Is Logo already doing what our prospective technology would be intended to do?

The correct answer appears to be a negative one. For although there is some evidence that learning Logo improves problem-solving and thinking skills (Solomon & Papert, 1976; Papert, 1980), the majority of researchers outside the Logo/constructionism camp have found precious little to be cheered by.

المنارة للاستشارات
www.manaraa.com

For example, Pea and colleagues systematically studied Logo's impact on children's programming skill and understanding of programming concepts, such as recursion; and they studied whether Logo programming developed planning skill beyond programming (Pea et al., 1987). The results, unfortunately, were negative. In their study of Logo's impact on cognitive planning skill, they compared two classes of 25 children/students who had one school year of extensive programming in Logo, with another group that had no programming. Quite remarkably, the performance of between two groups of students was not significantly different (Kurland et al., 1987; Pea et al., 1987).

In addition, Vaikakul (2005), one of Papert's students, conducted a study for the Maine Learning Technology Initiative (MLTI), and was found that Logo (as well as self-directed learning) did not fulfill the dream of Logo's educational power. Between 2002 and 2003, Maine's public education systems provided over 30,000 laptops (Apple iBooks) to 7–8th-grade grades students. The intention was that with one computer for every child/student, and the abundant and rich opportunities for self-directed work that would flow from that, students would use their machines as a tool to teach themselves problem-solving and critical thinking, among other skills. Some teachers went to LOGO workshops, and in turn provided LOGO workshops to their students, in their classrooms, or after school to willing kids. But after a while, many teachers as well as students lost interest. Teachers thought LOGO was too hard and had nothing like the math that they felt needed to be taught. Students, on the other hand, didn't find it sufficiently interesting. One year later, most of the teachers and students were using the computer the same way: mostly to access information in websites, create slides, etc.; not to pursue the original goals of the Logo camp: to explore mathematics, learn programming, and problem-solve. Interviews with students and teachers disclosed that neither group felt Logo made a connection to the math they were supposed to master.

In one of the studies from Khasawneh (2009), dedicated to examining the impact of Logo programming performance and school mathematics achievement, the results were rather disturbing: For instance, for a sample of 228 7th-grade students, all studied Logo programming languages though turtle geometry. After a credit semester, their Logo programming and mathematical tests were analyzed. The resulting correlation between Logo programming scores and their mathematical scores were very low, in fact near zero (0.053). This indicates that student mathematical performance and Logo-programming performance require different skills. For the perspective of our approach, which seeks to teach

المنارة للاستشارات
www.manaraa.com

programming as deep logico-mathematical thinking, this is unsavory.

3.1.3 Logo's Relatives

Logo has inspired and influenced myriad other systems: NetLogo, StarLogo, KTurtle, REBOL, Etoys, Scratch, etc. Still firmly in keeping with constructivism, these systems generally provide a graphical environment for children to learn by discovery themselves. During the last 20 years, different variations were being used and integrated with 130 different implementations of Logo, each of which has its own strength. Many of these implementations concern teaching and learning geometry as an important branch of mathematics (Khasawneh, 2009). One wonders whether students are really in any way learning the underlying formal structures of geometry.

3.1.3.1 StarLogo, LEGOsheets, Lego Mindstorms

Logo-based and Logo-inspired systems, all based on constructivism's mantra that learning by building is efficacious, are numerous. For example, StarLogo, also developed in MIT, is an extension of the Logo programming language. NetLogo, designed by Uri Wilensky at Northwestern University, is in turn Java-based, but still very much a Logo-like programming language and integrated modeling environment. PythonTurtle and Pynguin are two Logo-like turtle-graphics environments created on top of Python.

LEGOsheets, based on AgentSheets created by researchers at the University of Colorado in 1994, is a rule-based programming language and visual aid for programming in Lego Mindstorms. Lego Mindstorms includes a programmable brick (in so-called Brick Logo) computer that controls the system, a set of modular sensors and motors, and Lego parts from the Technics line to create the mechanical systems.

3.1.4 Recent Applications in the Same Vein

There are recent drop-and-drag, game-like applications in the same constructivist vein. For instance, there is Squeak, Scratch, and Snap. Squeak is an implementation of the Smalltalk programming language, which is an object-oriented, dynamically typed, reflective programming language. Scratch, in turn based on Squeak, is a visual, event-driven, and imperative educational programming language that was designed by Resnick et al. (2009). Heavily influenced by Logo, it too allows for turtle graphics; here the turtle moves "sprites" and changes angles, and the programming is event-driven. Resnick et al. (2009)

المنارة للاستشارات
www.manaraa.com

has explicitly affirmed three core design principles: "more tinkerable," "more meaningful," and "more social." Snap, deeply influenced by Scratch, is yet another educational programming language, designed and developed by Harvey & Mönig (2013).

3.1.4.1 Alice Projects

Another educational programming language/environment, Alice, lets students set up a narrative scene and create a simple animation — with minimal teacher facilitation, and using 3D objects and yet another drag-and-drop interface intended to be engaging and a very "supportive" coding environment. Alice was developed specifically to address collaborative programming (Al-Tahat, 2014). Jain et al. (2011) also devised a tool to learn programming languages and algorithms based on collaboration among young coders.

3.2 Conclusion

So, there are many, many systems in the educational-programming space. We have looked far and wide,for hard empirical evidence that any of them exceed the effectiveness of Logo (which, as we have reported above, is by any metric distressingly low), and have found no such thing, alas. We point out, finally, that absolutely none of the educational environments in the young-student space have been built in accordance with the logic-based paradigm of computer programming, in which programs *are* proofs (see §5.4.2). From the perspective of our theoretical basis, formal meta-cognitivism$^+$, it comes as no surprise that the educational programming systems of the present and past have severely limited pedagogical reach. Our explanation is directly in line with the overarching moral of Chapter 2, and the vindication therefrom of our chief hypothesis that the essence of excellence in computer programming is deep understanding of the formal, deep formal-logic terrain that underlies what programming is.

المنارة للاستشارات
www.manaraa.com

4. COGNITION AND AUTOMATIC PROGRAMMING

It seems reasonable to hold that in order to find out how humans manage to write impressive computer programs one should attempt to build systems that autonomously write programs. This attempt is known as *automatic programming* (AP, for short). Unfortunately, automatic programming, to this point at least, has ignored the human cognition associated with computer programs, in favor of two engineering approaches (evolving computer programs, and a particular logic-based approach). Neither of these approaches is based on study of, let alone the attempt to mechanically simulate or replicate, human computer programming. We now give a summary review and assessment of automatic programming.

4.1 Overview & Assessment of Automatic Programming

In brutal simplicity, automatic programming (AP) is the field devoted to creating computer programs smart enough to write significant computer programs, from — as we've of course said elsewhere — scratch. A visual overview of this challenge is provided in Figure 4.1. By and large, AP has not exactly made impressive strides over the last three decades.[4]

The aims of AP have fluctuated considerably over the decades; this has been pointed out by Rich & Waters (1988) and others. In the 1950s, the mechanical compilation of Fortran programs into machine code was actually considered "automatic programming." But in the 1960s, at the dawn of AI (officially 1956, at the famous Dartmouth conference), and in keeping with the rather ambitious dreams of this new and exciting discipline, a much more lofty goal was set for the field — the black-box version of AP pictured in our Figure 4.1, wherein only a non-implemented description of the desired relationship between input and output is provided to the program-generating software, and the latter then emits a ready-to-be-implemented, executable computer program that realizes the

Portions of this chapter previously appeared as: Bringsjord, S., Li, J.,Govindarajulu, S.N. and Arkoudas, K. (2012). On the Cognitive Science of Computer Programming in the Service of Two Historic Challenges. In De Mol L. & Primiero, G. (Eds.), Proceedings from AISB/IACAP World Congress 2012:*Symposium on the History and Philosophy of Programming* (pp. 17–23). Birmingham, UK: The Society for the Study of Artificial Intelligence and Simulation of Behaviour.

[4] The 'from scratch' phrase is the rub. Great progress has been made in the *semi*-automatic realm, where code is e.g. generated from libraries of pre-existing syntax, and from pre-engineered algorithms that generate code from mechanical, predictable triggers.

www.manaraa.com

given specification. In short, in goes something well short of a computer program or even a precise algorithm; but out comes a program ready to be executed.

There are several options available in the above framework. First, there are many choices concerning the medium in which the description can be expressed; for example:

1. *Natural Language*: Ideally, we would be able to just tell the machine in a so-called *natural language* like English what we want the program to accomplish. At least today, this is not practicable. One could of course restrict the input language to, say, a controlled, rigid fragment of English (or for that matter so other natural language, e.g. Chinese), but such subsets are in fact formally defined in the first place, and hence we would be "spoon feeding" the machine, and avoiding the fundamental challenge of AP.

2. *Formal Specification*: The input description could be expressed in a rigid, declarative formal language, such those that accompany a formal logic, for instance first- or second-order logic.

3. *Input-Output examples*: The "description" could be *illustrated* by way of sample input/output pairs. This would approach would be necessarily incomplete, as there will always be infinitely many programs that cover any presented finite set of examples.[5]

4. *Hybrid representations*: Here, there is mixture of the above, and also a reliance on diagrammatic or visual representations. (This option is relevant to what we discuss in §4.3.)

Most of the work on AP so far has been based on adoption of the second and third of these options. Typically, inductive techniques build programs from input-output examples; deductive techniques, on the other hand, typically construct programs from formal descriptions.

The following represent three of the most prominent lines of research pursued in the inductive thrust of AP:

1. *Recurrence Detection*. The seminal work in this area was carried out by Summers (1977). Importantly from the standpoint of cognitive science, this work is certainly among the most psychologically plausible lines taken in the field. Summer's ideas have been extended, most notably in the 1980s, by Kodratoff et al. (1989) and Wysotzki (1986); they augmented the basic scheme we have delineated above. Similar techniques have been used in "programming-by-demonstration" systems, such as the one provided by Tinker (Lieberman, 1993). Kitzelmann et al. (2006) and others are sustaining this line of investigation, but results so far have been somewhat meager.

[5]Theorems to this effect are provided in Jain et al. (1999).

www.manaraa.com

2. *Genetic Programming.* Genetic programming (GP) (Koza, 1992), the core idea of which is to evolve programs via algorithms that parallel mutation and natural selection seen in the biological realm, was invented in the 1980s (although the core concept behid evolutionary algorithms goes back to the 1950s). The basic algorithm-sketch underlying GP is as follows:

   (a) Initiate processing with a collection of random computer programs. Typically programs are purely functional in nature, often expressed in exclusively functional LISP, and represented as ASTs (abstract syntax trees).[6]

   (b) Now, parelleling the biological realm, assign a fitness value to each of the programs in 1.

   (c) Next, create a new collection, by performing "genetic operations" on selected programs in the original collection. Typical operations are crossover and mutation.

   The 1.–3. loop is continued until some program in the current collection reaches an acceptable level of finess, or until a maximum number of runs have been reached.

   The most common genetic operations are specifically these:

   - *Mutation* (performed on a single program): Randomly modify part of a program's structure.
   - *Crossover* (performed on a pair of programs): Randomly modify two parts of the two programs.
   - *Reproduction* (done on a single program): Move a program that has not been changed into the new collection.

   Crossover is probably the most important of these three operations, and is the operation performed most often.

   GP is generally well-suited for optimization and control problems, and for games. Unfortunately, it appears to be prohibitively expensive, computationally. Evaluating the fitness of programs entails evaluating the programs themselves, and that is extremely time-consuming. ADATE (Olsson, 1998), for instance, one of the most

[6]Excellent coverage of a purely functional version/approach in Common Lisp is provided in Shapiro (1992).

www.manaraa.com

prominent automatic-programming systems based based on genetic programming, requires a complete six-and-a-half days to evolve a program for simply computing the intersection of two lists of object. Like the other approaches to AP, GP has not scaled up to realistic programs. Not only that, but in contrast to inductive logic programming (briefly discussed below), GP offers human overseers and engineers no explanation. Usually, the generated programs are ridiculously convoluted — code that even novice programmers would not lapse into creating. Therefore, GP versions of AP, from the standpoint of cognitive science, are the least plausible of all well-known lines of attack on the automatic programming challenge.

3. *Inductive Logic Programming.* Inductive logic programming (ILP) (Muggleton, 1992) takes as input logic programs (not functional programs like those written e.g. in pure Lisp). The process of combinging this input makes use of these elements:

   (a) A background collection of formulae, B;

   (b) a collection of positive examples E^+ (almost always simple formulae in first-order logic); and

   (c) a set of negative examples.

   In this approach, he following are legislated as necessary conditions:[7]

   (a) $\forall\, e^- \in E^- \,.\, B \nvdash e^-$

   (b) $\neg \forall\, e^+ \in E^+ \,.\, B \vdash e^+$

   The output is then an hypothesis h such that:

   (a) $\forall\, e^+ \in E^+ \,.\, B \wedge h \vdash e^+$

   (b) $\forall\, e^- \in E^- \,.\, B \wedge h \nvdash e^-$

   Of course, the conjunction of all the positive examples constitutes a trivial solution to these equations. But what practitioners of the inductive logic programming (ILP) line are seeking is predictive power: they want h to perform well on brand-new data never seen by the system before.

[7] Following standard notation, we write $\Phi \vdash \phi$ to indicate that the collection of formulae Φ logically entails the formula ϕ.

المنارة للاستشارات
www.manaraa.com

The core algorithm-sketch underlying ILP is this: start with a very specific h, and gradually generalize it; or, alternatively, start with a very general h and gradually make it more specific.

Many successful ILP systems treat induction as the inverse of deduction; accordingly, such systems create hypotheses by "flipping" or reversing standard inference rules seen in proof theories. For instance, here is the inference rule known as absorption:

$$\frac{A \Rightarrow q \qquad A, B \Rightarrow p}{A \Rightarrow q \qquad B, q \Rightarrow p} \quad [Absorption]$$

The conclusion in this rule must logically entail the input premises (see note 7).

While ILP has been somewhat successful in data mining, and no doubt has a bright future elsewhere, in automatic programming the results have been underwhelming. In ILP-based work, like work based in competing paradigms, no computer programs beyond the usual simple examples that novice programmers quickly reach when for instance learning Logo have been have been generated. In addition, the generated programs are usually inelegant and inefficient.

What about purely deductive approaches, instead of approaches, like ILP, that essentially reverse deduction? In a deductive approach to AP, the input is in the form of a detailed, declarative description of the general connection between input and output (recall Figure 4.1). This description is typically couched in terms of formulae in the formal languages associated with first-order logic, or other higher-order descendants. The output is of course a computer program, in keeping with the nature of the AP challenge. Once again, popular types of computer programs here include purely functional versions of Lisp. The output is accompanied by "guarantees" that the code does behave as desired; these guarantees are step-by-step proofs.

Much of the work in the purely deductive approach has been carried out in the context of formal logic. The basic idea is that the desired computer program is extracted from a proof that asserts the existence of a suitable output (i.e. an output that meets the specification mentioned above). The main idea is straightforward: Given the formal specification and a background theory, both mentioned above, which together rigorously describe the relevant domain (e.g., some class of data structures or algorithms), an attempt is made to create a proof that the desired function satisfies the input specification.

المنارة للاستشارات
www.manaraa.com

It is probably worthwhile to be more rigorous, especially in the context of the overall cognitive-science approach (meta-cognitivism$^+$) advocated in the present work. So, let S represent the given specification:

$$\forall\, x : I, y : O \;.\; S(x, y) \tag{4.1}$$

where x and y are variables ranging over the input and output domains; that is, over, respectively, I and O. We do not require the specification to satisfy the traditional definition of a function: For any given input x, there may be zero, one, or multiple outputs y that stand in the desired relation to input x. Often the specification $S(x, y)$ is of this form:

$$Pre(x) \Rightarrow Post(x, y), \tag{4.2}$$

This form aasserts that if the input x accords with given preconditions (symbolized by '*Pre*'), the output y is related to x in accordwith some relevant postcondition ('*Post*').

The goal is of course to generate a Turing-computable definition of a function mapping from I to O, such that the aforementioned specification S is true of the input-output pairs. Put symbolically, the sought-for goal is this:

$$f : I \rightarrow O$$

, where the following formula holds.

$$\forall\, x : I \;.\; S(x, f(x)). \tag{4.3}$$

Specifically, when our specification S has exactly the syntax of (4.2), the desired goal can be equivalently stated like this:

$$\forall\, x : I \;.\; Pre(x) \Rightarrow Post(x, f(x)). \tag{4.4}$$

Unfortunately, it must be conceded even by fans of logic in the computer-programming space that certain disadvantages plague the purely deductive approach. For example, here are two:

- The purely deductive approach requires the relevant engineers/scientists to produce a formal specification of the relationship between the inputs and desired outputs;

المنارة للاستشارات
www.manaraa.com

given the equations set out above, this is patently clear, and quite unavoidable. The problem here is that creating such specifications can often be just as challenging as creating a program that computes the targeted function.

- The approach in question hinges on the power of automated theorem proving. But this field is itself an extremely challenging one, and while certainly much progress has been made and continues to be made Bringsjord (2008b), the power of automated theorem provers is still far shy of the level of power needed for the automated generation of substantive computer programs (i.e., programs routinely produced by human programmers$_o$) in the deductive manner we have described.

Despite these drawbacks, and in keeping with the approach to educational programming environments we advocate (Chapter 5), we believe that the purely deductive route will make significant headway against the AP challenge, as long as the burden on the machine for automated theorem proving is partly lifted by a human contribution in the crafting of proofs. Of course, by definition of the AP challenge itself, what the human supplies cannot require great ingenuity and insight. We believe that a framework in which the machine contributes the lion's share, but the human supplies some guidance, is an accurate reflection of how accomplished human programmers produce substantive programs. The empirical work, and associated results, set out in Chapter 2, does in fact generally support this human-machine symbiosis view, but we readily admit that much additional empirical work on our part is necessary to test our suspicion, let alone concretize it in a working AP system. At any rate, in order for challenge **T** to be met, some strong contribution from deduction must be made, since the only way, by definition, to prove that some AP-generated code does in fact compute the function it is supposed to is to use formal logic — and ultimately to use proof-checking technology (Arkoudas & Bringsjord, 2007, discussed in).

4.2 On Cognitive Science, Hypercomputation, and Automatic Programming

As noted earlier in the dissertation, cognitive science, from its sudden and revolutionary arrival on the behaviorism-dominated scene, has always proceeded under the assumption that human cognition is fundamentally computation (von Eckardt, 1995; Lepore & Pylyshyn, 1999). This is the assumption that is aligned with what we have called *cognitivism*. In this brief section, we point out that in light of the nature of computer programming, and specifically in light of how difficult computer programming is (formally

المنارة للاستشارات
www.manaraa.com

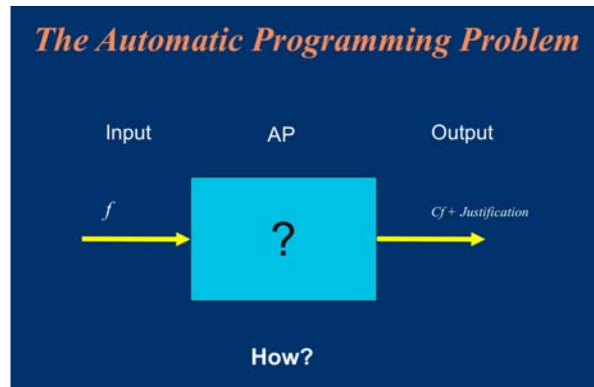


Figure 4.1: Automantic Programming Problem

speaking), it makes sense to consider the possible need to expand the cognitive science of computer programming in such a way that it brings within scope models of creativity in computer programming that are beyond standard, Turing-level computation.

Computation is standardly rendered precise within the space of functions from the natural numbers $N = \{0, 1, 2, \ldots\}$ (or pairs, triples, quadruples, $\ldots$ thereof) to the natural numbers; in other words, within

$$\mathcal{F} = \{f | f : N \times \ldots \times N \longrightarrow N\}.$$

This is a very large set. $\mathcal{F}$ is, as it is standardly said, an *uncountably* infinite set: one cannot specify a Turing machine (TM) (standard computer program, etc.) that, eventually, prints out every member of it Boolos et al. (2003). In short, $\mathcal{F}$ is the same size as the real numbers R. (Size would normally be regimented with help from cardinal numbers, but such precision is not needed here.) The set $\mathcal{T}$, on the other hand, is precisely the same size as the natural numbers N themselves. In the case of the natural numbers, each element can of course be eventually printed by this simple algorithm: print 0, increment by 1, print 1, increment by 1, print 2 $\ldots$. A very small (but infinite) proper subset of it, $\mathcal{T}$ (hence $\mathcal{T} \subset \mathcal{F}$), is composed of functions that Turing machines and their equivalents (register machines; programs written in modern-day programming languages like Java, Logo, Scheme, Prolog;the λ calculus, etc.; a discussion of these and others in the context of an account of standard computation can be found in Bringsjord (1994)) can compute; these are the *Turing-computable* functions. For example, multiplication is one function at this level: it is acutely laborious but conceptually trivial to specify a Turing machine that, with any pair of natural numbers m and n positioned on its tape before processing starts,

المنارة للاستشارات
www.manaraa.com

leaves $m \cdot n$ on its tape after its processing is wrapped up.[8]

Hypercomputation is the computing, by various extremely powerful devices or machines, of those functions in $\mathcal{F}$ that are beyond the so-called *Turing Limit*; i.e., those functions (composing $\mathcal{H}$) in $\mathcal{F}$ that are not in $\mathcal{T}$. The mathematics of hypercomputation is now very developed; the devices, machines, definitions, and lemmas/theorems in the relevant space are elegant and informative (e.g., see Siegelmann & Sontag, 1994; Siegelmann, 1999; Etesi & Nemeti, 2002; Copeland, 1998; Hamkins & Lewis, 2000; Bringsjord et al., 2006b; Bringsjord & Zenzen, 2003).

To provide here a glimpse into the nature of hypercomputation, we can use the traditional door opening into its simplest part: the famous halting problem, first presented in the seminal Turing (1937). Let us assume that we have some, as it is called, *totally computable* predicate $H^s(n, k, u)$ that holds exactly when, or if and only if, Turing machine n, taking as input u, halts in exactly k steps after its work is done. Predicate H^s is classified as totally computable because, given the first-order formula

$$\psi := H^s(n, k, u),$$

where u and k are constants, n is a variable or parameter, and H^s is a three-place predicate, there is some Turing machine $m^\star$ that is able determine this formula's truth-value. That is, the Turing machine here can infallibly give us a judgment, **T** ("true") or **F** ("false"), given assignments to the variable n.[9] This implies that $H^s \in \Sigma_0$, that is, that H^s is a member of the starting point in what is called the Arithmetic Hierarchy (AH) (AH),[10] a point composed of totally Turing-computable predicates. Put another way, a standard way, we say that ψ is a Σ_0 formula.

But now let us reflect upon the formula

$$\psi' := \exists k' H^s(n, k', u),$$

in which the former constant k becomes a variable k' within the scope of a lone existential quantifier. Since the ability to find out if there is a set of k' steps after which an arbitrary Turing machine halts cannot be accomplished without having on hand the power to solve

[8] One such specification is provided in classical form by Boolos & Jeffrey (1989).

[9] One widely known, successful approach is just for $m^\star$ to simulate n for k steps and see what ensues.

[10] For an introduction to AH, see any of the trio Bringsjord & Zenzen (2003), Davis et al. (1994), or Kugel (1986).

www.manaraa.com

the halting problem, we know that this new formula cannot be understood by a Turing machine. This formula is thus not a Σ_0 one. However, the formula here is, as it is standardly said, *partially* computable, because (for a given Turing machine n) if the formula is true, then the Turing machine alluded to above, denoted by $m^\star$, will produce **T** (see note 9). For this reason, we classify ψ' as Σ_1.[11] When one builds From this point up, harder and harder problems are defined, and this takes us further and further up the Arithmetic Hierarchy. All of these problems on the way up, of course, exceed the power of standard, Turing-machine-level computation to handle..

With this review in place, if we look at computer programming, from the standpoint of the attempt to automate it as in automatic programming, we see that there is indeed some evidence for the view that the human mind, when doing computer programming from scratch (what we have called 'programming$_o$') is doing more than computing at the level of a Turing machine. Programming a computer in ways allowing it to simulate human cognition remains the original driving dream of cognitivism-based cognitive science (e.g., see Anderson & Lebiere, 2003), and certainly many, many advances in cognitive science hinge on the ability of clever humans to write clever programs. But computer programming, from the standpoint of the Arithmetic Hierarchy, is extremely difficult. In order to be just a decent programmer, one must be able to decide whether two programs (written in, say, Logo) compute the same underlying function. But given two Turing machines m and m', the question of whether they compute the same function f is, believe it or not, a Π_2 problem.[12] It is hence not much of a surprise that automatic programming, the attempt, as we have explained earlier, to engineer (Turing-level) computational systems able to automatically write programs that compute certain functions, has proved to be

[11] To informally generalize this in standard fashion, the quantifier-based representation of (a "formulaized") AH is:

Σ_n The set of all formulae definable in terms of totally computable predicates that use at most n quantifiers, the first of them being an *existential* quantifier.

Π_n The set of all formulae definable in terms of totally computable predicates that use at most n quantifiers, the first of which is a *universal* one.

Δ_n Here we have: $\Sigma_n \cap \Pi_n$

[12] To formally demonstrate this, we have only to introduce $H(n, k, u, v)$, which holds exactly when Turing machine n halts in k steps, having begun with u on its tape to start with, and leaving v there on its tape as output. We can then build atop H to produce the full representation of the problem we are dealing with:

$$\psi'' := \forall u \forall v [\exists k H(n, k, u, v) \leftrightarrow \exists k' H(m, k', u, v)]$$

www.manaraa.com

fiendishly difficult, as we have also explained. And yet when it comes to humans, elegant programs of astonishing size are routinely written and verfified. This isn't *decisive* evidence in favor of the view that cognitive science needs to take seriously the possibility that humans hypercompute, but it *is* evidence nonetheless. The evidence is not conclusive because we lack some key information: We do not absolutely know whether the human race is capable, eventually, of judging in the *arbitrary case* whether two programs are equivalent. On the other hand, as to the strength of this evidence, it is worth noting that we have here considered only the situation where the human programmer_o judges whether *two* programs compute the very same function. But as a matter of fact, such judgments are routinely applied to cases of n programs. This is even more demanding, and presumably serves as additional evidence. To conclude, there thus remains open the possibility that a more mature cognitive science of computer programming, built atop the present dissertation and the work that it is in turn built atop, will need to consider models based in the mathematics of hypercomputation.

4.3 "Visual Logic" and Suggestive Aspect of Experiments

At the conclusion of Chapter 2, we mentioned that there is a need to explore further the obviously strong role that good human progammers give to visual or diagrammatic techniques. The role of diagrammatic thinking and reasoning in visualizing data structures and transformations on such data structures, during the creative, exploratory part of programming, seems very important. Could it be possible to give to machines something like the human ability to use visual techniques, so that the challenge **T** is met, at least partially, thereby? After all, we know that sorting algorithms have a strong visual side; it thus seems eminently reasonable to conjecture that discovering an innovative sorting algorithm would, in the human case, make crucial use of visualization. And it is not just sorting algorithms where this phenomena can be seen. For example, even number theorist theorists have been known to see patterns in terms of diagrams.[13] Could a computing machine do something like this? Perhaps. In fact, note that just about *any* algorithm that manipulates discrete data structures can be pictured diagrammatically. Arkoudas & Bringsjord (2009) developed a computational model (Vivid) of diagrammatic problem solving in previous research, and we believe, with them, that it could form the foundation for an attempt to give a computing machine the capacity to program_o via visual techniques

[13]Captivating examples, replete with figures, are given in Penrose (1994).

المنارة للاستشارات
www.manaraa.com

of the general sort that we have seen in our human subjects. Indeed we imagine a symbiosis as our research program moves forward: ascertain in more detail the visual techniques used by first-rate human programmers$_o$, and use this knowledge to engineer automatic-programming techniques, based on extensions of Vivid informed by what is seen in the human case.

www.manaraa.com

5. A NEW EDUCATIONAL PROGRAMMING LANGUAGE: REASON

Logicians and those using formal logic have for many centuries seen logic as the field able to (when suitably deployed) foster clear thinking in human persons. This can of course be easily confirmed by consulting any comprehensive introduction to logic, which invariably offer rationales such as that by study of logic one will be less likely to be hoodwinked by fallacious reasoning.[14] Put in terms of a key distinction that can be found as far back as 300 years BC (Aristotle), formal logic has been viewed as *prescriptive*, rather than *descriptive*. In other words, logic has been here considered to be a discipline charged with explaining, in detail, what representations, and processes over these representations, *ought* to be followed by those aspiring to be clear thinkers.[15] For people with this ambition, logic seems to be just what is needed, since it is designed to enable declarative content to be expressed in both a syntactically and semantically precise manner. In additional, logic also provides methods of reasoning that can be directly carried over to the process of computing and computer programming. Put in stark form, when someone today writes a Prolog program, logic is thereby incarnated on the spot.

Unfortunately, only an exceedingly small percentage of people officially study logic; this is true even if the population in question is restricted to the so-called "technologized" world, where educational programming environments (such the ones featured in Chapter 3) are common. Though there are some rare exceptions, the vast majority of pre-college mathematics curricula avoid logic, and the traditional introduction to formal logic is first available, almost invariably, to first-year college students only an elective — with the exception of some computer science programs. (Rensselaer is itself one of the exceptions: The BS in Computer Science requires study of logic.) Even those majoring in mathematics or philosophy can often obtain their degrees in these disciplines without having to take any official formal logic course. On the other hand, at least across the technologized

Portions of this chapter previously appeared as: Bringsjord, S. & Li, J. (2008). Toward Aligning Computer Programming with Clear Thinking via the Reason Programming Language. In K. Waelbers, B. Briggle & P. Brey (Eds.), *Current Issues in Computing and Philosophy* (pp. 156-170). Amsterdam, The Netherlands: IOS Press.

[14] See, e.g., the extensive discussion of fallacies in Copi & Cohen (1997). See also Barwise & Etchemendy (1999), which has the added benefit of setting out logic in a way designed to reveal its connection to computer science — a connection central to Reason.

[15] Aristotilean (syllogistic) logic is presented in *Organon*, part of McKeon (1941). Coverage of the relevant of Aristotelean logic to modern-day data structures, directly relevant to today's programming environments, a key focus of the present dissertation, is given in Glymour (1992).

المنارة للاستشارات
www.manaraa.com

world, computer programming, as of course noted earlier in this dissertation, *is* quite often introduced to young students.[16] And not only that, but certain computer programming languages, most prominently the Logo language and paradigm (and its many dialects and variants) discussed in Chapter 3, have long been billed as helping young people become clearer thinkers. It is somewhat doubtful that such languages can succeed in this regard (for reasons to be briefly discussed below), but at any rate, it seems sensible to explore an approach to programming that *guarantees* an intimate link between the clear thinking required to write first-rate programs, and the kind of clear thinking that logic has historically sought to cultivate. Accordingly, Bringsjord and Li have invented a new computer programming language standing at the nexus of computing and philosophy: Reason, a language firmly based in the logic-based programming paradigm,[17] and thus one offering the intimate link in question to all who would genuinely use it.

The plan of this chapter is as follows. The next section (5.1) is a barbarically quick and standard introduction to elementary deductive logic, given to ensure that the present chapter will be understandable to readers from fields other than logic and philosophy. In section 5.2, we define what we mean by 'clear thinking,' and provide two so-called *logical illusions* (Johnson-Laird et al., 2000; Bringsjord & Yang, 2003): that is, two examples of difficult problems which those capable of such thinking should be able to answer correctly (at least after they have Reason at their disposal).[18] The next section (5.3) is devoted to a brief discussion of Logo, and Prolog and logic programming. Logo and Prolog are essential to understanding the motivation for creating Reason. In section 5.4, Reason itself is introduced in action, as it is used to solve the two problems posed in section 5.2. The chapter ends with brief section on the future of Reason.

5.1 Elementary Logic Encapsulated

In formal, deductive logic,[19] declarative statements, or as is it sometimes said, propositions, are generally represented by formulas in one or more formal languages associated

[16]In the United States, there is an Advanced Placement exam available to any high school student seeking college credit for demonstrated competence in (Java) programming.

[17]Notice that we do not say 'the logic programming' paradigm. The paradigm we do not mention is a much narrower one, tightly connected to Prolog, and the inference procedure known as *resolution*. Prolog is discussed momentarily.

[18]These illusions appear in a larger list of puzzles given in Bringsjord (2008a), all developed and analyzed in Li-Bringsjord collaboration, with an eye to eventually using them to anchor and test new educational programming langauges for learning clear thinking.

[19]Informal logic is not directly connected to computer programming, and is left aside here. In addition, we leave aside probabilistic/probability logic, an excellent summary of which is provided in Adams (1998).

www.manaraa.com

with formal logics, and these formal languages provide precise strings for carrying out deduction over. The simplest formal languages (associated with deductive logics) that have provided sufficient raw material for building corresponding programming languages are: the propositional calculus, and the predicate calculus (or first-order logic, or just FOL); together, this pair comprises what is generally called *elementary deductive logic*. (The pair is often classified under 'mathematical logic,' because these formal logics have traditionally been used to formalized mathematics itself, but that is not an activity central to our programming-focused discussion.) We proceed now to give a very short review of how declarative content is represented and reasoned over in these elementary logics.

With respect to the specific logics called out in the previous paragraph, and indeed in general when it comes to any formal deductive logic, three main components are required: one is purely syntactic/linguistic or string-based; one is semantic or model-based; and one is meta-theoretical in nature. The syntactic component includes specification of the alphabet of a given formal language, the grammar for building well-formed formulas (as they are traditionally called: 'wffs') from this alphabet, and, more importantly, a so-called *proof theory* that precisely sets out how and when one or more formulae can be deductively inferred from a set of formulae. The semantic component includes a rigorous account of the conditions that must be met in order for a formula in a given formal language for a given logic to have a semantic value such as 'true' or 'false.' (Other values, e.g., 'indeterminate' and 'probable,' are possible, but we are simplifying here.) The meta-theoretical component includes theorems, lemmas, conjectures, hypotheses etc. concerning the trio just enumerated: that is, the syntactic component, the semantic component, and connections between them. In this chapter, we focus on the syntactic side of things. Ebbinghaus et al. (1994) give a thorough, but yet refreshingly economical, coverage of the formal semantics and meta-theory of elementary deductive logic.

We mentioned that an alphabet must be specified for each relevant formal language. In the case of the propositional calculus, this alphabet is standard and well-known. It is composed, first, of a collection of so-called propositional variables:

$$p_1, p_2, \ldots, p_n, p_{n+1}, \ldots$$

Standard practice is to make use of p, q, and r, by setting p_1 to p, p_2 to q, and p_3 to r. The alphabeth also contains a standard quintet of truth-functional operators or connectives; the are: $\neg, \rightarrow, \leftrightarrow, \wedge, \vee$. These operators can be intuitively understood, without danger,

www.manaraa.com

in the following manner, respectively: 'not,' 'implies' (or 'if then '), 'if and only if' (or 'provided that' or 'exactly when'), 'and,' and 'or.' Armed with this alphabet, one can construct formulae that represent declarative content. For instance, to say that 'if Alvin hits Bill, then Bill hits Alvin, and so does Charlie,' we could write

$$a_h \rightarrow (b_h \land k_h)$$

where the propositional variables here pick out the obvious underlying claims.

We can get to the more articulate FOL by introducing two standard and much-used (in e.g. computer science) *quantifiers* to the symbolic machinery given in the previous paragraph. The pair of quantifiers are: $\exists x$, which is normally read as: 'there exists an x such that ...') and $\forall x$ ('for all x the following ... holds'). The first of these quantifiers is is standardly called the *existential* quantifier, and the second is called the *universal* quantifier. In addition, FOL's language side also includes a collection of of variables (like x, y, etc., operating here pretty much as variables work in elementary algebra), names for individual objects (often called *constants*, predicates or relation symbols (used to denote attributes that individual things can have or lack), and symbols that denote functions. For an example, the linguistic machinery we now have available enables us to represent the English sentence 'Everyone loves anyone who is served by someone' is represented as

$$\forall x \forall y (\exists z Serves(y, z) \rightarrow Loves(x, y))$$

We have now defined the formal languages of the logics that we focus on, and by that have explained how declarative data is represented in wffs. Proof theory must be applied now to make plain how inference over this data works. We will discuss this later, at the point where we turn to coverage of the kind of proofs-as-programs that have inspired the attempt to build Reason. We will then see that in the approach embodied by Reason, computer programs are in the end instructions for how to generate linked chains of inference from formula to formula. Before turning to inference, though, we must as promised explicate the semantic sphere we said at the outset of the chapter is one of the main components of any classical deductive logic.

Recall that we presented above the standard quintet of boolean operators; that is, negation, conjunction, and so on. These operators are standardly defined by *truth tables*, to be found in any competent introduction to elementary deductive logic, for instance

المنارة للاستشارات
www.manaraa.com

Bergmann et al. (1997); Barwise & Etchemendy (1999).[20] (Slightly different notation may be used to specify these tables, but the differences are not significant, meaning-wise.) Truth tables define the semantic value of a formula that one wants to understand, as long as the components of the formula in question are assigned semantic values. This makes the entire process *compositional*. The easiest and simplest kind of truth-table is the one that defines negation. In negation, if some wff ϕ is assigned the semantic value `true` (**T**, then $\neg\phi$, which is the negation of ϕ, is declared to be **F**. The first row in the standard truth table given immediately below regiments this. The only other possibility to consider is of course when the formula ϕ has an "input value" of **F**. In this case, the semantic value of ϕ is flipped over to **T**; again, see the table immediately below.

ϕ	$\neg\phi$
T	F
F	T

The truth tables that define the other four boolean operators, or truth-functional connectives, are easy to understand, and are given now in sequence. Again, the reader can be confident that these table are an invariant part of the foundation for any standard exposition of formal deductive logics. Because of that, more detailed explanation can be found, as we have said, in introductory textbooks, such as Bergmann et al. (1997).

ϕ	ψ	$\phi \wedge \psi$
T	T	T
T	F	F
F	T	F
F	F	F

ϕ	ψ	$\phi \vee \psi$
T	T	T
T	F	T
F	T	T
F	F	F

ϕ	ψ	$\phi \rightarrow \psi$
T	T	T
T	F	F
F	T	T
F	F	T

ϕ	ψ	$\phi \leftrightarrow \psi$
T	T	T
T	F	F
F	T	F
F	F	T

The reader can observe that the truth-table defining disjunction says that when both sub-formulae in a disjunct are true, the composite disjunction is rendered true. Such a disjunction is standardly called an *inclusive* disjunction. In, as it is commonly called, an *exclusive* disjunction, one of the sub-formulae is true, but *only* one.

both. This distinction becomes particularly important if one is attempting to symbolize parts of English (or any other *natural language*). If we wanted to represent that

[20] The reader should not confuse truth *tables* with what are typically called truth *trees*. Thorough coverage of the latter is provided in Bergmann et al. (1997).

المنارة للاستشارات
www.manaraa.com

information conveyed by the English sentence

The Mets will either win or lose the Series.

it would be a poor idea to use a disjunctive formula such as

$$M_w \vee M_l,$$

for an obvious reason: It is impossible (given how baseball is defined) for the Mets to *both* win and lose (the Series). Sometimes, special symbols are used to indicate that the kind of disjunction being used is exclusive, not inclusive. For example, a common symbol used to denote exclusive disjunction is $\oplus$, which, following standard practice, is defined through the following kind of truth-table.

ϕ	ψ	$\phi \oplus \psi$
T	T	F
T	F	T
F	T	T
F	F	F

Assume now that we have on hand a truth-value assignment, v (that is, an assignment of **T** or **F** to each propositional variable p_i). Given this, we can follow standard parlance to declare that v "makes true," or "models," or — in terminology that is frequently used in computational contexts — "satisfies" a given formula ϕ. This relationship is standardly expressed by this equation:

$$v \models \phi.$$

When we have a formula which has the attribute that there is some model that satisfies it, this formula is said to be *satisfiable*. A formula that cannot possibly be true on any model (e.g., $\neg(\psi \rightarrow \psi)$) is classified as *unsatisfiable*. Of course, Some formulae hold on every single model. For example, the formula $((\phi \vee \psi) \wedge \neg\psi) \rightarrow \phi$ is such a formula. Such formulae are standardly classified *valid*, and are sometimes categorized as *validities*. To convey that a formula ϕ is valid one standardly writes

$$\models \phi.$$

المنارة للاستشارات
www.manaraa.com

Another important semantic concept is *consequence.* A particular formula ϕ is classified as a consequence of a set Φ of formulae just in case every single truth-value assignment on which all of members of Φ are true is also an assignment on which ϕ itself is true. This is traditionally conveyed by writing:

$$\Phi \models \phi.$$

The final very important concept that is in the semantic side of the propositional calculus is that of *consistency.* We follow tradition and say that a set Φ of formulae is *semantically consistent* if and only if there is a truth-value assignment on which all of the elements of Φ hold. As a check of understanding, we encourage the reader to verify that a conjunction of formulae extracted from a semantically consistent set must invariably be satisfiable.

And now, what about the semantic side of FOL, first-order logic?

Unfortunately, the formal semantics of FOL gets quite a bit more tricky than the truth table-based machinery we have given for the propositional (i.e., the non-quantificational) level. The central concept is that in FOL formulas are said to be true (or false) on *models* (this is nothing new, so far). That some first-order formula ϕ is true on a model is often written as $\mathcal{M} \models \phi$. (This is often read like this: "$\mathcal{I}$ satisfies, or models, ϕ.") For instance, the formula $\forall x \exists y Ryx$ might mean, on the standard model for arithmetic, that for every natural number n, there is a natural number m that is "to the right of" n on the natural-number line, which is to say, that $m > n$. In this case, the *domain* is the set of natural numbers, that is, $\mathbf{N}$; and R symbolizes 'right of.' Much more could of course be said about the formal semantics (or *model theory*) for FOL — but this is an advanced topic beyond the scope of the present, brief treatment, given only to make facilitate exposition of Reason. For a fuller treatment that uses the traditional notation of model theory, which we have pretty much followed, see Ebbinghaus et al. (1994).

5.2 What is Clear Thinking?

The concept of clear thinking, at least to a significant degree, can be operationally defined with help from psychology of reasoning; specifically with help from, first, a distinction between two empirically confirmed modes of reasoning: context-dependent reasoning, versus context-*in*dependent reasoning; and, two, from a particular class of stimuli used in experiments to show that exceedingly few people can engage in the latter mode. The class of stimuli are what have been called *logical illusions.* We now proceed to explain the distinction and the class.

المنارة للاستشارات
www.manaraa.com

5.2.1 Context-Dependent v. Context-Indepedent Reasoning

In an wide-ranging paper in *Behavioral and Brain Sciences* that draws upon empirical data accumulated over more than half a century, Stanovich & West (2000) explain that there are two dichotomous systems for thinking at play in the human mind: what they call System 1 and System 2.

Reasoning performed on the basis of System 1 thinking is bound to concrete contexts and is prone to error; reasoning on the basis of System 2 cognition "abstracts complex situations into canonical representations that are stripped of context" (Stanovich & West, 2000, p. 662), and when such reasoning is mastered, the human is armed with powerful techniques that can be used to handle the increasingly abstract challenges of the modern, symbol-driven marketplace. System 1 reasoning is context-dependent, and System 2 reasoning is context-independent. Recently, the presence of these two radically different modes in human thinking has been treated in a lively and readable way by Kahneman (2013). We now explain the difference in more detail.

Psychologists have devised many tasks to illuminate the distinction between these two modes of reasoning (without always realizing, it must be granted, that that was what they were doing). One such problem is the Wason Selection Task (Wason, 1966), which runs as follows.

Suppose that you are dealt four cards out of a larger deck, where each card in the deck has a digit from 1 to 9 on one side, and a capital Roman letter on the other. Here is what appears to you when the four cards are dealt out on a table in front of you:

E	K	4	7

Now, your task is to pick just the card or cards you would turn over to try your best at determining whether the following rule is true:

(R_1) If a card has a vowel on one side, then it has an even number on the other side.

Less than 5% of the educated adult population can solve this problem (but, predictably, trained mathematicians and logicians are rarely fooled). This result has been repeatedly replicated over the past 15 years, with subjects ranging from 7th grade students to illustrious members of the Academy (Bringsjord et al., 1998). About 30% of subjects do turn over the E card, but that isn't enough: the 7 card must be turned over as well. The reason is as follows. The rule in question is a so-called **conditional** in formal logic,

المنارة للاستشارات
www.manaraa.com

that is, a proposition having an if-then form, which is often symbolized as $\phi \rightarrow \psi$, where the Greek letters here are variables ranging over formulas from some logical system. As the truth-tables routinely taught to young pre-12 math students make clear (e.g., Bumby et al., 1995, chapter 1), a conditional is false if and only if its antecedent, ϕ, is true, while its consequent, ψ, is false; it's true in the remaining three permutations. So, if the E card has an odd number on the other side, (R_1) is overthrown. However, if the 7 card has a vowel on the other side, this too would be a case sufficient to refute (R_1). The other cards are entirely irrelevant, and flipping them serves no purpose whatsoever, and is thus profligate.

This is the abstract, context-independent version of the task. But now let's see what happens when some context-*de*pendent reasoning is triggered in you, for there is incontrovertible evidence that *if the task in question is concretized*, System 1 reasoning can get the job done (Ashcraft, 1994). For example, suppose one changes rule (R_1) to this rule:

(R_2) If an envelope is sealed for mailing, it must carry a 20 cent stamp on it.

And now suppose one presents four envelopes to you (keeping in mind that these envelopes, like our cards, have a front and back, only one side of which will be visible if the envelopes are "dealt" out onto a table in front of you), viz.,

sealed envelope	unsealed envelope	env. w/ 20 cent stamp	env. w/ 15 cent stamp

Suppose as well that you are told something analogous to what subjects were told in the abstract version of the task, namely, that they should turn over just those envelopes needed to check whether (R_2) is being followed. Suddenly the results are quite different: Most subjects choose the sealed envelope (to see if it has a 20 cent stamp on the other side), *and* this time they choose the envelope with the 15 cent stamp (to see if it is sealed for mailing).

5.2.2 The King-Ace Problem

Now we come to a logical illusion, the King-Ace Problem. As we present the problem, it's a slight variant[21] of a puzzle introduced by Johnson-Laird (1997). Here it is:

Assume that the following is true:

[21] The variation arises from disambiguating Johnson-Laird's 's or else s'' as 'either s or s', but not both.'

www.manaraa.com

> 'If there is a king in the hand, then there is an ace in the hand,' or 'If there is not a king in the hand, then there is an ace in the hand,' — but not both of these if-thens are true.
>
> What can you infer from this assumption? Please provide a careful justification for your answer.

Your are encouraged to record your own answer. We return to this problem later, when using Reason to solve it. But please note that the correct answer to the problem is not 'There is an ace in the hand,' but rather the (counterintuitive!) proposition that there *isn't* an ace in the hand. If Reason is on the right track, use of it will help students see that this is the right answer.

5.2.3 The Wine Drinker Problem

Now let us consider a second logical illusion, an interesting puzzle devised by Johnson-Laird & Savary (1995) that has the same general form as Aristotle's syllogisms:

> Suppose:
>
> - All the Frenchmen in the restaurant are gourmets.
> - Some of the gourmets are wine drinkers.
>
> Does it follow that some of the Frenchmen are wine drinkers? Please provide a careful justification for your answer.

We will return to this problem later, when using Reason to solve it. But note for now that the correct answer is 'No.' Reason will itself provide a justification for this negative answer.

5.3 The Logo Programming Language; Logic Programming

As noted earlier in the dissertation, when youth learn to program by using Logo,[22] by far the programming language most used in the States to teach programming in grades 6–12, almost without exception, they produce instructions designed drive a turtle through some sequence of states. For example, the procedure

```
to square
repeat 4 [forward 50 right 90]
end
```

[22] Logo Fundation (2012) http://el.media.mit.edulogo-foundation/logo/programming.html (Date Last Accessed November 12,2014).

المنارة للاستشارات
www.manaraa.com

causes the turtle to draw a square. In a second, more sophisticated mode of programming, the Logo programmer can process lists in ways generally similar to those available to the Lisp programmer. Ever since a seminal paper by Black et al. (1988), it has been known that while students who program in the second way do seem to thereby develop some clearer thinking skills, the improvement is quite slight, the cognitive distance from processing lists to better logical reasoning is great, and hence *transfer* from the first activity to the second is very problematic.[23] In an intelligent reaction to this transfer challenge, Black et al. make a move that is quite interesting from the perspective of our own objective, and the language Bringsjord and Li have built to meet it: viz., they consider whether teaching Prolog[24] might be a better strategy for cultivating in those who learn it a significant gain in clear thinking. Unfortunately, there are five fatal problems plaguing the narrow logic programming paradigm of which Prolog is a concretization. Here's the quintet, each member of which, as shall soon be seen, is overcome by Reason:

1. Logic programming is based on a fragment of full first-order logic: its inexpressive. Human reasoning, as is well-known, not only encompasses full first-order logic (and hence on this score alone exceeds Prolog), but also modal logic, deontic logic, and so on.

2. Logic programming is "lazy." By this we mean that the programmer doesn't herself construct an argument or proof; nor for that matter does she create a model or countermodel.

3. While you can issue queries in Prolog, all you can get back are assignments to variables, not the proofs that justify these assignments.

4. In addition, Prolog can't return models or counter-models.

5. Finally, as to deductive reasoning, Prolog locks those who program in it into the rule of inference known as *resolution*.[25] Resolution is not used by humans in the business of carrying out clear deductive thinking. Logic and mathematics, instead, are carried out in what is called *natural deduction* (which is why this is the form of deduction almost invariably taught in philosophy and mathematics, two prominent fields among those directly associated with the cultivation of clear thinking in students).

[23]In particular, it turns out that making a transition from "plug-and-chug" mathematics to being able to produce proofs is a very difficult one for students to achieve (e.g., see Moore, 1994). There is further negative data in the case of Logo (e.g., see Louden, 1989; Pea et al., 1987).

[24]There is of course insufficient space to provide a tutorial on Prolog. We assume readers to be familiar with at least the fundamentals. Clocksin & Mellish (2003) give a classic introduction to Prolog.

[25]All of resolution can essentially be collapsed into the one rule that from $p \vee q$ and $\neg p$ one can infer q.

المنارة للاستشارات
www.manaraa.com

5.4 The Reason Programming Language

5.4.1 Reason in the Context of the Four Paradigms of Computer Programming

There are four programming paradigms: *procedural*, reflected, e.g., in Turing machines themselves, and in various "minimalist" languages like those seen in foundational computer science texts (e.g., Davis et al. (1994), Pascal, etc.); *functional*, (e.g., Scheme, ML, and purely-functional Common Lisp (Shapiro, 1992; Abelson & Sussman, 1996)); *object-oriented*; and *declarative* (reflected, albeit weakly, in Prolog (Clocksin & Mellish, 2003)). Reason is in, but is an extension of, the declarative paradigm.[26]

Reason programs are specifically extensions and generalizations of the long-established concept of *logic programs* in computer science (presented succinctly in, e.g., Ebbinghaus et al., 1994, chapter "Logic Programming").

5.4.2 Proofs as Programs through a Simple Denotational Proof Language

In order to introduce Reason itself, we first introduce the syntax within it currently used to allow the programmer to build purported proofs, and to then evaluate these proofs to see if they produce the output (i.e., the desired theorem). This syntax is based on the easy-to-understand type-α denotational proof language NDL invented by Konstantine Arkoudas (for background, see Arkoudas, 2000; Bringsjord et al., 2006a) that corresponds for the most part to systems of Fitch-style natural deduction often taught in logic and philosophy. Fitch-style natural deduction was first presented in 1934 by two thinkers working independently to offer a format designed to capture human mathematical reasoning as it was and is expressed by real human beings such as Gentzen (1935) and Jaśkowski (1934). Streamlining of the formalism was carried out by Fitch (1952). The hallmark of this sort of deduction is that assumptions are made (and then discharged) in order to allow reasoning of the sort that human reasoners engage in.

Now here is a simple deduction in NDL, commented to make it easy to follow. This deduction, upon evaluation, produces a theorem that Newell and Simon's Logic Theorist,

[26] As is well known, in theory any Turing-computable function can be implemented through code written in any Turing-complete programming language. There is nothing in principle precluding the possibility of writing a program in assembly language that, at a higher level of abstraction, processes information in accordance with inference in many of the logical systems that Reason allows its programmers to work in. (In fact, as is well-known, the other direction is routine, as it occurs when a high-level computer program in, say, Prolog, is compiled to produce code corresponding to low-level code; assembly language, for example.) However, the mindset of a programmer working in some particular programming language that falls into one of the four paradigms is clearly the focus of the present discussion, and Turing-completeness can safely be left aside.

المنارة للاستشارات
www.manaraa.com

to great fanfare (because here was a machine doing what "smart" humans did), was able to muster at the dawn of AI in 1956, at the original Dartmouth AI conference.

```
// Here is the theorem to be proved,
// Logic Theorist's ''claim to fame'':
// (p ==> q) ==> (~q ==> ~p)

Relations p:0, q:0. // Here we declare that we have two
                    // propositional variables, p and q.
                    // They are defined as 0-ary relations.

// Now for the argument.  First, the antecedent (p ==> q)
// is assumed, and then, for contradiction, the antecedent
// (~q) of the consequent (~q ==> ~p).
assume p ==> q
   assume ~q
      suppose-absurd p
          begin
            modus-ponens p ==> q, p;
            absurd q, ~q
          end
```

If, upon evaluation, the desired theorem is produced, the program is successful. In the present case, sure enough, after the code is evaluated, one receives this back:

```
Theorem: (p ==> q) ==> (~q ==> ~p)
```

Now let us move up to programs written in first-order logic, by introducing quantification. As you will recall, this entails that we now have at our disposal the quantifiers $\exists x$ ('there exists at least one thing x such that ...') and $\forall x$ ('for all x ...'). In addition, there is now a supply of variables, constants, relations, and function symbols; these were discussed above. What follows is a simple NDL deduction at the level of first-order logic that illuminates a number of the concepts introduced to this point. The code in this case, upon evaluation, yields the theorem that Tom loves Mary, given certain helpful information. It is important to note that both the answer and the justification have been assembled, and that the justification, since it is natural deduction, corresponds to the kinds of arguments often given by human beings.

المنارة للاستشارات
www.manaraa.com

```
Constants mary, tom.  // Two constants announced.

Relations Loves:2. // This concludes the simple signature, which
                   // here declares Loves to be a two-place relation.

// That Mary loves Tom is asserted:
assert Loves(mary, tom).

// 'Loves' is a symmetric relation, and this is asserted:
assert (forall x (forall y (Loves(x, y) ==> Loves(y, x)))).

//Now the evaluable deduction proper can be written:
suppose-absurd ~Loves(tom, mary)
   begin
      specialize (forall x (forall y (Loves(x, y) ==> Loves(y, x)))) with mary;
      specialize (forall y (Loves(mary, y) ==> Loves(y, mary))) with tom;
      Loves(tom,mary) BY modus-ponens Loves(mary, tom)
                        ==> Loves(tom, mary), Loves(mary, tom);
   end;
Loves(tom,mary) BY double-negation ~~Loves(tom,mary)
```

When this program is executed, we obtain the result we want:

```
Theorem:  Loves(tom,mary).
```

As is desired, the answer, and the argument paired with it, are produced; and because the argument is supposition-driven in nature (called “natural deduction”, standardly), the output here is of a kind that human reasoners and programmers would tend to craft themselves.

To this point, we have understood programs to be proof-generating structures. But now, what about the semantic component of deductive logics, which we referred to above? In other words, what is the role of models or interpretations, which as we have noted, appear in this component, in the approach discussed here? Moving beyond NDL, but still inspired by it and related systems, in Reason, programs can be written to produce, and to manipulate, interpretations and models seen in the semantic side of deductive logics. In addition, while in NDL the full cognitive burden is borne by the programmer, Reason can be queried about whether certain claims are provable. In addition, in Reason, the programmer can set the degree to which the system is intelligent on a session-by-session

المنارة للاستشارات
www.manaraa.com

basis. This last property of Reason gives rise to the concept that the system can be set to be "oracular" at a certain level. That is, Reason can function as an oracle up to some pre-set threshold. One candidate threshold is to allow the oracle to reason autonomously, as long as that reasoning does not involve either of the two quantifiers $\exists$ and $\forall$. The idea here is to allow Reason to be able to prove on its own anything that requires only reasoning at the level of simple boolean logic.

5.4.3 Cracking King-Ace and Wine Drinker with Reason

5.4.3.1 Cracking the King-Ace Problem with Reason

In this example, the selected logic to be used with Reason is standard first-order logic as described above, with the specifics that reasoning is deductive and Fitch-style. The system is assumed to have oracular ability up to the level just beneath use of the quantifiers; that is, the programmer can ask Reason itself to prove things as long as only reasoning at the level of the propositional calculus is requested. This request is signified by use of `prop`.

To save space, we assume that the programmer has made these selections through prior interaction with Reason. Now, given the following two propositions, is there an ace in the hand? Or is it the other way around?

F1 If there is a king in the hand, then there is an ace in the hand; or: if there isn't a king in the hand, then there is an ace in the hand.

F2 Not both of the if-thens in F1 are true.

In this case, we want to write a Reason program that produces the correct answer, which is "There is not an ace in the hand." We also want to obtain certification of a proof of this answer as additional output from our program.

We can obtain what we want by first declaring in Reason our key alphabet for the purposes at hand, which in the present case consists in simply announcing two propositional variables, `K` (for 'There is a king in the hand') and `A` (for 'There is an ace in the hand'). For the next step, we present the facts to Reason. Note that Reason responds by saying that the relevant things are known, and added to a knowledge base (`KB1`).

```
> (known F1 KB1 (or (if K A) (if (not K) A)))
F1 KNOWN
```

www.manaraa.com

```
F1 ADDED TO KB1

> (known F2 KB1
   (not (and (if K A) (if (not K) A))))

F2 KNOWN
F2 ADDED TO KB1
```

Our next move is to present the following partial proof to Reason. (It's a *partial* proof because the system itself is called upon to infer that the negation of a conditional entails a conjunction of the antecedent and the negated consequent. More carefully put, from `(not (if P Q))` it follows that `(and P (not Q))`.)

```
(proof P1 KB1
  demorgan F2;
  assume (not (if K A))
    begin
      (and K (not A)) by prop on (not (if K A));
      right-and K, (not A)
    end
  assume (not (if (not K) A))
    begin
      (and (not K) (not A) by prop on (not (if (not K) A)));
      right-and (not K), (not A)
    end
  proof-by-cases (or (not (if K A)) (not (if (not K) A))),
                     (if (not (if K A)) (not A)),
                     (if (not (if (not K) A)) (not A)))
```

When this proof is evaluated, Reason responds with:

```
PROOF P1 VERIFIED
ADDITIONAL KNOWNS ADDED TO KB1:
THEOREM: (not A)
```

We now make the perhaps not-unreasonable assertion that anyone who takes the time to construct and evaluate this program (or for that matter any reader who takes the time to study it carefully to see why `(not A)` is provable) doesn't succumb to the logical illusion in question any longer. Now we can proceed to issue an additional query:

المنارة للاستشارات
www.manaraa.com

```
> (provable? A)

NO
DISPLAY-COUNTERMODEL OFF
```

If the flag for countermodeling was on, Reason would display a truth-table showing that `A` can be false while F1 and F2 are true.

5.4.3.2 Cracking the Wine Drinker Problem with Reason

Let us remember the three relevant statements, in English:

F3 All the Frenchmen in the restaurant are gourmets.

F4 Some of the gourmets are wine drinkers.

F5 Some of the Frenchmen in the restaurant are wine drinkers.

To speed the exposition, let us assume that the Reason programmer has asserted these into knowledgebase `KB2`, using the expected infix syntax of first-order logic, so that, for example, F3 becomes (where of course we are using `forall` for $\forall$):

```
(forall x (if (Frenchman x) (Gourmet x)))
```

In addition, let us suppose that Reason can once again operate in oracular fashion at the level of propositional reasoning, that the flag for displaying countermodels has been activated, and that we have introduced the names/constants `object-1` and `object-2` to Reason for this session. Given this, please study the following interaction.

```
> (proof P2 KB2
    begin
      assume (and (Frenchman object-1) (Gourmet object-1)
              (not (Wine-drinker object-1)));
      assume (and (Gourmet object-2) (In-restaurant object-2)
             (Wine-drinker object-2));
      not-provable (F3 F4 F5) => (some x (and (In-restaurant x)
             (Wine-drinker x)))
    end)
PROOF P2 VERIFIED
DISPLAY-COUNTERMODEL?
```

المنارة للاستشارات
www.manaraa.com

```
> Y
```

In the situation the programmer has imagined, all Frenchmen are gourmets, and there exists someone who is both a wine-drinker and a gourmet. This locks in the truth of both of the first two statements. Yet, it is *not* true that there exists someone who is both a Frenchman and a wine drinker. This means that F5 is false; more generally put, we have that F5 is not a valid deduction from the conjunction of F3 and F4.

When counter-examples are rendered in visual form, they can be more quickly grasped by human programmers. Figure 5.1 shows such a counter-example relevant to the present case.

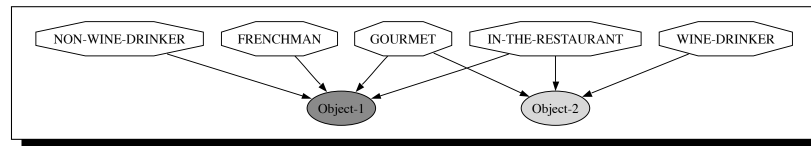


Figure 5.1: Visual Countermodel in Wine Drinker Puzzle (provided via a grammar and program written by Andrew Shilliday and Joshua Taylor that Reason can call). The reuse of this figure here, from Bringsjord (2008a), is permitted by Selmer Bringsjord.

5.5 The Future of Reason

The full and fully stable implementation of Reason is not yet complete. Fortunately, this implementation is not far off, as it is aided by the fact that this implementation is to a high degree meta-programming over computational building blocks that have been provided by others.[27] For example, resolution-based deduction is computed by the automated theorem provers Vampire (Voronkov, 1995), Otter (Wos, 1996; Wos et al., 1992), and SNARK (Stickel et al., 1994) (and others as well), while natural deduction-style automated theorem proving is provided by the Oscar system invented by philosopher Pollock (1989, 1995). As to automated model finding for first-order logic, a number of mature, readily available systems now compute this relation as well, for example Paradox and Mace (Claessen & Sorensson, 2003). At the propositional level, truth-value assignments are automatically found by many SAT solvers (e.g., see Kautz & Selman, 1999).

[27] Bringsjord & Ferrucci (1998) include a discussion of meta-programming in the logic programming field.

www.manaraa.com

While it seems sensible to strive for teaching clear thinking via programming languages that, by their very nature, are more intimately connected to the formalisms and processes that (from the perspective of logic, anyway) constitute clear thinking, noting this is not sufficient, obviously. One needs to empirically test determinate hypotheses. We need, specifically, to test the hypothesis that students who learn to program in Reason will as a result show themselves to be able, to a higher degree, to solve the kind of problems that are resistant to context-dependent reasoning. Accordingly, empirical studies of the sort we have carried out for other systems (e.g., Rinella et al., 2001; Bringsjord et al., 1998) are being planned for Reason. Finally, we are considering adding to curricula in which Reason is taught, explicit explanation, for students, of simplified versions of the Curry-Howard Isomorphism (Girard, 1989), a cluster of theorems and mappings that grounds the proofs-as-programs approach.

www.manaraa.com

6. THE FUTURE

6.1 A Necessary Confession

Certainly the research program of which this dissertation is a part is still embryonic. As stated at the outset, the goal for the dissertation was to take appreciable steps toward something that does not exist: a cognitive science of computer programming. Overall, we conclude, humbly, that the work reported above shows that we are making some significant but still-small progress toward meeting challenges **E** and **T**, and vindicating our core, driving approach of meta-cognitivism. We are far from being able to offer students an efficacious learn-to-program environment rooted in formalisms for representing programs and programmers, or in other words a "commercial-grade" K–12 programming environment rooted in meta-cognitivm; and we are far from being able to contribute to AP on the strength of dissecting human programming_2 ingenuity. But the springboard has been built by the results achieved and reported herein.

There is in particular much opportunity for future experiments that present subjects with *extremely* difficult programming problems, and it is perhaps in analyzing the cognition constitutive of solving such problems that real fruit for the advance of AP will be found. These problems would presumably be those which are such that, as far as anyone can tell, infinitary concepts and constructions are necessary as the human moves toward producing a program. We do have such examples in formal logic. For example, there currently is no finitary way of proving that certain theorems (e.g., Goodstein's Theorem) which are independent of Peano Arithmetic (therefore making Peano Arithmetic incomplete) are nonetheless true and provable.[28] It would be very interesting, and perhaps quite revealing, to pose programming problems that require the kind of computational ingenuity required to see that Goodstein's Theorem holds.[29]

6.2 Creativity, Computer Programming, and Automatic Programming

We begin this section by noting the brute fact that in AI, and in computational cognitive modeling (CCM), which, as we have noted earlier in the dissertation, is a direct incarnation of cognitivism, creativity is for the most part not investigated.

[28] A summary of Goodstein's Theorem is provided by Smith (2013).

[29] Smith (2007) provides a nice overview of Goodstein's Theorem in the context of Gödelian incompleteness.

المنارة للاستشارات
www.manaraa.com

The field of artificial intelligence (AI) is ultimately devoted to mechanizing human (and, for some, animal[30]) cognition. As Charniak & McDermott (1985, p. 7) express it in their classic introduction to the field:

> The ultimate goal of AI, which we are very far from achieving, is to build a person, or, more humbly, an animal.

Focusing on the less humble goal, we note that one of the hallmarks of (biologically normal) human persons is that they are creative; in some cases, *very* creative. And one of the hallmarks of the particular sciences (as well as, for that matter, of the humanities) is that human creativity is greatly, perhaps even singularly, revered, and desired. Whether it's the recent proof of Fermat's Last Theorem from Wiles (Wiles & Taylor, 1995; Wiles, 1995), *Othello* from Shakespeare, the theory of relativity from Einstein, or the minimum spanning tree algorithm from Prim (1957), it is the stunningly creative achievement that occupies a near-divine place in human culture. Even in everyday human life, far from the rarefied reaches of the particular sciences, creativity abounds, and is venerated: the 7th grade teacher who invents a new step-by-step approach to algebra word problems for his students, the 7th grade *student* who writes a short story never seen before, the cook who invents a new dish on the fly, the undergrad in *Comp Sci 101* who produces elegant code to compute some function, and so on, *ad infinitum*.

Despite the centrality and status of human creativity, nowhere do Charniak & McDermott (1985) cover the attempt to mechanize creativity; the index is devoid of the topic. Things are no less peculiar if we fast-forward to the present time: The dominant textbook today for introducing students to AI, a book which, whatever its defects might be, is frequently praised as remarkably comprehensive, is *Artificial Intelligence: A Modern Approach* (Russell & Norvig, 2009); and nowhere in this hefty tome is creativity discussed. In parallel with its predecessor from over two decades back, you will search in vain for `creativity` in the index.[31]

[30] In this dissertation, following the approach of Bringsjord and Arkoudas, as the reader will have noticed, the focus is on the *human* case. For weal or woe, this focus is typical of work in cognitive science (e.g., see von Eckardt, 1995), and in AI, as far as we can tell, work on creativity has traditionally been at the level of humans.

[31] Though we don't include full discussion here, the situation is fundamentally the same in what we called *cognitivism*. E.g., consider computational cognitive modeling (CCM), the part of cognitive science that is AI's sister field. CCM is devoted to providing computational simulations of various aspects of human cognition. Anderson & Lebiere (2003) offer an assessment of the attempt to computationally model, in working computer programs, "all" of human cognition. (They see this attempt as being inaugurated by Newell.) Remarkably, nowhere in the assessment is there a discussion of the degree to which human creativity has been computationally modeled. Were the assessment to be included, the result would be

المنارة للاستشارات
www.manaraa.com

6.2.1 An Exception: Story Generation

Despite what the AI textbooks indicate, some computationally oriented researchers have devoted considerable time and energy to the attempt to build a machine capable of creative behavior.[32] An example can be found in some of the work Bringsjord and colleagues have carried out (Bringsjord & Ferrucci, 2000; Bringsjord et al., 2001). This work was targeted specifically at mechanizing (humble forms of) *literary* creativity; specifically storytelling, or, as it is known in AI, *story generation*. In personal conversation, Peter Norvig and others in AI have remarked to Bringsjord since the advent of the Brutus story generation system that, despite claims by some that storytelling is at the very heart of human cognition (e.g., Schank, 1995), the brute empirical fact remains that storytelling is not at the heart of what computer scientists and AIniks are trained to do or in practice do, and story generation systems don't appear to offer a capability that would be a helpful addition to the toolkit of those attempting to build ever smarter machines. Whether or not these observations constitute fatal objections to the pursuit of automatic storytelling, they *are* certainly observations — and we have reflected upon them.[33]

As far as we can tell, None of the extant work in automatic programming has attempted to tackle the problem that drives this field by facing up to the fact that computer programming (on significant problems) requires remarkable creativity of human programmers. The underlying reason why creativity is required is that the programming problem is, as pointed out in §4.2 , a Turing-unsolvable problem.[34] Our suspicion, which goes back to the earliest days of thinking about these issues by Li and collaborators, is that whenever one attempts to render ϕ-ing from the human sphere in the form of computation, where ϕ-ing requires humans to solve problems the general form of which is Turing-unsolvable, cognitive science and computer science (particularly AI) will intersect. In our own case, the plan to exploit an understanding of human creativity in computer programming in order to advance automatic programming is clearly in this intersection.

that CCM has achieved exceedingly little in this direction.

[32]There is of course a large literature on creativity from the perspective of philosophy, psychology/psychometrics, and so on, but we are specifically concerned for the moment with explicit attempts to engineer creative machines.

[33]We suspect that complaints to the effect that story generation systems aren't exactly central to computation-based fields (like AI, computer science, large parts of cognitive science (e.g., CCM; see note 31, etc.) would be expressed by many against many other types of creative computational systems. For example, AARON (Cohen, 1995), the computational system that, instead of generating stories, generates images, is subject to the same complaints as those lodged against Brutus.

[34]Recall that automatic programming, which surely requires making decisions as to whether two computer programs compute the same function, is harder than the halting problem.

المنارة للاستشارات
www.manaraa.com

6.3 Programming East versus West

There is an obvious need to in the future extend our research program in such as way as to address the following four questions:

1. Are programming languages the same across the East and West? If not, what are the differences between these languages?

2. What about languages used to *teach* programming in the early grades, East versus West?

3. Given that Chinese characters number in the tens of thousands (80,000 characters, be conventional wisdom, are needed to achieve $\approx$ 98% coverage of Chinese linguistic communication), whereas English, in a parallel with the austerity of characters seen in mainsteam programming languages, uses very few (there are, after all, only 26 letters in the English alphabet), what are the consequences of this great and vast difference for the learning of programming by young minds in the East, versus the West?[35]

4. Nisbett (2004) has argued, by appeal to empirical evidence, that reasoning differs between East and West.[36] What are the consequences of this view for the cognitive science of computer programming?

It will be exciting to tackle these questions in the future, and the present work provides a foundation for doing so.

6.4 Final Remarks

In conclusion, then, given the work accomplished so far, it is perhaps safe to say that the future now appears bright, but there is a lot of work to be done to further advance the cognitive science of computer programming, and to harness that science so as to better educate programmers.

[35]This question is presumably greatly colored by the fact that there is no programming language, at least not one known to us or in the literature, that is a "natively Chinese" one.

[36]E.g., he specifically claims that cognizers in the East are much more willing to tolerate contradictions than those in Occidental culture. Given that at least mainstream programming langauges (as well as the sub-field of computer science known as 'programming languages' (or just 'PL' for short) are based on the outright unacceptability of contradictions, Nisbett's claim is certainly very interesting.

www.manaraa.com

LITERATURE CITED

Abelson, H., & Sussman, G. (1996). *Structure and interpretation of computer programs* (5th ed.). Cambridge, MA: MIT Press.

Adams, E. (1998). *A Primer of Probability Logic*. Stanford, CA: Center for the Study of Language and Information (CSLI).

Al-Tahat, K. (2014). An innovative instructional method for teaching object-oriented modelling. *International Arab Journal of Information Technology*, *11*(6), 540–549.

Anderson, J., & Lebiere, C. (2003). The Newell test for a theory of cognition. *Behavioral and Brain Sciences*, *26*(5), 587–601.

Anderson, J. R. (1993). *Rules of the mind*. Hillsdale, NJ: Lawrence Erlbaum.

Anderson, J. R., Corbett, A., Koedinger, K. R., & Pelletier, R. (1995). Cognitive tutors: Lessons learned. *Journal of the Learning Sciences*, *4*(2), 167–207.

Anderson, J. R., & Lebiere, C. (1998). *The Atomic Components of Thought*. Hillsdale, NJ: Lawrence Erlbaum.

Arkoudas, K. (2000). *Denotational proof languages*. (Unpublished doctoral dissertation). Massachusetts Institute of Technology, Cambridge, MA.

Arkoudas, K., & Bringsjord, S. (2007). Computers, justification, and mathematical knowledge. *Minds and Machines*, *17*(2), 185–202.

Arkoudas, K., & Bringsjord, S. (2009). Vivid: An AI framework for heterogeneous problem solving. *Artificial Intelligence*, *173*(15), 1367–1405.

Ashcraft, M. (1994). *Human memory and cognition*. New York, NY: HarperCollins.

Barwise, J., & Etchemendy, J. (1999). *Language, proof, and logic*. New York, NY: Seven Bridges.

Bergmann, M., Moor, J., & Nelson, J. (1997). *The Logic Book*. New York, NY: McGraw Hill.

المنارة للاستشارات
www.manaraa.com

Black, J., Swan, K., & Schwartz, D. (1988). Developing thinking skills with computers. *Teachers College Record*, *89*(3), 384–407.

Boolos, G. S., Burgess, J. P., & Jeffrey, R. C. (2003). *Computability and Logic (Fourth Edition)*. Cambridge, UK: Cambridge University Press.

Boolos, G. S., & Jeffrey, R. C. (1989). *Computability and logic*. Cambridge, UK: Cambridge University Press.

Bourdeau, R. N. J., & Mizoguchi, R. (Eds.). (2010). *Advances in intelligent tutoring systems*. New York, NY: Springer.

Bringsjord, S. (1994). Computation, among other things, is beneath us. *Minds and Machines*, *4*(4), 469–488.

Bringsjord, S. (2008a). Declarative/Logic-Based Cognitive Modeling. In R. Sun (Ed.) *The Handbook of Computational Psychology*, (pp. 127–169). Cambridge, UK: Cambridge University Press.

Bringsjord, S. (2008b). The Logicist Manifesto: At Long Last Let Logic-Based AI Become a Field Unto Itself. *Journal of Applied Logic*, *6*(4), 502–525.

Bringsjord, S., Arkoudas, K., & Bello, P. (2006a). Toward a general logicist methodology for engineering ethically correct robots. *IEEE Intelligent Systems*, *21*(4), 38–44.

Bringsjord, S., Bringsjord, E., & Noel, R. (1998). In defense of logical minds. In Proceedings from *The 20th Annual Conference of the Cognitive Science Society* (pp. 173–178). Hillsdale, NJ: Lawrence Erlbaum.

Bringsjord, S., & Ferrucci, D. (1998). Logic and artificial intelligence: Divorced, still married, separated...? *Minds and Machines*, *8*(2), 273–308.

Bringsjord, S., & Ferrucci, D. (2000). *Artificial intelligence and literary creativity: Inside the mind of Brutus, a storytelling machine*. Hillsdale, NJ: Lawrence Erlbaum.

Bringsjord, S., Ferrucci, D., & Bello, P. (2001). Creativity, the Turing test, and the (better) Lovelace test. *Minds and Machines*, *11*(1), 3–27.

Bringsjord, S., Kellett, O., Shilliday, A., Taylor, J., van Heuveln, B., Yang, Y., Baumes, J., & Ross, K. (2006b). A new Gödelian argument for hypercomputing minds based on the Busy Beaver problem. *Applied Mathematics and Computation*, *176*(2), 516–530.

المنارة للاستشارات
www.manaraa.com

Bringsjord, S., & Li, J. (2008). Toward aligning computer programming with clear thinking via the Reason programming language. In Briggle, A., Waelbers, K., & Brey, P.A.E. (Eds.), *Current Issues in Computing and Philosophy* (pp. 156–170). Amsterdam, The Netherlands: IOS Press.

Bringsjord, S., Li, J., Govindarajulu, N., & Arkoudas, K. (2012). On the cognitive science of computer programming in the service of two historic challenges. In De Mol, L. & Primiero, G. (Eds.), Proceedings from AISB/IACAP World Congress 2012: *Symposium on the History and Philosophy of Programming* (pp. 17–23). Birmingham, UK: The Society for the Study of Artificial Intelligence and Simulation of Behaviour.

Bringsjord, S., & Yang, Y. (2003). Logical illusions and the welcome psychologism of logicist artificial intelligence. In D. Jacquette (Ed.), *Philosophy, psychology, and psychologism: Critical and historical essays on the psychological turn in philosophy* (pp. 289–312). Dordrecht, The Netherlands: Kluwer.

Bringsjord, S., & Zenzen, M. (2003). *Superminds: People harness hypercomputation, and more*. Dordrecht, The Netherlands: Kluwer.

Bumby, Klutch, Collins, & Egbers (1995). *Integrated mathematics course 1*. New York, NY: Glencoe/McGraw Hill.

Charniak, E., & McDermott, D. (1985). *Introduction to artificial intelligence*. Reading, MA: Addison-Wesley.

Claessen, K., & Sorensson, N. (2003). New techniques that improve MACE-style finite model finding. In P. Baumgartner, & C. Fermueller (Eds.), Proceedings from the CADE-19 Workshop: *Model Computation — Principles, Algorithms, Applications* (pp. 11-27). Miami Beach, FL: Springer.

Clocksin, W., & Mellish, C. (2003). *Programming in Prolog (using the ISO standard)* (5th ed.). New York, NY: Springer.

Cohen, H. (1995). The further exploits of AARON, painter. *Stanford Humanities Review*, *4*(2), 141–158.

Copeland, B. J. (1998). Even Turing machines can compute uncomputable functions. In C. S. Calude, J. Casti, & M. J. Dinneen (Eds.), *Unconventional models of computation* (pp. 150–164). London, UK: Springer-Verlag.

المنارة للاستشارات
www.manaraa.com

Copi, I., & Cohen, C. (1997). *Introduction to logic* (10th ed.). Englewood Cliffs, NJ: Prentice-Hall.

Davis, M., Sigal, R., & Weyuker, E. (1994). *Computability, complexity, and languages: Fundamentals of theoretical computer science.* New York, NY: Morgan Kaufmann.

Ebbinghaus, H. D., Flum, J., & Thomas, W. (1994). *Mathematical logic* (2nd ed.). New York, NY: Springer-Verlag.

Elderton, W. P. (1902). Tables for testing the goodness of fit of theory to observation. *Biometrika*, *1*(2), 155–163.

Etesi, G., & Nemeti, I. (2002). Non-Turing computability via Malament-Hogarth space-times. *International Journal of Theoretical Physics*, *41*(2), 341–370.

Feurzeig, W. (2010). Toward a culture of creativity: A personal perspective on Logo's early years and ongoing potential. *International Journal of Computers for Mathematical Learning*, *15*(3), 257–265.

Fisher, R. A. (1922). On the mathematical foundations of theoretical statistics. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, *222*, 309–368.

Fitch, F. B. (1952). *Symbolic logic: An introduction.* New York, NY: The Ronald Press Company.

Gentzen, G. (1935). Untersuchungen über das logische Schließen. *Mathematische Zeitschrift*, *39*(1), 176–210, 405–431.

Girard, J.-Y. (1989). *Proofs and types.* (P. Taylor & Y. Lafont, Trans.). Cambridge, UK: Cambridge University Press.

Glymour, C. (1992). *Thinking things through.* Cambridge, MA: MIT Press.

Hamkins, J. D., & Lewis, A. (2000). Infinite time Turing machines. *Journal of Symbolic Logic*, *65*(2), 567–604.

Harvey, B., & Mönig, J. (2013). *Snap! Reference manual version 4.0.* Berkeley, CA: University of California at Berkeley.

المنارة للاستشارات

www.manaraa.com

Inhelder, B., & Piaget, J. (1958). *The growth of logical thinking from childhood to adolescence*. New York, NY: Basic Books.

Jain, A. K., Singhal, M., & Gupta, M. S. (2011). Algorithm building and learning programming languages using a new educational paradigm. *AIP Conference Proceedings*, *1373*, 149–158.

Jain, S., Osherson, D., Royer, J., & Sharma, A. (1999). *Systems That Learn: An Introduction to Learning Theory, Second Edition*. Cambridge, MA: MIT Press.

Jaśkowski, S. (1934). On the rules of suppositions in formal logic. *Studia Logica*, *1*, 5–32.

Johnson-Laird, P. (1997). Rules and illusions: A critical study of Rips's "The psychology of proof." *Minds and Machines*, *7*(3), 387–407.

Johnson-Laird, P., & Savary, F. (1995). How to make the impossible seem probable. In M. Gaskell, & W. Marslen-Wilson (Eds.), Proceedings from *The 17th Annual Conference of the Cognitive Science Society*, (pp. 381–384). Hillsdale, NJ: Lawrence Erlbaum.

Johnson-Laird, P. N., Legrenzi, P., Girotto, V., & Legrenzi, M. S. (2000). Illusions in reasoning about consistency. *Science*, *288*(5465), 531–532.

Kahneman, D. (2013). *Thinking, fast and slow*. New York, NY: Farrar, Straus, and Giroux.

Kautz, H., & Selman, B. (1999). Unifying SAT-based and graph-based planning. In Proceedings from IJCAI'99: *The 16th International Joint Conference on Artificial Intelligence* (pp. 318–325). New York, NY: Morgan Kaufmann.

Khasawneh, A. A. (2009). Assessing Logo programming among jordanian seventh grade students through turtle geometry. *International Journal of Mathematical Education in Science & Technology*, *40*(5), 619–639.

Kitzelmann, E., Schmid, U., & Kaelbling, L. P. (2006). Inductive synthesis of functional programs: An explanation based generalization approach. *Journal of Machine Learning Research*, *7*, 429–454.

Kleinberg, J., & Tardos, E. (2005). *Algorithm design*. Boston, MA: Addison-Wesley.

المنارة للاستشارات

www.manaraa.com

Kodratoff, Y., Franová, M., & Partridge, D. (1989). Why and how program synthesis? In K. P. Jantke (Ed.), *Analogical and Inductive Inference* (pp. 45–59). Berlin, Germany: Springer-Verlag.

Koza, J. (1992). *Genetic programming: On the programming of computers by means of natural selection*. Cambridge, MA: MIT Press.

Kugel, P. (1986). Thinking may be more than computing. *Cognition*, *22*, 137–198.

Kurland, D. M., Clement, C. A., Mawby, R., & Pea, R. D. (1987). Mapping the cognitive demands of learning to program. In D. Perkins, J. Bishop, & J. Lochhead (Eds.), *Thinking: Progress in research and teaching* (pp. 103–127). Hillsdale, NJ: Lawrence Erlbaum.

Lepore, E., & Pylyshyn, Z. (Eds.). (1999). *What is cognitive science?*. Oxford, UK: Blackwell.

Lewis, H., & Papadimitriou, C. (1981). *Elements of the theory of computation*. Englewood Cliffs, NJ: Prentice Hall.

Lieberman, H. (1993). Tinker: a programming by demonstration system for beginning programmers. In A. Cypher (Ed.), *Watch What I Do: Programming by Demonstration*(pp. 49–64). Cambridge, MA: MIT Press.

Louden, K. (1989). Logo as a prelude to Lisp: Some surprising results. *ACM SIGCSE Bulletin*, *21*(3), 35–38.

McHugh, M. L. (2013). The chi-square test of independence. *Biochemia Medica*, *23*(2), 143–149.

McKeon, R. (Ed.). (1941). *The basic works of Aristotle*. New York, NY: Random House.

Moore, R. C. (1994). Making the transition to formal proof. *Educational Studies in Mathematics*, *27*(3), 249–266.

Muggleton, S. (1992). Inductive logic programming. In S. Muggleton (Ed.), *Inductive logic programming* (pp. 3–27). San Diego, CA: Academic Press.

Nisbett, R. (2004). *The geography of thought: How Asians and Westerners think differently . . . and why*. New York, NY: Free Press.

المنارة للاستشارات
www.manaraa.com

ObjectPlanet, I. (2014). Opinio. Http://www.objectplanet.com/opinio/ (Date Last Accessed October 10,2014).

Olsson, J. R. (1998). The art of writing specifications for the ADATE automatic programming system. In J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, . . . H. I. R. Riolo (Eds.), Proceedings from *The Annual Genetic Programming Conference* (pp. 278–283). San Francisco, CA: Morgan Kaufmann.

Papert, S. (1980). *Mindstorms: Children, computers, and powerful ideas.* New York, NY: Basic Books.

Patt, Y. N., & Patel, S. J. (2004). *Introduction to computing systems: From bits and gates to C and beyond* (2nd ed.). New York, NY: McGraw-Hill.

Pea, R. D., & Kurland, D. M. (1984). On the cognitive effects of learning computer programming. *New Ideas in Psychology 2*(2), 137–168.

Pea, R. D., Kurland, D. M., & Hawkins, J. (1987). Logo and Development of Thinking Skills, In R. Pea & K.Sheingold (Eds.), *Mirrors of minds: Patterns of experience in educational computing*(pp. 178–197). Norwood, NJ: Ablex Publishing Corp.

Pearson, K. (1900). On the criterion that a given system of deviations from the probable in the case of a correlated system of variables is such that it can be reasonably supposed to have arisen from random sampling. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, *50*(302), 157–175.

Pearson, K. (1904). *On the theory of contingency and its relation to association and normal correlation.* London, UK: Dulau and Co.

Penrose, R. (1994). *Shadows of the Mind.* Oxford, UK: Oxford University Press.

Pollock, J. (1989). *How to build a person: A prolegomenon.* Cambridge, MA: MIT Press.

Pollock, J. (1995). *Cognitive carpentry: A blueprint for how to build a person.* Cambridge, MA: MIT Press.

Prim, R. C. (1957). Shortest connection networks and some generalizations. *Bell System Technical Journal*, *36*(6), 1389–1401.

Resnick, K., Maloney, J., & et al (2009). Scratch: Programming for all. *Communications of the ACM*, *52*(11), 60–67.

المنارة للاستشارات
www.manaraa.com

Rich, C., & Waters, R. C. (1988). Automatic programming: Myths and prospects. *Computer*, *21*(8), 40–51.

Rinella, K., Bringsjord, S., & Yang, Y. (2001). Efficacious logic instruction: People are not irremediably poor deductive reasoners. In J. D. Moore, & K. Stenning (Eds.), Proceedings from *The 23rd Annual Conference of the Cognitive Science Society* (pp. 851–856). Hillsdale, NJ: Lawrence Erlbaum.

Russell, S., & Norvig, P. (2009). *Artificial intelligence: A modern approach* (3rd ed.). Upper Saddle River, NJ: Prentice Hall.

Schank, R. (1995). *Tell me a story*. Evanston, IL: Northwestern University Press.

Shapiro, S. (1992). *Common Lisp: An interactive approach*. New York, NY: W. H. Freeman.

Siegelmann, H., & Sontag, E. (1994). Analog computation via neural nets. *Theoretical Computer Science*, *131*(2), 331–360.

Siegelmann, H. T. (1999). *Neural networks and analog computation: Beyond the Turing limit*. Boston, MA: Birkhäuser.

Smith, P. (2007). *An introduction to Gödel's theorems*. Cambridge, UK: Cambridge University Press.

Smith, P. (2013). *An introduction to Gödel's theorems* (2nd ed.). Cambridge, UK: Cambridge University Press.

Solomon, C. J., & Papert, S. (1976). A case study of a young child doing turtle graphics in Logo. In Proceedings from AFIPS'76: *The National Computer Conference and Exposition* (pp. 1049–1056). New York, NY: ACM.

Stanovich, K. E., & West, R. F. (2000). Individual differences in reasoning: Implications for the rationality debate. *Behavioral and Brain Sciences*, *23*(5), 645–665.

Stickel, M., Waldinger, R., Lowry, M., Pressburger, T., & Underwood, I. (1994). Deductive composition of astronomical software from subroutine libraries. In Proceedings from CADE-12: *The 12th International Conference on Automated Deduction* (pp. 341–355). New York, NY: Springer.

المنارة للاستشارات
www.manaraa.com

Summers, P. D. (1977). A methodology for Lisp program construction from examples. *Journal of the ACM*, *24*(1), 161–175.

Sun, R. (1999). Accounting for the computational basis of consciousness: A connectionist approach. *Consciousness and Cognition*, *8*(4), 529–565.

Turing, A. M. (1937). On Computable Numbers with Applications to the *Entscheidungsproblem*. *Proceedings of the London Mathematical Society*, *42*(2), 230–265.

Vaikakul, S. (2005). *Examining pervasive technology practices in schools: A mental models approach.* (Unpublished doctoral dissertation). Massachusetts Institute of Technology, Cambridge, MA.

von Eckardt, B. (1995). *What is cognitive science?* Cambridge, MA: MIT Press.

Voronkov, A. (1995). The anatomy of Vampire: Implementing bottom-up procedures with code trees. *Journal of Automated Reasoning*, *15*(2), 237–265.

Wason, P. C. (1966). Reasoning. In B. M. Foss (Ed.), *New horizons in psychology* (pp. 135–151), Baltimore, MD: Penguin Books.

Wiles, A. (1995). Modular elliptic curves and Fermat's last theorem. *Annals of Mathematics*, *141*(3), 443–551.

Wiles, A., & Taylor, R. (1995). Ring-theoretic properties of certain Hecke algebras. *Annals of Mathematics*, *141*(3), 553–572.

Wos, L. (1996). *The automation of reasoning: An experimenter's notebook with Otter tutorial.* San Diego, CA: Academic Press.

Wos, L., Overbeek, R., Lusk, E., & Boyle, J. (1992). *Automated reasoning: Introduction and applications.* New York, NY: McGraw Hill.

Wysotzki, F. (1986). Program synthesis by hierarchical planning. In P. Jorrand, & V. Sgurev (Eds.), *Artificial intelligence: Methodology, systems, applications* (pp. 3–11). Amsterdam, The Netherlands: Elsevier.

Yates, F. (1934). Contingency tables involving small numbers and the chi square test. *Supplement to the Journal of the Royal Statistical Society*, *1*(2), 217–235.

المنارة للاستشارات

www.manaraa.com

APPENDIX A
Experiment 1 Programming Problems

Description

Two programming problems are given below, and involve working with a very simple language, $\mathcal{L}$, a fragment of English. The words used in $\mathcal{L}$ are:

{Bill, Jane, likes, chases, makes, a, the, man, woman, cat, happy, thin, quickly}

Bill, Jane, man, woman, and cat, are nouns; happy and thin are adjectives; likes, chases, and makes are verbs; a and the are determiners; and quickly is an adverb.

The following grammar defines the sentences of $\mathcal{L}$.

S	::=	NOUN VERB NOUN	R1
	|	DET NOUN VERB NOUN	R2
	|	DET ADJ* NOUN VERB DET ADJ* NOUN	R3
	|	DET ADJ* NOUN ADV VERB DET ADJ* NOUN	R4
	|	DET ADJ* NOUN VERB DET ADJ* NOUN ADV	R5

where NOUN stands for any noun, VERB stands for any verb, DET stands for any determiner, ADV stands for any adverb, ADJ stands for any adjective, ADJ* stands for zero, one, or more adjectives, and S stands for a well-formed sentence of $\mathcal{L}$.

For instance, the sequence ⟨the, thin, cat, makes, a, Bill⟩ is a sentence of $\mathcal{L}$, because cat and Bill are nouns, likes is a verb, the and a are determiners, and thin is an adjective; so the sequence has the form DET ADV* NOUN VERB DET ADJ* NOUN, which, by rule R3, is a sentence of $\mathcal{L}$. (Notice that the first ADJ* is matched with the one adjective thin, while the second ADJ* is matched with the lack of adjectives between a and Bill.)

In each of the two problems you will be asked to write a program. You may use one of the following programming languages: BASIC; C; C++; Java; Lisp; Pascal; or you may use pseudo-code to describe your program in detail. Please indicate whether you are using pseudo-code or a programming language to solve the problem, and, if using a programming language, which programming language you are using.

المنارة للاستشارات
www.manaraa.com

Problem 1

Write a program P that takes as input a (finite) sequence of words used in $\mathcal{L}$ and outputs **yes** if the sequence is a sentence of $\mathcal{L}$, and outputs **no** otherwise. For example, given the sequence $\langle \mathsf{Bill}, \mathsf{likes}, \mathsf{Jane} \rangle$, P should output **yes** because the sequence is a sentence, according to R1. When given $\langle \mathsf{Bill}, \mathsf{Jane}, \mathsf{likes} \rangle$, P should output **no**, because this sequence is not a sentence of $\mathcal{L}$.

Problem 2

Write a program P that takes as input a (finite) sequence of words used in $\mathcal{L}$ and outputs **yes** if the sequences is a palindrome sentence of $\mathcal{L}$, and outputs **no** otherwise. A palindrome sentence is a sentence which reads the same in both directions. For example, given the sequence $\langle \mathsf{Bill}, \mathsf{likes}, \mathsf{Bill} \rangle$, P should output **yes**, since the sequence is a palindrome sentence. When given $\langle \mathsf{the}, \mathsf{cat}, \mathsf{likes}, \mathsf{Jane} \rangle$, P should output **no**, since the sequence, although a sentence, is not a palindrome sentence.

المنارة للاستشارات
www.manaraa.com

APPENDIX B
Experiment 2 Programming Problems

Description

First some basic terms: A logic gate is a basic building block of logic circuits. Each logic gate implement a Boolean function on one or more logic inputs and produces a single logic output (True or False). There are three basic logic gates: *NOT*, *OR* and *AND*.

- An *AND* gate acts in the same way as the logical *AND* operator. Given two inputs *A* and *B*, the output *O* is *true* when both *A* and *B* are *true*. Otherwise, the output *O* is *False*.This can be shown diagrammatically as in Figure B.1.

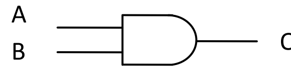


Figure B.1: *AND* Gate ($O = A \wedge B$).

The *AND* gate corresponds to a formula in the propositional calculus:

$$O = A \wedge B$$

- An *OR* gate acts in the same way as the logical *OR* operator. The output *O* is *true* if either or both of the inputs (*A*,*B*) are *true*. If both inputs are *false*, then the output *O* is *false*.

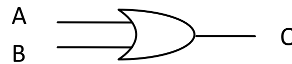


Figure B.2: *OR* Gate ($O = A \vee B$).

- A *Not* gate is a logic inverter. Assume an input *A*, the output *O* is the reversed logic state of input *A*.

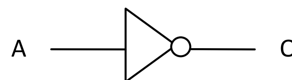


Figure B.3: *NOT* Gate ($O = \neg A$).

المنارة للاستشارات
www.manaraa.com

These are shown in Figures B.2 and B.3, and can be represented by formulae in the propositional calculus:

$$O = A \vee B$$

$$O = \neg A$$

repectively.

- Now, a *Programmable Logic Array (PLA)* consists of a programmable array of *AND* gates and a programmable array of *OR* gates, which can then be conditionally complemented to produce an output. Figure B.4 shows an example of pre-fabricated blocks of a PLA of three inputs and four outputs where all possible connections are available before programming.

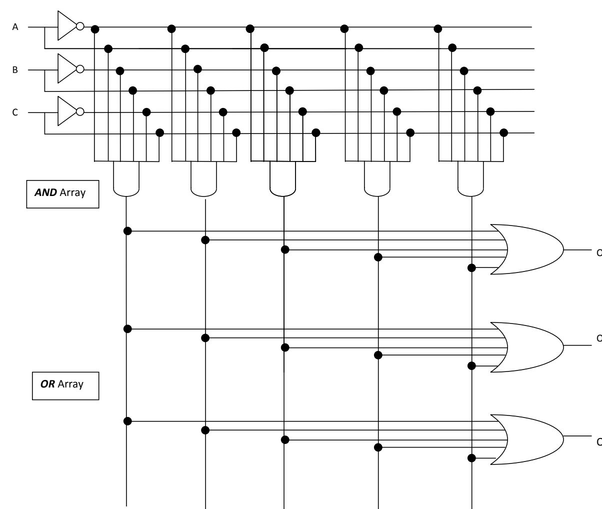


Figure B.4: An Example of Programmable Logic Array (PLA)

$$O_1 = o_1(A, B, C)$$

$$O_2 = o_2(A, B, C)$$

$$O_3 = o_3(A, B, C)$$

المنارة للاستشارات
www.manaraa.com

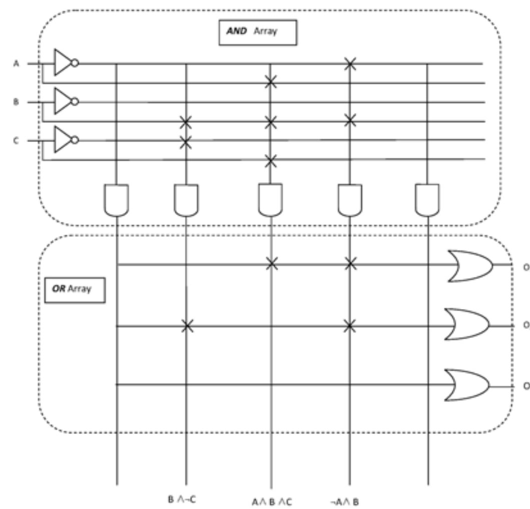


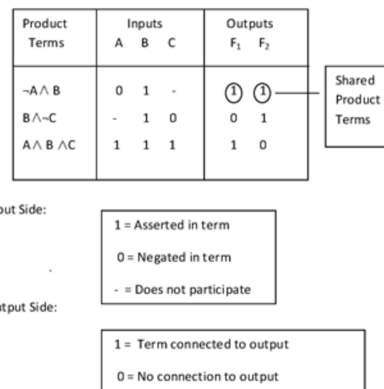
Figure B.5: An example of implementation of two functions (Xs mark connections).

An alternative representations of implementing two functions with unwanted connections blow out; connections marked by X (see Figure B.5).

$$O_1 = \neg A \wedge B \vee A \wedge B \wedge C$$

$$O_2 = B \wedge \neg C \vee \neg A \wedge B$$

The graphics presentations of PLA structure can also be represented as Figure B.6.



Product Terms	Inputs A	B	C	Outputs F_1	F_2
¬A∧ B	0	1	-	1	1
B∧¬C	-	1	0	0	1
A∧ B ∧C	1	1	1	1	0

Figure B.6: An example of implementation of two equations (matrix).

المنارة للاستشارات
www.manaraa.com

Programming Problems

1. Given a logic circuits as shown in Figure B.7 with three inputs (A, B, C). Please write a program to determine the output of the logic function of the O.

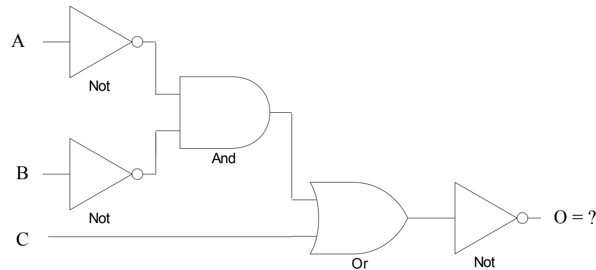


Figure B.7: Experiment 2. Programming Problem 1: A logic circuit with inputs (A, B, C) and output O.

2. For this problem, you are given a set of binary inputs: $(A_1, A_2, ..., A_n)$, please write a computer program that computes a function O $(A_1, A_2, ..., A_n)$, where $A_i = 0$,1 (i.e. false, true; see Figure B.8). You may use Programmable Logic Array (PLA) methods, with minimal number of AND gates.

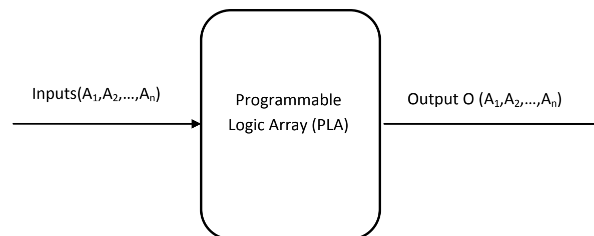


Figure B.8: Experiment 2. Programming Problem 2: A logic circuit with inputs (A, B, C) and a output O.

3. Given a set of functions below.

$$O_1 = A \vee \neg B \wedge C$$

$$O_2 = A \wedge \neg C \vee A \wedge B$$

$$O_3 = \neg B \wedge \neg C \vee A \wedge B$$

المنارة للاستشارات
www.manaraa.com

$$O_4 = \neg B \wedge \neg C \vee A$$

where (A, B, C) are inputs, (O_1, O_2, O_3, O_4) are outputs. Wrote a program to generate a logic that compute all above functions, using minimal number of AND gates. The output as represented as in Matrix with AND (product terms, input and outputs (see sample figure B.6).

للاستشارات المنارة
www.manaraa.com

APPENDIX C
Experiment 3 Programming Problems

Description [37]

The residents of the underground city of Thontville defend themselves through a combination of training, heavy artillery, and efficient algorithms. Recently, they become interested in automated methods that can help fend off attaches by swarms of robots. Here is what one of those robot attacks looks like.

- A swarm of robots arrives over the course of n seconds: in the ith second, x_i robots arrives. Based on remote sensing data, you know this sequences $x_1, x_2, \ldots, x_n$ advance.
- You have at your disposal an $electromagneticpulses(EMP)$, which can destroy some of the robots as they arrive; the EMP's power depends on how long it's been allowed to charge up. To make this precise, there is a function $f(.)$ so that if j seconds have passed since the EMP was last used, then it is capable of destroying up to $f(j)$ robots.
- So specifically, if it is used in the kth second, and it has been j seconds since it was previously used, then it will destroy $min(x_k, f(j))$ robots. (After this use, it will be completely drained.)
- We will also assume that the EMP starts off completely drained, so if it is used for the first time in the jth second, then it is capable of destroying up to $f(j)$ robots.

The problem: Given the data on robot arrivals $x_1, x_2, \ldots, x_n$, and given the recharging function $f(.)$, choose the points in time at which you are going to activate the EMP so as to destroy as many robots as possible.

Example: Suppose $n = 4$, and the values of x_i, and $f(i)$ are given by the following table:

I	L	2	3	4
x_i	1	10	10	1
$f(i)$	1	2	4	8

[37] The description and both of the problems were selected from Kleinberg & Tardos (2005, Question 8, Chapter 6, pp. 318–319).

المنارة للاستشارات
www.manaraa.com

The best solution would be to activate the EMP in the 3rd and the 4th seconds. In the 3rd second, the EMP has gotten to charge for 3 seconds, and so it destroys $min(10, 4) = 4$ robots; in the 4th second, the EMP has only gotten to charge for 1 second since its last use, and it destroys $min(1, 1) = 1$ robots. This is a total of 5.

Problem 1

Show that the following algorithm does not correctly solve this problem, by giving an instance on which it does not return the correct answer.

Schedule-EMP $(x_1, x_2, ..., x_n)$

Let j be the smallest number for which $f(j) => x_n$

(If no such j exists then set $j = n$)

Activate the EMP in the n_{th} second

If nâĂŞ$j => 1$ then

Continue recursively on the input $x_1, x_2, ..., x_{n-j}$

(i.e., invoke **Schedule-EMP**$(x_1, x_2, ..., x_{n-j})$)

In your example, say what the correct answer is and also what the above algorithm finds.

Problem 2

Given an algorithm that takes the data on Robot arrives $x_1, x_2, ..., x_n$ and the recharging function $f(.)$, and returns the maximum number of robots that can be destroyed by a sequences of EMP activations.

The running time of your algorithm should be polynomial in n. You should prove that your algorithm works correctly, and include a brief analysis of the running time.

المنارة للاستشارات

www.manaraa.com

APPENDIX D
List of Background Questions And Options

D.1 Background Information

- Identify your class year:
  - Undergraduate 1st Year
  - Undergraduate 2nd Year
  - Undergraduate 3rd Year
  - Undergraduate 4th Year
  - Graduate (Master Level)
  - Graduate (Doctoral Level)
  - Other
- Enrollment Status
  - Full Time
  - Part Time
- Gender
  - Male
  - Female
- Age
- Is English your native language?
  - Yes
  - No
- Do you identify yourself as having Hispanic origin?
  - Yes
  - No
- Please indicate your race (Check all that apply)

المنارة للاستشارات
www.manaraa.com

- White
- Black/African American
- Asian
- Native Hawaiian/Pacific Islander
- American Indian/Native Alaskan
- Other

D.2 Academic Information

- Academic Major
  - Current major: ______________________________
  - Second major (If applicable): ______________________________
  - Minor (If applicable): ______________________________
- GPA
  - Undergraduate GPA:
  - If a graduate student, enter Graduate GPA:
- Please enter the following test scores (if multiple, enter the highest score).
  - SAT Math
  - SAT Verbal
  - ACT Math
  - ACT Reading
  - ACT English
  - ACT Writing
  - ACT Science
  - ACT Composite
  - GRE Verbal
  - GRE Quantitative
  - GRE Writing

المنارة للاستشارات
www.manaraa.com

D.3 Education

D.3.1 Logic Education

- Check here if you have ever taken a logic class. If so, please answer questions below: Total number of logic courses taken:

- Please enter the information regarding your four most-recent logic classes:

  - Logic Course 1
    - Name: ____________
    - Calendar Year (e.g. 2006): ____________
    - Class Year (e.g., K-8, undergraduate 1st Year): ____________
    - School Name: ____________
    - Credit Hours: ____________
    - Grade: ____________
  - Logic Course 2
    - Name: ____________
    - Calendar Year: ____________
    - Class Year: ____________
    - School Name: ____________
    - Credit Hours: ____________
    - Grade: ____________
  - Logic Course 3
    - Name: ____________
    - Calendar Year: ____________
    - Class Year: ____________
    - School Name: ____________
    - Credit Hours: ____________
    - Grade: ____________
  - Logic Course 4
    - Name: ____________
    - Calendar Year: ____________

المنارة للاستشارات
www.manaraa.com

* Class Year: ______
* School Name: ______
* Credit Hours: ______
* Grade: ______

D.3.2 Computer Programming Language (PL) Education

- Check here if you have ever taken a computer programming language class.
- If so, please answer questions below:
  - Total number of a PL courses taken:
  - Please enter the information regarding your four most-recent PL classes:
    * PL Course 1
      · Name: ______
      · Calendar Year (e.g. 2006): ______
      · Class Year (e.g., K-8, undergraduate 1st Year): ______
      · School Name: ______
      · Credit Hours: ______
      · Grade: ______
    * PL Course 2
      · Name: ______
      · Calendar Year: ______
      · Class Year: ______
      · School Name: ______
      · Credit Hours: ______
      · Grade: ______
    * PL Course 3
      · Name: ______
      · Calendar Year: ______
      · Class Year: ______
      · School Name: ______
      · Credit Hours: ______

المنارة للاستشارات
www.manaraa.com

· Grade: ______

∗ PL Course 4

· Name: ______

· Calendar Year: ______

· Class Year: ______

· School Name: ______

· Credit Hours: ______

· Grade: ______

D.3.3 Programming Language Knowledge

- How many programming languages do you know?
- Number of Languages:
- Please indicate programming language(s) that you have taken a course on or used a project (check all that apply):
  - ALGOL
  - ADA
  - APL
  - Assembler
  - BASIC
  - C
  - C++
  - COBOL
  - Fortran
  - HTML
  - Hypertalk
  - Java
  - JCL
  - LISP
  - Logo

المنارة للاستشارات
www.manaraa.com

  - ML
  - PASCAL
  - Perl
  - PHP
  - PL/1
  - Prolog
  - Python
  - Ruby
  - S
  - Schema
  - Smalltalk
  - XML
  - Other (If there are others, please list their names):

- My favorite programming language(s) (check all that apply):
  - ALGOL
  - ADA
  - APL
  - Assembler
  - BASIC
  - C
  - C++
  - COBOL
  - Fortran
  - HTML
  - Hypertalk
  - Java
  - JCL
  - LISP

www.manaraa.com

- Logo
- ML
- PASCAL
- Perl
- PHP
- PL/1
- Prolog
- Python
- Ruby
- S
- Schema
- Smalltalk
- XML
- Other (If there are others, please list their names):

D.3.4 First Programming Language

Indicate the first programming language that you ever learned:

- Name of the language: ____________________
- Calendar Year of Initially Learned (e.g. 2006) ____________________
- Class Year: (e.g., K-8, undergraduate 1st Year etc.) ____________________
- Primary Reason for learning (check one):
  - Course Requirement
  - School âĂIJEnrichmentâĂİ/AP
  - School Club Activity
  - Research Project
  - Job Projects
  - Play/Design Games
  - Curiosity

المنارة للاستشارات

www.manaraa.com

  - Solve a game problem
  - Other; Please specify: ______________________

- Primary learning method (check one):
  - Books
  - Classroom training courses
  - Online documents
  - Online tutoring
  - Software tutoring
  - On on job training
  - Hands-on Self-Study
  - Other; Please specify: ______________________

- Other methods used (check all that apply):
  - Books
  - Classroom training courses
  - Online documents
  - Online tutoring
  - Software tutoring
  - On on job training
  - Hands-on Self-Study
  - Other; Please specify: ______________________

- Which learning method do you believe was best for learning your initial programming language?
  - Books
  - Classroom training courses
  - Online documents
  - Online tutoring

المنارة للاستشارات
www.manaraa.com

  - Software tutoring
  - On on job training
  - Hands-on Self-Study
  - Other; Please specify: ________________________

- Do you expect to use this language in a future project ?
  - Yes
  - No

- My favorite subject in school (check all that apply).
  - Art
  - Language
  - Math
  - Science
  - Engineering
  - Music
  - Other; Please specify: ________________________

Next please solve the attached programming problems.

www.manaraa.com

APPENDIX E
Post Problem Solving Feedback Questions

Please indicate your response to the following statements.

- Problem I was easy.
  - 1 Strongly agree
  - 2 Agree
  - 3 Somewhat agree
  - 4 Disagree
  - 5 Strongly disagree
- Problem 2 was easy.
  - 1 Strongly agree
  - 2 Agree
  - 3 Somewhat agree
  - 4 Disagree
  - 5 Strongly disagree
- A random check to make sure you are still paying attention.
  Please select option 3.
  - 1 Strongly agree
  - 2 Agree
  - 3 Somewhat agree
  - 4 Disagree
  - 5 Strongly disagree
- I had trouble solving the problems.
  - 1 Strongly agree
  - 2 Agree
  - 3 Somewhat agree

www.manaraa.com

  - 4 Disagree
  - 5 Strongly disagree

- The problems were easy overall.
  - 1 Strongly agree
  - 2 Agree
  - 3 Somewhat agree
  - 4 Disagree
  - 5 Strongly disagree

- We appreciate any other comments/feedbacks about your experiences of solving the above two programming problems.

____

____

____

____

____

المنارة للاستشارات
www.manaraa.com

APPENDIX F
List of Courses Input and Standard Names

المنارة للاستشارات
www.manaraa.com

Entry as Logic	Entry as PL	Course Name Raw	Course Name Standard
0	1	Beginning C for Engineers	Beginners C Programming for Engineers
0	1	AP Computer Science	AP Computer Science
0	1	AP Computer Science A	AP Computer Science
0	1	AP Computer Science AB	AP Computer Science
0	1	AP CS	AP Computer Science
0	1	AP Java	AP Computer Science
0	1	Beg. C Programming Eng	Beginners C Programming for Engineers
0	1	BegGinning C	Beginner Computer Programming
0	1	Beginner C Programming	Beginning C Programming
0	1	Beginner Computer Programming	Beginner Computer Programming
0	1	Beginners C Programming for Engineers	Beginners C Programming for Engineers
0	1	Beginning C	Beginning C Programming
0	1	Beginning C for Engineers	Beginners C Programming for Engineers
0	1	Beginning C prog for engineers	Beginners C Programming for Engineers
0	1	Beginning C Programming	Beginning C Programming
0	1	beginning c programming for engineers	Beginners C Programming for Engineers
0	1	Beginning C++ Programming for Engineers	Beginning C++ Programming for Engineers
0	1	C prog for eng	Beginners C Programming for Engineers
0	1	C Programming	Beginning C Programming
0	1	C programming 1	C Programming 1
0	1	C programming 2	C Programming 2
0	1	C Programming for Engineers	Beginners C Programming for Engineers
0	1	C++	C++
0	1	Comm Design for the WWW	Comm Design for the WWW
0	1	Comp Sci I	Comptuer Science 1
0	1	Comptuer Science	Comptuer Science 1
0	1	Comptuer Science 1	Comptuer Science 1
0	1	computer graph	Computer Graphics
0	1	Computer Graphics	Computer Graphics
0	1	computer org	Computer Organization
0	1	Computer Oranzation	Computer Organization
0	1	computer org	Computer Organization

المنارة للاستشارات

www.manaraa.com

Entry as Logic	Entry as PL	Course Name Raw	Course Name Standard
0	1	Computer orga	Computer Organization
0	1	computer organization	Computer Organization
0	1	Computer Programming	Computer Programming
0	1	Computer Programming I	Computer Programming 1
0	1	Computer Programming II	Computer Programming 2
0	1	Computer Programming Java	Programming in Java
0	1	Computer Science	Comptuer Science
0	1	Computer science 1	Comptuer Science 1
0	1	Computer science 1-ap	AP Computer Science
0	1	Computer Science 2	Comptuer Science 2
0	1	Computer Science I	Comptuer Science 1
0	1	Computer Science II	Comptuer Science 2
0	1	Computer Science III	Comptuer Science 3
0	1	CP	Computer Programming
0	1	CS	Comptuer Science
0	1	cs - 1	Comptuer Science 1
0	1	CS - AP	AP Computer Science
0	1	Cs 1	Comptuer Science 1
0	1	cs -1	Comptuer Science 1
0	1	CS 2	Comptuer Science 2
0	1	cs -ap	AP Computer Science
0	1	CS I	Comptuer Science 1
0	1	CS II	Comptuer Science 2
0	1	AP- Computer Science	AP Computer Science
0	1	cs using java	Computer Science using Java
0	1	cs1	Comptuer Science 1
0	1	CS-1	Comptuer Science 1
0	1	CS1: Java	Comptuer Science 1
0	1	cs2	Data Structures and Algorithms
0	1	CS-AP	AP Computer Science
0	1	CSCI 1	Comptuer Science 1
0	1	CSCI 1100	Comptuer Science 1

المنارة للاستشارات
www.manaraa.com

Entry as Logic	Entry as PL	Course Name Raw	Course Name Standard
0	1	CSCI 2	Data Structures and Algorithms
0	1	Csci 2400	Models of Computation
0	1	CSCI-1200	Data Structures
0	1	CSCI-2300	Introduction to Algorithms
0	1	CSCI-2400	Models of Computation
0	1	CSCI-2500	Computer Organization
0	1	csII	Comptuer Science 2
0	1	dara structure	Data Structures
0	1	data structure	Data Structures
0	1	data structure (c++)	Data Structures (C++)
0	1	Data Structures	Data Structures and Algorithms
0	1	Data Structures & Algorithms	Data Structures and Algorithms
0	1	Data Structures and Algorithms	Data Structures and Algorithms
0	1	data struture	Data Structures
0	1	DB systems	DB systems
0	1	DSA	Data Structures and Algorithms
0	1	Embedded Control	Embedded Controls
0	1	Emdeded Control	Embedded Controls
0	1	Engineering Tools	Engineering Tools
0	1	fortran	Fortran
0	1	high level Lang 1: C	high level Lang 1: C
0	1	high level Lang 1: C	high level Lang 1: C
0	1	into alg	Introduction to Algorithms
0	1	intro Algorithm	Introduction to Algorithms
0	1	intro Algrithm	Introduction to Algorithms
0	1	intro computer Programming	Introduction to Computer Programming
0	1	intro computer sc	Computer Science 1
0	1	intro cs	Computer Science 1
0	1	intro ro Pro(JAVA)	Programming in Java
0	1	intro to algorithm	Introduction to Algorithms
0	1	Intro to Algorithms	Introduction to Algorithms
0	1	intro to algorthm	Introduction to Algorithms

المنارة للاستشارات
www.manaraa.com

Entry as Logic	Entry as PL	Course Name Raw	Course Name Standard
0	1	intro to alorithms	Introduction to Algorithms
0	1	Intro to C	Introduction to C Programming
0	1	Intro to C Programming	Introduction to C Programming
0	1	Intro to C++	Introduction to C Programming
0	1	Intro to Computer Programming	Introduction to Computer Programming
0	1	Intro to Computer Science	Comptuer Science 1
0	1	intro to cs	Comptuer Science 1
0	1	intro to cs 2	Comptuer Science 2
0	1	intro to db	Introduction to DB
0	1	intro to game design	Introduction to Game Design
0	1	intro to java	Programming in Java
0	1	intro to organization	Computer Organization
0	1	intro to P	Introduction to Computer Programming
0	1	intro to Phython	Programming in Python
0	1	intro to programm (C++)	Introduction to Programming (C++)
0	1	Intro to Programming	Introduction to C Programming
0	1	intro to programming design	Introduction to Computer Programming
0	1	intro to VB	Introduction to Visual Basic
0	1	Intro to Visual Basic	Introduction to Visual Basic
0	1	Intro. Computer Programming	Introduction to Computer Programming
0	1	Introduction to Algorithm	Introduction to Algorithms
0	1	Introduction to C++	Beginning C++ Programming
0	1	Introduction to Computer Science	Computer Science 1
0	1	Introduction to Java	Programming in Java
0	1	Java	Programming in Java
0	1	Java I	Programming in Java
0	1	Java II	Programming in Java 2
0	1	java programming	Programming in Java
0	1	LITEC	Embedded Control
0	1	Models of Computation	Models of Computation
0	1	Matlab Programming for engineers	Matlab Programming for engineers
0	1	Microsoft Apps	Microsoft Apps

المنارة للاستشارات
www.manaraa.com

Entry as Logic	Entry as PL	Course Name Raw	Course Name Standard
0	1	model of computation	Models of Computation
0	1	Numerical computing	Numerical Computing
0	1	numerucal computing	Numerical Computing
0	1	OOP	Object Oriented Programming
0	1	Operating Sys	Operating Systems
0	1	Operating Systems	Operating Systems
0	1	Operatng system	Operating Systems
0	1	OS	Operating Systems
0	1	Perl	Programming in Perl
0	1	pl	Proggramming Languages
0	1	PL in java	Programming in Java
0	1	Pls	Proggramming Languages
0	1	Proframming Languages	Proggramming Languages
0	1	prog in Python	Programming in Python
0	1	Proggramming Languages	Proggramming Languages
0	1	program in Python	Programming in Python
0	1	Programing In Java	Programming in Java
0	1	Programming and logic 1	Programming and logic 1
0	1	Programming and logic 2	Programming and logic 2
0	1	Programming in Java	Programming in Java
0	1	programming in python	Programming in Python
0	1	psthon	Programming in Python
0	1	System Level Programming	System Level Programming
0	1	Visual Basic	Introduction to Visual Basic
0	1	Web and Database Programming	Web and Database Programming
0	1	Web Design	Web Systems Design
0	1	Web Sys	Web Systems Design
0	1	web system	Web Systems Design
0	1	web system design	Web Systems Design
0	1	Web Systems	Web Systems Design
0	1	Web Systems Development	Web Systems Design
1	0	calclus 1	calclus 1

المنارة للاستشارات
www.manaraa.com

Entry as Logic	Entry as PL	Course Name Raw	Course Name Standard
1	0	Calculus II	Calculus II
1	0	Chemistry	Chemistry
1	0	COCO	Computer Components and Operations
1	0	compitability and logic	Computability and Logic
1	0	Computer Components and Operations	Computer Components and Operations
1	0	computer componets and org	Computer Components and Operations
1	0	Computer Org	Computer Components and Operations
1	0	descreate math	Introduction to Discrete Structures
1	0	digital electonics	Digital electonics
1	0	discreate math	Introduction to Discrete Structures
1	0	discreate Structure	ntroduction to Discrete Structures
1	0	Discrete Mathematics	ntroduction to Discrete Structures
1	0	discrete structure	Introduction to Discrete Structures
1	0	discrete structures	Introduction to Discrete Structures
1	0	Embedded Control	Embedded Controls
1	0	Embedded Controls	Embedded Controls
1	0	Faith and Reason	Faith and Reason
1	0	General Psychology	General Psychology
1	0	General Pyschology	General Psychology
1	0	IEA	Introduction to Engineering Analysis
1	0	intermed logic	Intermediate Logic
1	0	intro philosophy and anarchy	introduction to philosophy and anarchy
1	0	Intro to Civil/Environmental Engineering	Intro to Civil/Environmental Engineering
1	0	Intro to Discrete Structures	Introduction to Discrete Structures
1	0	intro to discrye structure	Introduction to Discrete Structures
1	0	intro to logic	Introduction to Logic
1	0	Intro To Management	Intro To Management
1	0	Intro to Philosophy	Intro to Philosophy
1	0	Introduction to Discrete Structures	Introduction to Discrete Structures
1	0	Introduction to Logic	Introduction to Logic
1	0	Introduction to Programming in C	Introduction to Programming in C
1	0	Logic 101	Logic 101

المنارة للاستشارات
www.manaraa.com

Entry as Logic	Entry as PL	Course Name Raw	Course Name Standard
1	0	Math for Management	Math for Management
1	0	Micro Computers and Applications	Micro Computers and Applications
1	0	mind and machine	Minds and Machines
1	0	Minds + macn	Minds and Machines
1	0	minds and machine	Minds and Machines
1	0	Minds and Machines	Minds and Machines
1	0	model of computation	Models of Computation
1	0	models of computation	Models of Computation
1	0	PHIL-1100	Introduction to Philosophy
1	0	Probability Theory & Applications	Probability Theory & Applications
1	0	programing on Logic	Programing on Logic
1	0	Programming and logic 1	Programming and logic 1
1	0	Programming and logic 2	Programming and logic 2
1	0	Psychology	Psychology
1	0	Statistics and State. Programming	Statistics and State. Programming
1	0	symbolic logic	Symbolic logic

المنارة للاستشارات
www.manaraa.com

APPENDIX G
List Logic and Programming Courses Entries and Groups

المنارة للاستشارات
www.manaraa.com

Entry as Logic	Entry as PL	Course Name Standard	is Logic	Logic Group	is PL	Is CS	PL Group	PL Group2	Course Group
0	1	AP Computer Science	0		1	1	AP_CS	AP_CS	AP_CS
0	1	Beginner Computer Programming	0		1	1	PL	C	C
0	1	Beginners C Programming for Engineers	0		1	1	PL	C	C
0	1	Beginning C Programming	0		1	1	PL	C	C
0	1	Beginning C++ Programming	0		1	1	PL	C++	C++
0	1	Beginning C++ Programming for Engineers	0		1	1	PL	C++	C++
0	1	C Programming 1	0		1	1	PL	C	C
0	1	C Programming 2	0		1	1	PL	C	C
0	1	C++	0		1	1	PL	C++	C++
0	1	Comm Design for the WWW	0		1	1	CS	CS	WEB
0	1	Comptuer Science	0		1	1	CS	CS	CS
0	1	Comptuer Science 1	0		1	1	CS	CS	CS
0	1	Comptuer Science 2	2	Logic2	1	1	PL	PL2	CS
0	1	Comptuer Science 3	0		1	1	CS	CS	CS
0	1	Computer Graphics	0		1	1	CS	CS	CS
0	1	Computer Organization	5	Logic_CS	1	1	CS	CS	CS
0	1	Computer Programming	0		1	1	PL	PL	PL
0	1	Computer Programming 1	0		1	1	PL	PL	PL
0	1	Computer Programming 2	0		1	1	PL	PL2	PL
0	1	Computer Science using Java	0		1	1	PL	Java	Java
0	1	Data Structures	4	Logic_DSA	1	3	DSA	DSA	DSA
0	1	Data Structures (C++)	4	Logic_DSA	1	3	DSA	DSA	DSA
0	1	Data Structures and Algorithms	4	Logic_DSA	1	3	DSA	DSA	DSA
0	1	DB systems	0		1	1	CS	CS	CS
0	1	Engineering Tools	0		1	1	CS	Other	Other
0	1	Fortran	0		1	1	PL	Fortran	Fortran

المنارة للاستشارات
www.manaraa.com

Entry as Logic	Entry as PL	Course Name Standard	is Logic	Logic Group	is PL	Is CS	PL Group	PL Group2	Course Group
0	1	high level Lang 1: C	0		1	1	PL	C	C
0	1	high level Lang 1: C++	0		1	1	PL	C++	C++
0	1	Introduction to Algorithms	4	Logic_DSA	1	3	DSA	DSA	DSA
0	1	Introduction to C Programming	0		1	1	PL	C	C
0	1	Introduction to Computer Programming	0		1	1	PL	PL	PL
0	1	Introduction to DB	0		1	1	CS	CS	CS
0	1	Introduction to Game Design	0		1	1	CS	CS	CS
0	1	Introduction to Programming (C++)	0		1	1	PL	C++	C++
0	1	Introduction to Visual Basic	0		1	1	PL	VB	VB
0	1	Matlab Programming for engineers	0		1	1	PL	CS	CS
0	1	Microsoft Apps	0		1	1	CS	CS	CS
0	1	Models of Computation	5	Logic_CS	1	1	CS	CS	CS
0	1	Numerical Computing	0		1	1	CS	CS	Math
0	1	Object Oriented Programming	0		1	1	PL	PL	PL
0	1	Operating Systems	0		1	1	CS	CS	CS
0	1	Proggramming Languages	0		1	1	PL	PL	PL
1	1	Programming and logic 2	2	Logic2	1	1	PL	PL	PL
0	1	Programming in Java	0		1	1	PL	Java	Java
0	1	Programming in Java 2	0		1	1	PL	Java	Java
0	1	Programming in Perl	0		1	1	PL	Perl	Perl
0	1	Programming in Python	0		1	1	PL	Python	Python
0	1	System Level Programming	0		1	1	PL	CS	CS
0	1	Web and Database Programming	0		1	1	PL	Web	Web
0	1	Web Systems Design	0		1	1	Web	Web	Web
1	0	calclus 1	0		0	0			
1	0	Calculus II	0		0	0			

المنارة للاستشارات
www.manaraa.com

Entry as Logic	Entry as PL	Course Name Standard	is Logic	Logic Group	is PL	Is CS	PL Group	PL Group2	Course Group
1	0	Chemistry	0		0	0			Other
1	0	Computability and Logic	3	Logic3	0	0			Logic3
1	0	Computer Components and Operations	5	Logic_CS	0	0			Logic_CS
1	0	Digital electonics	0		0	1			Other
1	1	Embedded Controls	5	Logic_CS	1	1	CS	PL	PL
1	0	Faith and Reason	1	Logic1	0	0			Logic1
1	0	General Psychology	0		0	0			Other
1	0	Intermediate Logic	2	Logic2	0	0			Logic2
1	0	Intro to Civil/Environmental Engineering	0		0	0			Other
1	0	Intro To Management	0		0	0			Other
1	0	Intro to Philosophy	0		0	0			Other
1	0	Introduction to Discrete Structures	4	Logic_DSA	1	3	DSA	DSA	DSA
1	0	Introduction to Engineering Analysis	0		0	0			Other
1	0	Introduction to Logic	1	Logic1	0	0			Logic1
1	0	Introduction to Philosophy	1	Logic1	0	0			Logic1
1	0	introduction to philosophy and anarchy	1	Logic1	0	0			Logic1
1	0	Introduction to Programming in C	0		1	1	PL	C	C
1	0	Logic 101	1	Logic1	0	0			Logic1
1	0	Math for Management	0		1	0	Other	Other	Other
1	0	Micro Computers and Applications	0		0	1			CS
1	0	Minds and Machines	5	Logic_CS	0	1			CS
1	0	Probability Theory & Applications	0		0	0			Math
1	1	Programming and logic 1	1	Logic1	1	1	PL	PL	PL
1	0	Psychology	0		0	0			Other
1	0	Statistics and State. Programming	0		1	1	PL	PL	PL
1	0	Symbolic logic	3	Logic3	0	0			Logic3

المنارة للاستشارات
www.manaraa.com

APPENDIX H

Additional Data

المنارة للاستشارات

www.manaraa.com

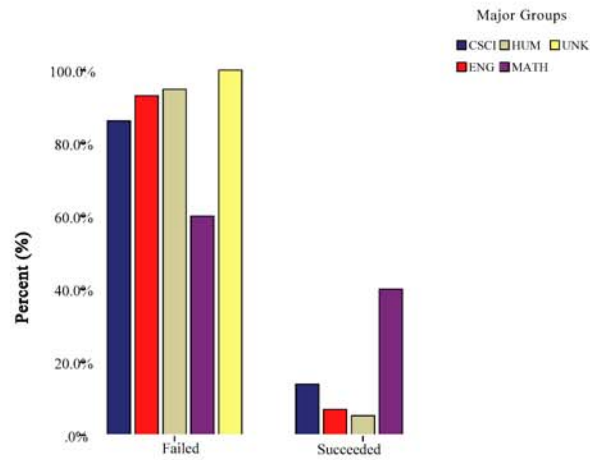


Figure H.1: Success and Academic Major
$\chi^2(4) = 18.7, p = 0.001.$

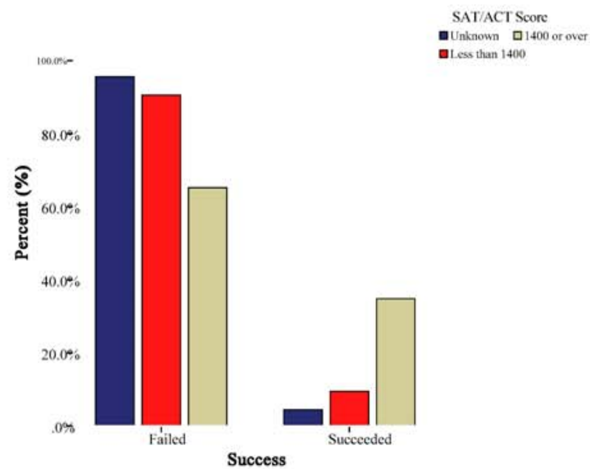


Figure H.2: Success and SAT/ACT Scores
$\chi^2(2) = 36.3, p < 0.001.$

www.manaraa.com

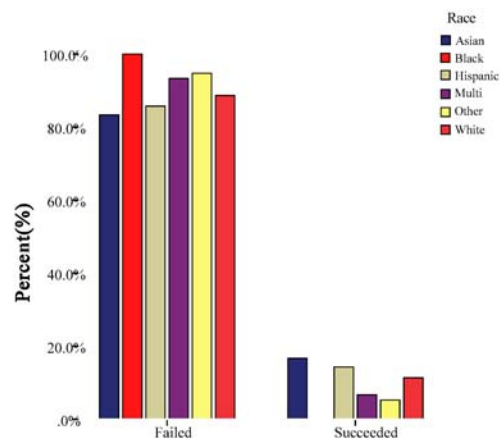


Figure H.3: Success and Race/Ethnicity. $\chi^2(5) = 3.8, p < 0.58$. No significance difference of success rate among different race/ethnicity groups.

المنارة للاستشارات

www.manaraa.com