

INTELLIGENT TUTORING SYSTEM FOR LEARNING PHP

Dinesha Samanthi Weragama
B.Sc. (Eng) Hons.

Submitted in fulfilment of the requirements for the degree of
Doctor of Philosophy

School of Electrical Engineering & Computer Science
Science & Engineering Faculty
Queensland University of Technology
October 2013

Keywords

Computerised learning systems, Domain module, Intelligent Tutoring Systems, PHP, Program Analysis

Abstract

Teaching introductory programming has challenged educators for decades. Of the many suggested methods of improving the teaching process, individual tutoring has proven to be very effective. However, individual tutoring with human tutors requires a large amount of resources and is therefore impractical to use with the vast number of students who want to learn programming. A viable alternative is to use Intelligent Tutoring Systems (ITSs) for this purpose. Although some ITSs have been built to teach programming, none have been developed to address the subject of web programming, which is becoming increasingly popular. This thesis addresses this gap by designing, building and evaluating an Intelligent Tutoring System to teach web development using PHP.

Any system that teaches programming needs to provide practical exercises for the students. In order for the students to learn from the system, it is necessary for them to receive feedback on their solutions to the exercises. A major challenge here is that a programming problem rarely has a unique solution. For a system to be effective, it is necessary that it be capable of handling many alternative solutions to a given programming exercise. This thesis concentrates on achieving this objective using the theories of artificial intelligence. The system converts the student's solution into a set of predicates. These predicates are then compared against an overall goal which is also depicted as a set of predicates. Any missing predicates are used to identify sub-goals of the programming exercise that are not met and to provide relevant feedback.

The PHP ITS customises the instructions for individual students by providing guidance for each student on the next best exercise he/she should attempt. This is done by dividing the subject matter into topics and storing a probabilistic estimate as to each student's current knowledge of that topic. The estimates are updated based on each solution that the student submits for the exercises. The knowledge level of each topic and the topics covered by each exercise are utilised to find the exercise that has the least number of topics that are not known to the current student. This ensures that the student will learn something new by attempting this exercise, while reducing the amount of new material so as not to overload the student.

These concepts were used to build a web-based Intelligent Tutoring System. The system was evaluated on two sets of students at the Queensland University of Technology. The students were given a pre-test to measure their knowledge of the subject matter. Then, they used the system for six weeks during their own time to solve exercises. Finally, they were given a post-test to gauge whether their knowledge had improved. They were also given a questionnaire to measure their acceptance of the system.

The results of a paired t-test showed that the student's knowledge increased significantly as a result of using the system. They also showed that the system's gauge of the knowledge level of each student was successful in predicting their final test scores, indicating that the gauge was fairly accurate. Analysis of qualitative data also showed that the students were relatively satisfied with the system overall.

Although it is possible to improve the system further, the evaluation process showed that the PHP ITS can be used effectively to teach PHP web development to beginners in web programming.

Table of Contents

Keywords	i
Abstract	ii
Table of Contents	iii
List of Figures	vii
List of Tables	xi
List of Abbreviations.....	xiii
Statement of Original Authorship	xiv
Acknowledgements	xv
CHAPTER 1 : INTRODUCTION	1
1.1 Background.....	1
1.2 Context.....	1
1.3 Research Goal and Objectives	3
1.4 Significance	4
1.5 Scope of the Thesis	5
1.6 Thesis Outline	6
CHAPTER 2 : LITERATURE REVIEW	9
2.1 Teaching Introductory Programming.....	9
2.1.1 Cognitive Requirements	9
2.1.2 Syntax and Semantics	10
2.1.3 Orientation	10
2.1.4 Auxiliary Skills.....	11
2.1.5 Resource Constraints	11
2.2 Intelligent Tutoring Systems	11
2.2.1 Background.....	11
2.2.2 Architecture of Intelligent Tutoring Systems	13
2.2.3 Domains Taught by Existing ITSs.....	15
2.3 The Domain Module	16
2.3.1 Static and Dynamic Program Analysis	16
2.3.2 Knowledge Representation	17
2.3.3 Intention Based Analysis	22
2.4 The Teaching Module	25
2.4.1 Feedback.....	25
2.4.2 Next Problem Selection	28
2.4.3 Other Forms of Support	29
2.4.4 Summary.....	30
2.5 The Student Module.....	30
2.5.1 Bayesian Student Modelling	32
2.5.2 Open Learner Models	33
2.6 Comparison of Existing ITSs to Teach Programming	34
2.7 Summary and Implications	37
CHAPTER 3 : RESEARCH DESIGN.....	39
3.1 Methodology	39

3.2	Research Design	40
3.2.1	Phase One	40
3.2.2	Phase Two	40
3.2.3	Phase Three	43
3.2.4	Phase Four	43
3.3	Timeline	44
3.4	Chapter Summary	45
CHAPTER 4 : BASICS OF PROGRAM ANALYSIS		47
4.1	Theoretical Concepts	49
4.1.1	Concepts in Artificial Intelligence	49
4.1.2	Concepts in Database Design	50
4.1.3	Concepts in Language Parsing	52
4.2	Conventions Used in this Thesis	53
4.3	Outline of the Basic Program Analysis Process	53
4.4	Knowledge Base Structure	55
4.4.1	Predicates and Rules	55
4.4.2	Exercise Specification	65
4.4.3	Actions	66
4.5	Program Analysis	68
4.5.1	Initial State	69
4.5.2	Abstract Syntax Tree	70
4.5.3	Walking the AST	73
4.5.4	Goal Checking	75
4.5.5	Checking for Unnecessary Program Statements	77
4.6	Special Situations	80
4.6.1	Multiple <i>OnPage</i> Predicates	80
4.6.2	Pre and Post Increment and Decrement Operators	82
4.6.3	HTML Embedded Within PHP	84
4.7	Chapter Summary	84
CHAPTER 5 : SELECTION STRUCTURES		87
5.1	Goal Specification	88
5.2	Program Analysis	89
5.2.1	Incorrect Solutions	92
5.3	Alternative Solutions	94
5.4	Other Forms of Conditional Expressions	96
5.4.1	Simple Expressions Behaving as Conditional Expressions	97
5.4.2	Conditional Expressions with <i>And</i> , <i>Or</i> and <i>Not</i>	100
5.5	Nested Selection Structures	104
5.5.1	Analysis of <i>Program a</i>	105
5.5.2	Analysis of <i>Program b</i>	108
5.5.3	Correct Overall Goal for Nested Selection Structures	109
5.6	Switch Statements	111
5.6.1	Special Considerations	113
5.7	Handling Unnecessary Statements in Selection Structures	114
5.8	Chapter Summary	116
CHAPTER 6 : ARRAYS, FUNCTIONS AND FORMS		117
6.1	Arrays	117
6.1.1	Assigning to Array Variables	120
6.1.2	Array Construct	123

6.2	Functions.....	123
6.2.1	Predicates for Handling Functions.....	124
6.2.2	The Scope of Variables.....	127
6.2.3	Analysis of Programs that Use Functions.....	134
6.2.4	Pre-Defined Functions.....	141
6.2.5	Conditional Expressions Where the Condition is a <i>FunctionCall</i>	142
6.2.6	Unnecessary Statements in Functions.....	143
6.3	Forms.....	145
6.3.1	Form Definition.....	145
6.3.2	Accessing Values Passed Through Forms.....	148
6.3.3	Handling Standard Form Definitions.....	151
6.4	Chapter Summary.....	152
CHAPTER 7 : LOOPS		153
7.1	Types of Loops.....	153
7.2	Definite Loops.....	159
7.2.1	Predicate Definition.....	159
7.2.2	Overall Goal Specification.....	161
7.2.3	Program Analysis.....	163
7.2.4	Unnecessary Statements in Loops.....	167
7.2.5	While Loops that Behave as For Loops.....	169
7.3	Special Situations.....	170
7.3.1	Loops Where the Counter Variable Changes According to an Arithmetic Sequence.....	170
7.3.2	Loop where the Execution of Statements Depends on the Results of Previous Iterations.....	178
7.4	Collection Based Loops that Perform Some Action Against Every Item in the Collection Independently Without Summarising.....	185
7.4.1	<i>For</i> and <i>While</i> Constructs.....	186
7.4.2	<i>Foreach</i> Construct.....	192
7.5	Collection Based Loops that Perform Some Action Against Every Item in the Collection Independently While Summarising.....	199
7.5.1	Overall Goal Specification.....	200
7.6	Chapter Summary.....	220
CHAPTER 8 : IMPLEMENTATION OF THE PHP INTELLIGENT TUTORING SYSTEM.....		221
8.1	The PHP Intelligent Tutoring System.....	221
8.1.1	Exercise Selection.....	223
8.1.2	Solving an Exercise.....	225
8.2	Student module.....	228
8.2.1	Equations for Updating the Student Model.....	228
8.2.2	Assumptions.....	231
8.2.3	Updating the Student Model in the PHP ITS.....	234
8.3	Teaching Module.....	235
8.3.1	Assistance for Solving Exercises.....	236
8.3.2	Assistance for Selecting Next Exercise.....	239
8.3.3	Viewing the Suggested Solution.....	240
8.4	Implementation Details.....	240
8.4.1	Software and Tools.....	241
8.4.2	Database Structure.....	244
8.4.3	Implementation Issues.....	245
8.5	Chapter Summary.....	248
CHAPTER 9 : SYSTEM EVALUATION.....		251
9.1	Evaluation Process of the PHP Intelligent Tutoring System.....	251

9.1.1	Participants	251
9.1.2	Procedures and Instruments	252
9.2	Different Versions of the PHP intelligent Tutoring System	254
9.2.1	Feedback to Students' Solutions.....	254
9.2.2	Selecting the Next Exercise	255
9.2.3	Handling Students' Doubt Regarding Program Analysis	255
9.2.4	User Interface	256
9.3	Results and Discussion	256
9.3.1	Effectiveness of the System.....	256
9.3.2	Validity of the Student Model	268
9.3.3	System Usage	270
9.3.4	Satisfaction	274
9.4	Chapter Summary	279
CHAPTER 10 : CONCLUSIONS.....		281
10.1	Research Contributions.....	282
10.1.1	Knowledge Representation.....	283
10.1.2	Student Model	283
10.1.3	Feedback and Individualised Instruction	284
10.1.4	Publications and Talks.....	285
10.2	Lessons Learned	286
10.2.1	System Design.....	286
10.2.2	Evaluation.....	288
10.3	Future Directions	288
BIBLIOGRAPHY		291
APPENDICES		299
	Appendix A Introduction to Bayesian Belief Networks	299
	Appendix B PHP Grammar	301
	Appendix C Combined Assign Actions	311
	Appendix D HTML Grammar	315
	Appendix E Examples of Analysis of Selection Structures	326
	Appendix F Examples for Analysis of Functions and Forms	344
	Appendix G Examples for Analysis of Loops	350
	Appendix H Implementation Details	359
	Appendix I Pre and Post Test.....	361
	Appendix J Questionnaire.....	366
	Appendix K Focus Group Questions	371
	Appendix L Complete ORM Diagram.....	373

List of Figures

<i>Figure 2.1.</i> Main modules of an Intelligent Tutoring System.	14
<i>Figure 2.2.</i> Generic architecture for ITS to teach programming.	15
<i>Figure 4.1.</i> Example programming exercise.	47
<i>Figure 4.2.</i> Some ORM symbols and their meanings.	51
<i>Figure 4.3.</i> Basic program analysis.	54
<i>Figure 4.4.</i> ORM diagram of key components of the assignment statement.	56
<i>Figure 4.5.</i> ORM diagram of expression subtypes of simple and calculate expressions.	61
<i>Figure 4.6.</i> ORM diagram of Boolean expression subtypes.	62
<i>Figure 4.7.</i> Predicates relevant to addition expression.	63
<i>Figure 4.8.</i> Rules for calculating the <i>ValueOf</i> expressions.	64
<i>Figure 4.9.</i> <i>Display</i> action.	66
<i>Figure 4.10.</i> <i>Assign</i> action.	68
<i>Figure 4.11.</i> Detailed version of <i>AssignAdd</i> action.	68
<i>Figure 4.12.</i> Subtype version of <i>AssignAdd</i> action.	68
<i>Figure 4.13.</i> Initial state for example program.	69
<i>Figure 4.14.</i> AST for example program.	72
<i>Figure 4.15.</i> Final state of example program.	76
<i>Figure 4.16.</i> Overall goal of example exercise.	76
<i>Figure 4.17.</i> A program with unnecessary statements.	77
<i>Figure 4.18.</i> Rules used to calculate the <i>ValueOf</i> the right-hand expression.	78
<i>Figure 4.19.</i> Rule used to find the <i>ValueOf</i> the echoed expression.	79
<i>Figure 4.20.</i> Status flow for example program.	79
<i>Figure 4.21.</i> Status flow for example program with unnecessary statements.	80
<i>Figure 4.22.</i> ORM diagram for pre and post fix expressions.	82
<i>Figure 4.23.</i> Rules for calculating the <i>ValueOf</i> pre and post fix expressions.	83
<i>Figure 4.24.</i> Example of HTML embedded within PHP.	84
<i>Figure 4.25.</i> New HTML input stream.	84
<i>Figure 5.1.</i> Boolean predicates used for comparison.	88
<i>Figure 5.2.</i> Initial state and overall goal of example program for selection.	89
<i>Figure 5.3.</i> AST for example program for selection.	89
<i>Figure 5.4.</i> Rules for converting Boolean expressions into comparison predicates.	90
<i>Figure 5.5.</i> Incorrect solution to example exercise for selection structures.	93
<i>Figure 5.6.</i> AST for incorrect solution to exercise.	93
<i>Figure 5.7.</i> AST for Program b in Table 5.1.	94
<i>Figure 5.8.</i> Rules for converting between equivalent expression subtypes.	96
<i>Figure 5.9.</i> A solution to the example exercise for selection structures using a conditional statement with a <i>SimpleExpression</i>	97

<i>Figure 5.10.</i> Rules to convert <i>VariableExprs</i> into comparison predicates.	99
<i>Figure 5.11.</i> Rule to handle mathematical equality.	99
<i>Figure 5.12.</i> Rules for handling complex conditional expressions.	101
<i>Figure 5.13.</i> Example exercise for selection structures with Boolean operators in the condition.....	101
<i>Figure 5.14.</i> Overall goal for example exercise for selection structures with Boolean operators in the condition.....	102
<i>Figure 5.15.</i> Solution to example exercise.....	102
<i>Figure 5.16.</i> Example exercise for nested selection structures.	104
<i>Figure 5.17.</i> Suggested initial state and overall goal for example exercise for nested selection structures.	105
<i>Figure 5.18.</i> Relevant facts for final state of Program b.	109
<i>Figure 5.19.</i> Overall goal for example exercise for nested selection structures.	109
<i>Figure 5.20.</i> Example exercise for switch statements.	111
<i>Figure 5.21.</i> Suggested overall goal for example exercise.	112
<i>Figure 5.22.</i> Simplified overall goal for example exercise.	112
<i>Figure 5.23.</i> Example program for comparison operators within switch statements.	113
<i>Figure 5.24.</i> Example switch statement with execution falling through to next case.	114
<i>Figure 5.25.</i> Status flow for example selection program.	115
<i>Figure 6.1.</i> ORM diagram for arrays.	118
<i>Figure 6.2.</i> Example array exercise.	120
<i>Figure 6.3.</i> Overall goal of example array exercise.	120
<i>Figure 6.4.</i> <i>AssignArrayVariable</i> action.	121
<i>Figure 6.5.</i> Subtype version of <i>AssignAddArrayVariable</i> action.	122
<i>Figure 6.6.</i> Two forms of the array construct.	123
<i>Figure 6.7.</i> ORM diagram for functions.	124
<i>Figure 6.8.</i> Example program for function use.	125
<i>Figure 6.9.</i> Rules for handling variable scope.	129
<i>Figure 6.10.</i> A PHP program with a global variable.....	130
<i>Figure 6.11.</i> Rules to calculate <i>ValueOf</i> expressions with scope considered.	131
<i>Figure 6.12.</i> Modified <i>Assign</i> action to include variable scope.	132
<i>Figure 6.13.</i> Modified <i>AssignArrayVariable</i> action to include scope.....	132
<i>Figure 6.14.</i> Example exercise for functions.	134
<i>Figure 6.15.</i> Initial state for example exercise for functions.	135
<i>Figure 6.16.</i> Overall goal specification for example exercise for functions.	135
<i>Figure 6.17.</i> Solution to example exercise for functions.	136
<i>Figure 6.18.</i> Rule to set initial value of parameter variables.	137
<i>Figure 6.19.</i> Rule to calculate the <i>ValueOf</i> parameter variables.....	140
<i>Figure 6.20.</i> Rule for calculating the <i>ValueOf FunctionExprs</i>	141
<i>Figure 6.21.</i> PHP code that has a function expression as the condition within a selection statement.	143
<i>Figure 6.22.</i> Rules used to find conditional expressions for <i>FunctionExprs</i>	143

<i>Figure 6.23.</i> Flow of statuses for example program using functions.	144
<i>Figure 6.24.</i> ORM diagram for forms.	146
<i>Figure 6.25.</i> Example exercise for forms.	146
<i>Figure 6.26.</i> Example solution to exercise for forms.	147
<i>Figure 6.27.</i> Rule to create array elements from form input elements.	148
<i>Figure 6.28.</i> Rule to set the value of the parameter variable when the value of a 'isset' function expression is True.	150
<i>Figure 7.1.</i> Classification of loops.	154
<i>Figure 7.2.</i> Predicates for handling loops with counters.	160
<i>Figure 7.3.</i> Example exercise for simple counted loop.	162
<i>Figure 7.4.</i> Overall goal for example exercise for simple counted loop.	162
<i>Figure 7.5.</i> Rules for finding the end value of a <i>CountedLoop</i>	165
<i>Figure 7.6.</i> Rule to find the increment of a <i>CountedLoop</i>	166
<i>Figure 7.7.</i> Rules to consolidate results of loop execution.	167
<i>Figure 7.8.</i> Flow of statuses example program for loops.	169
<i>Figure 7.9.</i> Overall goal for example program for loops that do not execute for all values of the counter variable.	172
<i>Figure 7.10.</i> Rules for consolidating loops that do not execute for all values of the counter variable.	175
<i>Figure 7.11.</i> Example exercise for loops where execution depends on previous iterations.	178
<i>Figure 7.12.</i> Example solution for factorial exercise.	178
<i>Figure 7.13.</i> Initial state and overall goal for factorial exercise.	179
<i>Figure 7.14.</i> Rule to aggregate factorial as repeated multiplication.	183
<i>Figure 7.15.</i> Initial state and overall goal for multiplication as repeated addition.	184
<i>Figure 7.16.</i> Example solution for multiplication exercise.	184
<i>Figure 7.17.</i> Rule to aggregate multiplication as repeated addition.	185
<i>Figure 7.18.</i> Example exercise for for-each loop using the <i>for</i> construct.	186
<i>Figure 7.19.</i> Initial state and overall goal for example exercise for for-each loop using <i>for</i> construct.	187
<i>Figure 7.20.</i> Example solution to exercise for for-each loop using <i>for</i> construct	188
<i>Figure 7.21.</i> Predicates for handling the <i>foreach</i> construct.	193
<i>Figure 7.22.</i> Example program for <i>foreach</i> construct.	194
<i>Figure 7.23.</i> Overall goal for example exercise for <i>foreach</i> construct.	195
<i>Figure 7.24.</i> Rules for consolidating <i>foreach</i> constructs.	198
<i>Figure 7.25.</i> Example exercise for a search loop.	199
<i>Figure 7.26.</i> Initial state for example exercise for collection based loops that perform some action against every item in the collection.	199
<i>Figure 7.27.</i> Overall goal for example exercise for collection based loops that perform some action against every item in the collection.	201
<i>Figure 7.28.</i> Rules for handling loop unrolling of the first or last element of the array.	209
<i>Figure 7.29.</i> Facts and rules for finding search results.	210
<i>Figure 7.30.</i> Rule for handling direct method of array access in search loops.	219

<i>Figure 8.1. Banner.</i>	222
<i>Figure 8.2. Skillometer.</i>	222
<i>Figure 8.3. Exercise selection page.</i>	223
<i>Figure 8.4. Exercise search page</i>	224
<i>Figure 8.5. The solution page.</i>	226
<i>Figure 8.6. Equations for first phase of updating the student model.</i>	231
<i>Figure 8.7. Equations for calculating combined effect of two phase updating of the student model.</i>	232
<i>Figure 8.8. Modified equations for two phase update of the student model.</i>	233
<i>Figure 8.9. Empirical parameter values.</i>	233
<i>Figure 8.10. Final equations for two phase updating of student model.</i>	234
<i>Figure 8.11. Single phase update of the student model for the first interaction.</i>	235
<i>Figure 8.12. Final equations for updating the student model based on the pre-test.</i>	235
<i>Figure 8.13. Software architecture used in the PHP ITS.</i>	242
<i>Figure 8.14. Database model of the PHP ITS – 1.</i>	246
<i>Figure 8.15. Database model of the PHP ITS – 2.</i>	247
<i>Figure 9.1. Average pre and post-test score.</i>	259
<i>Figure 9.2. Normal probability plot for regression analysis.</i>	263
<i>Figure 9.3. Types of help used by students.</i>	271
<i>Figure 9.4. Exercise selection mode used by each student.</i>	273
<i>Figure 9.5. Frequency of Skillometer usage by students.</i>	273
<i>Figure 9.6. Overall impression of the system.</i>	274
<i>Figure 9.7. Ease of use of the system.</i>	275
<i>Figure 9.8. Programming exercises.</i>	276
<i>Figure 9.9. Speed of response of the system.</i>	276
<i>Figure 9.10. Feedback messages.</i>	277
<i>Figure 9.11. Success in gaining student knowledge and understanding.</i>	278
<i>Figure 9.12. Look and feel.</i>	278
<i>Figure A1. A Bayesian Belief Network for pre-requisite topics.</i>	300
<i>Figure L1. Complete ORM diagram</i>	373
<i>Figure L2. Complete ORM diagram – left half</i>	375
<i>Figure L3. Complete ORM diagram – right half</i>	377

List of Tables

Table 2.1 <i>Existing ITSs to Teach Programming</i>	35
Table 3.1 <i>Timeline for Completion of Each Phase</i>	44
Table 4.1 <i>Alternative Correct Solutions for Example Exercise</i>	47
Table 4.2 <i>Different Methods of Displaying “Hello World” on a PHP Web Page</i>	81
Table 4.3 <i>Modified ASTs for Pre and Post Increment and Decrement</i>	83
Table 5.1 <i>Programs to Illustrate Different Forms of the Same Conditions</i>	87
Table 5.2 <i>Alternative Solutions to Example Exercise for Nested Selection Structures</i>	105
Table 5.3 <i>Alternative Programs for Example Exercise</i>	111
Table 5.4 <i>Modified AST for Switch Statements</i>	113
Table 6.1 <i>AST Conversion for Array Construct</i>	123
Table 7.1 <i>Definite and Indefinite Loops in PHP</i>	155
Table 7.2 <i>Counted Loops in PHP</i>	155
Table 7.3 <i>Perform Action Against Every Item in Collection Independently</i>	156
Table 7.4 <i>Types of Loops Based on the Independence of Actions Performed Within the Loop</i>	157
Table 7.5 <i>Equivalent For and While Loops</i>	170
Table 7.6 <i>Examples of Loops That Do Not Execute for all Integer Values of the Counter Variable within the Specified Range</i>	171
Table 7.7 <i>Solutions to Example Exercise for Collection Based Loops the Perform Some Action on Every Element while Summarising using Indirect and Direct Methods of Array Access</i>	200
Table 8.1 <i>List of Knowledge Components</i>	230
Table 8.2 <i>Feedback Messages for Checking if a Value Entered in a Textbox is Greater than 10</i>	238
Table 8.3 <i>Feedback Messages for an Unnecessary Assignment Statement in Line 10</i>	239
Table 9.1 <i>Pre and Post-test Results for Version 1</i>	257
Table 9.2 <i>Pre and Post-Test Results for Version 2</i>	258
Table 9.3 <i>Initial and Final Average Learned Probabilities for Version 1</i>	259
Table 9.4 <i>Initial and Final Average Learned Probabilities for Version 2</i>	260
Table 9.5 <i>Number of Help Requests for Each Student</i>	262
Table 9.6 <i>Correlations Results for Improvement in Test Score and Number of Help Requests</i>	263
Table 9.7 <i>Total Duration of System Use for Each Student</i>	265
Table 9.8. <i>Correlation Results for Minutes Used and Improvement in Test Score</i>	266
Table 9.9. <i>The Number of Problems Attempted and the Number of Problems Correct for Each Student</i>	267
Table 9.10. <i>Correlation Results for Number of Problems Attempted, Number of Problems Correct and Improvement in Test Score</i>	268
Table 9.11 <i>Predicted and Actual Post Test Scores for Each Student for Version 1</i>	269
Table 9.12 <i>Predicted and Actual Post Test Scores for Each Student for Version 2</i>	270
Table 9.13 <i>Correlation results of Post-test Score and Predicted Post-test Score for Version 1</i>	270

Table 9.14 *Correlation results of Post-test Score and Predicted Post-test Score for Version 2*271

List of Abbreviations

AI	Artificial Intelligence
AST	Abstract Syntax Tree
BBN	Bayesian Belief Network
EBNF	Extended Backus-Naur Form
FOL	First-Order Logic
FOPC	First-Order Predicate Calculus
HTML	Hypertext Markup Language
ITS	Intelligent Tutoring System
KB	Knowledge Base
KC	Knowledge Component
ORM	Object Relational Model
PDDL	Planning Domain Definition Language
PHP	PHP Hypertext Processor
PHP ITS	PHP Intelligent Tutoring System
QUT	Queensland University of Technology

Statement of Original Authorship

The work contained in this thesis has not been previously submitted to meet requirements for an award at this or any other higher education institution. To the best of my knowledge and belief, the thesis contains no material previously published or written by another person except where due reference is made.

Signature: 02/10/2013

Date: Balarinja

Acknowledgements

First and foremost, I would like to express my sincere gratitude to my principal supervisor Dr. Jim Reye for all the support and encouragement given during the course of my PhD studies. I would also like to thank my associate supervisor Dr. Hasmukh Morarji for helping me to complete my studies.

I would like to especially mention Dr. Mahinda Alahakoon of the University of Peradeniya, Sri Lanka without whose guidance and encouragement I would never have embarked on a PhD degree. You really helped me to move forward in my career and my life at a time when I felt that all my previous efforts were fruitless.

I would like to thank the Queensland University of Technology for providing me with the infrastructure to complete this project as well as providing me with a QUT Postgraduate Award and a QUT Fee Waiver Scholarship to help me through my studies. My thanks go out to Kerrie Petersen of the Language and Learning Support unit, all the staff at the Research Student Centre and at International Student Services for the support extended during various times during my time at the Queensland University of Technology.

My thanks also go out to all my friends, both in Australia and abroad, who helped me cope with the ups and downs of my PhD studies.

Last but not least, I extend my heartfelt appreciation to my mother, my husband Punchandra, and my two daughters Malithi and Rashmi, who stood by me through the triumphs and defeats of the past three years. Your love and support contributed in no small amount to the completion of this thesis.

Chapter 1: Introduction

This chapter outlines the background (Section 1.1) and context (Section 1.2) of the research, and its goals and objectives (Section 1.3). Section 1.4 describes the significance of this research. Section 1.5 identifies the scope of the thesis. Finally, Section 1.6 includes an outline of the remaining chapters of the thesis.

1.1 BACKGROUND

Programming is a fundamental component of any Computer Science course. It is also incorporated into many other disciplines such as Business, Finance and Accounting due to its widespread use in industry. However, many beginning students find programming a very difficult subject. This is shown by the fact that many students either drop out or fail programming courses (Miliszewska & Tan, 2007). Therefore, it is necessary to find means of making this subject less challenging to the novice student.

The large number of people who show an interest in learning to program are very diverse. They differ in many aspects such as age, gender, educational level and aptitude for solving logical problems. Experience as a teacher of beginning Computer Science students has shown that it is extremely difficult to create a single course that caters to all their differing needs. Although one-to-one tutoring would be a suitable means of addressing this problem, it is not a financially viable alternative. A much better solution is to use Intelligent Tutoring Systems (ITS). An ITS is a computerised teaching system, that offers one-to-one tutoring, by altering its interaction with the student based on the individual's personal characteristics (Woolf, 2009).

1.2 CONTEXT

With the current popularity of the World Wide Web (WWW), more and more students are showing an interest in learning to create web pages. The number of programming languages that can be used to create web pages is very large. Of these, PHP continues to be one of the most popular ("TIOBE Programming Community

Index for December 2012,"). Therefore, PHP is a popular programming language taught to beginning web developers.

Developing web based applications requires different methods than does developing stand-alone applications (Wang, 2006). This usually requires the application of several software tools such as server side scripting languages and HTML. These technologies are often included in a single file. PHP causes difficulty for beginning programmers since it permits HTML statements to be embedded within PHP statements and vice versa ("PHP Manual,"). This two way transition from one language to the other increases the number of possible ways to write code that result in the same web page. Beginners of PHP programming need to be aware of these possibilities and be able to transition smoothly from one tool to another.

The students who want to learn PHP are very diverse. They vary in many of the aspects described above as well as in their previous experience in programming in other environments. Such previous experience in developing or using other programming environments very often causes difficulties for beginners of PHP.

Students coming from a non-web based programming background have difficulty understanding that web pages are stateless. This means that additional programming methods have to be used for passing data from one web page to another. A single web page can have two sets of input data: before submitting and after submitting. Before the page is submitted, it usually contains some display elements. However, once the page is submitted, it contains some user-supplied data as well. This makes it necessary to write different code for the different situations of the web page, thereby complicating the programming task.

Unlike many other languages, PHP is a dynamically typed language. This means that the type of a PHP variable is not fixed but can change with the value that is held by the variable at any given time ("PHP Manual,"). This complicates the process of comparing variables once they are given a value.

Another peculiarity in PHP is that it handles single quoted and double quoted strings in different manners ("PHP Manual,"). Single quoted strings are taken as standard literals. Double quoted strings can contain variables. Corresponding literals are obtained by replacing these variables with the values they contain.

The syntax of associative array elements within double quoted strings is different from their syntax in all other instances. This makes the syntax rules for PHP very confusing, especially for students coming from other programming backgrounds.

This means that any course designed to teach beginners of PHP, needs to address these peculiarities of PHP in addition to the concepts of programming in order to ensure that the students learn effectively.

This research was undertaken with the aim of finding a solution to the problem of teaching dynamic web development using PHP to a diverse range of individuals, in an effective and economically feasible manner.

1.3 RESEARCH GOAL AND OBJECTIVES

Based on the analysis above, the research problem addressed by this thesis can be defined as below.

Is it possible to create an Intelligent Tutoring System to effectively teach web development using PHP?

In order to answer this question, the goal of this study was to build and evaluate an Intelligent Tutoring System to teach the PHP web development language. Since programming is a very practical subject, it was decided that programming exercises would be used by the ITS to facilitate the process of learning. Students' solutions to such exercises would be analysed for correctness, and feedback would be provided based on the results of this analysis. Throughout this thesis, any answer submitted to a programming exercise will be considered as a 'solution'. More support for learning would be provided through web links to relevant web pages. The most suitable exercise for the current student would be suggested based on his/her current level of knowledge, in order to individualise the instruction.

A significant challenge here is the variety of correct solutions for a single programming exercise. For example, a programming exercise requiring a student to display a grade based on marks obtained can be written in many forms. It can be written as a series of *if* statements, a series of nested *if-else* statements or a *switch* statement. Additionally, the condition within the selection structure can be written in many forms. Similarly, a program which uses a loop can be written using *for* loops,

while loops or *do* loops. Again, the condition for exiting the loop can be written in many forms. In the case of PHP, any text to display on a web page can be written using HTML only, PHP only or a combination of both. Therefore, the number of correct solutions to a programming exercise can be very large. It is necessary that the system be capable of identifying all such variations. Although simple pattern matching mechanisms can be used to identify several equivalent solutions, this method becomes impractical with the large number of variations that are possible in PHP programming. Therefore, the main emphasis of the research is on representing the subject matter in a manner that makes it possible to identify such alternate solutions. It is also important to find methods of representing the student's current knowledge in order to individualise the instruction. These representations then need to be integrated to create an actual computerised system to effectively teach PHP to beginners with different levels of knowledge.

This meant that the main objectives of the research were to answer the following research questions.

1. *What is the best method of knowledge representation that can be used to model the subject matter necessary to effectively teach basic PHP programming while achieving the following?*
 - a. *Analysing alternative solutions to a given programming problem, both correct and incorrect*
 - b. *Providing feedback based on the specific errors made by the student*
2. *What is a suitable student model for the above system?*
3. *What methods of feedback and individualised interactions are useful to teach the above subject matter effectively through an ITS?*

1.4 SIGNIFICANCE

Research in Intelligent Tutoring Systems has been growing in momentum over the past few decades. Yet, ITSs are not a concept that is known extensively by educators. One of the main reasons for this is that, although many ITSs have been built, only a few are used in practical teaching situations. This indicates that there is

significant room for improvement in the field of ITS. This research attempts to improve on existing ITSs at least to a certain degree.

Existing ITSs teach in many different domains, from primary school reading to programming and electronic circuit design. The ITSs that teach programming languages such as Pascal, Prolog, C and Java, focus mainly on developing console and windows applications (Corbett, 2000; Song, Hahn, Tak, & Kim, 1997; E. Sykes, 2007). On the other hand, many computerised teaching systems that target web programming are available ("PHP Tutorial," undated; "PHP tutorial - free," ; "PHP/MySQL Tutorial,"). However, they present subject matter in the same way to each student, i.e. they do not individualise the instruction. The literature does not reveal any instance of the integration of these two ideas: i.e. ITSs that are designed to teach web programming. Therefore, this study addresses a domain that is totally new to ITS research.

It is essential that any ITS that teaches programming be capable of analysing computer code written by students. As described earlier, a programming exercise rarely has a unique solution. In order to analyse students' programs correctly, it is vital that the analysis process is able to accept alternative solutions to each programming exercise. This is true for any programming paradigm. However, programming for the web involves added complexities to program analysis. Web pages very often contain many technologies integrated within a single page. They also make it necessary to consider two states for every page: one before submitting the page and one after. Using PHP as the server side scripting language adds further complexity since it is possible to interleave HTML and PHP code using many combinations. All in all, the process of analysing programs written in PHP is very complex. This thesis develops methods of program analysis that are capable of accepting alternative solutions to a given programming exercise while also dealing with the complexity of PHP web development.

1.5 SCOPE OF THE THESIS

Building an ITS is a very time consuming task. It has been estimated that 200 to 300 hours are required to build an ITS to do one hour of teaching (Aleven, Sewall, & Koedinger, 2006; Murray, 1999). One of the main reasons for this is that the knowledge base of the subject matter is usually very specific to the subject

being taught. This means that a new knowledge base has to be created in order to teach a new subject. Therefore, creating a new knowledge base to analyse programs written in PHP is a significantly time consuming task.

As can be seen from the above description, building an ITS that teaches all the intricacies of PHP would be extremely difficult. It is simply not possible to complete such a task within the time constraints of a PhD. Therefore, this ITS only teaches the main aspects of the PHP language that are required by a beginning programmer. In particular, it teaches the concepts of assignment, selection using *if*, nested *if* and *switch* statements, predefined (some) and user defined functions, HTML form processing for *text*, *select* and *submit* inputs, *for* loops with only a single condition, *while* loops which can be converted into *for* loops, associative and indexed arrays and *foreach* loops for accessing array elements. Other parts of the PHP language are not handled in this tutoring system. However, it does handle a few other HTML elements such as tables and also a few HTML attributes such as *name* and *border*. A website designed using PHP usually incorporates Javascript or a similar client side language to handle validation and other aspects. The PHP ITS does not teach any sort of client side scripting whatsoever.

An ITS contains a student module in order to customise its interaction for the current student. To do this, the system ideally needs to have very good knowledge about the student, including his/her age, gender, capabilities, emotions and numerous other characteristics. The focus of this research is not on the detailed design of the student module. Therefore, the student module used here considers only the student characteristic that is most directly related to learning: i.e. the current knowledge level of the student in the subject matter being taught.

Another function of many ITSs is to provide feedback to the student. The feedback in this system is provided using several levels. The feedback would support better learning if the level of feedback provided was customised based on the current knowledge level of the student. However, since this thesis does not focus on an advanced teaching module, such customisation is not provided.

1.6 THESIS OUTLINE

This section outlines the remainder of the thesis. Chapter 2 contains a review of the literature that is pertinent to this research. Chapter 3 looks at the research

design used and the reasons for this. The next four chapters (Chapter 4,Chapter 5,Chapter 6 and Chapter 7) provide a detailed description of the knowledge base and how it is used to analyse PHP programs. Each of these four chapters looks at a different set of PHP constructs and how they are modelled. Chapter 8 describes the user interface of the system as well as the design of the student and teaching modules. Chapter 9 discusses how the PHP ITS was evaluated and the results obtained from this evaluation. Chapter 10 discusses the results of this evaluation and the implications for future work. The rest of the thesis contains a set of Appendices that further support the explanations provided throughout the thesis. It provides detailed diagrams, further examples and detailed data that are too lengthy to incorporate within the main body of the thesis.

Chapter 2: Literature Review

This chapter investigates the existing body of literature that is related to this research project. It begins with an examination of why introductory programming is a difficult subject to teach (Section 2.1). Section 2.2 looks at the concept of Intelligent Tutoring Systems (ITSs) and how they can be used to overcome some of these problems. Section 2.3 discusses how the domain models of existing ITSs to teach programming are designed. Section 2.4 investigates typical features of the teaching module of an ITS. Section 2.5 looks at the common methods of modelling students in ITSs. Section 2.6 compares the features of existing ITSs, that teach programming, that are pertinent to this thesis. Finally, Section 2.7 presents a summary of the chapter and its implications.

2.1 TEACHING INTRODUCTORY PROGRAMMING

A large number of students have difficulty in learning to program. This is shown by the fact that in 2003, 35% of students at the Queensland University of Technology failed their first programming course (Truong, Bancroft, & Roe, 2003). The situation is similar in Victoria University, where a large number either drop out or fail programming courses (Miliszewska & Tan, 2007). Understanding the reason for this difficulty has been the focus of a large body of research. This section investigates some probable causes as to why beginning students find programming so difficult.

Five main areas in learning computer programming, as identified by Mow (2008), form a good basis for understanding why programming can be difficult for beginners. These five areas are cognitive requirements, syntax and semantics, orientation, auxiliary skills and resource constraints.

2.1.1 Cognitive Requirements

In order to write correct computer programs, students need to understand abstract concepts then convert these into concrete solutions (Gomes & Mendes, 2007; Miliszewska & Tan, 2007). The problem must first be solved using a conceptual approach before a computer program can be written using a particular programming language (Mow, 2008). In doing so, students need to utilise skills in

program design and creative thinking (Al-Imamy, Alizadeh, & Nour, 2006). When creating a solution, they need to concentrate simultaneously on the syntax and the algorithm construction (Gomes & Mendes, 2007). This means that the entire process requires the interaction of many cognitive skills, making it very challenging for beginners.

2.1.2 Syntax and Semantics

The fact that exact syntax rules must be followed in order to write a correct computer program is a concept that is very difficult for many beginners to grasp. They often find the semantics of the many programming constructs very complicated. Ebrahimi (1994) found that many novices had questions such as “what is the difference among loops” and “how is a value bound to its variable”. Research has found that some programming constructs are more difficult than others for novices. For example, students make more mistakes with loops and conditionals than they do with other types of statements (Spohrer et al. as cited in Robins, Rountree, & Rountree, 2003). Arrays are another problem area with many having trouble with confusing the subscript and the actual value stored (du Boulay as cited in Robins et al., 2003). The fact that some programming language constructs use words similar to standard English, but having a different semantic meaning is a common source of confusion (Ebrahimi, 1994). The concepts of Object Oriented Programming (OOP) are another major cause of concern for beginning programmers. Many tend to assume that objects are automatically created and do not need to be instantiated (Robins et al., 2003).

2.1.3 Orientation

Many students come to their first programming course with the pre-conceived idea that programming is a difficult subject (Gomes & Mendes, 2007). Others fail to understand the importance of the practical aspect of computer programming and attempt to pass the subject by simply memorising textbooks (Gomes & Mendes, 2007) without writing any actual code. Such inappropriate orientation based on incorrect practices and attitudes make programming a difficult subject for some students.

2.1.4 Auxiliary Skills

One of the most widely accepted impediments to beginning students is the fact that a good programmer needs to have a many auxiliary skills, among them logical reasoning, problem solving (Gomes & Mendes, 2007; Miliszewska & Tan, 2007) and planning (Al-Imamy et al., 2006; Ebrahimi, 1994; Ebrahimi & Schweikert, 2006; Robins et al., 2003). In fact, Spohrer & Soloway (1986) found that the most common source of bugs in computer programs is in plan composition. Attempting to learn programming without developing these auxiliary capabilities is an oversight made by many beginning students.

2.1.5 Resource Constraints

In addition to the above difficulties, the structure of existing introductory programming courses makes it very difficult for beginners to learn the subject in any depth. Since students in many disciplines need to know how to write computer programs, a first programming course typically has students from varying backgrounds with varying degrees of relevant skills. Strict time constraints imposed by the semester system employed in many universities worldwide (Al-Imamy et al., 2006) makes it extremely difficult to cater to the diverse needs of these students. A large amount of the available time needs to be spent in explaining the basic concepts and syntax, leaving very little time to build up the other necessary skills to be able to write correct and efficient programs.

The above classification identifies the main reasons why teaching introductory programming is proving challenging. Any solutions that are proposed to make the subject easier must find means of overcoming at least some of these difficulties.

2.2 INTELLIGENT TUTORING SYSTEMS

2.2.1 Background

Researchers have classified the approaches to the problem of teaching novice programmer using several methods. Mow (2008) categorised the potential approaches to these problems into three groups: pedagogical, technological and content-based. *Pedagogical* solutions focus on using different teaching strategies to maximise learning. *Technological* solutions employ computer technology to create more effective learning environments. *Content-based* solutions make use of the

different types of content knowledge required by students in order to facilitate learning.

A different classification of approaches was employed by Miliszewska & Tan (2007). They mentioned four main pedagogical techniques to support beginning programmers: analogy, relevance, continuous reinforcement of concepts and use of technology for teaching. *Analogy* refers to the use of illustrative examples of concepts that students have seen before. *Relevance* refers to showing students a purpose for what they are learning. *Continuous reinforcement of concepts* refers to repeatedly reminding the students of what they have learnt. Finally, *using technology* refers to the use of computer technology to support learning.

It can be seen that technology plays an important role in both these classifications. This shows that computerised learning systems have been seen as a way forward to the problem of teaching programming to novices.

Computerised learning systems take many forms – among them web resources and desktop learning environments. Websites that teach various programming subjects are ever popular because of their wide availability. They are accessible from anywhere in the world and available for the students to use at whatever time they require. PHP Tutorial ("PHP Tutorial," undated), PHP/MySQL Tutorial ("PHP/MySQL Tutorial,"), PHP Tutorial – free ("PHP tutorial - free,") and PHP:A simple tutorial – manual ("PHP: A simple tutorial - manual,") are just a few of the large number of PHP tutorials that are freely available on the Internet.

Although the websites described above provide good factual data for beginning programmers, they provide the same set of facts in the same order for each student. However, each student is an individual who learn at his own pace, based on his own style. This means that, when using the above systems, as well as in a typical classroom situation, many students are at a disadvantage since their learning needs may differ from those considered by the class tutor. That is why individual human tutoring is the most effective form of instruction (Corbett, 2001). In fact, a seminal study in the field of education by Bloom (1984) found that students taught on an individual basis achieved a final examination score that is two standard deviations higher than those taught in a traditional classroom situation. Although individual tutoring has this high success rate, it is extremely expensive in terms of both physical and human resources. Therefore, such instruction is usually difficult to provide in

novice programming courses which typically consist of a large number of students. The solution to this dilemma is to use what are known as Intelligent Tutoring Systems (ITSs). These are computerised learning systems that alter their interaction based on the requirements of each individual student. Therefore, they provide the benefit of individual tutoring while reducing the additional costs.

2.2.2 Architecture of Intelligent Tutoring Systems

In order to function properly, an Intelligent Tutoring System needs to have many modules. One common classification of the modules that comprise an ITS (Woolf, 2009) is shown in *Figure 2.1*. In order to understand the functionality of each of these modules, consider a situation where the ITS provides a problem for a student to solve. The problem is presented through the communication module which is what handles all interactions between the student and the ITS. Next, the student enters his/her solution via the communication module. The teaching module then considers this solution together with information it obtains from the student and domain modules. The domain module contains details of the subject matter that is taught by the ITS and therefore contains information about the correct solution to the problem. Based on this information, the teaching module decides whether the solution is correct or not. The student module contains information regarding the characteristics of the current student. The teaching module uses this information to decide what sort of feedback it should provide to the student. Whatever the decision of the teaching module, the feedback is provided to the student through the communication module. Meanwhile, the system forms an opinion about the student's knowledge of the subject matter being taught by the current problem. This information is updated to the student module in order to have a more accurate model of the student.

Although this is the architecture that is used most commonly for ITSs, some other architectures have been suggested by researchers. Of these, the architecture suggested by Pillay (2003) is of interest since it has been developed with ITSs that teach programming in mind. This is actually an extension of the architecture described above and contains 10 modules as shown in *Figure 2.2*. The interface module in this architecture is the same as the communication module in the previous architecture. The domain module in the architecture in *Figure 2.1* is made up of the domain module, the problems module, the expert module and the code specification

module in *Figure 2.2*. In this case, the domain module is where the skills that are tutored are stored. The problems module contains the actual problems used to teach these skills and the expert module analyses students' solutions to the problems. The code specification module is used to ensure that the architecture is independent of the programming language used. This module stores solutions algorithms to programming problems in a language independent manner. The explanations module, pedagogical module and the instructional strategies module in *Figure 2.2* corresponds to the single teaching module in *Figure 2.1*. The pedagogical and instructional strategies contain the relevant strategies employed by the system while the explanations module is responsible for generating the actual explanations of any errors made by the students. The learning/ experience module is used to improve the tutor performance over time by learning from student actions.

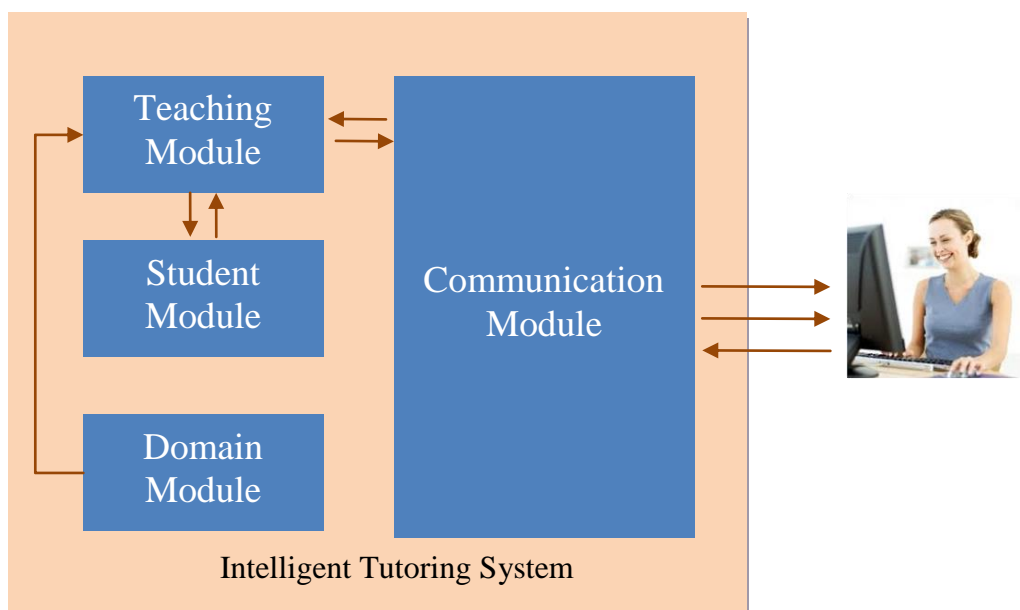


Figure 2.1. Main modules of an Intelligent Tutoring System.

This architecture is specific for a tutor to teach programming. Some modules have been incorporated in order to make it generic for all tutors that teach programming. In general, it is not necessary to use such a generic architecture as the requirements of a particular tutor are pre-defined. Therefore, this thesis uses the more commonly used architecture shown in *Figure 2.1*.

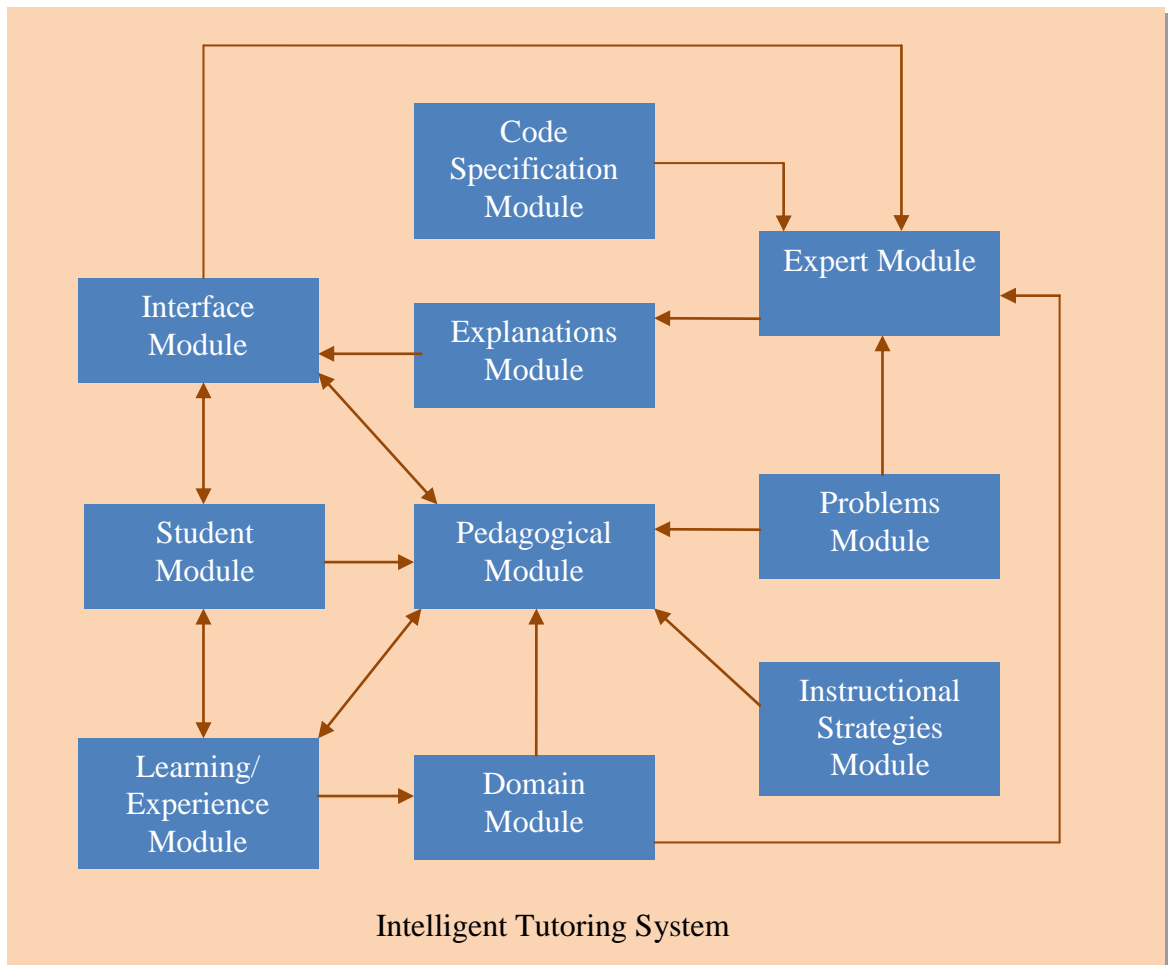


Figure 2.2. Generic architecture for ITS to teach programming.

2.2.3 Domains Taught by Existing ITSs

The concept of Intelligent Tutoring Systems have been used to teach subject matter in many domains. The Practical Algebra Tutor (Koedinger, Anderson, Hadley, & Mark, 1997; Koedinger & Sueker, 1996) and Ms Lindquist (Heffernan) both teach Mathematics. The Andes Physics Tutoring System (VanLehn et al., 2005) is a successful ITS which teaches Physics. The Cardiac Tutor (Eliot, Williams, & Woolf, 1996) teaches medical personnel how to handle cardiac arrests while CAPIT (M. Mayo, Mitrovic, & McKenzie, 2000) teaches English grammar.

Database theory is another area that has attracted the design of many ITS. The SQL-Tutor (Mitrovic, 1998) is one of the most successful ITS of all time and teaches students how to write SQL queries. KERMIT and NORMIT (Mitrovic, Suraweera, Martin, & Weerasinghe, 2004; Suraweera & Mitrovic, 2002) are ITSs that teach theories of database design.

The Personal Access Tutor (Risco & Reye, 2009) is somewhat different from other tutors in that it functions as an Add-in to Microsoft Access to teach students how to design forms and reports.

Many tutors that teach computer programming in many different languages have also been developed. These include the ACT Programming Tutors (Corbett, 2000), ELM-ART (Weber & Brusilovsky, 2001), C Tutor (Song et al., 1997), CPP Tutor (Naser, 2008), JITS (E. Sykes, 2007) and J-LATTE (Holland, Mitrovic, & Martin, 2009). It can be seen that although ITS covering many domains have been developed previously, none of them teach web programming in any form.

2.3 THE DOMAIN MODULE

The domain module is an important component of an Intelligent Tutoring System as it is what contains the subject matter that is being taught by the system. Additionally, the other modules are generally built around the domain module so the representation used here becomes very important.

When creating an ITS to teach programming, an important concept is the fact that programming is a very practical skill and students need to be given ample opportunity to practice in order to learn programming well (Gomes & Mendes, 2007). However, just supplying programming exercises to the students is insufficient. In order for the students to learn from them, it should be possible to analyse their solutions to determine if they are correct. This process of automated program analysis is very important in ITSs that teach programming. However, it should be noted that some other computerised systems that teach programming, and not only ITSs, perform program analysis. This section discusses methods that have been used by previous systems for such analysis.

2.3.1 Static and Dynamic Program Analysis

Computer programs can be analysed using static and dynamic methods. In *static analysis*, the program code is typically tested against programming standards, acceptable programming practice and efficiency guidelines. The code is not executed at any time during the analysis. The MENO-II system (Soloway, E.M., Woolf, B.P., Rubin, E. and Barth., P. as cited in Wenger, 1987), the Prolog Programming Environment (Gegg-Harrison, 1991) and the Prolog Tutor (Hong, 2004) are examples of systems that carried out static analysis. However, the process

used for static analysis is able to identify only a small number of specific solutions to a given programming problem. This is not the scenario in the real world. Each programming problem has many correct solutions since the many programming constructs can be manipulated in different ways to obtain the same output.

This is the idea that is utilised in *dynamic program analysis*. Here, the program is directly executed on a set of test data to see whether the expected output is obtained. Whatever the program statements used, the program is identified as correct as long as the output is correct. This makes dynamic analysis suitable for identifying alternative solutions to a given problem. However, it does not consider the exact means of arriving at the output. Therefore, the program is identified as correct even if a very roundabout method is used. This may not be an acceptable solution within the bounds of acceptable program practice. Additionally, it is possible that the output for the test data is correct due to the characteristics of the actual data values selected. It may give an incorrect output when a different set of input data is used. This means that there is no guarantee that the program works correctly for all values of data.

The shortcomings of each of these methods of program analysis can be reduced by combining both for program analysis. The C-Tutor (Song et al., 1997), ELM-PE (Weber & Möllenberg, 1995), the ELP system (Truong et al., 2003), the Basic Instructional Program (BIP) system (Barr, A., Beard, M., and Atkinson, R.C., as cited in Wenger, 1987) and Analyser for PROlog Programs of Students (APROPOS2) (Looi, 1991) are examples of systems that utilise a combination of these methods. Although the analysis provided in these systems is useful for identifying alternate solutions to a single problem, the methods used are limited to a particular programming language and cannot be used for others.

2.3.2 Knowledge Representation

Static analysis requires that the program code itself is analysed. In doing so, it becomes necessary to separate the syntax rules from the semantic aspects of the program. This means that the program needs to be represented in a manner which enables this distinction to be made. Methods of representing the program for this purpose have been the focus of much research in the field of using technology to teach programming. The characteristics of a suitable representation depend on the

requirements of the user as well as external factors that affect how the knowledge is obtained, as described below.

One main external factor of how the subject matter is represented is the origin of that subject matter. It is the domain expert that provides the subject matter that needs to be taught by a computerised system, based on his/her experience in working in the application field. As pointed out by Hatzilygeroudis & Pretzas (2004), this domain expert is typically a tutor who does not necessarily have familiarity with knowledge engineering concepts. Therefore, it is essential that the method of representation used is as natural as possible. It is also important that the knowledge represented is easily updateable by the domain expert and that a solution can be analysed in a time efficient manner. It should also support the possibility of the system offering appropriate explanations based on the results of the program analysis.

One of the earliest methods used to analyse computer programs was to maintain libraries of bugs. In this method, a list of possible errors in a students' program are stored. The student's solution is analysed to see whether any of these bugs are present. A student can write a program in an infinite number of ways, and it is simply not possible to enumerate all possible bugs in a program. Additionally, in order to initially develop the bug library, it is necessary to conduct empirical studies on the types of errors made by students. Even if a library was constructed in this manner, such bug libraries do not typically transfer well between different student populations (Ohlsson & Mitrovic, 2007). Another problem is that it becomes necessary to create a library of possible bugs for each exercise that needs to be analysed. This means that a system created using this method cannot easily be expanded to handle additional exercises. Due to all these reasons, the bug library approach is not a very suitable method of analysing student programs for an Intelligent Tutoring System.

The MENO-II system uses a very natural and simple method of knowledge representation (Soloway, E.M., Woolf, B.P., Rubin, E. and Barth., P. as cited in Wenger, 1987). Here, the student's solution is compared against a series of templates that represents the correct solution. The analysis is carried out by comparing a stored correct solution against the student's answer.

This method of analysis is quite effective for basic computer programs since they were typically written using code in a single method. However, a programming exercise very rarely has a unique solution. This becomes more apparent as the programs become more complex. The method of analysis used in the MENO-II system is incapable of accepting such alternate solutions to a single exercise. Therefore, it is not a very suitable method especially with more complicated programs.

Concepts of programming language semantics have been used to address the problem of identifying different solutions to a single programming problem (Winskel, 1993). Here, mathematical models of programs are used to serve as a basis for understanding and reasoning about their behaviour. Programming language semantics are categorised into three main strands: operational semantics, denotational semantics and axiomatic semantics. Both operational and denotational semantics focus on describing the meaning of the programming language. Axiomatic semantics try to fix the meaning of a programming construct by giving proof rules for it within a program logic. Therefore, this form is useful for proving that a program is correct. Axiomatic semantics uses a set of logical rules known as Hoare logic for program verification (Huth & Ryan, 2004). This process basically starts at the end of the program and proceeds backwards. For each program statement it encounters, it uses a rule to find the pre-condition based on the post-condition. This method is meant to be used manually and can only be partly automated. Additionally, it cannot be used as a basis for explaining the reason for any identified errors. This means that axiomatic semantics by itself is not sufficient as a basis for analysing student programs and providing appropriate feedback.

Symbolic rules are another commonly used formalism for knowledge representation (Hatzilygeroudis & Prentzas, 2004). These are in the form of if-then rules. They follow inference steps and are highly modular.

The cognitive tutors designed to teach programming in LISP, Pascal and Prolog (Anderson, Corbett, Koedinger, & Pelletier, 1995; Corbett, 2000, 2001; Corbett & Anderson, 1992), use symbolic rules of this form for knowledge representation. They are based on the ACT-R theory of cognition (Anderson, 1993, 1996). This theory concerns declarative knowledge and procedural knowledge. *Declarative knowledge* is in the form of facts. On the other hand, *procedural*

knowledge applies this declarative knowledge to solve problems. Therefore, it is goal-oriented and impacts problem solving behaviour. Such knowledge can be represented as a set of independent production rules that associate problem states and problem-solving goals with actions and state changes. The solution to an exercise in the cognitive tutors is stored as a set of these production rules incorporating the underlying skills that are required to solve the exercise. As the student enters a solution to the exercise, it is matched against this set of production rules. If the entire solution is typed in and the production rules are matched, the solution is identified as correct. Any discrepancy from the production rules is immediately identified as incorrect. This approach to solution analysis is known as model tracing since it traces through an ideal model stored in the system. Several sets of production rules that constitute a correct solution can be stored in this manner. However, the number of solutions that can be stored are limited and it becomes more difficult to accept alternative solutions as the exercise get more complex.

Constraint Based Modelling (CBM) (Ohlsson & Mitrovic, 2006) is another method of modelling the knowledge base that uses symbolic rules to represent knowledge and identify alternative solutions to a single exercise. They contain a set of if-then conditions which are known as relevance conditions and satisfaction conditions. Each part of the student's solution is analysed to see whether the relevance conditions are met. If so, it is again checked to see whether the corresponding satisfaction conditions are met. Any rule where the relevance condition is met but the satisfaction condition is not met indicates that there is an error in the solution. This means that CBM does not rely on explicitly storing alternative correct answers but works on analysing the result of the solution. For this reason, it is capable of accepting multiple solution paths that arrive at the correct answer. Although CBM is a very important method of knowledge representation in computerised tutoring systems, its use has been primarily in the domain of database concepts. Although it has been used to create a system to teach programming (Holland et al., 2009), its use has been very limited compared to the much more successful systems for database concepts.

A main disadvantage of using symbolic rules for knowledge representation is that, when the number of rules becomes very large, inference and knowledge acquisition becomes difficult (Hatzilygeroudis & Prentzas, 2004).

An approach that is used extensively to identify alternate solutions to a given problem is to convert the program code into a standardised form. The standardised form is then compared against a solution that is stored in the same standardised form. Different standardised forms have been proposed.

One such approach is to convert a submitted solution into a linkage graph (Jin et al., 2012). A linkage graph is a directed acyclic graph whose nodes represent program statements and directed edges indicate dependencies between the different statements. This graph is represented as a two dimensional matrix. Equivalent programs have equal matrices, thereby allowing accepting alternative solutions to a single exercise. However, the published work only deals with the assignment statement. The probability that this method will be able to produce equal matrices for logically equivalent programs using other programming structures is, as yet doubtful.

An older system which uses the approach of converting programs into standardised form is LAURA (Adam & Laurent, 1980). Here, the system is given an implementation of a correct solution to a programming problem. The system converts this into an internal representation of the corresponding calculus processes. This is stored in the form of a graph. The student's program is also represented as a graph using the calculus process that is implied by it. The graphs are then transformed using certain rules and compared. Any differences in the transformed graphs are used to identify errors in the student's program. This method of program analysis uses heuristics for certain graph transformations. Additionally, for accurate analysis, it is necessary to have a great knowledge of the field in which the task has a meaning in addition to the task the program has to perform. These requirements make this method unable to identify certain differences automatically.

Programming statements can be written in many forms using a variety of structures. It is often useful to convert the flat structure represented by a program's source code, into a structure that better represents the overall structure of the program. An Abstract Syntax Tree (AST) is such a representation. The form of the AST is dependent on the structure of the program. Therefore, alternative solutions to a single program have different ASTs. Several researchers have proposed converting these ASTs into a canonicalised form (Rivers & Koedinger, 2012; Truong et al., 2003). This means that the AST obtained from the student's program is converted to

a standard form using a set of rules. Although this method seems suitable for identifying alternative solutions to small programming problems, it is difficult to see it being expandable for larger programs.

Computerised tutoring systems to teach Prolog have been developed with the aim of standardising student's solutions in order to identify alternative solutions. The Prolog Programming Environment (Gegg-Harrison, 1991) contains a set of schemata that can be used to represent a Prolog program. When a student submits a solution, it is converted to a canonical form using these schemata and is then compared against the expected solution which is also stored in this canonical form. In the Prolog Tutor (Hong, 2004), a single reference program is stored against each programming exercise. A set of programming techniques, that can be used to write Prolog programs, are also stored. When a student solution is submitted, the programming techniques used are first identified. Then, both the student's solution and the reference program are parsed using the same set of programming techniques. The results of the two parses are then compared to identify any errors. Although these methods of standardising solutions have proved useful, they are restricted to Prolog programs since the concepts of schemata and programming techniques do not translate across all programming languages.

All the methods of program analysis described in this section analyse the student's code itself and do not attempt to identify what the student intended to do by writing that particular piece of code. However, trying to understand what the student intended to do from a particular piece of code has proved to be useful in providing appropriate instruction.

2.3.3 Intention Based Analysis

Novice programmers make many errors while writing programs. An important concept that is used in many teaching systems is that bugs are not properties of programs alone but properties of the relationship between the programs and the intentions (W. L. Johnson & Soloway, 1985). This is the idea used in systems that perform intention based analysis. Such systems attempt to identify the purpose of the student when writing a specific programming statement. Based on this, it then decides whether the student is on a correct solution path even if the final solution is not correct.

One of the oldest and most famous systems that used this approach is PROUST (W.L. Johnson, 1990; W. L. Johnson & Soloway, 1985). Here, implementation methods that are commonly used in writing programs are identified and stored in the form of programming plans. These plans include both correct and incorrect versions. Expected solutions to exercises are stored as goal decompositions consisting of these plans and form the various interpretations of the solution. When a student submits a solution to an exercise, it is analysed against the goal decompositions to try to identify which interpretation the program fits into. A set of transformation rules are also maintained to modify the code to match existing plans. Heuristics are used to determine which interpretation a solution most closely fits into in order to determine the intention of the student's program. PROUST is able to analyse many alternative solutions by generating new interpretations based on the program it is currently analysing. However, with the increase in the number of programming plans stored in the system, it becomes harder to identify the actual plans used by the student. This is mainly because the system takes a lot of time to consider all these solutions and decide on a probable interpretation of the student's solution.

Results of an evaluation of PROUST showed that it was sometimes unable to interpret the programs of the students based on the set of plans that it contained. This meant that it could not provide appropriate error diagnosis in such cases (W. L. Johnson, 1985). The evaluation process considered only two programs and therefore, there is no evidence regarding how it would perform on analysing other programs. Also, its explanations were somewhat difficult to use. It was not coupled with a tutoring module or a student module and was described as a program debugger and not an ITS.

The weaknesses of PROUST inspired the building of another programming debugger – CHIRON (Sack, Soloway, & Weingrad, 1992). This attempted to solve the problem of not being able to interpret certain programs by identifying what is correct in the program and not what is incorrect. This meant that it did not contain a set of mal-rules as did PROUST. A hierarchical representation of knowledge made it possible to describe errors using a better approach than PROUST. However, the error messages were still somewhat difficult to understand. Also, the knowledge level of each student was not considered when displaying error messages or for any

other purpose. Although the literature describes a prototype of CHIRON, it does not include details of any empirical evaluation to see whether it was successful.

Intention based analysis is also used in the more recent CPP-Tutor (Naser, 2008). This tutor stores a correct solution to each problem. When a student submits a solution to a programming problem, the system calculates an edit distance between the student's solution and the correct solution using pattern matching in order to identify the intent of the student. Feedback is then provided based on this analysis. In the C-Tutor (Song et al., 1997), the intention of each problem is stored as a goal plan hierarchy. Once a student enters a solution to the problem, it is converted into a similar plan hierarchy, in canonical form, by the system. This plan is then compared against the above goal plan hierarchy in order to identify the intention of the student.

The Prolog Intelligent Tutoring System (PITS) also uses intention based analysis to analyse Prolog programs (Looi, 1991). The program debugger of this system first uses heuristic code matching to analyse different aspects of the program. The errors identified here are general and apply to any programming task. During the next stage, errors specific to the particular programming task are identified. If code matching fails to detect the errors, dynamic analysis is performed to see if the objective of the program is satisfied. This multi-level approach makes PITS very versatile in identifying errors. The Java Intelligent Tutoring System (JITS) is another e-learning system that uses intention based analysis to guide each student towards a potentially unique solution (E. Sykes, 2007). A problem specification is stored in the system but not a corresponding solution. JITS identifies the intent of the student based on his/her program and attempts to guide him/her towards a solution that is correct. JavaBugs (Suarez & Sison, 2008) is another tutor that uses intention based analysis. Here, the student's solution is analysed against a set of stored correct programs. Intentions are identifying by comparing matching classes, attributes and methods. Any discrepancies are identified as errors. This method is only suitable for analysing programs written using Object Oriented concepts.

Whatever the method of analysis used, the final goal of a system designed to teach programming should be to properly identify a student's program as correct or incorrect. The different methods of analysis discussed here are successful in achieving this, to different degrees. When designing the domain module of an ITS to

teach programming, it is necessary to select a method that is suitable for the proposed system.

2.4 THE TEACHING MODULE

As described in Section 2.2.2, the teaching module is an important component of an ITS. This module concentrates on methods to provide better learning to students. It utilises concepts from many different disciplines, mainly Cognitive Science and Education.

The teaching module in an ITS to teach programming concentrates on different aspects of teaching. This section concentrates on the aspects of the teaching module that are relevant to the this thesis.

2.4.1 Feedback

Analysing a program alone is not sufficient for learning to occur. It is necessary for the system to analyse the program and provide appropriate feedback. This idea has been utilised in many systems to provide different types of feedback and other methods of support to students using ITSs to learn programming.

One interesting study to understand what causes learning was carried out by analysing hours and hours of tutorial dialogue (VanLehn, Siler, Murray, & Baggett, 1998). This study indicated that, in order to achieve some form of learning, students need to make an error or reach an impasse. Many computerised tutoring systems for programming identify such an error or impasse when a student makes a mistake in answering an exercise on writing a computer program or asks for help. During such events, the tutor can either indicate that an error has been made (verification) or provide more detailed explanations about the error (elaboration) (Mason & Bruning, 2001). Although both these methods have been used in previous tutors, studies have shown that elaboration provides better learning than simple verification (Singh et al., 2011).

The timeliness of the feedback is another factor that plays an important role in learning (Singh et al., 2011). Providing the proper feedback at the wrong time could result in students becoming confused. Some tutors, such as the cognitive tutors mentioned earlier (Anderson et al., 1995) provide feedback as and when a student makes an error. This type of feedback is known as *proactive feedback*. However,

research has shown that too much help can actually prevent learning (VanLehn et al., 1998). Additionally, proactive feedback does not allow students to realise on their own that they have reached an impasse. Research has shown that for effective learning, it is necessary for the students to be aware that they have some form of knowledge deficit (VanLehn et al., 1998). Therefore, it is more beneficial to let the student ask for feedback when s/he realises that such a deficit is present. This type of feedback is known as *on-request feedback* and has been utilised in many computerised learning systems to teach programming (referred to as ‘systems’ for the rest of this literature review) (Chee, 1994; Gegg-Harrison, 1991; Hong, 2004; Looi, 1991; Weber & Brusilovsky, 2001; Weber & Möllenberg, 1995). Such feedback has proved beneficial since the students themselves need to determine that they need help and are therefore more open to accepting assistance from the system.

When using elaboration to provide descriptive feedback, two main methods are used: scaffolding and hinting (Razzaq & Heffernan, 2006). In the scaffolding situation, students are asked questions, thereby allowing them to determine the reason for their error. This is useful to build up the cognitive abilities of the student as well as to correct the more immediate problem in the program. In hinting, the system indicates to the student what is wrong. Studies have shown that students forced to do scaffolding perform better than those given hints (Razzaq & Heffernan, 2006). However, students need a longer time to work with scaffolding. This means that any system that uses scaffolding over hinting should ensure that students have plenty of time to work on the problems.

Providing feedback alone is not sufficient for students to learn. Many IDEs used for programming provide some sort of feedback through compiler error messages. However, these messages are not very user friendly and can cause confusion to novices. Therefore, it is essential that any feedback provided by the system is user friendly (Truong, 2007). On the other hand, research has shown that providing very strong hints can actually result in the students missing the opportunity to learn (VanLehn et al., 1998). Therefore, the level of feedback is an important consideration when providing error messages.

An important theory in Education, related to the level of feedback, is the Zone of Proximal Development (ZPD). In this theory, Vygotsky (1978, p. 84) describes ZPD as “the distance between the actual development level as determined by

independent problem solving and the level of potential development as determined through problem solving under adult guidance or in collaboration with more capable peers.” In other words, ZPD refers to the range of tasks that are too difficult for an individual to master with his current level of knowledge, but can be mastered with the assistance of a more skilled person. Learning occurs best when a tutor gives guidance in the ZPD of the student. This enables the student to improve his knowledge, thereby altering his ZPD.

The ZPD of each individual student is different. This means that, an error message may be suitable for certain students to increase their knowledge while it can be useless to others. Maintaining a single level of error messages, like in the ELM-ART system (Weber & Brusilovsky, 2001), is not very beneficial to improve the knowledge of a wide cross-section of students. In order to avoid this problem, many systems provide a multi-levelled approach to feedback (Chee, 1994; Garner, 2007; Kemp, Kemp, & Todd, 2009; Mason & Bruning, 2001). The actual number of levels varies from system to system but the general concept used is the same. Students can then obtain the most suitable level of feedback based on their particular knowledge level. In some systems, the system automatically determines the most suitable level for the current student and displays the error message (Suraweera & Mitrovic, 2002). This could prove problematic as students sometimes want more detailed error messages while at other times want just a very general hint. Therefore, it is better to allow the students to choose the level of feedback themselves. Many systems first provide a very general error message but allow students to manually move on to more detailed descriptions if they desire it (Chee, 1994; Koedinger et al., 1997; VanLehn et al., 1998; Weber & Möllenberg, 1995). This method has proved more successful since students are in charge of their learning.

In addition to the many types of feedback provided when a student makes an error, some systems actually go a step further and correct the errors in the student's programs. The CPP-Tutor (Naser, 2008) and JITS (E. Sykes & Franek, 2004) are examples of such systems. Both these systems first ask the student questions to determine whether the error correction suggested by the system is acceptable to the student before making the actual change. However, automatically correcting errors can actually hinder student learning since students miss an opportunity to learn by themselves (VanLehn et al., 1998).

2.4.2 Next Problem Selection

Any system that teaches using exercises needs some method to determine the next exercise to present to the student. Many systems present the exercises to a student in a preset order (Weber & Brusilovsky, 2001). Others present a list of exercises and allow the students to select which exercise they want to attempt next (Weber & Möllenberg, 1995). In both these types of systems, the next exercise presented to the student does not depend on the abilities of the student. This method is not very suitable since it has been shown that continually encountering problems that they are unable to solve results in a negative psychological effect on students (Mow, 2008).

In order to cater to this need, some systems look at the current knowledge level of the student and provide the exercise that is most suitable for his/her current level of knowledge. This is done using the concept of ZPD described in Section 2.4.1. The subject matter is broken down into knowledge components (KCs) and the KCs covered by each exercise are maintained. The most suitable problem for the current student is considered to be the one with the least number of unknown KCs for that student. Many systems automatically select the next best exercise in this manner and present it to the student (Song et al., 1997; Weber, 1996; Wenger, 1987). Others function on the concept of mastery learning (Anderson et al., 1995; Corbett, 2000). These systems provide students with more and more exercises that cover the same KCs until the student has achieved mastery of those KCs. Then, they select the next best exercise as described above. Although this method of selecting the next best exercise is useful in individualising the interaction, it has some disadvantages. The system is not always a hundred percent correct in its estimate of the student's knowledge. Additionally, even if the system is correct, some students may simply not feel confident enough in using some KCs and may simply wish to practice them more. In such situations, the system does not allow them to do so, forcing them to work on the next best exercise. The solution to this problem is for the system to suggest the next best exercise based on the KCs, but allow the student to select either that or a different exercise based on his/her requirements (Naser, 2008). This method supports the student by individualising the instruction while allowing the student to also control his own learning.

2.4.3 Other Forms of Support

In addition to customised levels of feedback and method of next problem selection, some systems provide other forms of support for the students to learn effectively. This section discusses such forms of support that contribute to this thesis.

As discussed in Section 2.1, the process of writing an entire computer program from scratch requires the integration of many cognitive skills. This makes it more difficult for novices (Chee, 1994; Kolling, 2010; Miliszewska & Tan, 2007; Truong, 2007). The number of skills required can be minimised by requiring beginners to complete segments of code rather than to write complete computer programs (Al-Imamy et al., 2006; Kolling, 2010; Truong et al., 2003). This makes it easier for them to concentrate on fewer aspects of programming since they do not then need to worry about the more complicated issues of designing an entire program. This concept has been converted to computerised learning system by Kolling (2010) who stressed the fact that such systems should never start with a blank screen. This notion has been utilised in the Environment for Learning Programming (ELP) (Truong et al., 2003) which presented gap exercises for students to complete. An interesting variation of this was utilised by Garner (2007), where the students were mainly required to select from a list of provided program statements and order them. In addition to reducing the cognitive load, this method also made it unnecessary for the students to remember the exact syntax of statements, thereby allowing them to concentrate more on program design.

Many other methods have been used in technology based systems to make it easier for the students to follow the syntax rules of the language. Some systems provide coding templates for the students to fill in so that they are not required to remember the intricacies of the syntax. These templates can be obtained by selecting appropriate program statements from a set of menus (Kelleher & Pausch, 2005; Weber & Möllenberg, 1995). The templates can then be filled in using data that is relevant to the current exercise. Sometimes, skeleton code in the form of templates is provided for the entire program (Al-Imamy et al., 2006). The students are then free to insert lines into, or delete lines from, the template. More support is provided for inserting lines by allowing the students to request templates of program statements that are valid at a particular point. When a programming statement is

selected, the students can then complete the template as appropriate. Templates have also been utilised to assist OOP in the Greenfoot Programming Environment (Kolling, 2010). Here, a class template is created each time a new class is required, thereby allowing the students to work on the implementation class without worrying about the class declaration.

Gegg-Harrison (1991) proposed an interesting variation on the use of templates to teach introductory programming. His environment to teach Prolog programming uses a set of program schemata, or standard structures, used commonly in Prolog programs. Each complex program is thought of as an extension to one or more of these schemata. Each problem presents a combination of these schemata with blanks that needed to be filled in by the students. A similar approach is used in the Prolog Tutor (Hong, 2004). Instead of schemata, it uses the concept of Prolog programming techniques, which are language dependent but specification independent coding techniques used by Prolog programmers (Brna et al., 1991). ProPAT (Delgado & Barros, 2004) uses a variation to this by providing a plug-in to the Eclipse development environment. This plug-in contains templates for some well thought of programming patterns. The focus of all of these methods is to reduce the problems many novice students encounter due to the complex syntax of programming languages.

2.4.4 Summary

This section described the various features of the teaching module that have been used in systems to teach programming. Although different approaches have been used, the actual features used in any given system depend on many factors. These include, but are not limited to, the programming language taught, the knowledge representation used in the system, and the variability of the students that use the system. These factors need to be considered carefully when deciding on the exact features of the teaching module that is suitable for any system.

2.5 THE STUDENT MODULE

Section 2.2.2 described the overall architecture of Intelligent Tutoring Systems. It can be seen that the student module is an important component in such systems in order to individualise the interactions based on the characteristics of the students. Students are human beings who have many different traits such as knowledge levels,

learning styles, motivation, likes and many more. All these traits contribute to their preferred methods of learning and should therefore theoretically be modelled in order to individualise the interaction. In practice, this is a very difficult problem due to many reasons (Self, 1990). Since the focus of this research is not on the design of the student module, only the characteristic that is most directly related to learning, the current level of knowledge of the student on the subject matter, is considered in this thesis.

The knowledge level of a student regarding a certain domain is difficult to measure. In order to make this measure more accurate, it is usually broken down into separate topics or cognitive skills known as knowledge components (KCs). The knowledge level for each of these KCs is then considered instead of an overall knowledge level.

When measuring the knowledge level of a KC, the obvious measure is to gauge whether the KC is known or unknown. However, in practice, it is difficult to observe whether a person does or does not have a certain piece of knowledge. There is always uncertainty since a person can make a mistake due to a slip or get an answer correct by luck. This means that this uncertainty must be accounted for when measuring the knowledge level of a certain KC (Woolf, 2009). Therefore, the knowledge level is usually maintained as a probability that the KC is known. A value of zero indicates that it is not known for sure, while a value of 1 indicates that the KC is known without doubt. In practice, the knowledge level is somewhere between these two extremes, indicating the level of confidence of the system that the person knows the KC. In summary, this makes it possible to deal with uncertain and vague knowledge to make evaluations.

The student model is often designed as an overlay model of the domain model. This means that the domain being taught by the ITS is divided into certain knowledge components and the student model measures each student's knowledge level of the KC as described above. In the cognitive tutors mentioned in Section 2.3.2, knowledge tracing is commonly used practice when updating the student knowledge. In this method, at each opportunity that the student gets to apply a production rule, the student either knows or does not know the associated skill and therefore gives either a correct or incorrect response (Corbett & Anderson, 1992). However, there is always the possibility that the student applied the rule correctly by

chance or that s/he simply made a slip and did not apply the rule correctly, even if it was known. Each production rule is associated with a single skill, thereby making it possible to gauge the student's current knowledge of that skill (Corbett & Anderson, 1995).

Many methods of student modelling have been suggested in the literature. These utilise many different theories such as Bayesian Belief Networks (BBN) (Beck, Chang, Mostow, & Corbett, 2008; Corbett & Anderson, 1992, 1995; Hatzilygeroudis & Prentzas, 2004; Reye, 2004), Item Response Theory (IRT) (Galvez, Guzman, Conejo, & Millan, 2009; Johns, Mahadevan, & Woolf, 2006), and many more. Although these theories and their combinations have been used in many systems, BBNs are the basis for many successful Intelligent Tutoring Systems due to their many useful features.

2.5.1 Bayesian Student Modelling

As described above, BBNs are very often used to model the knowledge of students using an ITS. For the reader who is not familiar with BBNs, a concise description is provided in Appendix A. In Bayesian evaluation, a Belief Network of how the student gains knowledge is first constructed. They usually consider such factors as the student's previous knowledge of the KC and the response (correct or incorrect) during the current interaction. A set of equations to calculate the current knowledge level of a student after an interaction have been developed (Reye, 1998). These equations are actually a more generalised version of the equations specific to a situation where the outcome of an interaction can only be correct or incorrect, that are used in the successful cognitive tutors (Corbett & Anderson, 1992, 1995).

The above models based on BBNs assume that each KC is independent of the other. However, in actual programming practice, this is not the case. Topics are generally dependent on each other and form a pre-defined order. For example, it is necessary that the student has knowledge about simple sequential statements before s/he can proceed on to more advanced selection statements. Therefore, the student's knowledge level of the sequential statement affects his/her knowledge level of the selection statement. A method of modelling this relationship between KCs has been proposed by Reye (2004). Since this takes additional factors into account, it produces a more accurate measure of the student's knowledge of a KC.

In addition to the highly successful tutors mentioned above, variations of the Bayesian modelling technique have been used in many other tutors (Beck et al., 2008; Michael Mayo & Mitrovic, 2001).

Overall, the reason for using this method so extensively is that it can accurately handle uncertainty. The theoretical basis for BBNs is also highly developed and therefore it is expected to provide a relatively accurate student model, when sufficient observations are available.

2.5.2 Open Learner Models

The information from the student model affects the feedback provided to the student. However, there could be times when students disagree with the system's gauge of their knowledge. This could happen due to inaccuracies in the student model, as well as other reasons such as a student deliberately providing wrong code in order to understand what happens better. In such situations, the feedback provided by the system may not be appropriate for the student. If the student is unaware what the system thinks of his/her knowledge, the student may be confused as to why this is happening. Therefore, it can be beneficial to the students to let them know the system's gauge of their knowledge. Additionally, many students feel that they have a right to view data about themselves (Bull, 2012). Researchers have shown that students that are provided with such a view of their student model and meta-cognitive tips performed much better than other students (Long & Alevan, 2011). A student model which has been made accessible to the student in this manner is known as an 'Open Learner Model' (OLM).

OLMs can be of three main types: inspectable, negotiated and editable. An *inspectable* student model allows the student to view the system's idea of his/her knowledge but does not allow him/her to alter it. An *editable* student model allows the student to view as well as change his/her knowledge level manually. A *negotiated* student model is in-between these two, allowing the student to negotiate his/her knowledge with the system by providing some sort of dialogue. Research has shown that, of these three methods, a majority of students prefer an inspectable student model (Bull, 2012).

Open Learner Models can take many forms. It should be noted that the method in which the model is externalised to the user can be very different from the

underlying model (Bull, 2012). The externalised model should be understandable to the user. Many students prefer to have an overview and a detailed view as well as details of their misconceptions. This makes it easier for students to be aware of their general difficulties. However, students do differ in their preference, so it has been suggested that students are offered a choice as to what views they would prefer to see (Bull, 2012). Research has shown that students also like the model to include details of what is expected at the current stage (Bull, 2012). Another feature that has been used in OLMs is the ability to release it to others so that the students can make a comparison between themselves and their counterparts (Bull, 2012).

Therefore, it can be seen that an open learner model in some form is a preferred feature of an Intelligent Tutoring System and has proved to be valuable to students.

2.6 COMPARISON OF EXISTING ITSS TO TEACH PROGRAMMING

The above sections described the difficulties of teaching programming to beginners and how Intelligent Tutoring Systems can be used to help this process. Many ITSs have been built with this in mind. This section compares the features of some of these systems.

Table 2.1 compares several existing ITSs that teach programming. It does not look at all possible features but examines some of the main features that have been discussed in previous sections. Note that the PROUST system mentioned in section 2.3.2 has been omitted here since it does not provide feedback based on the abilities of each student.

One main feature that can be identified here is that some ITSs focus more on providing instruction in a textbook-like fashion while others concentrate on practical programming exercises. ELM-ART is the only system among these that is textbook-like but with programming exercises incorporated into the system. This emphasises the fact that exercises are an important feature in any ITS that teaches programming.

When analysing the systems in the table, it can be seen that many of the ITSs provide delayed feedback. The main reason for this is that programming exercises have many solutions. Therefore, it is very difficult to identify whether the student is on a correct path as and when the solution is typed in. This problem does not occur in the ACT programming tutors since it compares the student's program to an ideal solution and can therefore immediately identify any deviations.

Table 2.1

Existing ITSs to Teach Programming

<i>System</i>	<i>Domain</i>	<i>Feedback and Hints/Special Features</i>	<i>Program Analysis</i>	<i>Next Task Selection</i>
ACT Programming Tutor (Corbett, 2000)	LISP	Immediate feedback	Compares against a set of production rules	Predefined set of exercises presented at the end of each section
	Prolog	Three levels of hints		More exercises to achieve mastery presented based on knowledge of current student
	Pascal	Inspectable OLM		
ELM-ART (Weber & Brusilovsky, 2001)	LISP	Adaptive hypermedia	Identifies semantically equivalent solutions using plan transformation and bug rules	Pages on electronic textbook are colour coded to indicate ones suitable for the student
		Feedback on request		
		Identifies complete and incomplete solutions and provides hints		
		Several levels of hints		Student may select differently
		Example based problem solving support		
Editable OLM				
C-Tutor (Song et al., 1997)	C	Feedback on request	Intention based analysis using goal/plan hierarchies	System selects an exercise or concept to teach based on knowledge of current student
		Bugs described using cause-effect relationships	Programs converted into canonical form	

<i>System</i>	<i>Domain</i>	<i>Feedback and Hints/Special Features</i>	<i>Program Analysis</i>	<i>Next Task Selection</i>
			Static and dynamic analysis	
CPP-Tutor (Naser, 2008)	C++	Feedback on request	Intention based analysis based on edit distance between student's solution and probable intent	System selects an exercise based on knowledge of current student
		Modifications made to student code based on identified intent		
		Modifications done only after consultation with student		
		Possibility to run the code without analysing program		
JITS (E. Sykes, 2007)	Java	Feedback on request	Intention based analysis using parse trees	System selects an exercise based on knowledge of current student
		Guides student towards a potentially unique solution based on identified intention		
		Automatic correction of code where appropriate		
Prolog Tutor (Hong, 2004)	Prolog	Feedback on request	Compare parsed version of both the student's solution and reference program using a set of common Prolog programming techniques	System selects an exercise based on knowledge of current student
		Guided programming provides templates of relevant programming techniques		
		Uses error messages based on incorrect programming techniques		

Another interesting fact is that many of the systems automatically select the next best task for the student based on his/her current knowledge level. Since a main task of an ITS is to individualise the interaction, this is an important feature. However, there is evidence that some students do not feel comfortable with accepting the system's suggestions. This may be due to the student preferring to practice more. Therefore, it is good practice to provide such help while allowing the student to select a different task as in ELM-ART and the CPP-Tutor.

Each of these systems use a different method for program analysis. Some of the methods specified are very dependent on the programming language used while others may be generalised across several languages. However, in order to generalise the analysis, the languages need to have a similar structure.

Another feature is that several of the systems contain open learner models. Although the OLM in ELM-ART is editable, the one in the highly successful ACT programming tutors is inspectable.

These features needed to be considered carefully when deciding on features that were desirable for the PHP Intelligent Tutoring System.

2.7 SUMMARY AND IMPLICATIONS

Teaching programming to beginners is a complex task which has proved challenging to educators through decades. Intelligent Tutoring Systems, that customise their instruction based on the characteristics of the current student, have been proposed as a method of overcoming some of these challenges. An ITS consists of four main modules: the domain module, teaching module, student module and communications module. Each of these modules play an important role in teaching the subject matter effectively to the students.

Many methods have been used to design the domain module in ITSs that teach programming. A main challenge encountered here is that a programming exercise can have many correct solutions. Although many methods have been proposed to solve this problem, it is clear that more work needs to be carried out in this area.

In addition to analysing programs, an ITS should be capable of providing pedagogical support for students to learn. Existing ITSs achieve this by a variety of

means. These features need to be analysed carefully to decide which of them are most suitable for the system under consideration.

ITSs focus on customising the interaction based on the requirements of each student. In order to do this, it is necessary to find methods of modelling the students. Different methods have been utilised for this purpose. These methods need to be studied to identify which of them are appropriate for the proposed system.

Although the literature showed that many ITSs have been developed to teach programming, none focus on the intricacies of web development.

Therefore, the review of existing literature supports the fact that there is no previous work that addresses the research problem of the thesis as defined in Section 1.3. The rest of the thesis discusses how the research project was carried out to answer this problem.

Chapter 3: Research Design

This chapter describes the research design adopted to achieve the aims and objectives stated in Section 1.3. Section 3.1 discusses the overall methodology used in the study. Section 3.2 goes on to discuss the different phases by which the methodology was implemented and the research methods used in each phase. Section 3.3 gives the timeline for the implementation of the research design. Finally, Section 3.4 gives a brief summary of the chapter.

3.1 METHODOLOGY

Research can be divided into two main categories: basic research and applied research. Basic research involves the developing and testing of theories or hypothesis to satisfy intellectual interests. Applied research applies knowledge to solve practical problems. This usually results in the development of new artefacts which utilise new theories that are formulated during the research process. The new artefacts are then tested to obtain proof for or against the hypothesis that they solve the underlying practical problem (Nunamaker Jr., Chen, & Purdin, 1990).

As mentioned in Section 1.2, this research addresses the practical problem of teaching dynamic web development using PHP in an efficient and economical manner. The artefact resulting from this research is an Intelligent Tutoring System that teaches PHP programming. In developing this artefact, it is necessary to develop new theories on how the different components of the ITS need to be modelled. Therefore, this research falls into the category of applied research. It followed the three stage model of: concept, development and impact (Nunamaker Jr. et al., 1990). This is apparent by the objectives of the research as described in Section 1.3. The first objective - to design a knowledge base - falls into the concept stage. The second objective - to build the system - falls into the development stage and the third objective - to evaluate the system - falls into the impact stage. The developed system serves as both a proof of the concept of the fundamental research as well as an artefact for continued research. Therefore, this research closely followed the concepts of the Systems Development research methodology (Nunamaker Jr. et al., 1990).

3.2 RESEARCH DESIGN

Systems Development uses a multi-methodological approach of three stages described in the previous page. It encompasses theory building with experimentation and observation to validate hypotheses. The research process consists of five main steps (Nunamaker Jr. et al., 1990): construction of a conceptual framework, development of a systems architecture, analysis and design of the system, system building, and observation and evaluation. This research project was divided into four phases to incorporate these five main steps of the Systems Development research methodology.

3.2.1 Phase One

The first phase encompassed the first two stages of the research methodology: construction of a conceptual framework and development of a systems architecture. This phase included the formulation of the research question and the identification of system requirements. The main research method used was an extensive literature survey. Available literature in ITS design was studied in detail to understand the current state of the discipline. Since it was obvious that Artificial Intelligence (AI) plays an important role in the design of ITSs, this subject was also studied at some length. Existing ITSs and other computerised teaching systems that teach programming were studied to identify the requirements of the system. The information obtained from this literature was also used to develop a systems architecture based on the standard architecture of an Intelligent Tutoring System as described in Section 2.2.2.

As described earlier, a major challenge in building an ITS that teaches programming is the ability for it to analyse computer programs written by students. An architecture to achieve this, using theories of AI, was developed during this phase and was used as a basis for the rest of the thesis. A detailed description of this architecture is given in Section 4.3.

3.2.2 Phase Two

The second phase of the research design was the analysis and design of the system. This major phase included the design of schema and knowledge bases necessary for the system. The methods of modelling the different components of the systems architecture were carefully studied.

Building an ITS is a complex task which is very time consuming. It has been estimated that it takes 200 to 300 hours to build an ITS that provides one hour of instruction (Aleven et al., 2006). The subject matter taught by the PHP ITS is web development using PHP. The number of hours of instruction necessary to teach this subject matter exhaustively is very large. Therefore, it was impossible to create a system that was capable of achieving this within the time limitation of a PhD. Consequently, it became necessary to narrow down the subject matter taught by the PHP ITS. It was decided that the system would cater for novice programmers with no prior knowledge of PHP programming. Subject matter relevant to this was then identified to be included in the system. A PHP Grammar in Extended Backus-Naur Form showing the sub-set of PHP that is handled by the system is shown in Appendix B .

The next step in the design process was to find suitable representations for the selected subject matter. It was necessary to represent this subject matter in a manner which made it possible to analyse student answers and identify different solutions to a given problem. Literature surveys were used to study the different methods that had previously been used to represent subject knowledge. Of the many methods that had been used previously, it was necessary to find a method that was flexible enough to handle the multitude of variations possible when writing computer programs. It was also necessary that the selected method facilitated the process of providing appropriate feedback based on particular errors made by students. Consequently, it was decided to use First Order Predicate Logic (FOPL) to model the knowledge base. Although this representation had been used previously for Intelligent Tutoring Systems, it had not been used to represent subject matter in order to analyse computer programs.

The subject matter selected for including in the system was then studied in detail to see how it could be modelled using FOPL. This was an iterative process. At each step, a type of programming construct was selected and a suitable model proposed. Then, a set of examples that demonstrate the use of this construct was considered. These examples were traced through manually to ensure that the proposed model could be used to analyse these programs. If any problems were found, the model was refined and the process was repeated. This was done for all constructs that were going to be included in the implemented system.

An effective ITS needs several modules as described in Section 2.2.2. However, the core of this is the domain module. This is because it is impossible to tutor properly without having a good representation of the subject matter. The representation of the student knowledge also depends on how the subject matter is represented. Therefore, the main focus of this thesis, and an important contribution, is the proper design of the domain module. The other modules can then be built on top of this core module. However, as mentioned previously, it was impossible to create an ITS where all the modules are in their best possible form, during the time limitations of a PhD. Therefore, it was decided that the student and teaching modules would receive less emphasis in this thesis.

The student module models the characteristics of a student using the system. Students are human beings who differ in many characteristics such as subject knowledge, level of education, learning style, motivation and age. Modelling all these traits is a very difficult problem. Therefore, this study focused on only modelling each student's current knowledge of the subject matter taught by the system.

As described in Section 2.5 many methods of student modelling have been used in successful ITS. These representations were studied through literature surveys. Of these methods, Bayesian student modelling was selected as the method most suitable for the PHP ITS, based on the selected subject matter representation as well as the functionality and success of the methods.

The teaching module concentrates on teaching methods that are adapted by the system. Again, the many teaching methods used in previous ITSs were analysed based on literature reviews. Of the many methods described in Section 2.4, certain methods suitable for the system were selected. A detailed description of the methods incorporated in the teaching module of the PHP ITS can be found in Section 8.3. However, it was not possible to incorporate all the best methods identified from the literature due to time limitations.

The end result of the second phase was a thorough theoretical basis for the PHP ITS.

3.2.3 Phase Three

The third phase of the research was the actual system building process. During this phase, the design and architecture of the previous two phases were put into actual practice using available software tools and technologies. In order to do this, existing development platforms were compared, together with their available programming languages and tools. While identifying a suitable set of software tools, they were studied to see how they could be integrated to build the actual system (see Section 8.4.1).

One main component of any software system is its database. The database was designed to meet the requirements identified in the previous phases. The system was then built using these technologies. During this process, some issues related to implementing the designs using the selected tools were encountered. Suitable methods of overcoming these difficulties were also identified (see Section 8.4.2). The final outcome of this phase was the developed PHP ITS.

3.2.4 Phase Four

The fourth phase of the research was to evaluate the system under practical use. In order to carry out the evaluation, the PHP ITS was deployed in a QUT unit to teach web development using PHP. The participants were selected on a voluntary basis with the additional condition that they satisfy certain qualifying criteria to study PHP. Ethical clearance was obtained from the University Ethics Committee as the research required gathering data from human participants. The students that participated in the unit were awarded marks that counted towards their final GPA and graduation. Therefore, ethical problems would have occurred if only some students enrolled in the unit were allowed to use the ITS. In order to avoid this problem, all participants in the unit were allowed to use the ITS. This meant that it was impossible to have a control group to compare against the students that were using the ITS.

During the evaluation process, data was gathered from the students who used the system. There are many aspects that the evaluation of an ITS needs to consider. One important aspect was whether the students actually gained knowledge by using the system. Pre and post-test results were the main form of data used for this purpose. It was also necessary to evaluate the validity of the student model (Mark &

Greer, 1993). For this purpose, details of student interactions with the system were recorded and analysed together with the pre and post-test results. The usability of the system was also an important consideration. This was analysed using both qualitative and quantitative responses to a questionnaire and also to questions at a focus group discussion.

An iterative approach was taken for the evaluation and improvements. After deploying the system during one semester, improvements were made based on the responses received from the students who used the system. This improved system was then deployed during the next semester. It was then evaluated using the same methods as in the first semester. The only difference was that a focus group discussion was not carried out during the second semester. The system was improved further using feedback received from the second evaluation.

3.3 TIMELINE

As described in Section 3.2, the research design consists of four main phases. Table 3.1 shows the timeline to complete each of these phases. Phase one, which consisted of the initial literature review and identification of requirements continued for the first 12 months of the research project. However part of phase two - the design of knowledge base - started in parallel, during the fourth month. Phases two and three (system building) continued throughout most of the research project and overlapped. Phase four - the system evaluation - took place during two short periods at 24 months and 30 months into the research respectively. Both phases two and three were revisited after these two periods of evaluation in order to improve the system further.

Table 3.1
Timeline for Completion of Each Phase

Time Elapsed (in months)	3	6	9	12	15	18	21	24	27	30	33	36
Phase One	█	█	█	█								
Phase Two		█	█	█	█	█	█	█		█		█
Phase Three				█	█	█	█	█		█		█
Phase Four									█		█	

3.4 CHAPTER SUMMARY

This chapter described the research methodology used in the research and a detailed analysis of the research design used. It also explored the time line of the research project. The next chapters go on to discuss the outcomes of the research process in much more detail.

Chapter 4: Basics of Program Analysis

The PHP Intelligent Tutoring System is designed to teach basic web development to beginning programmers. This is mainly done through providing programming exercises for the students to answer. In order to teach the subject effectively, it is necessary for the system to analyse any answers provided by students and provide constructive feedback.

One major issue encountered when trying to analyse program code is that a programming exercise does not have a unique solution. Consider the example programming exercise described in *Figure 4.1*. Although this is a very simple exercise, the program can be written in many ways. Table 4.1 shows three programs that all result in the web page described in the exercise, although some of them use very round-about methods.

Write a PHP program to display the string 'Welcome!' on a web page. Next, add 3 to the value in the variable \$y and store it into the variable \$x. Finally, display the value of \$x. Note that the variable \$y already contains a value when execution reaches the point where the code needs to be completed.

Figure 4.1. Example programming exercise.

Table 4.1

Alternative Correct Solutions for Example Exercise

<i>Program a</i>	<i>Program b</i>	<i>Program c</i>
Welcome! <?php \$x=\$y+3; echo(\$x); ?>	<?php echo('Welcome!'); \$x=\$y+3; echo(\$x); ?>	Welcome! <?php \$z=\$y+1+2; \$x=\$z; echo(\$x); ?>

This shows that matching a program line by line is not a very effective method of analysing it. If a single ideal solution was maintained, many of these programs

will be identified as incorrect, although they create the required web page. In order to reason about a program, a more formal method of representation is required. The representation selected needs to support logical reasoning about the structure of a program. It also needs, not only to analyse the program for correctness, but to allow providing appropriate feedback based on the actual errors made by the students. Since this process involves logical reasoning by the computer, Artificial Intelligence (AI) techniques are a suitable means of achieving this. Of the many representations available in AI, First Order Predicate Logic (FOPL) is a simple representation with a lot of flexibility. However, the literature does not reveal any attempt to use FOPL to analyse computer programs. This thesis investigates the possibility of using FOPL for this purpose. Chapter 4, Chapter 5, Chapter 6 and Chapter 7 explain the formal representation used to represent in the PHP ITS and how it helps to analyse programs. This is the knowledge base (KB) that functions as the domain module of the PHP ITS.

This chapter concentrates on the basics of program analysis. It discusses the process used by the PHP ITS to decide whether a student's answer to an exercise is correct. It deals with some of the PHP constructs that are used in very basic programs and how they are represented within the KB. Specifically, it discusses display statements and assignment statements. These statements form the basis for more advanced PHP constructs such as selections and loops which are detailed in later chapters.

This chapter is organised in the following manner. First, Section 4.1 introduces some theoretical concepts which are important to understanding the rest of the thesis. Section 4.2 then goes on to explain some conventions that have been used throughout the thesis. Section 4.3 gives an outline of the process used by the PHP ITS to analyse computer programs written by students. A more detailed description of the process then follows. Section 4.5 expands on the process discussed in the previous section to explain how the knowledge base of the PHP ITS is structured and describes how this is used in program analysis. Section 4.6 discusses how some special situations are handled and finally, Section 4.7 summarises the chapter.

4.1 THEORETICAL CONCEPTS

In order to understand the process of program analysis, it is first necessary to have a knowledge of certain theoretical concepts that are used extensively throughout this thesis. This section gives a brief introduction to these theoretical concepts.

4.1.1 Concepts in Artificial Intelligence

Artificial intelligence (AI) techniques attempt to build intelligent agents. The definitions of the AI concepts described herein are taken from the book “Artificial Intelligence a Modern Approach” (Russell & Norvig, 2010).

Logic is a general class of representations used to design knowledge bases (KB). A knowledge base is actually a set of sentences where each sentence is expressed in a knowledge representation language. The representation language used throughout this research project is First-Order Logic (FOL), also called First-Order Predicate Calculus (FOPC) or First-Order Predicate Logic (FOPL). It is assumed that the reader is familiar with FOL, at least to the level discussed in Chapter 8 of the above book (Russell & Norvig, 2010). Inference procedures in FOL can be used to check whether some sentence is true given that a set of facts is true. This is the fundamental basis for the theoretical framework of this research.

Throughout the research project, database semantics (Russell & Norvig, 2010, pp. 299-300) in First-Order Logic have been used. This means that it is assumed that every constant symbol refers to a distinct object (unique-names assumption). Secondly, sentences not known to be true are assumed to be false (closed-world assumption). Thirdly, domain-closure, which means that the model contains no more domain elements than those named by the constant symbols, is assumed.

A state in AI is a set of facts which are true at the given point in time. The state changes by addition or deletion of facts to the state. A searching or planning **problem** in AI consists of five parts: the initial state, a set of actions, a transition model, a goal test and a path cost (Russell & Norvig, 2010, pp. 66-68). The **initial state** is the set of facts that correspond to the state the problem-solving agent starts in. **Actions** are a set of actions that are applicable or that can be executed in a given state. The **transition model** is a description of what each action does. The **goal test** determines whether a given state is a goal state. The **path cost** assigns a

numeric value to each path where a path is a sequence of states connected by a sequence of actions. Since the system developed in this thesis does not do planning, the path cost is not considered here.

A **plan** is a sequence of actions that can be used to achieve a given goal state. An action in such a plan can be at an abstract level and can be decomposed into more actions at a later stage. Creating plans with such high level actions is known as **hierarchical planning**. The actions that comprise the high level action at the abstract level, is then called a **sub-plan**.

4.1.2 Concepts in Database Design

Object Role Modeling (ORM) (Halpin & Morgan, 2008) is a graphical method which can be used to provide a diagram of the predicates used in the knowledge base. Although ORM is primarily used for database design, this method has been used for a different purpose in this research. The notations used in ORM have been used to depict the various predicates and their relationships. The easy graphical design of ORM makes it a suitable method of representation to be easily understood.

Only certain notations in ORM have been used in this research. *Figure 4.2* shows how these symbols have been adopted to depict object types and predicates defined in AI.

Object types have been categorised into two main groups: **entity types** and **value types**. Entity types refer to object types that can be instantiated to create instances of objects and are depicted using rounded rectangles with continuous lines. Value types refer to types that can only take one of a specific set of values and are represented using rounded rectangles with dotted lines. **Predicates** are relationships between one or more of these object types. They are represented using a rectangle divided into the number of object types that form the arguments of the predicate. Each section of the rectangle is connected to the corresponding object type. In the given diagram, *Expression* is a entity type since many expressions can be created. *ExpressionId* is a value type since it will contain specific values. *HasId* is a predicate that shows the relationship between the *Expression* and the *ExpressionId*. Since it relates two object types, the rectangle is divided into two sections.

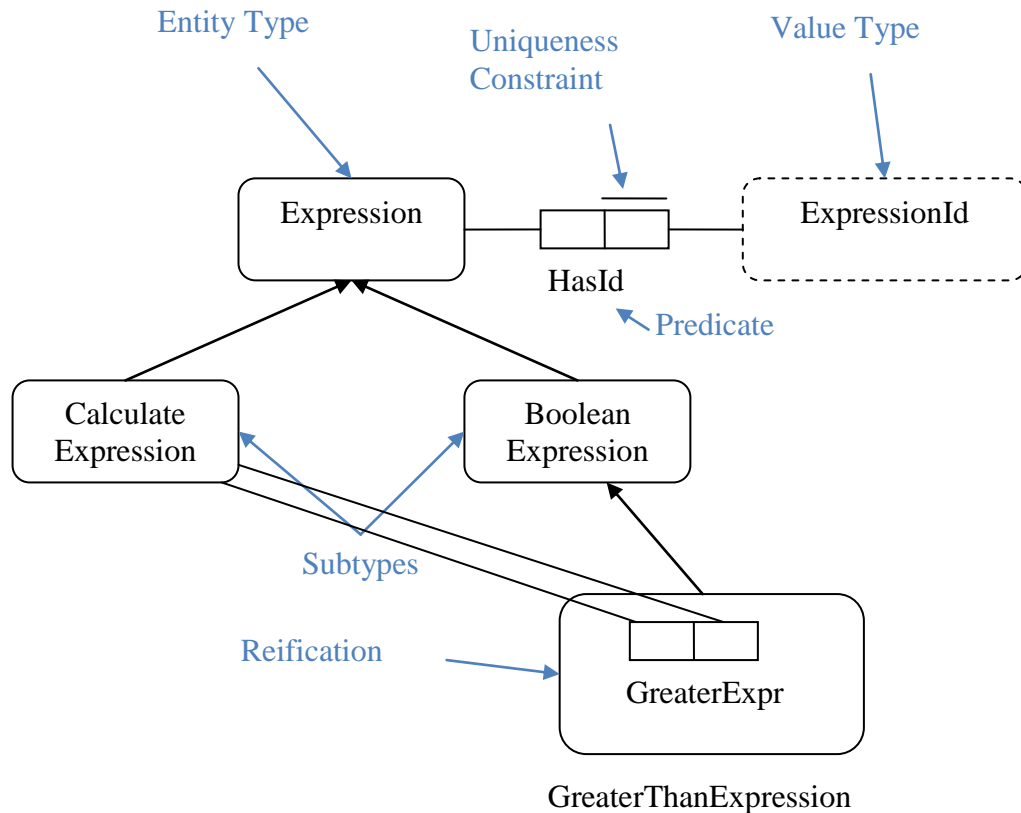


Figure 4.2. Some ORM symbols and their meanings.

Uniqueness constraints can be defined for predicates. These define which object types contain unique values for each instance of the predicate, or in other words, each fact. A line above a section of the rectangle indicates that the corresponding object type has a unique value for each fact based on that predicate. The constraints can exist for one or more object types that form the arguments of a predicate. In *Figure 4.2*, the line above the *HasId* predicate is placed on the side of the rectangle connected to *ExpressionId*. This means that each fact created based on the *HasId* predicate has a unique *ExpressionId*.

Sub-types are object types that contain the characteristics of the main type as well as some characteristics of their own. This is depicted by joining the sub-types to the main-type using arrows. For example, both *CalculateExpression* and *BooleanExpression* are sub-types of *Expression* in *Figure 4.2*. This means that both these object types have a *HasId* predicate which is defined for the main type. In addition to this, they can each have their own predicates which are specific to that particular object type.

Sometimes, it becomes necessary for a predicate to behave as an object. Considering the given example, *GreaterExpr* is a predicate which relates two *CalculateExpressions*. However, this predicate is also a sub-type of *BooleanExpression*. In such cases, the predicate is reified or objectified so that it can be used as another object. The reified predicate is given a new object name, in this case *GreaterThanExpression*.

4.1.3 Concepts in Language Parsing

Language parsing concepts are used in this research project to analyse programs written in HTML and PHP. Some definitions of key language parsing concepts are given here. These are taken from the book 'The Definitive ANTLR Reference' (Parr, 2007).

A **translator** is a program that reads some input and emits some output. An **input** is a sequence of vocabulary symbols. An input sequence is called a **sentence**. A **language** is a well defined set of sentences. A **translator** is a program that maps each input sentence in its input language to a specific output sequence. A **grammar** describes the syntax of a language. It is a set of rules where each rule describes some phrase of the language.

Grammars can be expressed using many notations. The notation used in this research project is Extended Backus-Naur Form (EBNF).

Sometimes different portions of the input can conform to different grammars (eg:- PHP and HTML). This is handled using the concept of island grammars. Each grammar is defined separately and a link to the other grammar is established. In each grammar, the input that conforms to the other grammar is defined imprecisely as a set of characters or tokens. When this imprecise portion is encountered, the other grammar is used to parse the input.

A program can be translated into an Abstract Syntax Tree (AST) using a grammar. An AST is simply an internal data structure that represents a program as a tree structure. ASTs are used in this research project for analysing programs written by students.

4.2 CONVENTIONS USED IN THIS THESIS

This section outlines some conventions that have been used in this thesis. It highlights conventions that are somewhat different in meaning to the standard conventions used in the relevant disciplines. Such differences are sometimes necessary since concepts from several disciplines are integrated into this research.

The conventions used in FOL specify that all constant symbols, predicate symbols and function symbols begin with uppercase letters. The knowledge base in this research does not contain any function symbols. All predicate symbols used here also begin with uppercase letters. Constants in this research are defined using a somewhat different notation. When the constants refer to the id of a particular object, they begin with an uppercase letter. However, when referring to a literal value, they are surrounded with single quotes and can begin with either an uppercase or a lowercase letter depending on the usage. For example, the names of variables are represented using the same case as used to define the variable within the program code.

In standard FOL, variables begin with lowercase letters. Although many AI variables used during this research also begin with lowercase letters, there is an exception. Variables used within the definition of the overall goal (as described in Section 4.4.2) are defined using all uppercase letters. This is done to eliminate the need for using existential quantifiers in front of a large number of FOL variables. It is assumed that FOL variables specified using all uppercase letters are existentially quantified.

4.3 OUTLINE OF THE BASIC PROGRAM ANALYSIS PROCESS

In this research, concepts in AI are used as a basis for the formal representation described above. The process of converting a program into this representation and analysing it for correctness is modelled as a problem in AI. *Figure 4.3* is a schematic representation of how the AI problem is formulated. As described in Section 4.1.1, a classical AI problem is based on states. In the PHP ITS, these states are represented by a set of facts. Each fact is a specific instance of a predicate. The initial state is the set of facts that are valid before the student's code is analysed. In situations where the student is required to write the entire code for an exercise, the initial state is the empty set. However, the PHP ITS contains some gap exercises. This means that

part of the program is already provided by the system and corresponding facts exist before the student's program is analysed. These facts form the initial state for the exercise. The goal state of the exercise is the set of facts that must be matched if the answer submitted by the student is correct. This set of facts is defined in the overall goal. The process of setting up the initial state and overall goal using facts in a simple PHP exercise is described in detail in Section 4.4.

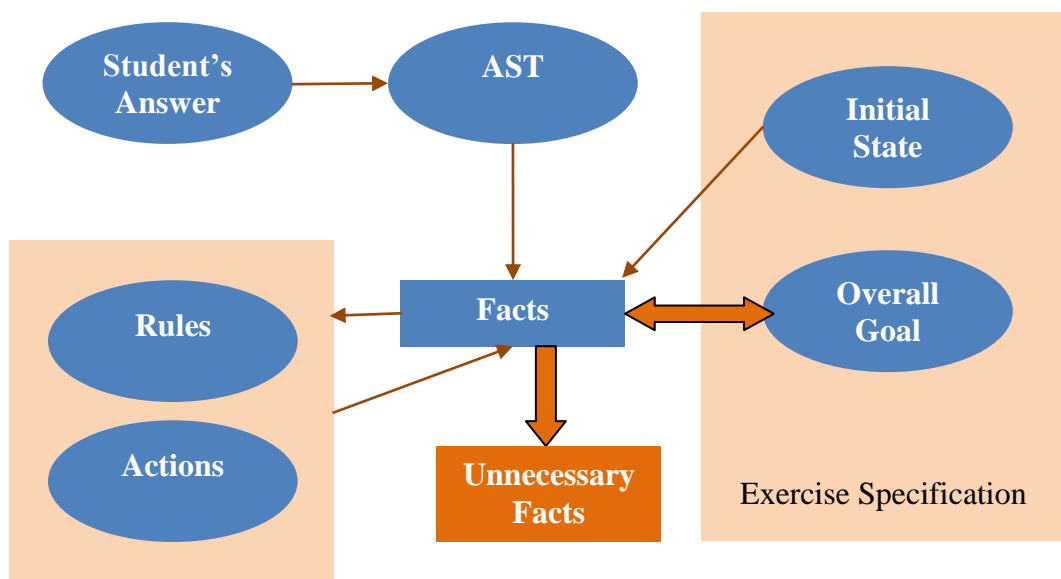


Figure 4.3. Basic program analysis.

Once the student submits an answer to an exercise, it is first converted into an Abstract Syntax Tree (AST) to make it easier to analyse. This process is explained in more detail in Section 4.5.2. The AST is then converted into a set of corresponding facts. The KB also contains a set of rules and actions that are used to transition from one state to another. These are activated as and when necessary while walking the AST. The process of walking the AST and of activating the rules and actions are described in detail in Section 4.5. This section also describes how the final state arrived at in this manner is compared against the overall goal to determine if the student's answer is correct.

A common mistake made by students is to include unnecessary code in their programs. In such cases, the final set of facts contains unnecessary facts that do not contribute to the overall goal. Before reaching a decision as to whether a student's

program is correct, the final state is examined to see if such unnecessary facts are present. The process of identifying such extra facts is described in detail in Section 4.5.5.

It can be seen that this method of program analysis depends on the facts created during the AST walking process. As long as the AST depicts the functionality of the program, the method should be capable of analysing programs no matter what the original programming language used. Therefore, this method should be extendable to analyse programs written in other 3GL programming languages. The amount of work involved would depend on the number of differences between the nodes of the AST produced by whatever the other language and PHP. This should involve using appropriate grammars as described in section 4.5.2 although this has not been investigated during this PhD project.

4.4 KNOWLEDGE BASE STRUCTURE

As described above, the states in the AI problem are represented by a set of facts. These facts are instantiations of predicates. During this research work I was successful in defining a suitable set of predicates, rules and actions that can be used to represent computer programs written in PHP and analyse them for correctness. This section describes the structure of the predicates used in this KB and how they are used to describe the initial and goal states of an exercise in the PHP ITS.

4.4.1 Predicates and Rules

The knowledge base of the PHP ITS uses a set of predicates to identify PHP object types and their relationships. In order to make it easier to understand, these are shown in the form of an ORM diagram (Section 4.1.2). The entire ORM diagram that shows all the predicates is very complex and is included in Appendix L Appendix L. In this chapter, the relevant parts of this diagram are presented with explanations as to how the different predicates are used in program analysis.

Figure 4.4 shows the key predicates that are used in the analysis of the most basic PHP statements, mainly display statements and assignment statements. Three main knowledge base object types used in PHP programs are identified here: *Variables*, *Literals* and *Expressions*.

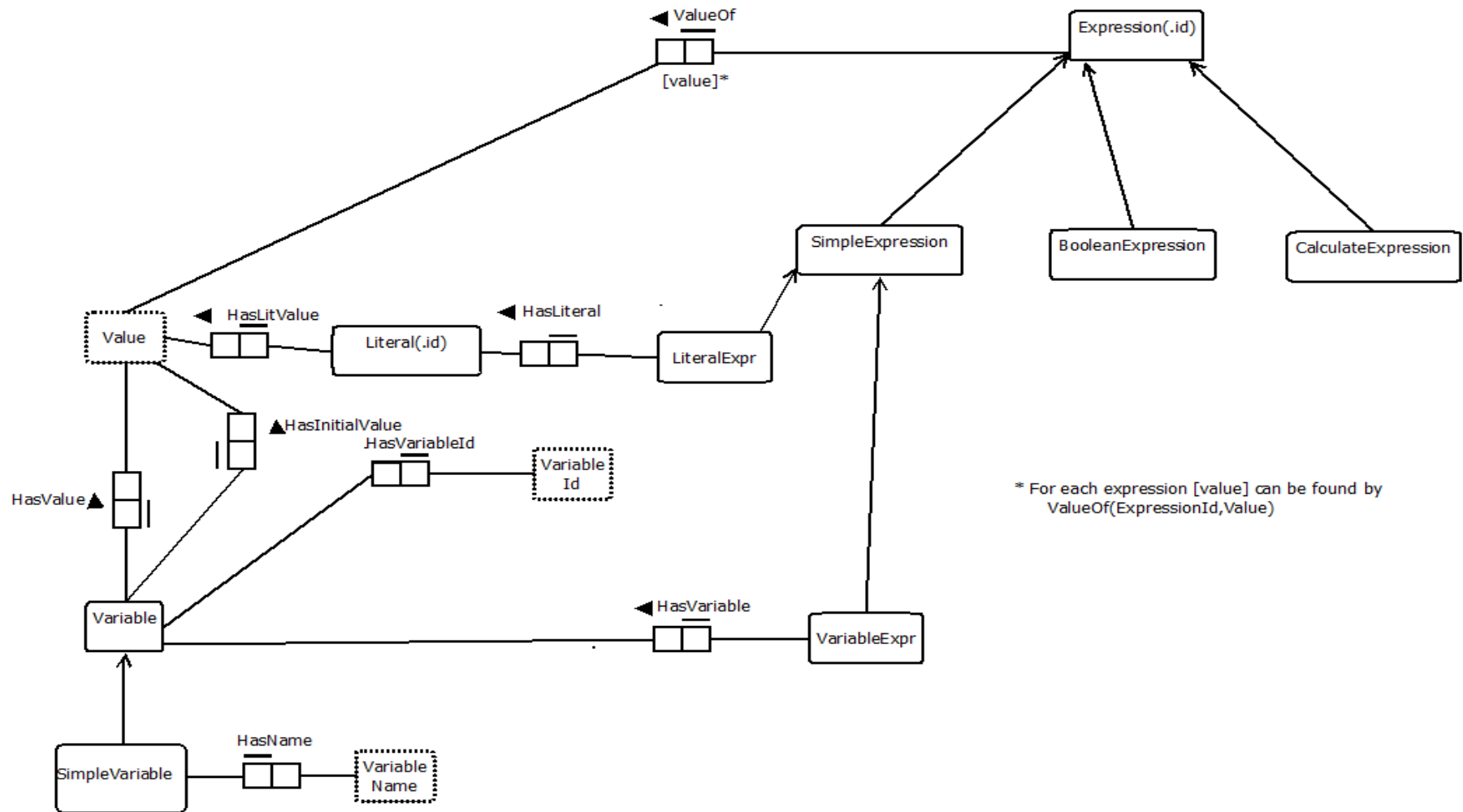


Figure 4.4. ORM diagram of key components of the assignment statement.

A variable is a fundamental concept in most programming languages. All but the extremely simple computer programs use variables. Therefore, it is an important object in the analysis of programs. Whenever a new variable is encountered, a new knowledge base *Variable* object is created. The system automatically assigns each new variable a *VariableId* in order to identify it uniquely. The most common type of variable is a variable with a symbolic name such as \$employee. In this research, such variables are referred to as *SimpleVariables* and are modelled as a subtype of the *Variable* object.

Although other subtypes of the *Variable* object are considered in later sections, only *SimpleVariables* are considered in this chapter. Each *SimpleVariable* has a symbolic name given by the *HasName* predicate. This name is just the name of a local PHP variable and ignores any associated class names. Also, PHP variable names are prefixed with a '\$' sign. This is not included in the variable name used for program analysis. For example, the name of the \$employee variable mentioned earlier is stored as 'employee'.

A variable contains a value except in the case when it is null. This value is represented by the *HasValue* predicate. The value of a variable may change during the life time of a program. However, the initial value of the variable sometimes becomes important. A good example for this is when the output of the program is dependent on the initial value of the variable. The *HasInitialValue* predicate is used to preserve the very first value of the variable for this purpose. It should be noted that a predicate to define the variable type is not used here. The reason for this is that PHP is a loosely typed language and therefore, each variable takes the type of the value it is holding at any given time. The type of the variable can change during the life cycle of the variable and is not modelled in this knowledge base.

As an example, consider a situation when a variable named \$x contains an initial value of 10. Assume that the unique *VariableId* assigned to this *Variable* by the system is VarId1. Then, based on the above description, the following facts are created in the system.

HasName(VarId1,'x')

HasValue(VarId1,10)

HasInitialValue(VarId1,10)

Literals are another object type that is often used when writing programs. A literal is a notation for representing a fixed value. A *Literal* object is also given a unique *LiteralId* by the system. The fixed value of the literal is given by the *HasLitValue* predicate.

As an example, consider the literal '5'. Let the *LiteralId* assigned to this Literal by the system be LitId1. Then, the following fact is created in the system.

$$\text{HasLitValue}(\text{LitId1}, 5)$$

4.4.1.1 Expressions

Expressions are a key concept used in programming in most programming languages. They are used extensively in many programming constructs. The right hand side of an assignment statement is an expression. The comparison statements used in selection and repetition constructs are expressions. They are used to pass parameters to functions. A KB that cannot handle expressions would be of very little use for program analysis. Therefore, the third key concept modelled in *Figure 4.4* is the *Expression*. Each *Expression* is again assigned a unique id known as the *ExpressionId* by the system.

As described above, expressions have many forms. In order to analyse programs correctly, it is necessary to categorise the expressions based on their type. This categorisation is done by dividing the *Expression* object into subtypes as shown in *Figure 4.5*. The following section describes the various subtypes of the *Expression* object type.

Variables and *Literals* are often used as expressions in PHP programs. These are used as all or part of the right hand side of an assignment statement or as a part of a conditional expression. A *LiteralExpr* is created each time a literal is encountered. The connection between the *Literal* and the *LiteralExpr* is established using the *HasLiteral* predicate. For example, if the literal described in Section 4.4.1 is used in an expression with an *ExpressionId* of ExprId1, the following fact is created.

$$\text{HasLiteral}(\text{ExprId1}, \text{LitId1})$$

A *VariableExpr* is created each time a *Variable* is used where any type of expression is acceptable. A *Variable* on the left hand side of an assignment expression does not result in a *VariableExpr* being created since an l-value is not an expression. A *VariableExpr* is connected to the corresponding *Variable* through the

HasVariable predicate. For example, if the variable described in Section 4.4.1 is used in an expression with an *ExpressionId* of *ExprId2*, the following fact is created.

$$\textit{HasVariable}(\textit{ExprId2}, \textit{VarId1})$$

It is important to note that several *VariableExprs* can refer to the same *Variable* as the same variable can be used in many expressions. Similarly, several *LiteralExprs* can refer to the same *Literal* as the same literal value can be used in many expressions. Both *LiteralExprs* and *VariableExprs* are modelled as a subtype of a special type of expression known as a *SimpleExpression*.

The right hand side of assignment statements often contain some form of calculation resulting in a value. Such calculations are also used in other types of programming statements. These calculations are modelled as a subtype of *Expression* known as a *CalculateExpression*.

CalculateExpressions are actually a combination of one or two other expressions. For example, consider the expression $\$x+5$. This is actually an addition expression (*AddExpr*) which has two other expressions, $\$x$ and 5 on either side of the addition operation. The expression on the left hand side is the *VariableExpr* described above, and the expression on the right hand side is the *LiteralExpr* described above. The *AddExpr* expression subtype is actually a predicate with one or two expression subtypes as its arguments. This predicate is reified as the *Expression* object type. Considering the above example, let the *ExpressionId* of the *AddExpr* be *ExprId3*. Then, using the expressions described previously, the reified expression is represented as below.

$$\textit{HasId}(\textit{AddExpr}(\textit{ExprId2}, \textit{ExprId1}), \textit{ExprId3})$$

Similarly, all other *CalculateExpression* subtypes are also predicates with one or two other expression subtypes as arguments. If more than two expressions are connected, they are broken into groups of two where the sub expressions are again broken down into more sub expressions. The subtypes of the *CalculateExpression* that have been implemented in the PHP ITS are shown in *Figure 4.5*. It can be seen that this includes the normal mathematical expressions of *AddExpr*, *SubtractExpr*, *MultiplyExpr*, *DivideExpr* and *ModulusExpr*. Although other types of mathematical expressions are not implemented here, the same theory can be utilised in many cases such as integer division, absolute value, factorial etc. The number of sub expressions

may vary but the general format remains the same. In addition to mathematical expressions, the *ConcatenateExpr* and the *DoubleStringExpr* are also modelled as a *CalculateExpr*. This is necessary to deal with PHP strings. In PHP, double quoted string may contain variables within them. In such cases the variables need to be replaced with their relevant values to obtain the value of the expression. Since this can also be considered a form of calculation, such expressions are modelled as a subtype of *CalculateExpression*.

Comparison statements that return a Boolean value are another common type of expression used in computer programming. Such statements are modelled as a subtype of an *Expression* known as a *BooleanExpression*. As with *CalculateExpressions*, *BooleanExpressions* are also a combination of sub expressions. The not expression (*NotExpr*) contains only one sub expression while the others are made up of two sub expressions. The most common types of *BooleanExpressions* are comparison expressions such as *GreaterThanExpr*, *GreaterThanOrEqualExpr*, *LessThanExpr*, *LessThanOrEqualExpr*, *EqualToExpr* and *NotEqualExpr*. These are then combined with Boolean operators to form **not** (*NotExpr*), **and** (*AndExpr*) and **or** (*OrExpr*) Expressions. All these are modelled as subtypes of the *BooleanExpression* and are shown in *Figure 4.6*.

Whatever the type of expression, it always has a value represented by the *ValueOf* predicate. Very often, this value is calculated using a set of rules in the knowledge base. The rules operate in an iterative manner to calculate the value of expressions that contain other sub expressions.

The rules used to calculate the value of the common expression subtypes are shown in *Figure 4.8*. In order to understand how they work, consider the PHP expression $x+5$ where x already contains the value 10. The facts resulting from this PHP code are explained above. *Figure 4.7* shows the list of these facts.

Next, the rules defined in *Figure 4.8* are invoked to find the value of the expression. First, the value of the variable expression is found. This is done using the second rule.

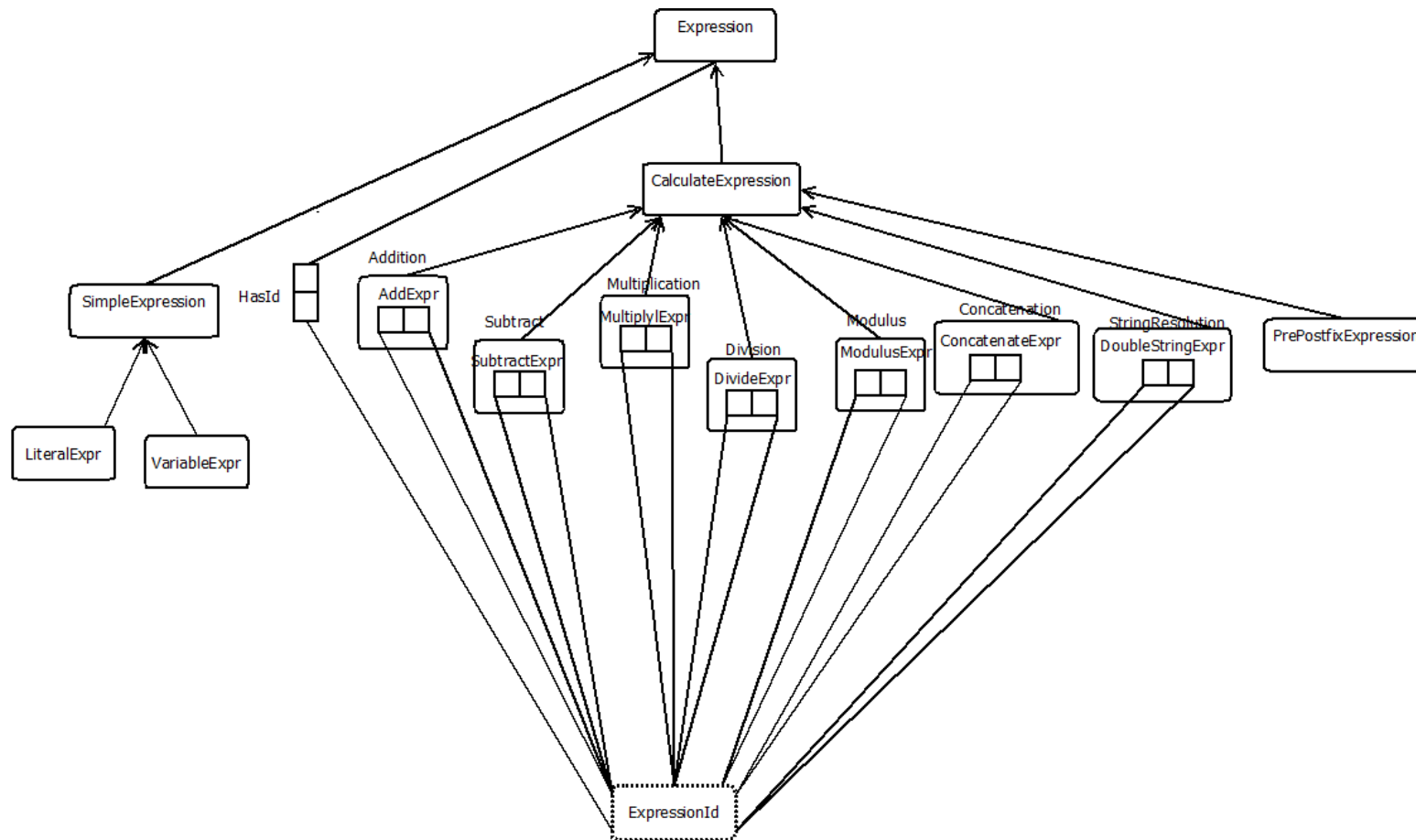


Figure 4.5. ORM diagram of expression subtypes of simple and calculate expressions.

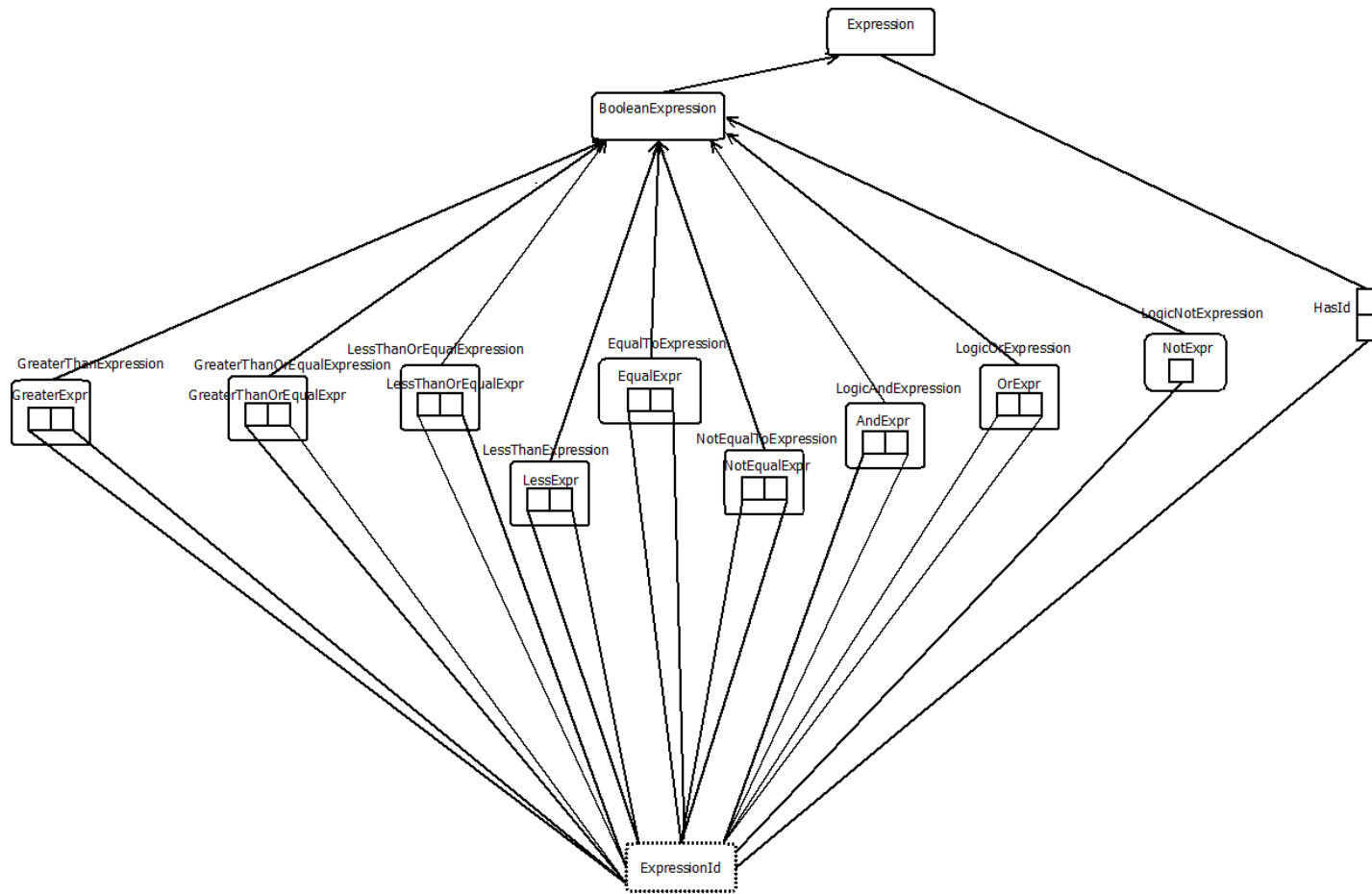


Figure 4.6. ORM diagram of Boolean expression subtypes.


```

HasName(VarId1,'x')
HasValue(VarId1,10)
HasInitialValue(VarId1,10)
HasLitValue(LitId1,5)
HasLiteral(ExprId1,LitId1)
HasVariable(ExprId2,VarId1)
HasId(AddExpr(ExprId2,ExprId1),ExprId3)

```

Figure 4.7. Predicates relevant to addition expression.

The result of applying this rule to the currently existing predicates is shown below.

$$\begin{aligned} & \text{ValueOf}(\text{ExprId2},10) \\ & \leftarrow \text{HasVariable}(\text{ExprId2},\text{VarId1}) \wedge \text{HasValue}(\text{VarId1},10) \end{aligned}$$

Similarly, the first rule in Figure 4.8 is used to calculate the value of the literal expression.

$$\begin{aligned} & \text{ValueOf}(\text{ExprId3},5) \\ & \leftarrow \text{HasLiteral}(\text{ExprId1},\text{LitId1}) \wedge \text{HasLitValue}(\text{LitId1},5) \end{aligned}$$

Finally, the third rule in Figure 4.8 is used to calculate the value of the entire expression. In this case, the $\text{Add}(x,y,z)$ predicate is a predicate that returns true if the sum of x and y result in z . So the value of the addition expression is as below.

$$\begin{aligned} & \text{ValueOf}(\text{AddExpr}(\text{ExprId2},\text{ExprId1}),15) \\ & \leftarrow \text{ValueOf}(\text{ExprId2},10) \wedge \text{ValueOf}(\text{ExprId1},5) \wedge \text{Add}(10,5,15) \end{aligned}$$

A similar method is used to calculate the value of all other expression subtypes.

$$\begin{aligned}
\text{ValueOf}(\text{literalExpr},v) &\leftarrow \text{HasLiteral}(\text{literalExpr},\text{literalId}) \wedge \text{HasLitValue}(\text{literalId},v) \\
\text{ValueOf}(\text{variableExpr},v) &\leftarrow \text{HasVariable}(\text{variableExpr},\text{VarId}) \wedge \text{HasValue}(\text{varId},v) \\
\text{ValueOf}(\text{AddExpr}(\text{exprIda},\text{exprIdb}),v) &\leftarrow \text{ValueOf}(\text{exprIda},va) \wedge \text{ValueOf}(\text{exprIdb},vb) \wedge \text{Add}(va,vb,v) \\
\text{ValueOf}(\text{SubtractExpr}(\text{exprIda},\text{exprIdb}),v) &\leftarrow \text{ValueOf}(\text{exprIda},va) \wedge \text{ValueOf}(\text{exprIdb},vb) \wedge \text{Subtract}(va,vb,v) \\
\text{ValueOf}(\text{MultiplyExpr}(\text{exprIda},\text{exprIdb}),v) &\leftarrow \text{ValueOf}(\text{exprIda},va) \wedge \text{ValueOf}(\text{exprIdb},vb) \wedge \text{Multiply}(va,vb,v) \\
\text{ValueOf}(\text{DivideExpr}(\text{exprIda},\text{exprIdb}),v) &\leftarrow \text{ValueOf}(\text{exprIda},va) \wedge \text{ValueOf}(\text{exprIdb},vb) \wedge \text{Divide}(va,vb,v) \\
\text{ValueOf}(\text{ConcatenateExpr}(\text{exprIda},\text{exprIdb}),v) &\leftarrow \text{ValueOf}(\text{exprIda},va) \wedge \text{ValueOf}(\text{exprIdb},vb) \wedge \text{Concatenate}(va,vb,v) \\
\text{ValueOf}(\text{GreaterExpr}(\text{exprIda},\text{exprIdb}),v) &\leftarrow \text{ValueOf}(\text{exprIda},va) \wedge \text{ValueOf}(\text{exprIdb},vb) \wedge \text{Greater}(va,vb,v) \\
\text{ValueOf}(\text{GreaterEqualExpr}(\text{exprIda},\text{exprIdb}),v) &\leftarrow \text{ValueOf}(\text{exprIda},va) \wedge \text{ValueOf}(\text{exprIdb},vb) \wedge \text{GreaterEqual}(va,vb,v) \\
\text{ValueOf}(\text{LessExpr}(\text{exprIda},\text{exprIdb}),v) &\leftarrow \text{ValueOf}(\text{exprIda},va) \wedge \text{ValueOf}(\text{exprIdb},vb) \wedge \text{Less}(va,vb,v) \\
\text{ValueOf}(\text{LessEqualExpr}(\text{exprIda},\text{exprIdb}),v) &\leftarrow \text{ValueOf}(\text{exprIda},va) \wedge \text{ValueOf}(\text{exprIdb},vb) \wedge \text{LessEqual}(va,vb,v) \\
\text{ValueOf}(\text{EqualExpr}(\text{exprIda},\text{exprIdb}),v) &\leftarrow \text{ValueOf}(\text{exprIda},va) \wedge \text{ValueOf}(\text{exprIdb},vb) \wedge \text{Equal}(va,vb,v) \\
\text{ValueOf}(\text{NotEqualExpr}(\text{exprIda},\text{exprIdb}),v) &\leftarrow \text{ValueOf}(\text{exprIda},va) \wedge \text{ValueOf}(\text{exprIdb},vb) \wedge \text{NotEqual}(va,vb,v) \\
\text{ValueOf}(\text{NotExpr}(\text{exprIda}),v) &\leftarrow \text{ValueOf}(\text{exprIda},va) \wedge \text{Not}(va,v) \\
\text{ValueOf}(\text{AndExpr}(\text{exprIda},\text{exprIdb}),v) &\leftarrow \text{ValueOf}(\text{exprIda},va) \wedge \text{ValueOf}(\text{exprIdb},vb) \wedge \text{And}(va,vb,v) \\
\text{ValueOf}(\text{OrExpr}(\text{exprIda},\text{exprIdb}),v) &\leftarrow \text{ValueOf}(\text{exprIda},va) \wedge \text{ValueOf}(\text{exprIdb},vb) \wedge \text{Or}(va,vb,v) \\
\text{ValueOf}(\text{DoubleStringExpr}(\text{exprIda},\text{exprIdb}),v) &\leftarrow \text{ValueOf}(\text{exprIda},va) \wedge \text{ValueOf}(\text{exprIdb},vb) \wedge \text{Concatenate}(va,vb,v)
\end{aligned}$$

Figure 4.8. Rules for calculating the *ValueOf* expressions.

4.4.2 Exercise Specification

The main function of the domain module of the PHP ITS is to analyse programs. In order to identify whether a program is correct or not, it is first necessary to know what the program is required to do. This is defined in the exercise specification.

The exercise specification contains a description of what needs to be done. Additionally, it contains a goal state or overall goal that needs to be achieved for the program to be considered correct. In order to understand how the overall goal is specified, consider an exercise where the value of the variable \$x needs to be set to 10. This means that the execution of the answer to this exercise should result in a variable containing the value 10. In terms of the predicates described in Section 4.4.1, this is equivalent to a fact of the form *HasValue(VARID,10)*. This component of the overall goal that is a direct result of execution of the program code is known as the ‘goal’.

However, matching the final state against the goal does not necessarily mean that the program code is correct. Sometimes, certain other aspects of the program such as the structure of the program need to be of the form given in the description for the program to be considered correct. Such structural constraints are specified in the component of the overall goal known as ‘constraints’. In this case, the name of the variable where the required value is stored should be x. In terms of predicate logic, this constraint is represented as *HasName(VARID,'x')*. So the overall goal of the exercise, containing both the execution goal and the constraints can be given as shown below.

Goal : *HasValue(VARID,10)*

Constraints : *HasName(VARID,'x')*

Note that the ids in the overall goal are given in uppercase. All components of facts given in uppercase represent existentially quantified first order variables. This convention has been assumed throughout this thesis to avoid the repeated use of existential quantifiers in the overall goal with the intention that this notation would be easier to understand. First order variables are especially necessary when specifying goals and constraints since the actual ids that are created by the ITS can take any value.

In addition to a goal state, the exercise specification may also contain the initial state of the program. This becomes necessary when the exercise is a gap exercise as explained in the introduction to this section.

4.4.3 Actions

As explained in the introduction to this section, a change of program state needs to be modelled in order to go from the initial state to the final state. AI actions are used to model such changes. In this chapter, actions are used to model two main program statements: assignment and display of elements on a web page.

In PHP, displaying elements in a web page is mainly achieved through the ‘echo’ and ‘print’ statements which are basically synonymous except for the fact that ‘print’ behaves as if it returns a value. This difference is immaterial to the basic PHP taught using the PHP ITS. Therefore, the *Display* action is executed each time an ‘echo’ or ‘print’ statement is encountered in the program. The Planning Domain Definition Language (PDDL) description of the *Display* action is shown in *Figure 4.9*. The resultant predicate is *OnPage*, which takes two arguments: a *value* and a *running counter*. The *value* is the value of whatever expression forms the argument for the ‘echo’ or ‘print’ statement. It is necessary to model this as an expression since the argument does not necessarily have to be a literal string. It can be any form of expression. The *running counter* is necessary during goal checking to ensure that whatever is necessary is displayed on the web page in correct order. It starts at one and is incremented by one each time a new *OnPage* predicate is created. This ensures that there is a record of the order in which the statements are displayed on the web page and is used in the final goal to ensure that the required output is obtained.

```

Action(Display(expressionId),
PRECOND:  $\exists$  value,rC,x
          (ValueOf(expressionId,value))
           $\wedge$  HasValue(rC,x)
EFFECT:  OnPage(value,x)
           $\wedge$  Add(x,1,y)
           $\wedge$  HasValue(rC,x)  $\leftarrow$  HasValue(rC,y)

```

Figure 4.9. Display action.

A PHP program can also have HTML statements. HTML statements are either tags or text. Any text in HTML is displayed on the web page in the same manner as PHP 'echo' statements. Therefore, HTML text statements are also handled using the *Display* action.

The second type of action used in the knowledge base models each assignment statement. The basic form of the assign action is shown in *Figure 4.10*. In this case, the first argument is the variable on the left hand side of the assignment statement and the second argument is the id of the expression on the right hand side of the assignment statement. The effect of this action is based upon whether or not a variable with the given name already exists. If it does, its value is updated to the value of the expression on the right hand side. If not, a new variable is generated, assigned a name and the value of the expression.

PHP also allows assignments using combined operators such as $+=$, $-=$, $*=$, $/=$ and $\%=$. In this case, the right hand side expression is incomplete by itself and needs to be combined with the variable on the left hand side. The $+=$ operator is considered here to explain how these statements are modelled.

The detailed action schema used to model the combined add assignment statement is shown in *Figure 4.11*. Here, the original value of the variable x is added to the value of the expression before assigning the new value to the variable x . A careful comparison of *Figure 4.10* and *Figure 4.11* shows that the combined assignment action is actually a specialised case of the normal add action. Therefore, the combined action is modelled as a subtype of the main add action. This version of the action where it is modelled as a subtype is shown in *Figure 4.12*. In this case, since the *AssignAdd* action is a subtype of the *Assign* action, only the facts that are different from the *Assign* action are shown in the action specification.

The same method is used to model the *AssignSubtract*, *AssignMultiply*, *AssignDivide* and *AssignModulus* actions. These actions are given in Appendix C.

Action(Assign(x,expressionId).

PRECOND: \exists value ValueOf(expressionId,value)

EFFECT: When \exists variableId (HasName(variableId,'x')):

HasValue(variableId,_) \leftarrow HasValue(variableId,value)

\wedge when $\neg \exists$ variableId(HasName(variableId,'x')):

Generate(newVariableId)

HasName(newVariableId,'x')

HasValue(newVariableId,value)

HasInitialValue(newVariableId,value))

Figure 4.10. Assign action.

Action(AssignAdd(x,expressionId).

PRECOND: \exists value ValueOf(expressionId,value)

EFFECT: When \exists variableId (HasName(variableId,'x'))

\wedge HasValue(variableId,value2) \wedge Add(value2,value,value1):

HasValue(variableId, value2) \leftarrow HasValue(variableId,value1)

\wedge when $\neg \exists$ variableId(HasName(variableId,'x'))

\wedge HasValue(variableId,value2) \wedge Add(value2,value,value1):

Generate(newVariableId)

HasName(newVariableId,'x')

HasValue(newVariableId,value)

HasInitialValue(newVariableId,value))

Figure 4.11. Detailed version of AssignAdd action.

AssignAdd(x,expressionId) \subset Assign(x,expressionId)

Action(AssignAdd(x,expressionId).

PRECOND:

EFFECT: When \exists variableId (HasName(variableId,'x'))

HasValue(variableId,value2) \wedge Add(value2,value,value1):

HasValue(variableId, value2) \leftarrow HasValue(variableId,value1))

Figure 4.12. Subtype version of AssignAdd action.

4.5 PROGRAM ANALYSIS

This section describes how the predicates, rules and actions described in Section 4.4 are used to decide whether a PHP program is correct according to the specifications. It goes into more detail of how the predicates, rules and actions map to the program analysis as an AI problem as shown in Figure 4.3. The initial state in

this case is a set of facts which are created in the system at the start of program analysis. The goal state is the final goal describe in Section 4.4.2.

In order to analyse the student's solution, it is first converted into an Abstract Syntax Tree (AST). This AST is then walked through, node by node, creating facts that are appropriate for each node. When the preconditions for a rule become true, this is activated to create more facts. When the AST indicates that an action needs to be performed and the preconditions of the action are satisfied, the relevant action comes into effect, creating the facts that are specified in the effects of the action. Once the walking of the AST is completed, the resulting facts represent the final state. This final state is then compared against the overall goal. If all the facts in the overall goal are present in the final state the overall goal is met. However, it is possible that the program contains program statements that do not contribute towards the final goal. The analysis process next checks to ensure that all program statements are necessary to ensure that the overall goal is satisfied. If so, the program is considered correct.

In order to study this process in more detail, consider the example PHP exercise described in *Figure 4.1*. For the purpose of the analysis, assume that the student's solution to this exercise is Program a in Table 4.1.

4.5.1 Initial State

In the given example, the variable \$y contains a value at the beginning of the program. This means that this exercise contains an initial state as described in Section 4.4.2. The initial state in this case is specified in *Figure 4.13*. This uses the predicates described above to specify that a variable named y already exists in the system and contains a value of val_y. This symbolic value is used since no specific value has been given in the description. Using a symbolic value ensures that the final goal is valid, no matter what the actual value contained in the variable at this point is.

```
HasName(VARID1,'y')
 $\wedge$  HasValue(VARID1,val_y)
 $\wedge$  HasInitialValue(VARID1,val_y)
```

Figure 4.13. Initial state for example program.

The first step during program analysis is to create the initial state in the system. When this is done, the variable symbols (denoted by the upper case letters as described in Section 4.4.2) are replaced with actual ids. Let the id of the variable created at this point be *VarId1*. Then, the list of facts after creating the initial state is as below.

HasName(VarId1, 'y')

HasValue(VarId1, val_y)

HasInitialValue(VarId1, val_y)

4.5.2 Abstract Syntax Tree

In order to analyse the solution to the exercise, it is necessary to create a list of corresponding predicates. The first step in this process is to convert the written PHP program code into an Abstract Syntax Tree (AST). A major barrier to convert a PHP program into an AST is that the PHP language allows PHP and HTML code to be embedded within each other. This means that a single grammar cannot be used to convert the entire code to as AST. The solution to this problem is to use two island grammars (Section 4.1.3), one for HTML and one for PHP.

The outermost part of any web page written in PHP can be thought to be HTML. Even if the coding starts with PHP, the `<HTML>` tag is implicitly present in the outermost level of the web page. This feature of HTML which allows some tags to be present even if they are not explicitly written down is another major challenge when converting a PHP program to an AST. Several other problems are encountered when dealing with HTML code. Although most HTML tags have a beginning and ending tag, some tags do not have or do not require ending tags. Others allow self-closing (eg:- `
`). HTML tags can be written in both lower and upper case forms without any error in the program. All these issues make it very difficult to write grammars that are capable of converting programs written in PHP into ASTs.

Keeping all these in mind, an HTML grammar to handle all the tags that are used in the PHP ITS was developed (Appendix D). This grammar also considers attributes that are pertinent to the ITS. When a beginning PHP tag (`<?php`) is encountered in the input, the grammar automatically transfers control to the PHP grammar. The PHP grammar used in the system has not been developed during the course of this research project. It is a grammar that is freely available on the web

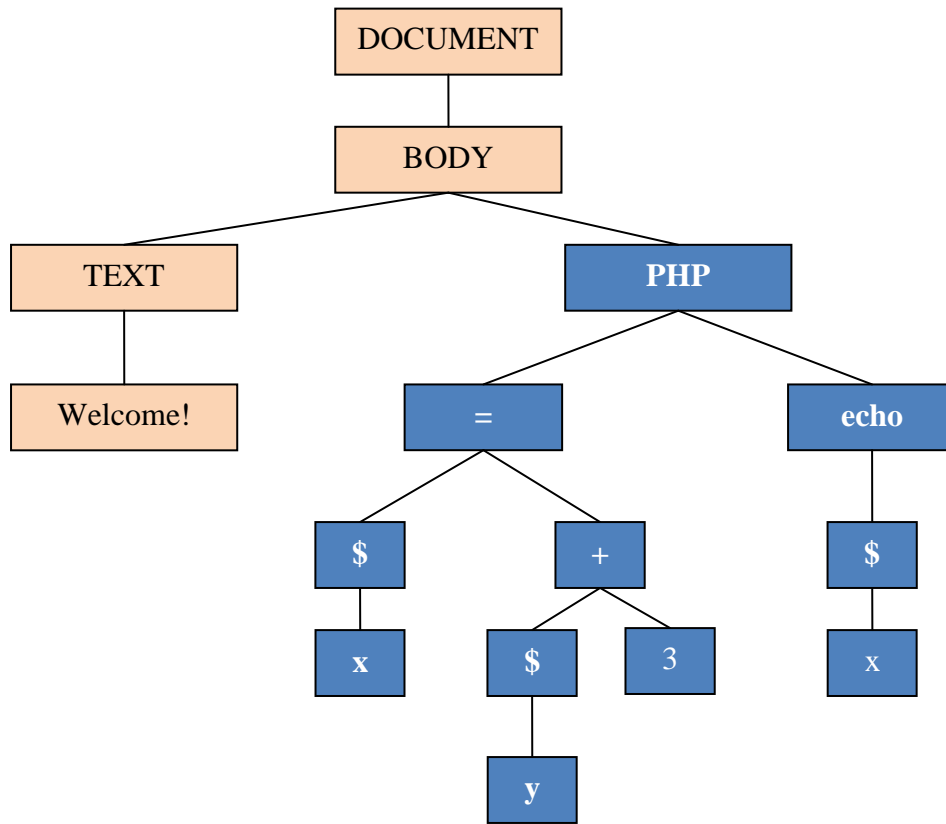
(Kuruville, 2009). However, minor modifications have been done to handle the return to the HTML grammar when an end PHP tag (?>) is encountered and to eliminate some PHP constructs which are not included in the PHP ITS (Appendix B).

To analyse Program a in Table 4.1, it is converted into an AST using the two grammars described above. This program uses HTML to write the string ‘Welcome!’ onto the web page while using PHP to perform the other operations. This exemplifies the fact that a PHP program is an integration of both HTML and PHP code.

The resulting AST is shown in *Figure 4.14*. The top part of the figure shows the graphical representation of the AST. This is a hierarchical representation. The bottom part shows a more concise, textual representation of the AST. This becomes useful for very large ASTs that would otherwise occupy a large space. In this form, each opening and closing bracket pair show a node of the AST. The first item within the bracket is the root while the rest are child nodes. Hierarchy is shown using nested brackets.

The top two nodes of all ASTs created using these two grammars are ‘DOCUMENT’ and ‘BODY’. This does not change based on whether the actual code contains the <html> and <body> tags or not. If the code contains a <head> tag, a ‘HEAD’ node is created, parallel with the ‘BODY’ node. These are created using the HTML grammar described above. When an open PHP code is encountered, control is passed to the PHP grammar. This results in an AST with a root node of ‘PHP’. Therefore, the light section of the AST in *Figure 4.14* is created using the HTML grammar and the dark section is created using the PHP grammar.

This mechanism allows handling PHP code that is embedded within HTML. However, it is also common for HTML code to be embedded within PHP. This is usually achieved by writing the HTML code within PHP echo statements. In such situations, it is sometimes necessary to know the result of certain PHP operations before it is possible to convert the HTML to an AST. For example, if the HTML code refers to a value contained in a PHP variable, this value needs to be known before the HTML code can be converted to the relevant AST. Therefore, it is not possible to achieve this during the first conversion. Any HTML code embedded within PHP is treated as simple echo statements at this point.



(DOCUMENT (BODY (TEXT Welcome!) (PHP (= (\$ x) (+ (\$ y) 3)) (echo (\$ x))))))

Figure 4.14. AST for example program.

Although this mechanism makes it possible to handle the change of code between HTML and PHP, there are certain situations which it cannot handle. If PHP code is embedded within HTML attribute lists, the grammars given here or the mechanisms described in subsequent sections are incapable of handling this.

If the program contains any syntax errors, the grammars generate errors during the AST creation process. It is possible to identify at which line and token the error occurred. However, the grammar files sometimes return incorrect positions, mainly when it cannot match a token or even guess which token the program is attempting to match. Therefore, the error position returned by the grammar is not always accurate.

If the AST creation process returns an error, the program is identified to have syntax errors. The rest of the analysis process can only continue in a program free from syntax errors.

4.5.3 Walking the AST

Once the program is converted to an AST, it is easy to walk through it node by node. Each node is then analysed and converted to the relevant facts. The tree walking happens from top to bottom, left to right.

The first two nodes encountered are ‘DOCUMENT’ and ‘BODY’. These are nodes are just used to add structure to the AST and no predicates are created as a result. The next node is a ‘TEXT’ node which specifies that a *Display* action occurs. This action operates on an expression. Therefore, an expression is created for the actual text. Since the actual text is a literal in this case a literal is created. As described in Section 4.4.1.1, a literal always works in conjunction with a literal expression. Therefore, a literal expression is also created at this point. Let the id of the generated literal be *LitId1* and the id of the expression be *LitExprId1*. Then, the following facts are created as described in Section 4.4.1.1.

$$\textit{HasLiteral}(\textit{LitExprId1}, \textit{LitId1})$$

$$\textit{HasLitValue}(\textit{LitId1}, \textit{'Welcome!'})$$

Next, the value of this literal expression is found using the relevant rule as described in Section 4.4.1.1.

$$\textit{ValueOf}(\textit{LitExprId1}, \textit{'Welcome!'})$$

$$\leftarrow \textit{HasLiteral}(\textit{LitExprId1}, \textit{LitId1}) \wedge \textit{HasLitValue}(\textit{LitId1}, \textit{'Welcome!'})$$

Now, the precondition for the *Display* action is met (Section 4.4.3) since a *ValueOf* fact is present for the expression that is the argument of the *Display* action. Therefore, the action is invoked resulting in creating a new fact equal to its effect. The second argument of the created *OnPage* fact is 1 since no other *OnPage* facts exist in the current state.

$$\textit{OnPage}(\textit{'Welcome!'}, 1)$$

The next node analysed is ‘PHP’ which has no effect on the state. Next, the ‘=’ node is analysed. This results in invoking the *Assign* action. The first argument of this action is the name of the variable to which a value is assigned. This is found by

following the AST along the left hand child of the '=' node. The variable name is found as the child node of the '\$' node, in this case x. The second argument of the action is an expression id for the right hand side of the assignment statement. This means that an expression is created for the right hand branch of the '=' node. In this case, this is a '+' node signifying that an add expression is created. Let the id of this add expression be ExprId1 and the ids of the left and right hand sub expressions of the add expression be ExprId2 and ExprId3 respectively. The child node of the left hand expression is a '\$' indicating that the left hand expression, or the one corresponding to ExprId2 is a variable expression. The child node of the '\$' node indicates that the actual variable used in the expression is y. Considering the facts that have already been created, it can be seen that the id of the variable y is VarId1 so this is the variable that is connected to the variable expression. ExprId3 corresponds to a literal expression, resulting in the creation of a literal with LitId3. The set of resultant facts is given below.

HasId(AddExpr(ExprId2,ExprId3),ExprId1)

HasVariable(ExprId2,VarId1)

HasLiteral(ExprId3,LitId3)

HasLitValue(LitId3,3)

Now, the rules are invoked to calculate the value of all the expressions as described in Section 4.4.1.1. This results in the following facts being created in the system.

ValueOf(ExprId2,val_y)

ValueOf(ExprId3,3)

ValueOf(ExprId1,value1) where Add(val_y,3,value1)

Now, the preconditions for the *Assign* action given in Section 4.4.3 are met. Therefore, this action is invoked. The effect of the *Assign* action is dependent on whether or not a variable with the name of the first argument exists. In terms of predicates, this means that it depends on whether or not the fact *HasName(VariableId,'x')* exists for some value of VariableId. Considering the current state, no such variable exists, so the second part of the effect of the *Assign*

action is invoked resulting in the generation of a variable. Let the id of this generated variable be *VarId2*. Then, the following facts are created.

HasName(VarId2,'x')

HasValue(VarId2,value1)

HasInitialValue(VarId2,value1)

The next node during the walking of the AST is the ‘echo’ node. This again results in a *Display* action with an expression. In this case, since the child node of the ‘echo’ node is a ‘\$’ node, the expression is a variable expression and the variable corresponding to the expression is *x*. Let the id of the created variable expression be *VarExprId1*. Then, the following fact is created.

HasVariable(VarExprId1,VarId2)

Next, the rule to calculate the value of the variable expression is invoked as below.

ValueOf(VarExprId1,value1)

$\leftarrow \text{HasVariable(VarExprId1,VarId2)} \wedge \text{HasValue(VarId2,value1)}$

So the *Display* action is now be invoked, resulting in the following fact.

OnPage(value1,2)

Based on this analysis, the final list of facts or the final state is shown in *Figure 4.15*.

4.5.4 Goal Checking

The final step in the program analysis process is goal checking. Based on the requirements of the exercise given in *Figure 4.1*, the overall goal can be specified as shown in *Figure 4.16*. It should be noted that this goal should be read in conjunction with the initial state specification in *Figure 4.13*. Common values in both specifications refer to the same value. The specification (*j>i*) specifies the required ordering of the output. This says that the value stored in the variable *x* should be displayed after the ‘Welcome!’ message. The overall goal also contains a constraint in this instance. This is to specify the requirement that the name of the variable to which the result of the calculation is assigned is *x*.

```

HasName(VarId1,'y')
HasValue(VarId1,val_y)
HasInitialValue(VarId1,val_y)
OnPage('Welcome!',1)
HasId(AddExpr(ExprId2,ExprId3),ExprId1)
HasVariable(ExprId2,VarId1)
HasLiteral(ExprId3,LitId3)
HasLitValue(ExprId3,3)
ValueOf(ExprId2,val_y)
ValueOf(ExprId3,3)
ValueOf(ExprId1,value1) where Add(val_y,3,value1)
HasName(VarId2,x)
HasValue(VarId2,value1)
HasInitialValue(VarId2,value1)
HasVariable(VarExprId1,VarId2)
ValueOf(VarExprId1,value1)
OnPage(value1,2)

```

Figure 4.15. Final state of example program.

```

Goal :      OnPage('Welcome!',i)
           ∧ Add(val_y,3,VALUE1)
           ∧ HasValue(VARID2,VALUE1)
           ∧ OnPage(VALUE1,j)
           ∧ (j>i)
Constraints :  HasName(VARID2,x)

```

Figure 4.16. Overall goal of example exercise.

When comparing the final state in *Figure 4.15* against this overall goal specification, it can be seen that all facts in the overall goal are present in the final state when VALUE1='value1', VARID2=VarId2, i=1 and j=2. The constraint is also

satisfied for this set of values and therefore, the program conforms to the specifications.

4.5.5 Checking for Unnecessary Program Statements

A common mistake made by many students is to include unnecessary program statements that are not necessary for the program to conform to the specification. Consider the example exercise in *Figure 4.1*. *Figure 4.17* shows an example program with an unnecessary echo statement to display the string “The value of x is :”. This is not a requirement specified in the exercise specification. Although this may make the output more attractive, some such statements may actually make the execution of the program code inefficient. Therefore, it is inadvisable to include such unnecessary statements in program code. The system is capable of identifying such extra statements and indicating this as an error.

```
Welcome!  
<?php  
$x=$y+3;  
echo("The value of x  
is : ");  
echo($x);  
?>
```

Figure 4.17. A program with unnecessary statements.

Such statements are identified in this analysis by maintaining a series of status transitions. A new status is created, each time a PHP program statement that results in a significant outcome is encountered. In the basic programs considered in this chapter, a new status is created each time an ‘echo’ statement or an assignment statement is reached during analysis. An association is then created between all facts that are newly created and the current status.

When the facts created in one status are utilised to create a new fact in another status, a link is created between the related statuses. When a rule is activated, the statuses associated with the facts that make up the premise of the rule are linked to the current status. When an assignment state is encountered, any previous statuses are linked with facts encountered when finding the value of the right hand side expression of the assignment statement are linked to the current status. Also, if the variable on the left hand side of the assignment expression was created in a previous

status, that status is linked to the current status. In the case of an ‘echo’ statement, any fact linked to finding the value of the expression being echoed is used to link previous statuses to the current status.

For example, consider Program a in Table 4.1. At the beginning of the analysis, a new status (known here as Status 0) is created. The facts related to the initial state, as described in Section 4.5.1, are associated with this state. Next, a new status (known here as Status 1) is created as soon as the assignment expression is encountered. Any new facts created as a result of the assignment expression are linked to Status 1.

Next consider rules used to find the value of the expression on the right hand side of the assignment statement as described in Section 4.4.1.1. A summary of these rules is show in *Figure 4.18*. When considering the first rule, the first premise was created in the current status so there is no need to link a previous status to the current status. However, the second premise was created as a result of the initial state and is therefore associated with Status 0. This results in a link been created between Status 1, which is the current status, and Status 0. Since all the premises of the other rules are created during the analysis of the assignment statement, they do not result in more links between statuses.

<p>ValueOf(ExprId2,10) \leftarrow HasVariable(ExprId2,VarId1) \wedge HasValue(VarId1,10)</p> <p>ValueOf(ExprId3,5) \leftarrow HasLiteral(ExprId1,LitId1) \wedge HasLitValue(LitId1,5)</p> <p>ValueOf(AddExpr(ExprId2,ExprId1),15) \leftarrow ValueOf(ExprId2,10) \wedge ValueOf(ExprId1,5) \wedge Add(10,5,15)</p>
--

Figure 4.18. Rules used to calculate the *ValueOf* the right-hand expression.

The next step in the analysis process is to create facts relevant to the echo statement. As described above, this results in the creation of a new status, known here as Status 2. The analysis of the echo statement results in the activation of the rule in *Figure 4.19* as described in Section 4.5.3. The second premise of this rule is a

result of the assignment expression and therefore is associated with Status 1. This results in a link being created between the current status (Status 2) and Status 1.

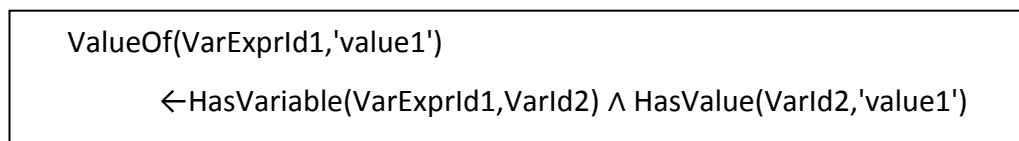


Figure 4.19. Rule used to find the *ValueOf* the echoed expression.

So the final flow of statuses resulting from the example program is as shown in Figure 4.20. This shows that a path exists from all existing statuses to the status in which the overall goal is satisfied (Status 2), indicating that all the statuses contribute to the final goal. In such cases, the program is identified as not having any unnecessary program statements.

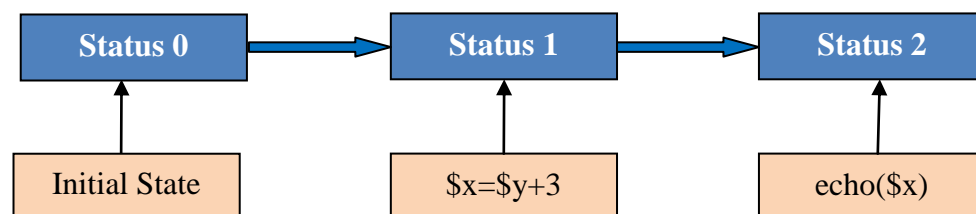


Figure 4.20. Status flow for example program.

Next consider the similar status flow model for the PHP program shown in Figure 4.17. This model is shown in Figure 4.21. The *ValueOf* the expression in the unnecessary echo statement depend on any previous statuses since it is an independent literal expression. This echo statement does not contribute to the satisfaction of the overall goal and is therefore not associated with Status 3, which is where the overall goal is satisfied. This means that Status 2 is unnecessary to achieving the overall goal of the program. Therefore, the program is identified as incorrect and the program statement leading to Status 2 is identified as an unnecessary program statement.

A similar status flow model is created during the walking of the AST for any program as described in Section 4.5.3. Once the overall goal is satisfied, this model is inspected to ensure that every status has a link, either direct or indirect, to the status in which the overall goal is satisfied. If this is the case, the program is identified as correct. If any statuses that do not link to the status where the overall

goal is satisfied are encountered, the program statements that resulted in these statuses are identified as unnecessary and the program is taken to be incorrect.

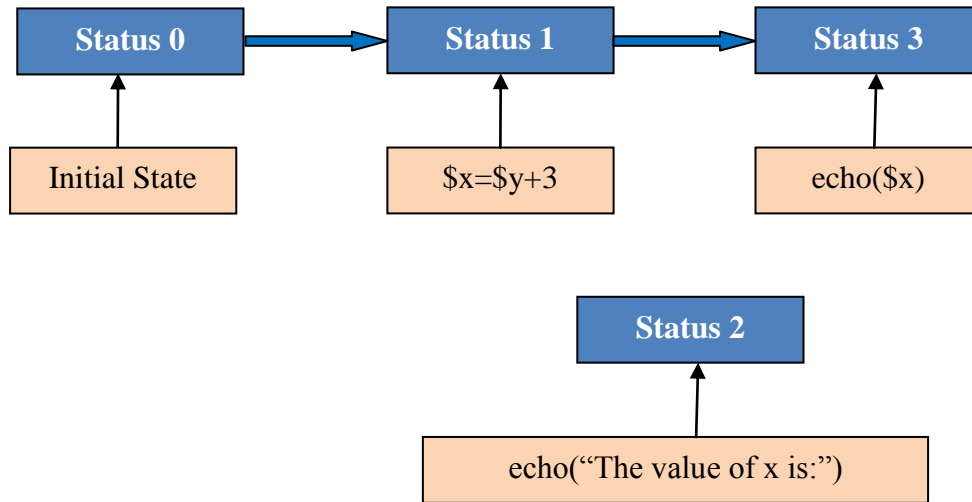


Figure 4.21. Status flow for example program with unnecessary statements.

The strength in this method of program analysis lies in the fact that it can accept many alternative solutions to the given exercise. As long as the facts defined in the goal are present in the final state, any program will be accepted as correct, no matter what actual statements were used. For example, if the ‘Welcome!’ line was written using an echo statement in PHP instead of as an HTML text as in Program b in Table 4.1, the *OnPage('Welcome!',1)* fact would still exist. Similarly, if the assignment statement was something of the form $\$x=\$y+1+2$ as in Program c in Table 4.1, the necessary facts would still exist.

4.6 SPECIAL SITUATIONS

The preceding section described how a simple program is analysed using the knowledge base in the PHP ITS. However, there are certain situations where the analysis of even simple PHP programs becomes more complicated. This section describes uses of PHP that need to be handled in special ways.

4.6.1 Multiple *OnPage* Predicates

Data on a web page can be displayed using either HTML statements or PHP ‘echo’ and ‘print’ statements. These statements can take argument strings of varying length. Therefore, a single string can be displayed on a web page using many

combinations of statements. Table 4.2 shows two methods that can be used to display the string “Hello World” on a PHP web page.

Table 4.2

Different Methods of Displaying “Hello World” on a PHP Web Page

<i>Program a</i>	<i>Program b</i>
<pre><?php echo("Hello World"); ?></pre>	<pre><?php echo("Hello"); echo(" World"); ?></pre>

If these two programs are converted to facts as described in Section 4.5, the first program results in a single fact *OnPage('Hello World',1)* while the second program results in two facts *OnPage('Hello',1)* and *OnPage(' World',2)*. If the objective is to display the string “Hello World” on a web page, both these programs are correct. When specifying the overall goal, it is not possible to enumerate all the possible combinations of facts. In this case, the overall goal is specified as *OnPage('Hello World',x)*. Therefore, when matching the final state against the overall goal, the second program is identified as incorrect.

The knowledge base handles this problem by using a special method to check for *OnPage* predicates included in the overall goal. First, it checks to see whether the exact string specified in the goal is present in any *OnPage* facts in the system. If so, the goal is taken to be satisfied. If this is not the case, It concatenates the *OnPage* predicates in order of their second argument to see whether the string given in the goal can be obtained. If this can be achieved, the goal is taken to be satisfied. If not, the program is identified as incorrect.

When the overall goal is achieved by the concatenation of the first arguments in several *OnPage* facts, the statuses associated with each of these predicates contribute to achieving the overall goal. Therefore, links are created between the current status (where the overall goal is being checked) and the statuses associated with the contributing *OnPage* facts. This ensures that the new statuses created by the corresponding echo statements are taken to contribute to the overall goal and are not considered unnecessary.

4.6.2 Pre and Post Increment and Decrement Operators

Pre and post increment and decrement operators are used very often in PHP programs. Variables qualified with a pre or post increment or decrement operator can be used as two types of PHP constructs: either as expressions or as complete statements.

When applied to the right hand side of an assignment statement or within an ‘echo’ statement, they behave as other types of expressions. Therefore, they are modelled as a subtype of a calculate expression. The *PrePostFixExpr* in Figure 4.5 is used to model this behaviour of pre and post increment and decrement operators. In this case, the expression is not a combination of other expressions but is connected to a variable using the *HasPrePostVariable* predicate, and a fix type using the *HasFixType* predicate. The *FixType* can take the values INCREMENT or DECREMENT. *PrePostFixExpr* is divided into two further subtypes, *PreFixExpr* and *PostFixExpr*. The relevant ORM diagram is shown in Figure 4.22.

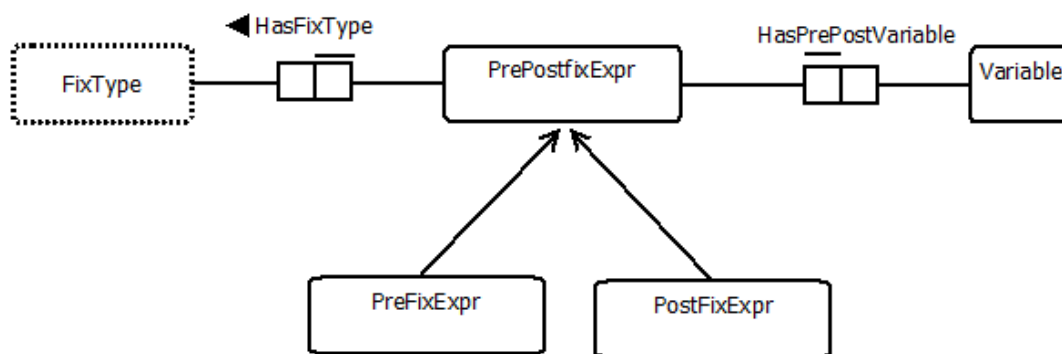


Figure 4.22. ORM diagram for pre and post fix expressions.

The value of the expression is calculated in a similar manner to other expressions. The fix type is unimportant to calculate the value of any prefix expression since the value of the expression is the value of the variable after the operation. However, the fix type plays an important role in the calculation of the value of a post fix expression since the expression value is the value of the expression before performing the necessary operation on the variable. The rules to calculate these values are given in Figure 4.23.

$\text{ValueOf}(\text{preExprId},v) \leftarrow \text{HasPrePostVariable}(\text{preExprId},\text{varId})$ $\wedge \text{HasValue}(\text{varId},v)$
$\text{ValueOf}(\text{postExprId},v) \leftarrow \text{HasPrePostVariable}(\text{postExprId},\text{varId})$ $\wedge \text{HasFixType}(\text{postExprId},\text{'INCREMENT'})$ $\wedge \text{HasValue}(\text{varId},\text{val1})$ $\wedge \text{Subtract}(\text{val1},1,v)$
$\text{ValueOf}(\text{postExprId},v) \leftarrow \text{HasPrePostVariable}(\text{postExprId},\text{varId})$ $\wedge \text{HasFixType}(\text{postExprId},\text{'DECREMENT'})$ $\wedge \text{HasValue}(\text{varId},\text{val1})$ $\wedge \text{Add}(\text{val1},1,v)$

Figure 4.23. Rules for calculating the ValueOf pre and post fix expressions.

As mentioned previously, a pre or post increment or decrement statement can also behave as a statement of its own. In such cases, and even in the case where it behaves as an expression, it also changes the value of the corresponding variable. In other words, the associated variable is associated a new value based on the operator used. This is similar to an assignment statement with the variable on the left hand side and the variable plus or minus one on the right hand side. This is modelled using the same principle as the assignment statement. Each time a pre or post increment or decrement operator is encountered the AST is modified to correspond to the relevant assignment statement. The AST created by the pre or post increment statement and the corresponding modified AST is shown in Table 4.3.

This AST is then used in the walking process, thereby ensuring that the value of the variable is changed appropriately.

Table 4.3
Modified ASTs for Pre and Post Increment and Decrement

Operation	Original AST	Modified AST
Post Increment	(Postfix ++ (\$ i))	(= (\$ i) (+ (\$ i) 1))
Post Decrement	(Postfix -- (\$ i))	(= (\$ i) (- (\$ i) 1))
Pre Increment	(Prefix ++ (\$ i))	(= (\$ i) (+ (\$ i) 1))
Pre Decrement	(Prefix -- (\$ i))	(= (\$ i) (- (\$ i) 1))

4.6.3 HTML Embedded Within PHP

As mentioned in Section 4.5.2, the grammar files for HTML and PHP can only handle PHP code embedded within HTML. However, it is common practice to embed HTML within PHP code. For example, HTML tags can be embedded within PHP echo statements as shown in *Figure 4.24*. In this program, the `<body>` tag is opened within the HTML code while it is closed within the PHP code. Although this seems a little unusual, it is perfectly legitimate PHP code. This becomes useful in situations such as when the attributes of the starting tag differ based on a condition.

```
<body>
<?php
echo("Hello World</body>");
?>
```

Figure 4.24. Example of HTML embedded within PHP.

If this program is passed directly through the HTML grammar, it results in an error since the HTML code is incorrect by itself. It only forms valid HTML when the PHP echo statement is first converted to its equivalent HTML form. In order to handle this problem, the process of AST walking is done more than once. In the first walk through the AST, any PHP statements are converted to the corresponding predicates. However, all HTML statements and the output of PHP echo statements are used to create a new input stream for the HTML grammar. The new input stream created in this manner for the example program is shown in *Figure 4.25*. Then, the resultant input stream is the continuous HTML stream that is displayed on the web page. This is then converted to another AST using the HTML grammar before walking through this new AST.

```
<body>
Hello World
</body>
```

Figure 4.25. New HTML input stream.

4.7 CHAPTER SUMMARY

This chapter provided an introduction to the domain module used in the PHP ITS. It discussed how the parts of the KB were created in a manner that enabled it to

analyse answers to exercises written in PHP. The chapter also looked at how alternative solutions to a given problem were accepted by the analysis process. The step by step process of analysing a program was discussed. Finally, it outlined a few special situations that were encountered in simple PHP programs and how they were handled.

The next chapter looks at how more advanced PHP programs are analysed. It explores how arrays, different types of selection structures and PHP functions are modelled in the KB and how programs containing these constructs are analysed.

Chapter 5: Selection Structures

The previous chapter described how the system analyses a simple program written in PHP. It concentrated on displaying data and assignment statements. Selection structures are a more advanced type of statement that are used extensively in writing computer programs. This chapter looks in detail at how the system handles the different types of selection structures available in PHP. Section 5.1 describes how the goal is specified for an exercise that requires selection structures. Section 5.2 discusses how programs with selection structures are analysed. Section 5.3 investigates how alternative solutions to a given problem are handled when the required program uses selection structures. Section 5.4 looks at how conditional expressions with the **and**, **or** and **not** Boolean operators are handled. Section 5.5 describes the analysis process for nested selection structures while Section 5.6 discusses switch statements. Section 5.7 examines how unnecessary statements in selection structures are identified by the system. Finally, Section 5.8 summarises how the system handles the processing of selection structures.

Selection statements check to see whether some condition is true or false before executing a list of other statements based on the result. One of the main challenges in handling such structures is that the same condition can be given in many forms as shown in Table 5.1, which is an excerpt from Weragama & Reye (2012). In this case, all three programs achieve the same objective of setting the variable \$y to 0 if the value of \$x is greater than 10 and to 1 in all other instances, given that \$x is an integer. The main difficulty in the design of the knowledge base is to be able to identify this fact since it is very likely that different solutions be supplied by different students.

Table 5.1
Programs to Illustrate Different Forms of the Same Conditions

Program a	Program b	Program c
<pre>if(\$x>10) \$y=0; else \$y=1;</pre>	<pre>if(\$x>=11) \$y=0; else \$y=1;</pre>	<pre>if(\$x<=10) \$y=1; else \$y=0;</pre>

5.1 GOAL SPECIFICATION

As described in Section 4.5, in order to analyse whether a program is correct, it is necessary to set an actual goal using a set of predicates. In order to do this, a set of Boolean predicates are defined. These predicates are shown in *Figure 5.1*. The facts based on these predicates come into existence only if the given condition is true. For example, if x is greater than y , the fact $GreaterThan(x,y)$ is created in the system. It should be noted that x and y represent symbolic or numeric values and not PHP variables.

GreaterThan(x,y) GreaterThanOrEqual(x,y) LessThan(x,y) LessThanOrEqual(x,y) EqualTo(x,y) NotEqualTo(x,y)

Figure 5.1. Boolean predicates used for comparison.

The conditional nature of the goal is modelled using the First Order Logic (FOL) concept of implication. Consider the example discussed in Table 5.1. It can be seen that before this program can be analysed, it is necessary for the variable x to have a value. This is specified by the initial state of the program as described in Section 4.5.1. The initial state and the overall goal for this program are shown in *Figure 5.2*. It can be seen that the initial value of the variable x is taken to be val_x . The constraint specifies that $VARID2$ represents the variable y . The goal specifies that when val_x is greater than 10, variable y should have a value 0. The value of variable y should be 1 when val_x is not greater than 10. However, in order to make it easier to analyse programs, the goal is never specified using the negative forms of predicates. This is because in practice, it is difficult to have the ‘not’ form of facts as the ‘not’ form usually means the fact is false or that it is not present. Therefore, the logical equalities in mathematics are considered and ‘not greater than’ is taken to be equivalent to ‘less than or equal to’. This is apparent in the overall goal specification in *Figure 5.2*.

Initial State	: HasName(VARID1,'x') \wedge HasValue(VARID1,val_x) \wedge HasInitialValue(VARID1,val_x)
Goal	: (GreaterThan(val_x,10) \rightarrow HasValue(VARID2,0)) \wedge (LessThanOrEqualTo(val_x,10) \rightarrow HasValue(VARID2,1))
Constraints	: HasName(VARID2,'y')

Figure 5.2. Initial state and overall goal of example program for selection.

It is important to note that there is an exception to this rule as shown in *Figure 5.1*. Although the not form of the other predicates are different predicates, there is no such not form for the *EqualTo* predicate. In order to avoid this problem, a separate predicate, *NotEqualTo*, is defined.

5.2 PROGRAM ANALYSIS

Consider how the system analyses Program a in Table 5.1. Since an initial state is defined, the following facts are created in the system. Assume that the id assigned to *Variable* x is VarId1. This results in the following list of facts.

HasName(VarId1,'x')

HasValue(VarId1,val_x)

HasInitialValue(VarId1,val_x)

Next consider how the AST created by the program is walked through. The textual representation of the AST created by this program is given in *Figure 5.3*. It can be seen that the 'If' node contains three child nodes, the first for the condition, the second for what to do if the condition is satisfied, and the third for what to do if the condition is not satisfied.

(DOCUMENT (BODY (PHP (If (> (\$ x) 10) (= (\$ y) 0) (= (\$ y) 1))))))

Figure 5.3. AST for example program for selection.

GreaterThan(value1,value2)
 $\leftarrow \text{HasId}(\text{GreaterExpr}(\text{exprId1},\text{exprId2}),\text{exprId3})$
 $\wedge \text{ValueOf}(\text{exprId3},\text{True}) \wedge \text{ValueOf}(\text{exprId1},\text{value1}) \wedge \text{ValueOf}(\text{exprId2},\text{value2})$

LessThanOrEqual(value1,value2)
 $\leftarrow \text{HasId}(\text{GreaterExpr}(\text{exprId1},\text{exprId2}),\text{exprId3})$
 $\wedge \text{ValueOf}(\text{exprId3},\text{false}) \wedge \text{ValueOf}(\text{exprId1},\text{value1}) \wedge \text{ValueOf}(\text{exprId2},\text{value2})$

GreaterThanOrEqual(value1,value2)
 $\leftarrow \text{HasId}(\text{GreaterEqualExpr}(\text{exprId1},\text{exprId2}),\text{exprId3})$
 $\wedge \text{ValueOf}(\text{exprId3},\text{True}) \wedge \text{ValueOf}(\text{exprId1},\text{value1}) \wedge$
 $\text{ValueOf}(\text{exprId2},\text{value2})$

LessThan(value1,value2)
 $\leftarrow \text{HasId}(\text{GreaterEqualExpr}(\text{exprId1},\text{exprId2}),\text{exprId3})$
 $\wedge \text{ValueOf}(\text{exprId3},\text{false}) \wedge \text{ValueOf}(\text{exprId1},\text{value1}) \wedge \text{ValueOf}(\text{exprId2},\text{value2})$

LessThanOrEqual(value1,value2)
 $\leftarrow \text{HasId}(\text{LessEqualExpr}(\text{exprId1},\text{exprId2}),\text{exprId3})$
 $\wedge \text{ValueOf}(\text{exprId3},\text{True}) \wedge \text{ValueOf}(\text{exprId1},\text{value1}) \wedge \text{ValueOf}(\text{exprId2},\text{value2})$

GreaterThan(value1,value2)
 $\leftarrow \text{HasId}(\text{LessEqualExpr}(\text{exprId1},\text{exprId2}),\text{exprId3})$
 $\wedge \text{ValueOf}(\text{exprId3},\text{false}) \wedge \text{ValueOf}(\text{exprId1},\text{value1}) \wedge \text{ValueOf}(\text{exprId2},\text{value2})$

LessThan(value1,value2)
 $\leftarrow \text{HasId}(\text{LessExpr}(\text{exprId1},\text{exprId2}),\text{exprId3})$
 $\wedge \text{ValueOf}(\text{exprId3},\text{True}) \wedge \text{ValueOf}(\text{exprId1},\text{value1}) \wedge \text{ValueOf}(\text{exprId2},\text{value2})$

GreaterThanOrEqual(value1,value2)
 $\leftarrow \text{HasId}(\text{LessExpr}(\text{exprId1},\text{exprId2}),\text{exprId3})$
 $\wedge \text{ValueOf}(\text{exprId3},\text{false}) \wedge \text{ValueOf}(\text{exprId1},\text{value1}) \wedge \text{ValueOf}(\text{exprId2},\text{value2})$

EqualTo(value1,value2)
 $\leftarrow \text{HasId}(\text{EqualExpr}(\text{exprId1},\text{exprId2}),\text{exprId3})$
 $\wedge \text{ValueOf}(\text{exprId3},\text{True}) \wedge \text{ValueOf}(\text{exprId1},\text{value1}) \wedge \text{ValueOf}(\text{exprId2},\text{value2})$

NotEqualTo(value1,value2)
 $\leftarrow \text{HasId}(\text{EqualExpr}(\text{exprId1},\text{exprId2}),\text{exprId3})$
 $\wedge \text{ValueOf}(\text{exprId3},\text{false}) \wedge \text{ValueOf}(\text{exprId1},\text{value1}) \wedge \text{ValueOf}(\text{exprId2},\text{value2})$

Figure 5.4. Rules for converting Boolean expressions into comparison predicates.

When the condition node is reached, a *BooleanExpression* is created as explained in Section 4.4.1.1. Let the id of this expression be *ExprId1*. The left hand side of the *BooleanExpression* is a *VariableExpr* and the right hand side is a *LiteralExpr*. Let the ids of these expressions be *VarExprId1* and *LitExprId1* respectively. Let the id of the created *Literal* be *LitId1*. Then, the following set of facts is created.

HasId(GreaterThanExpr(VarExprId1,LitExprId1),ExprId1)

HasVariable(VarExprId1,VarId1)

HasLiteral(LitExprId1,LitId1)

HasLitValue(LitId1,10)

The values of the *VariableExpr* and *LiteralExpr* are calculated using the rules in *Figure 4.8*, resulting in the following facts.

ValueOf(VarExprId1,val_x)

ValueOf(LitExprId1,10)

Considering the semantics of the selection statement, the value of the conditional expression is True for the second child node in the AST and False for the third child node. Therefore, separate sets of facts are maintained for the two nodes.

First consider the second node of the AST where the conditional expression is true. Therefore, inside this node, the following fact is present.

ValueOf(ExprId1,True)

As mentioned previously, it is possible to write this condition in many different ways. Therefore, working with a specific type of expression in a predicate will result it being impossible to accept other equivalent conditional expressions. In order to avoid this problem, the set of rules in *Figure 5.4* are used to find the corresponding more generalised predicate explained in *Figure 5.1*. Using the first rule here and considering the case when the conditional expression is true, the following fact is obtained.

GreaterThan(val_x,10)

This predicate implies whatever facts created in the second node of the conditional AST, i.e. the facts created by the assignment node. This node results in

the following fact using the *Assign* action in *Figure 4.10*. Assume that the id of the newly created *Variable* is *VarId2*.

$$\textit{HasName}(\textit{VarId2}, 'y')$$

$$\textit{HasValue}(\textit{VarId2}, 0)$$

$$\textit{HasInitialValue}(\textit{VarId2}, 0)$$

So the combined result for the second node of the selection section of the AST can be written as below.

$$\textit{GreaterThan}(\textit{val}_x, 10) \rightarrow \textit{HasValue}(\textit{VarId2}, 0)$$

Similarly, considering the third node of the selection section of the AST, the value of the expression is *False*. Therefore, the following fact is created.

$$\textit{ValueOf}(\textit{ExprId2}, \textit{False})$$

Using the second rule in *Figure 5.4*, the following fact is created.

$$\textit{LessThanOrEqual}(\textit{val}_x, 10)$$

Following the same procedure as above, the combined result for the third node of the selection section of the AST can be written as below.

$$\textit{LessThanOrEqual}(\textit{val}_x, 10) \rightarrow \textit{HasValue}(\textit{VarId2}, 1)$$

So the final state contains the following facts.

$$\textit{HasName}(\textit{VarId2}, y)$$

$$\wedge (\textit{GreaterThan}(\textit{val}_x, 10) \rightarrow \textit{HasValue}(\textit{VarId2}, 0))$$

$$\wedge (\textit{LessThanOrEqual}(\textit{val}_x, 10) \rightarrow \textit{HasValue}(\textit{VarId2}, 1))$$

Therefore, the overall goal is satisfied when *VARID2=VarId2*. This means that the program is identified as correct.

5.2.1 Incorrect Solutions

It is essential that the system not only identifies correct programs but also incorrect programs. In order to see how this is done, consider the example PHP program for the above exercise shown in *Figure 5.5*. The corresponding AST is shown in *Figure 5.6*. Upon comparison with *Figure 5.3*, it can be seen that only the conditional expression is different.

```

if($x>9)
{
    $y=0;
}
else
{
    $y=1;
}

```

Figure 5.5. Incorrect solution to example exercise for selection structures.

```

(DOCUMENT (BODY (PHP (If (> ($ x) 9) (= ($ y) 0) (= ($ y) 1))))))

```

Figure 5.6. AST for incorrect solution to exercise.

When this node is reached, a *BooleanExpression* containing a *VariableExpr* on the left hand side and a *LiteralExpr* on the right hand side is created as described above. Let the ids of the *BooleanExpression*, *VariableExpr* and *LiteralExpr* be *ExprId1*, *VarExprId1* and *LitExprId1* respectively. Let the id of the created *Literal* be *LitId1*. Then, the following set of facts is created.

HasId(GreaterEqualExpr(VarExprId1,LitExprId1),ExprId1)

HasVariable(VarExprId1,VarId1)

HasLiteral(LitExprId1,LitId1)

HasLitValue(LitId1,9)

The following facts are again created when the *ValueOf* each of these expressions are calculated as explained in Section 4.4.1.1.

ValueOf(VarExprId1,val_x)

ValueOf(LitExprId1,9)

Considering the section of the AST where the conditional expression is true, the following fact is created.

ValueOf(ExprId1,True)

Using the first rule in Figure 5.4 the following fact is obtained.

GreaterThan(val_x,9)

Similarly, analysing the else part of the AST results in the following fact being created.

$$\textit{LessThanOrEqual}(\textit{val_x},9)$$

Since the rest of the AST is the same, the resulting final state contains the following facts.

$$\textit{HasName}(\textit{VarId2},y)$$

$$\wedge (\textit{GreaterThan}(\textit{val_x},9) \rightarrow \textit{HasValue}(\textit{VarId2},0))$$

$$\wedge (\textit{LessThanOrEqual}(\textit{val_x},9) \rightarrow \textit{HasValue}(\textit{VarId2},1))$$

When comparing this set of facts against the overall goal in *Figure 5.2*, it can be seen that it is not satisfied for any value of VARID2. The system therefore identifies this program as incorrect.

Appendix E shows the analysis of several other incorrect solutions for this programming exercise.

5.3 ALTERNATIVE SOLUTIONS

As mentioned at the start of the section, a main strength of the knowledge base is the ability to identify different correct solutions to the same problem. In order to illustrate this, consider how Program b in Table 5.1 is analysed. The AST created is shown in *Figure 5.7*. When comparing this AST with the one in *Figure 5.3* it can be seen that the only difference is in the section corresponding to the condition of the selection statement.

```
(DOCUMENT (BODY (PHP (If (>= ($ x) 11) (= ($ y) 0) (= ($ y) 1))))))
```

Figure 5.7. AST for Program b in Table 5.1.

Again, when the condition node is reached, a *BooleanExpression* is created as explained in Section 4.4.1.1. Let the id of this expression be ExprId1. The left hand side of the *BooleanExpression* is again a *VariableExpr* and the right hand side is a *LiteralExpr*. Let the ids of these expressions be VarExprId1 and LitExprId1 respectively. Let the id of the created *Literal* be LitId1. Then, the following set of facts is created.

HasId(GreaterEqualExpr(VarExprId1,LitExprId1),ExprId1)

HasVariable(VarExprId1,VarId1)

HasLiteral(LitExprId1,LitId1)

HasLitValue(LitId1,11)

The values of the *VariableExpr* and *LiteralExpr* are calculated again using the rules in *Figure 4.8*, resulting in the following facts.

ValueOf(VarExprId1,val_x)

ValueOf(LitExprId1,11)

Next consider the second node of the AST where the conditional expression is true. Therefore, inside this node, the following fact is present.

ValueOf(ExprId1,True)

Using the third rule in *Figure 5.4* the following fact is obtained.

GreaterThanOrEqual(val_x,11)

A set of rules are included in the KB to handle equivalent expressions. These rules are shown in *Figure 5.8*. Using the second rule in this figure, since *Subtract(11,1,10)*, the above fact creates the new fact given below.

GreaterThan(val_x,10)

Similarly, for the third node of the selection section of the AST, the fact *LessThan(val_x,11)* is created. Again using the rules in *Figure 5.8*, this converts to *LessThanOrEqual(val_x,10)*. Since the rest of the AST is identical to that in *Figure 5.3*, the facts created for the two separate states are the same as before. The final resulting state contains the following facts.

HasName(VarId2,y)

$\wedge (GreaterThan(val_x,10) \rightarrow HasValue(VarId2,0))$

$\wedge (LessThanOrEqual(val_x,10) \rightarrow HasValue(VarId2,1))$

This is identical to the final state of Program a as described in Section 5.2. Therefore, although Program b uses a different condition than Program a, this program is also identified as correct by the system.

A similar analysis of Program c in Table 5.1 can be found in Appendix E. This same method of program analysis is used to identify any solution that is made up of equivalent expressions. This is a very powerful feature when analysing computer programs.

$\text{LessThanOrEqual}(\text{value2}, \text{value1}) \leftarrow \text{GreaterThanOrEqual}(\text{value1}, \text{value2})$
$\text{GreaterThan}(\text{value1}, \text{value3}) \leftarrow \text{GreaterThanOrEqual}(\text{value1}, \text{value2})$
$\quad \wedge \text{Subtract}(\text{value2}, 1, \text{value3})$
$\text{LessThan}(\text{value3}, \text{value1}) \leftarrow \text{GreaterThanOrEqual}(\text{value1}, \text{value2})$
$\quad \wedge \text{Subtract}(\text{value2}, 1, \text{value3})$
$\text{LessThan}(\text{value2}, \text{value1}) \leftarrow \text{GreaterThan}(\text{value1}, \text{value2})$
$\text{GreaterThanOrEqual}(\text{value1}, \text{value3}) \leftarrow \text{GreaterThan}(\text{value1}, \text{value2})$
$\quad \wedge \text{Add}(\text{value2}, 1, \text{value3})$
$\text{LessThanOrEqual}(\text{value3}, \text{value1}) \leftarrow \text{GreaterThan}(\text{value1}, \text{value2})$
$\quad \wedge \text{Add}(\text{value2}, 1, \text{value3})$
$\text{GreaterThanOrEqual}(\text{value2}, \text{value1}) \leftarrow \text{LessThanOrEqual}(\text{value1}, \text{value2})$
$\text{LessThan}(\text{value1}, \text{value3}) \leftarrow \text{LessThanOrEqual}(\text{value1}, \text{value2})$
$\quad \wedge \text{Add}(\text{value2}, 1, \text{value3})$
$\text{GreaterThan}(\text{value3}, \text{value1}) \leftarrow \text{LessThanOrEqual}(\text{value1}, \text{value2})$
$\quad \wedge \text{Add}(\text{value2}, 1, \text{value3})$
$\text{GreaterThan}(\text{value2}, \text{value1}) \leftarrow \text{LessThan}(\text{value1}, \text{value2})$
$\text{LessThanOrEqual}(\text{value1}, \text{value3}) \leftarrow \text{LessThan}(\text{value1}, \text{value2})$
$\quad \wedge \text{Subtract}(\text{value2}, 1, \text{value3})$
$\text{GreaterThanOrEqual}(\text{value3}, \text{value1}) \leftarrow \text{LessThan}(\text{value1}, \text{value2})$
$\quad \wedge \text{Subtract}(\text{value2}, 1, \text{value3})$

Figure 5.8. Rules for converting between equivalent expression subtypes.

5.4 OTHER FORMS OF CONDITIONAL EXPRESSIONS

Section 5.3 discusses how the knowledge base handles alternative solutions to selection structures. However, this method only works if the conditional expression within the *if* statement is an expression consisting of a comparison statement with two expressions on either side. Several other types of conditional expressions are also permissible within PHP. This section looks at how the knowledge base handles this type of conditional expressions.

5.4.1 Simple Expressions Behaving as Conditional Expressions

Sometimes, the conditional expression can be a single *SimpleExpression*. It can be either a *LiteralExpr* or a *VariableExpr* evaluating to True or False. Such a program which accomplishes the same objective as the programs in Table 5.1 is shown in *Figure 5.9*.

```
$z=$x>10;
if($z)
{
    $y=0;
}
else
{
    $y=1;
}
```

Figure 5.9. A solution to the example exercise for selection structures using a conditional statement with a *SimpleExpression*.

In this program, a *BooleanExpression* is assigned to a *Variable* which is then used as a conditional statement in the selection structure. In order to see how this program is analysed, consider that the initial state is as mentioned in Section 5.2. In this case, an assignment is encountered before the selection structure. The right hand side of the assignment is a *GreaterExpr*. Let the id of this be *ExprId1* and the ids of the two sides of the expression be *VarExprId1* and *LitExprId1* respectively. The left hand side of the *GreaterExpr* is actually a *VariableExpr* referring to the variable in the initial state and the right hand side is a *LiteralExpr*. Let the id of the corresponding *Literal* be *LitId1*. Then, the following facts are created.

HasId(GreateExpr(VarExprId1,LitExprId1),ExprId1)

HasVariable(VarExprId1,VarId1)

HasLiteral(LitExprId1,LitId1)

HasLitValue(LitId1,10)

The values of the *VariableExpr* and *LiteralExpr* are calculated again using the rules in *Figure 4.8*, resulting in the following facts.

ValueOf(VarExprId1,val_x)

ValueOf(LitExprId1,10)

Then, the ValueOf the entire expression is calculated, again using the rules in *Figure 4.8*. For this purpose assume the fact *Greater(val_x,10,value)* is true. Then, the following fact is created.

ValueOf(ExprId1,value)

Next, the value of this expression is assigned to a new *Variable* z using the *Assign* action in *Figure 4.10*. Assume that the id of the newly created *Variable* is VarId2.

HasName(VarId2,'z')

HasValue(VarId2,value)

HasInitialValue(VarId2,value)

Next, an expression is created for the conditional expression in the *if* statement as before. However, in this case, the conditional expression is a *VariableExpr*. Let this expression have an id of VarExprId2. Since it refers to the variable created earlier, the following fact is created.

HasVariable(VarExprId2,VarId2)

Inside the first part of the *if* condition, this conditional expression is true so the following fact is valid inside this section.

ValueOf(VarExprId2,True)

The set of rules to convert Boolean expressions into corresponding comparison predicates shown in *Figure 5.4* is extended to handle situations where the conditional expression is a simple expression as shown in *Figure 5.10*. The first rule in this figure now operates on the existing facts to create the following fact.

EqualTo(value,True)

In order to handle this situation, it is also necessary to identify the mathematical fact that if two values are equal, one of them can be used in place of the other. The first rule in *Figure 5.11* is used to achieve this. Using this rule on the existing set of facts, the following additional fact is created.

ValueOf(ExprId1,True)

```

EqualTo(value,True)
  ← HasId(variableExpr,varExprId1)
  ∧ HasVariable(varExprId1,varId1)
  ∧ ValueOf(varExprId1,True)
  ∧ HasValue(varId1,value)

```

```

EqualTo(value,False)
  ← HasId(variableExpr,varExprId1)
  ∧ HasVariable(varExprId1,varId1)
  ∧ ValueOf(varExprId1,False)
  ∧ HasValue(varId1,value)

```

```

EqualTo(value,True)
  ← HasId(literalExpr,litExprId1)
  ∧ HasLiteral(litExprId1,litId1)
  ∧ ValueOf(litExprId1,True)
  ∧ HasLitValue(litId1,value)

```

```

EqualTo(value,False)
  ← HasId(literalExpr,litExprId1)
  ∧ HasLiteral(litExprId1,litId1)
  ∧ ValueOf(litExprId1,False)
  ∧ HasLitValue(litId1,value)

```

Figure 5.10. Rules to convert VariableExprs into comparison predicates.

```

ValueOf(exprId1,True) ← ValueOf(exprId1,value) ∧ EqualTo(value,True)
ValueOf(exprId1,False) ← ValueOf(exprId1,value) ∧ EqualTo(value,False)

```

Figure 5.11. Rule to handle mathematical equality.

Now, the previous rules to convert Boolean expressions to corresponding comparison predicates shown in Figure 5.4 are activated. Using the first rule here, the following fact is created.

GreaterThan(val_x,10)

When this condition is satisfied, the variable *y* is assigned a value 0. This results in the following facts as explained in Section 5.2. Here, the id of the newly created *Variable* is taken to be *VarId3*.

HasName(VarId3,'y')

HasValue(VarId3,0)

HasInitialValue(VarId0,0)

Similarly, for the *else* part of the selection structure, the following fact is true.

ValueOf(VarExprId2,False)

Again using the rules in *Figure 5.10* and *Figure 5.11*, the following facts are created.

EqualTo(value,False)

ValueOf(ExprId1,False)

Next, using the rules in *Figure 5.4*, the following fact is created.

LessThanOrEqual(val_x,10)

When this condition is satisfied, the variable *y* is set to the value 1, resulting in the following facts.

HasName(VarId3,'y')

HasValue(VarId3,1)

HasInitialValue(VarId0,1)

So, the final state of the program in this case can be written as below.

HasName(VarId3,y)

$\wedge (Greater\ Than(val_x,10) \rightarrow Has\ Value(VarId3,0))$

$\wedge (Less\ Than\ Or\ Equal(val_x,10) \rightarrow Has\ Value(VarId3,1))$

Therefore, the overall goal is satisfied when VARID2=VarId3. This means that the program is identified as correct.

5.4.2 Conditional Expressions with *And*, *Or* and *Not*

Section 5.3 discussed how to handle situations where the conditional expression consists of a single comparison expression. However, it is common to group several such statements with '&&', '||' and '!' operators to form more complex conditional statements. A set of rules that allow handling these situations are shown in *Figure 5.12*. These rules are first used to find the values of the sub expressions and then, the rules in *Figure 5.4* are used to find the relevant conditional facts.

In order to illustrate this, consider the example exercise given in *Figure 5.13*. The overall goal for this program is given in *Figure 5.14*. An example solution is given in *Figure 5.15*.

ValueOf(exprId1,True) \wedge ValueOf(exprId2,True) \leftarrow HasId(AndExpr(exprId1,exprId2),exprId3) \wedge ValueOf(exprId3,True)
ValueOf(exprId1,False) \leftarrow HasId(AndExpr(exprId1,exprId2),exprId3) \wedge ValueOf(exprId3,False) \wedge ValueOf(exprId2,True)
ValueOf(exprId2,False) \leftarrow HasId(AndExpr(exprId1,exprId2),exprId3) \wedge ValueOf(exprId3,False) \wedge ValueOf(exprId1,True)
ValueOf(exprId1,True) \leftarrow HasId(OrExpr(exprId1,exprId2),exprId3) \wedge ValueOf(exprId3,True) \wedge ValueOf(exprId2,False)
ValueOf(exprId2,True) \leftarrow HasId(OrExpr(exprId1,exprId2),exprId3) \wedge ValueOf(exprId3,True) \wedge ValueOf(exprId1,False)
ValueOf(exprId1,False) \wedge ValueOf(exprId2,False) \leftarrow HasId(OrExpr(exprId1,exprId2),exprId3) \wedge ValueOf(exprId3,False)
ValueOf(exprId1,False) \leftarrow HasId(NotExpr(exprId1),exprId2) \wedge ValueOf(exprId2,True)
ValueOf(exprId1,True) \leftarrow HasId(NotExpr(exprId1),exprId2) \wedge ValueOf(exprId2,False)

Figure 5.12. Rules for handling complex conditional expressions.

Write a PHP program to set the variable \$x to 0 if the value of \$x is between 10 and 20. Note that when execution reaches the point where the code needs to be completed, the variable \$x already contains a value.

Figure 5.13. Example exercise for selection structures with Boolean operators in the condition.

$$((\text{GreaterThanOrEqual}(\text{val_x},10) \wedge (\text{LessThanOrEqual}(\text{val_x},20)) \rightarrow \text{HasValue}(\text{VARID1},0)))$$

Figure 5.14. Overall goal for example exercise for selection structures with Boolean operators in the condition.

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

Figure 5.15. Solution to example exercise

Let the initial value of the variable \$x be val_x. Then, the following facts are created as the initial state in the system.

HasName(VarId1, 'x')

HasValue(VarId1, val_x)

HasInitialValue(VarId1, val_x)

When the condition node for the if condition is reached, a *BooleanExpression* is created as in previous cases. However, in this case, the *BooleanExpression* is an *AndExpr* with a *GreaterEqualExpr* on the left hand side and a *LessEqualExpr* on the right hand side. Let the ids of the three expressions be ExprId1, ExprId2 and ExprId3 respectively. Then, the following facts are created in the system.

HasId(AndExpr(ExprId2, ExprId3), ExprId1)

ExprId2 represents a *GreaterEqualExpr* with a *VariableExpr* on the left hand side and a *LiteralExpr* on the right hand side. Let the ids of the *VariableExpr* and *LiteralExpr* be VarExprId2 and LitExprId2 respectively. Let the id of the created *Literal* be LitId2. Then, the following facts are created.

HasId(GreaterEqualExpr(VarExprId2, LitExprId2), ExprId2)

HasVariable(VarExprId2, VarId1)

HasLiteral(LitExprId2, LitId2)

HasLitValue(LitId2, 10)

Using the rules in *Figure 4.8*, the *ValueOf* the *VariableExpr* and *LiteralExpr* can be found as below.

$$\text{ValueOf}(\text{VarExprId2}, \text{val_x})$$

$$\text{ValueOf}(\text{LitExprId2}, 10)$$

Similarly, ExprId3 represents a *LessEqualExpr* with a *VariableExpr* on the left hand side and a *LiteralExpr* on the right hand side. Let the ids of the *VariableExpr* and the *LiteralExpr* be VarExprId3 and LitExprId3 respectively. Let the id of the created *Literal* be LitId3. Then, the following facts are created.

$$\text{HasId}(\text{LessEqualExpr}(\text{VarExprId3}, \text{LitExprId3}), \text{ExprId3})$$

$$\text{HasVariable}(\text{VarExprId3}, \text{VarId1})$$

$$\text{HasLiteral}(\text{LitExprId3}, \text{LitId3})$$

$$\text{HasLitValue}(\text{LitId3}, 20)$$

As before, the *ValueOf* the *VariableExpr* and *LiteralExpr* can be found as below.

$$\text{ValueOf}(\text{VarExprId3}, \text{val_x})$$

$$\text{ValueOf}(\text{LitExprId3}, 20)$$

When considering the case where the condition is satisfied, the *ValueOf* ExprId1 becomes True so the following fact is created.

$$\text{ValueOf}(\text{ExprId1}, \text{True})$$

Now, since ExprId1 represents an *AndExpr*, the first rule in *Figure 5.12* can be applied to create the following facts.

$$\text{ValueOf}(\text{ExprId2}, \text{True}) \wedge \text{ValueOf}(\text{ExprId2}, \text{True})$$

But since this is an **and** condition, it means that each of these facts exist independently of each other so they can be used to generate corresponding comparison facts using the rules in *Figure 5.4*.

$$\text{GreaterThanOrEqual}(\text{val_x}, 10) \wedge \text{LessThanOrEqual}(\text{val_x}, 20)$$

When this condition is true, the variable \$x is set to zero. This results in the following implication being created.

$$\begin{aligned}
& \text{GreaterThanOrEqual}(\text{val_x}, 10) \wedge \text{LessThanOrEqual}(\text{val_x}, 20) \\
& \qquad \qquad \qquad \rightarrow \text{HasValue}(\text{VarId1}, 0)
\end{aligned}$$

Comparing this final state with the overall goal shown in *Figure 5.14*, it can be seen that the overall goal is satisfied when VARID1=VarId1. Therefore, this program is identified as correct.

It should be noted that these rules cannot handle all expressions combined by using ‘&&’ and ‘||’. If an ‘&&’ expression is known to be true, it is easy to ascertain that all its sub expressions are also true. However, if an ‘&&’ expression is False, all that can be ascertained is that at least one of its sub expressions is False. If it is known that one of the sub expressions is true, it is possible to ascertain that the other is False. However, in all other cases, it is not possible to determine the value of the sub expressions. Similarly, if an ‘||’ expression is False, both its sub expressions are False. However, if it is true, it is not possible to determine the value of the sub expressions unless it is known that one of them is False. Therefore, this method of program analysis cannot generally handle situations where an ‘&&’ expression is false or an ‘||’ expression is true.

5.5 NESTED SELECTION STRUCTURES

Nested if-else structures are commonly used in programming to account for multiple conditions. These are handled in the same manner as normal if-else structures. The only significant aspect is the specification of the overall goal for these structures.

Consider the example exercise given in *Figure 5.16*. The expected program is program that contains a nested if-else structure as shown in Program a in *Table 5.2*.

Write a PHP program to display ‘A’ if \$marks is greater than 80. Otherwise, if \$marks is greater than 50, display ‘B’. Display ‘F’ in all other instances. Note that when execution reaches the point where the code has to be completed, the variable \$marks already contains a value.

Figure 5.16. Example exercise for nested selection structures.

Table 5.2

Alternative Solutions to Example Exercise for Nested Selection Structures

Program a	Program b	Program c
<pre> if(\$marks>80) { echo('A'); } else if (\$marks>50) { echo('B'); } else { echo('F'); } </pre>	<pre> if(\$marks<=50) { echo('F'); } else if (\$marks<=80) { echo('B'); } else { echo('A'); } </pre>	<pre> if(\$marks>80) { echo('A'); } if(\$marks<=80 && \$marks>50) { echo('B'); } if(\$marks<=50) { echo('F'); } </pre>

Figure 5.17 shows the overall goal for this program written in the same manner as explained in Section 5.1. In this case, the overall goal is given using a nesting structure, similar to the one in Program a of Table 5.2. Therefore, this program is identified as correct.

Initial State	: HasName(VARID1,'marks') \wedge HasValue(VARID1, val_m) \wedge HasInitialValue(VARID1, val_m)
Goal	: (GreaterThan(val_m,80) \rightarrow OnPage('A',i)) \wedge (LessThanOrEqualTo(val_m,80) \rightarrow (GreaterThan(val_m,50) \rightarrow OnPage('B',j)) \wedge (LessThanOrEqualTo(val_m,50) \rightarrow OnPage('F',k)))

Figure 5.17. Suggested initial state and overall goal for example exercise for nested selection structures.

5.5.1 Analysis of Program a

Consider how Program a in Table 5.2 is analysed. As before, the initial state results in the following facts since the variable \$marks already contains a value.

HasName(VarId1,'marks')

HasValue(VarId1, val_m)

HasInitialValue(VarId1, val_m)

The first conditional expression results in a *BooleanExpression* consisting of a *VariableExpr* and a *LiteralExpr* being created. Let the ids of these expressions be *ExprId1*, *VarExprId1* and *LitExprId1* respectively. Let the id of the created *Literal* be *LitId1*. Then, the following facts are created.

HasId(GreaterExpr(VarExprId1, LitExprId1), ExprId1)

HasVariable(VarExprId1, VarId1)

HasLiteral(LitExprId1, LitId1)

HasLitValue(LitId1, 80)

Finding the value of these expressions as explained in Section 4.4.1.1 results in the following facts being created.

ValueOf(VarExprId1, val_m)

ValueOf(LitExprId1, 80)

When considering the case when the condition is satisfied, the following fact is created.

ValueOf(ExprId1, True)

This fact results in the following fact being created using the rules in *Figure 5.4*.

GreaterThan(val_m, 80)

When this condition is satisfied, an ‘echo’ statement is executed. This results in the *Display* action being used to create the following fact.

OnPage('A', 1)

So the entire state for when the condition is satisfied can be written as below.

GreaterThan(val_m, 80) → OnPage('A', 1)

When the condition is not satisfied, i.e. in the else section, the following fact is created.

ValueOf(ExprId1, False)

Again using the rules in *Figure 5.4*, the following fact is then created in the system for the case where the condition is not satisfied.

$$\text{LessThanOrEqual}(\text{val_m}, 80)$$

At this point, another selection structure is encountered. This means that whatever facts are created after this are implied by the above fact. The condition for this second selection structure results in the following set of facts being created. Let the ids of the relevant *BooleanExpression*, *VarExpr* and *LitExpr* be *ExprId2*, *VarExprId2* and *LitExprId2* respectively. Let the id of the created *Literal* be *LitId2*.

$$\text{HasId}(\text{GreaterExpr}(\text{VarExprId2}, \text{LitExprId2}), \text{ExprId2})$$

$$\text{HasVariable}(\text{VarExprId2}, \text{VarId2})$$

$$\text{HasLiteral}(\text{LitExprId2}, \text{LitId2})$$

$$\text{HasLitValue}(\text{LitId2}, 50)$$

Finding the value of these expressions as explained in Section 4.4.1.1 results in the following facts being created.

$$\text{ValueOf}(\text{VarExprId2}, \text{val_m})$$

$$\text{ValueOf}(\text{LitExprId2}, 50)$$

When this second condition is satisfied the *ValueOf* of the expression is set to true and this results in a comparison fact being created using the rules in *Figure 5.4*. This means that the following facts are created.

$$\text{ValueOf}(\text{ExprId2}, \text{True})$$

$$\text{GreaterThan}(\text{val_m}, 50)$$

When the second condition is satisfied, a *Display* action is again used to create the following fact.

$$\text{OnPage}(\text{'B'}, 2)$$

So the result of the second condition being true can be written as below.

$$\text{GreaterThan}(\text{val_m}, 50) \rightarrow \text{OnPage}(\text{'B'}, 2)$$

When the second condition is not satisfied, the *Display* action is used to create the following facts.

$ValueOf(ExprId2, False)$

$LessThanOrEqual(val_m, 50)$

For this situation, the *Display* action results in the following fact.

$OnPage('F', 3)$

So the state when the second condition is not satisfied is as below.

$LessThanOrEqual(val_m, 50) \rightarrow OnPage('F', 3)$

Using the above description, it can be seen that the entire state for the second condition is as below.

$(GreaterThan(val_m, 50) \rightarrow OnPage('B', 2))$

$\wedge (LessThanOrEqual(val_m, 50) \rightarrow OnPage('F', 3))$

But as described earlier, the second condition is only satisfied if the first one is not so this entire state is an implication of when the first condition is not satisfied. Therefore, the final state of this program is as below.

$(GreaterThan(val_m, 80) \rightarrow OnPage('A', 1))$

$\wedge (LessThanOrEqual(val_m, 80) \rightarrow (GreaterThan(val_m, 50) \rightarrow OnPage('B', 2))$

$\wedge (LessThanOrEqual(val_m, 50) \rightarrow OnPage('F', 3)))$

When comparing this final state against the overall goal in *Figure 5.17*, it can be seen that it is satisfied when $i=1$, $j=2$ and $k=3$. Therefore, Program a is identified as a correct solution to the exercise.

5.5.2 Analysis of Program b

Next consider another correct solution to the exercise, Program b in Table 5.2. Using the same approach as above, it can be seen that the final state of this program is as shown in *Figure 5.18*. A detailed analysis of how this final state is obtained is given in Appendix E.

$$\begin{aligned}
& (\text{LessThanOrEqual}(\text{val_m},50) \rightarrow \text{OnPage}('F',k)) \\
& \wedge (\text{GreaterThan}(\text{val_m},50) \rightarrow \\
& \quad (\text{LessThanOrEqual}(\text{val_m},80) \rightarrow \text{OnPage}('B',j)) \\
& \quad \wedge (\text{GreaterThan}(\text{val_m},80) \rightarrow \text{OnPage}('A',i)))
\end{aligned}$$

Figure 5.18. Relevant facts for final state of Program b.

When comparing this final state against the overall goal given in *Figure 5.17*, it can be seen that it is in a different form and is therefore identified as incorrect. On careful observation, it can be seen that the final state is dependent on the nesting structure of the program. Different nesting structures can be used to obtain the same final result but specifying the goal in the manner given in *Figure 5.17* results in many of these programs being identified as incorrect.

5.5.3 Correct Overall Goal for Nested Selection Structures

Due to the above difficulty, it is necessary to specify the overall goal in a manner that makes it possible to identify all these alternatives as correct. The solution used in this case is to remove all nesting from the overall goal and express it using implications where the left hand side is a combination of conditional facts. The correct overall goal for this exercise is shown in *Figure 5.19*.

$$\begin{aligned}
& (\text{GreaterThan}(\text{val_m},80) \rightarrow \text{OnPage}('A',i)) \\
& \wedge (\text{LessThanOrEqual}(\text{val_m},80) \wedge \text{GreaterThan}(\text{val_m},50) \rightarrow \text{OnPage}('B',j)) \\
& (\text{LessThanOrEqual}(\text{val_m},50) \rightarrow \text{OnPage}('F',k))
\end{aligned}$$

Figure 5.19. Overall goal for example exercise for nested selection structures.

Within a nested node, all the conditional predicates along the path of the nesting are true. Therefore, the nesting guarantees that combined conditional facts on the left hand side of the overall goal are true. This means that whatever method of nesting is used, as long as the correct output is obtained, the program is identified as correct.

For example, consider the situation in Program a where the first condition is false. As apparent from the analysis process in Section 5.5.1, this results in the following fact being created.

$$\text{LessThanOrEqual}(\text{val_m},80)$$

This fact is now valid for all situations where the first condition is false. Next consider the case where the second condition is true. As above, this results in the following fact.

$$GreaterThan(val_m,50)$$

This means that both these facts are valid in the case where the first condition is false but the second condition is true and together they imply the result of actions performed during this situation. So the state corresponding to this situation can be written as below.

$$LessThanOrEqual(val_m,80) \wedge GreaterThan(val_m,50) \rightarrow OnPage('B',1)$$

Similarly, the state when both the conditions are false is as below.

$$LessThanOrEqual(val_m,80) \wedge LessThanOrEqual(val_m,50) \rightarrow OnPage('F',2)$$

However, considering the laws of Mathematics, the $LessThanOrEqual(val_m,80)$ has no effect here since it is always true when $LessThanOrEqual(val_m,50)$ is true. Therefore, the last statement can be modified as below.

$$LessThanOrEqual(val_m,50) \rightarrow OnPage('F',2)$$

So the final state of Program a can now be written as below.

$$(GreaterThan(val_m,80) \rightarrow OnPage('A',1))$$

$$\wedge (LessThanOrEqual(val_m,80) \wedge GreaterThan(val_m,50) \rightarrow OnPage('B',1))$$

$$\wedge (LessThanOrEqual(val_m,50) \rightarrow OnPage('F',2))$$

When comparing against the overall goal in *Figure 5.19*, it can be seen that this is satisfied when $i=1$, $j=2$ and $k=3$ so the program is again identified as correct.

Using a similar analysis, it can be seen that Program b in Table 5.2 results in the following final state.

$$(LessThanOrEqual(val_m,50) \rightarrow OnPage('F',1))$$

$$\wedge (GreaterThan(val_m,50) \wedge LessThanOrEqual(val_m,80) \rightarrow OnPage('B',1))$$

$$\wedge (GreaterThan(val_m,80) \rightarrow OnPage('A',2))$$

Again comparing with *Figure 5.19* it can be seen that the overall goal is satisfied although the final facts are given in a different order. A detailed analysis of

how Program c in Table 5.2 is analysed to obtain the same final state is given in Appendix E. It can be seen that this method of specifying the overall goal is suitable to handle all possible nesting combinations in students' programs.

5.6 SWITCH STATEMENTS

Switch statements are commonly used to handle situations where the processing differs based on the value of a variable. This is similar to nested if-else structures where the conditional expression is testing for equality. Therefore, the same method as for nested if-else structures is used here.

Consider the example exercise given in *Figure 5.20*. Table 5.3 shows two alternative solutions to this exercise. Program a uses a nested if-else structure while Program b uses a switch statement.

Write a PHP program to display 'Excellent' if the grade is 'A'. Otherwise, if the grade is 'B' display 'Good'. In all other instances display 'Try Harder'. Note that when execution reaches the point where the code has to be completed, the variable \$grade already contains a value.

Figure 5.20. Example exercise for switch statements.

Table 5.3

Alternative Programs for Example Exercise

Program a	Program b
<pre> if(\$grade=='A') { echo('Excellent'); } else if (\$grade=='B') { echo('Good'); } else { echo('Try Harder'); } </pre>	<pre> switch(\$grade) { case 'A': echo('Excellent'); break; case 'B': echo('Good'); break; default: echo('Try Harder'); } </pre>

As described in Section 5.5.3, the overall goal for this exercise can be written as shown in *Figure 5.21*. It has been assumed that the initial value of \$grade is val_g. However, in this case, the combined conditions on the left hand side of some of the sub-goals consists of combinations of *NotEqualTo* and *EqualTo* predicates with the same first argument, joined using the *And* operator. In practice, if a value is equal to a certain value, it is obviously not equal to another value. Therefore, the *NotEqualTo* predicate can be left out of the overall goal specification in such cases. *Figure 5.22* shows the overall goal, simplified in this manner.

```
(EqualTo(val_g,'A') → OnPage('Excellent',i))
∧ (NotEqualTo(val_g,'A') ∧ EqualTo(val_g,'B') → OnPage('Good',j))
∧ (NotEqualTo(val_g,'A') ∧ NotEqualTo(val_g,'B') → OnPage('Try Harder',k))
```

Figure 5.21. Suggested overall goal for example exercise.

```
(EqualTo(val_g,'A') → OnPage('Excellent',i))
∧ (EqualTo(val_g,'B') → OnPage('Good',j))
∧ (NotEqualTo(val_g,'A') ∧ NotEqualTo(val_g,'B') → OnPage('Try Harder',k))
```

Figure 5.22. Simplified overall goal for example exercise.

Although the above section explains how the overall goal is specified for switch statements, the AST for switch statements causes some inconvenience. Since the case statements only contain the value of the variable that is considered and not the equality check itself, it is necessary to manually change the AST to include equality expressions. This is done during the AST walking process. Each time a case expression is encountered, it is combined with the variable of the switch statement to create a new AST that is then used to create an equality expression. The entire original AST and the modified version of the first case expression is shown in Table 5.4.

```

switch($marks)
{
    case($marks>80): echo('A');
                    break;
    case($marks>50): echo('B');
                    break;
    default: echo('F');
}

```

Figure 5.23. Example program for comparison operators within switch statements.

Table 5.4
Modified AST for Switch Statements

Original AST	Modified ASTs
(PHP (switch (\$ grade) (case 'A' (echo 'Excellent') break) (case 'B' (echo 'Good') break) (default (echo 'Try Harder')))))	(== (\$ grade) 'A') (== (\$ grade) 'B')

It is possible for switch statements to contain default cases. This means that under this node, none of the equalities tested are true. This is modelled by setting all the equality expressions that were encountered during the switch to false. Therefore, when analysing Program b in Table 5.3, the default case results in the facts *NotEqualTo('val_g','A')* and *NotEqualTo('val_g','B')* being created.

A detailed analysis of Program b in Table 5.3 is given in Appendix E. When the programs are modelled in this manner, both Program a and Program b in Table 5.3 are accepted as correct solutions to the programming exercise in Figure 5.20.

5.6.1 Special Considerations

It should be noted that PHP is somewhat different to many other programming languages in that it allows comparison operators within switch statements. Figure 5.23 is an example of such a switch statement to solve the exercise given in Figure 5.16. This type of switch statement results in an AST that is somewhat different

from the normal case. Since this is a more advanced PHP topic, it has been eliminated from the scope of statements handled in this thesis.

Another issue that arises when analysing switch statements is that, unlike in nested if-else structures, program execution can flow through from one case to another if no 'break;' keyword is used. *Figure 5.24* shows an example of such a program. In this case, the text 'Pass' is displayed in both the first two case statements, i.e. if marks are greater than 80 or greater than 50. This is handled when walking the AST. The same set of facts is created against each case that falls through to the actual execution statements. For example, an *OnPage('Pass',i)* fact is created against the conditions where the marks are greater than 80 or 50.

```
switch($marks)
{
    case($marks>80):
    case($marks>50): echo('Pass');
                    break;
    default: echo('Fail');
}
```

Figure 5.24. Example switch statement with execution falling through to next case.

5.7 HANDLING UNNECESSARY STATEMENTS IN SELECTION STRUCTURES

The analysis process described above is capable of identifying alternative solutions to a given exercise using selection structures. However, a common mistake made by many students is to include additional program statements that do not contribute to achieving the overall goal. As described in Section 4.5.5, this is handled by maintaining a set of statuses.

In case of selection statements, several new statuses are created in order to identify the flow of execution. A new status is created immediately, when a selection structure is encountered. The *BooleanExpression* corresponding to the condition is created within the status. Two separate statuses are created for the 'if' and 'else' parts of an if-else structure. These statuses are linked to the status of the

main selection statement created above. The flow of statuses for Program a in Table 5.1 is shown in *Figure 5.25*.

The ‘if’ part of the program can contain many statement and these can result in the creation of one or more new statuses. Any statuses created in this manner are linked to the main status corresponding to the ‘if’ part. Similarly, any new statuses created during the ‘else’ part of the program are linked to the status corresponding to ‘else’. This process ensures that relevant links are maintained between statuses created using nested structures.

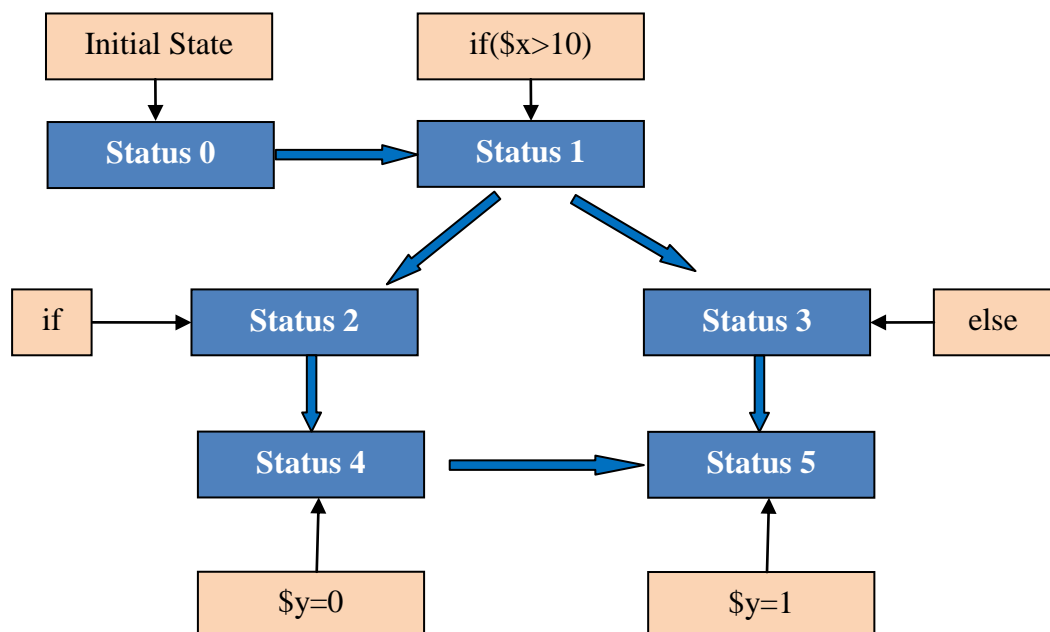


Figure 5.25. Status flow for example selection program.

In the case of selection structures, the status flow described above creates a problem when identifying the status where the overall goal is satisfied. In the above example, Status 5 is the status where the entire overall goal is satisfied. However, Status 4 also contributes to satisfying the overall goal of the system. The divergent paths of the structure do not depict the fact that Status 4 contributes to the goal. Therefore, once the final status where the overall goal is satisfied is identified, any previous statuses in the structure that contribute to satisfying the goal are linked to

this status. In this case, a link is created between Status 4 and Status 5. This ensures that there is a path from all statuses contributing to the overall goal to the goal status.

Statuses for switch statements are handled in a similar manner. The only difference is that a new status is created for each 'case' and these are linked to the main status created at the beginning of the selection statement.

This flow of statuses is then used to identify any statuses that do not correspond to achieving the overall goal. Such extra statuses are then indicated as unnecessary statements in the student's program.

5.8 CHAPTER SUMMARY

This chapter explored how the knowledge base of the PHP ITS deals with selection structures which are used extensively during programming. It discussed how selection structures can be used in a multitude of ways to achieve the same result and how the KB identified all these as correct. It looked at nested selection structures as well as switch statements that are used to handle a multitude of conditions.

The next chapter looks at some more advanced structures used in PHP, namely arrays, functions and forms. The process of depicting these structures and their analysis is described in detail.

Chapter 6: Arrays, Functions and Forms

The previous chapter looked at how commonly used selection structures are handled in the system. This chapter goes on to investigate more advanced topics in PHP. Section 6.1 looks at how arrays are modelled using predicates and how they are analysed. Section 6.2 describes how both predefined and user defined PHP functions are handled in the KB. Section 6.3 discusses how forms are modelled and how the KB handled passing information from one web page to another. Finally Section 6.4 summarises the chapter.

6.1 ARRAYS

In programming, arrays are used to handle collections of similar objects. They are basically a systematic arrangement of objects. Each array element has the same functionality as a variable. In other words, it can be used anywhere a variable is used, on the left hand side of assignment statements as well as in expressions. Therefore, array elements are modelled as a subtype of a *Variable* as shown in *Figure 6.1*. An array element is called an *ArrayVariable* to easily identify it as a *Variable*. An *ArrayVariable* is actually a relationship between an array, and a key. This relationship is shown by the *HasElement* predicate which is reified into the *ArrayVariable* object type.

PHP arrays are somewhat different from arrays found in most other programming languages in that both indexed and associative array referencing is permitted within the same array. This means that the key can be either an integer or a string. Therefore, the key is divided into two further subtypes, *Index* for indexed access and *KeyString* for associative access. When accessing array elements in a PHP program, it is not necessary to explicitly specify the key. It is possible to use an expression that returns a value in place of the key. This expression can take the form of any expression such as a *LiteralExpr*, *VariableExpr* or *CalculateExpression*. The association between this expression and the actual value of the key is maintained through the *HasKeyExpression* predicate. When indexed access is used to access an array element, the expression in the *HasKeyExpression* predicate refers to an expression specifying the *Index*. Similarly, when associative access is used, the

expression in the *HasKeyExpression* predicate refers to an expression specifying the *KeyString*. A peculiarity in PHP is that, sometimes, it is possible to access an associative array using both indexed and associative access. This is handled by creating two separate facts, one for each type of access, in the system.

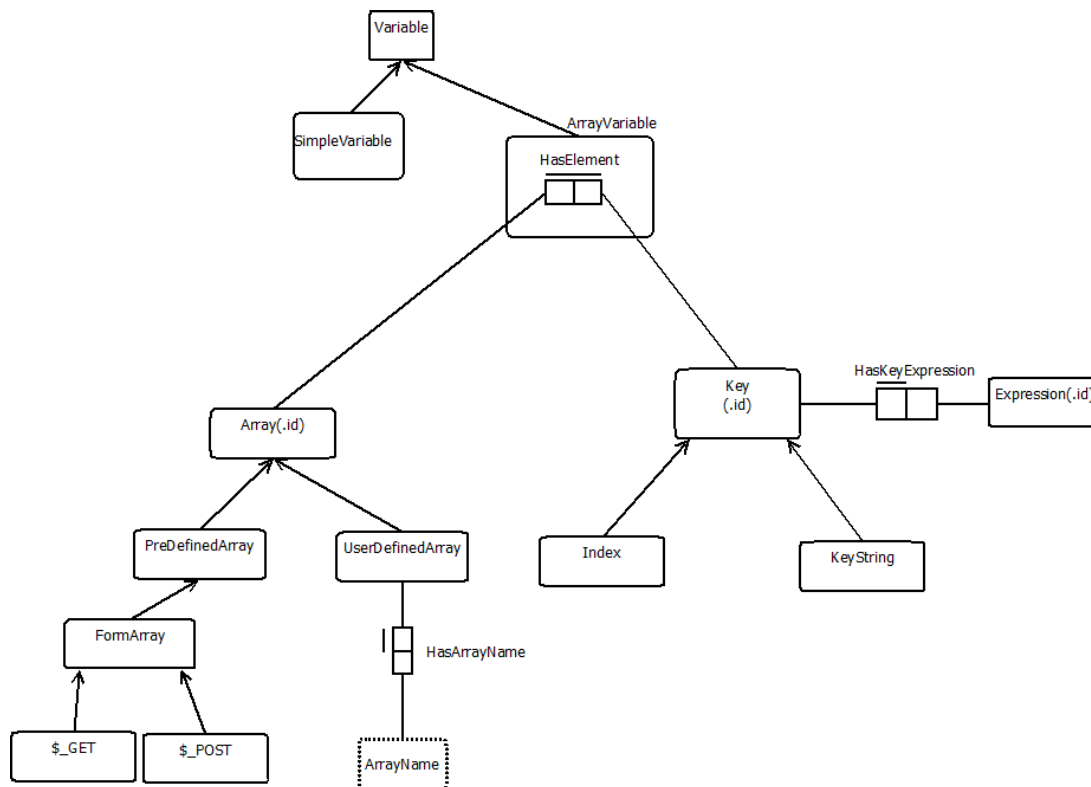


Figure 6.1. ORM diagram for arrays.

Another interesting feature of PHP is that it contains some predefined arrays in addition to user defined arrays. Therefore, the *Array* is divided into two subtypes, *PreDefinedArray* and *UserDefinedArray*. There are many predefined arrays such as *\$_SERVER*, *\$_ENV*, *\$_GLOBALS* and many more. Most of these arrays are rarely used in basic PHP programming and have therefore not been modelled in the knowledge base. However, two types of predefined arrays, *\$_POST* and *\$_GET* are associated with HTML form processing in PHP. These are defined under a further subtype of *PreDefinedArray* know as *FormArray*. The *FormArray* is divided into two further subtypes *\$_GET* and *\$_POST*. In principle, it seems likely that other types of *PreDefinedArrays* can be modelled in a similar manner. *UserDefinedArrays* have a name that is given by the *HasArrayName* predicate.

In order to understand the relationship between these predicates, consider a case where a PHP program contains a reference to `$myarray[5]`. Since `$myarray` is not a *PreDefinedArray*, a *UserDefinedArray* object is created. Let the id of this array be `ArrId1`. Then, the following fact is created to specify the name of the array.

$$\text{HasArrayName}(\text{ArrId1}, \text{'myarray'})$$

The key in this case is an index – the value 5 - so an *Index* object is created. Let the id of the created *Index* be `KeyId1`. The association between the *Array* and the *Key* is then given by the following fact.

$$\text{HasElement}(\text{ArrId1}, \text{KeyId1})$$

But as described earlier, this is reified into a *Variable*. Let the id of the relevant *Variable* be `VarId1`. This results in the following reified fact.

$$\text{HasVariableId}(\text{HasElement}(\text{ArrId1}, \text{KeyId1}), \text{VarId1})$$

So the value of the corresponding variable can now be accessed using the *HasValue*(`VarId1`, `n`) fact, where `n` is the value assigned to the array element.

As described earlier, each key is associated with an expression. In this case, the expression is a *LiteralExpr*. Let the id of the created *LiteralExpr* be `LitExprId1`. Then, the following fact is created to show the relationship between the key and the expression.

$$\text{HasKeyExpression}(\text{KeyId1}, \text{LitExprId1})$$

As described in Section 4.4.1.1, each *LiteralExpr* is associated with a *Literal*. Let the id of the created *Literal* be `LitId1`. Then, the following facts are created.

$$\text{HasLiteral}(\text{LitExprId1}, \text{LitId1})$$
$$\text{HasLitValue}(\text{LitId1}, 5)$$

Using the rules in *Figure 4.8*, the following fact is created for the *LiteralExpr*.

$$\text{ValueOf}(\text{LitExprId1}, 5)$$

Therefore, it can be seen that a single array element results in a large number of facts in the system.

Consider the example PHP exercise with an array given in *Figure 6.2*. Using the above predicates, the overall goal for this exercise is as shown in *Figure 6.3*. It specifies that the key of the *ArrayVariable* should have a value of 0 while the *Variable* itself should have a value of 1.

Write a PHP program to create an array named \$myarray. Assign the value 1 to the 0th element of the array.

Figure 6.2. Example array exercise.

Goal : HasVariableId(HasElement(ARRID1,KEYID1),VARID1)
 \wedge HasKeyExpression(KEYID1,EXPRID1)
 \wedge ValueOf(EXPRID1,0)
 \wedge HasValue(VARID1,1)
 Constraints : HasArrayName(ARRID1,'myarray')

Figure 6.3. Overall goal of example array exercise.

6.1.1 Assigning to Array Variables

As described above, *ArrayVariables* are similar to other *Variables* in most operations within the knowledge base. However, when an assignment is done to an *ArrayVariable*, there are several differences from a *SimpleVariable*. When assigning to a *SimpleVariable*, it may already exist or not. When assigning to an *ArrayVariable*, there are three situations that need to be considered. The first is that the *ArrayVariable* already exists. The second is that the *Array* exists but the corresponding *ArrayVariable*, i.e one with the relevant key, does not exist. The third is that neither the array nor the key exist. In order to allow for these differences, a separate action is used when assigning to *ArrayVariables*. This action is shown in *Figure 6.4*.

Action(AssignArrayVariable(x,y,expressionId),
PRECOND: \exists value ValueOf(expressionId,value)
EFFECT: when \exists varId,arrayId,keyId,exprId
 (HasVariableId(HasElement(arrayId,keyId),varId)
 \wedge HasKeyExpression(keyId,exprId)
 \wedge ValueOf(exprId,y)
 \wedge HasArrayName(arrayId,'x')):
 HasValue(varId,_) \leftarrow HasValue(varId,value)
 \wedge when $\neg \exists$ varId,arrayId,keyId,exprId
 (HasVariableId(HasElement(arrayId,y),varId)
 \wedge HasKeyExpression(keyId,exprId)
 \wedge ValueOf(exprId,y)
 \wedge HasArrayName(arrayId,'x')):
 Generate(newArrId)
 Generate(newVarId)
 Generate(newKeyId)
 Generate(newExprId)
 HasArrayName(newArrId,'x')
 HasVariableId(HasElement(newArrId,newKeyId),newVarId)
 HasKeyExpression(newKeyId,newExprId)
 ValueOf(newExprId,y)
 HasValue(newVarId,value)
 HasInitialValue(newVarId,value)
 \wedge when $\neg \exists$ varId, keyId,exprId \exists arrayId
 (HasVariableId(HasElement(arrayId,keyId),varId)
 \wedge HasKeyExpression(keyId,exprId)
 \wedge ValueOf(exprId,y)
 \wedge HasArrayName(arrayId,'x')):
 Generate(newVarId)
 Generate(newKeyId)
 Generate(newExprId)
 HasVariableId(HasElement(arrayId, newKeyId),newVarId)
 HasKeyExpression(newKeyId,newExprId)
 ValueOf(newExprId,y)
 HasValue(newVarId,value)
 HasInitialValue(newVarId,value))

Figure 6.4. AssignArrayVariable action.

This action is very similar to the *Assign* action discussed in *Figure 4.10*. One main difference is that it takes in three arguments, the array name, the value of the key and the expression id of the right hand side expression, instead of the two arguments of the standard *Assign* action. It then uses these arguments to check if an *ArrayVariable* already exists for the given array and key. If so, it updates this variable. The next main difference from the standard *Assign* action is that this action contains two alternatives in the case where a corresponding *ArrayVariable* is not found. In the first case, neither the array nor the key given in the arguments exist. In such a case, the action creates a new *Array*, *Key*, key expression and *Variable* before assigning the value of the right hand side expression. In the second case, the array exists but no corresponding key exists. In this case, a new *Key* and key expression are created before assigning the value to the *ArrayVariable*.

ArrayVariables as well as a *SimpleVariable* can make up the left hand side of the combined assignment operators discussed in Section 4.4.3. Therefore, actions corresponding to *Figure 4.12* are defined for *ArrayVariables* as well. *Figure 6.5* shows the subtype version of the *AssignAddArrayVariable* action. The only difference from the *AssignArrayVariable* action is that the value that is assigned to the variable when it already exists is the value of the expression plus the original value of the variable. Similar actions are written for all the other combined assignment operators as well (Appendix C).

$\text{AssignAddArrayVariable}(x,y,\text{exprId}) \subset \text{AssignArrayVariable}(x,y,\text{exprId})$

Action(*AssignAddArrayVariable*(*x,y,exprId*))
 PRECOND: $\exists \text{value ValueOf}(\text{exprId},\text{value})$
 EFFECT: when $\exists \text{arrayId,varId,keyId,exprId}$
 $(\text{HasVariableId}(\text{HasElement}(\text{arrayId},\text{keyId}),\text{varId})$
 $\wedge \text{HasKeyExpression}(\text{keyId},\text{exprId})$
 $\wedge \text{ValueOf}(\text{exprId},y)$
 $\wedge \text{HasArrayName}(\text{arrayId},'x')$
 $\wedge \text{HasValue}(\text{varId},\text{value2}) \wedge \text{Add}(\text{value2},\text{value},\text{value1})) :$
 $\text{HasValue}(\text{varId},_) \leftarrow \text{HasValue}(\text{varId},\text{value1})$

Figure 6.5. Subtype version of *AssignAddArrayVariable* action.

6.1.2 Array Construct

PHP offers a special construct, ‘array’ to assign values to an entire array in one go. This construct is placed on the right hand side of an assignment operator, where the left hand side contains the name of the array. The ‘array’ construct can take two forms. The first form contains a list of values separated by commas creating an indexed array starting from index 0. The second form contains a list of key value pairs separated by commas creating an associative array. Examples for both the forms of use of the ‘array’ construct are given in *Figure 6.6*.

```
$a=array(25,32)
$b=array('Emily'=>25,'Bob'=>32')
```

Figure 6.6. Two forms of the array construct.

The ‘array’ construct is handled by manipulating the AST. When an ‘=’ node is encountered while walking the AST, the right hand node is inspected to see whether it is an ‘array’ node. If so, the assignment statement in AST form is converted into several separate assignment statements in AST form with corresponding array elements and values as child nodes. Table 6.1 shows the original AST and the converted AST for the second example shown in *Figure 6.6*. Here, the ‘=’ node in the original AST is converted into two separate ‘=’ nodes in the modified AST. The left hand side of each of the converted nodes contain the array as well as the key while the right hand side contains the relevant value. Now, the modified AST is handled as a normal assignment to two separate *ArrayVariables*.

Table 6.1

AST Conversion for Array Construct

<i>Original AST</i>	<i>Modified AST</i>
(PHP (= (\$ b) (array (=> 'Emily' 25) (=> 'Bob' 32))))	(PHP (= ([(\$ b) 'Emily') 25) (= ([(\$ a) 'Bob') 32)))

6.2 FUNCTIONS

PHP functions can be divided into two main groups: pre-defined and user-defined functions. Therefore, the *Function* object type is divided into two main subtypes, *PreDefinedFunction* and *UserDefinedFunction* as shown in *Figure 6.7*. In writing PHP programs, it is always possible to refactor a section of code into a

function and call that function from other code. The scope of this thesis does not include the analysis of such arbitrarily defined functions. *UserDefinedFunctions* are only analysed if they are specified in the exercise specification. Also, such *UserDefinedFunctions* are only accepted as correct if they carry out the exact tasks given in the specification. E.g. programs are considered incorrect if parts of the main program, as given in the specification, are transferred into a function. This is a shortcoming in the program analysis process used here.

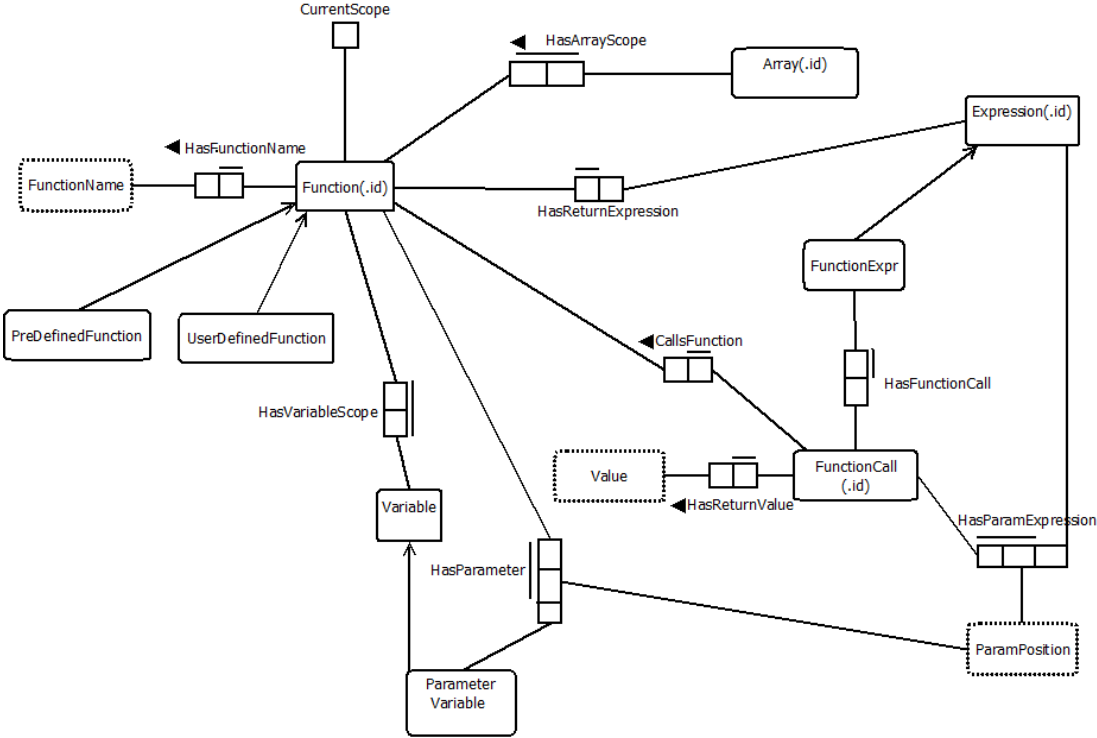


Figure 6.7. ORM diagram for functions.

6.2.1 Predicates for Handling Functions

When modelling functions, it is necessary to consider two distinct aspects. Consider the PHP program with a function given in *Figure 6.8*. The first block of code containing the ‘function’ keyword is the function definition. This block defines the name of the function, its parameters and what it actually does. The next block of code is outside the function but the last line is a function call. Separate sets of predicates are defined in the KB to handle these two situations: function definitions and function calls.

```

function findTotal($num1,$num2)
{
    $tot=$num1+$num2;
    return($tot);
}

$x=5;
$y=10;
$z=findTotal($x,$y);

```

Figure 6.8. Example program for function use.

6.2.1.1 Function Definition

A function definition results in the creation of a *Function* object. As in other types of objects, each *Function* is assigned a unique id. A *Function* always has a name which is given by the *HasFunctionName* predicate. Very often, functions have parameters. These are defined using the *HasParameter* predicate. This predicate takes three arguments: the function id, the parameter position given by *ParamPosition*, and the *ParameterVariable*. The *ParameterVariable* behaves like other variables once it is defined in the function signature. However, its value is taken from any values passed into the function during a function call. Therefore, it is a *Variable* with some special characteristics. Due to this reason, *ParameterVariables* are modelled as a third subtype of *Variables*. Some functions also return a value. This value is an expression. This is modelled as a return expression given by the *HasReturnExpression* predicate.

In order to illustrate this, consider the function definition in *Figure 6.8* again. Let the id of the created *Function* be *FuncId1*. This function has two *ParameterVariables*. Let their ids be *ParamVarId1* and *ParamVarId1* respectively. This function also has a ‘return’ keyword so it returns a value of an expression. Let the id of the return expression be *RetExprId1*. Then, the following facts are created.

HasFunctionName(FuncId1,'findTotal')

HasParameter(FuncId1,1,ParamVarId1)

HasName(ParamVarId1,'num1')

HasParameter(FuncId1,2,ParamVarId2)

HasName(ParamVarId2,'num2')

HasReturnExpression(FuncId1,RetExprId1)

6.2.1.2 *Function Call*

Each function call is represented as a collection of facts. The call itself is modelled as a *FunctionCall* object instance with a unique id. Each *FunctionCall* calls a function that has already been defined, either by the program itself (*UserDefinedFunction*) or by PHP (*PreDefinedFunction*). This relationship is established through the *CallsFunction* predicate. When calling a function, it is necessary to pass values to the *ParameterVariables*. These values could be *Literals*, other variables or even other expressions. In order to cover all these types, these are modelled as expressions. The relationship between the *FunctionCall*, the position of the passed expression and the expression itself is given using the *HasParamExpression* predicate. The *ParamPosition* used here is the same as that used in the *HasParameter* predicate in the function definition. It is possible that functions also return some value. The value returned by the function for a particular function call is modelled using the *HasReturnValue* predicate.

In order to illustrate this, again consider the program in *Figure 6.8*. In this case, a call is made to the function defined in Section 6.2.1.1. Let the id of the created *FunctionCall* be *FuncCallId1*. Then, the relationship to this function is established using the following fact.

CallsFunction(FuncCallId1,FuncId1)

The next aspect that must be captured are the parameters that are passed. Two parameters are passed in the example in *Figure 6.8*. Both are simply the values of variables and are therefore modelled as *VariableExprs*. Let the ids of these expressions be *VarExprId1* and *VarExprId2* respectively. Also, let the ids of the *Variables* corresponding to *\$x* and *\$y* be *VarId1* and *VarId2* respectively. Then, the following facts are created.

HasParamExpression(FuncCallId1,1,VarExprId1)

HasVariable(VarExprId1,VarId1)

HasParamExpression(FuncCallId1,2,VarExprId2)

HasVariable(VarExprId2,VarId2)

Assume that, after executing of the function based on the given parameters, it returns a value of Value1. Then, the following fact is created.

HasReturnValue(FuncCallId1,Value1)

Function calls can be made in two ways: as stand-alone calls to perform some processing, or as parts of expressions that return a value. The above predicates are sufficient to model stand-alone function calls. However, when the return value of a function is used for some purpose, the function call behaves as any other type of expression. For example, in the example program in *Figure 6.8*, the function call forms the right hand side of an assignment statement. As described in Section 4.4.3, the right hand side of assignment statements are always modelled as Expressions. Therefore, a fourth subtype of *Expression* known as *FunctionExpr* is modelled as explained in Section 4.4.1.1. The relationship between the function expression and the actual *FunctionCall* is given through the *HasFunctionCall* predicate. In the above case let the value of the created expression be FuncExprId1. This results in the following fact being created.

HasFunctionCall(FuncExprId1,FuncCallId1)

All the facts described in this section are used together to analyse programs that use PHP functions.

6.2.2 The Scope of Variables

When dealing with PHP programs that do not contain any functions, any *Variable* that is defined once is accessible from anywhere within the program. However, when functions are included in a program, it is necessary to consider the scope of variables. The scope indicates which area of the program each variable is accessible from. Several predicates and rules are used in order to model the scope of variables.

CurrentScope is a predicate with a single argument. This argument specifies which function is in scope at the current time during program analysis. There is no function within the main PHP program. Therefore, the argument of *CurrentScope* during the analysis of the main program is taken to be Null. Whenever a function definition is encountered, the argument of *CurrentScope* becomes the id of this

function. At any given state during the fact creation process, there is only one *CurrentScope* fact. After all the AST nodes for the function are walked through, the argument of *CurrentScope* returns to Null.

The scope of each *Variable* is established using the *HasVariableScope* predicate. This predicate forms a relationship between the ids of the variable and the function. Again, for *Variables* that are in scope within the main program, the id of the function is replaced by Null. For example, again consider the example program in *Figure 6.8*. The moment the function definition is encountered and the *Function* object is created, the *CurrentScope* is set to the id of the function as below.

$$\text{CurrentScope}(\text{FuncId1})$$

The *ParameterVariables* can both be accessed only within the function. Therefore, each time a *ParameterVariable* is created, its scope is set to the *CurrentScope*. This results in the following facts.

$$\text{HasVariableScope}(\text{ParamVarId1}, \text{FuncId1})$$
$$\text{HasVariableScope}(\text{ParamVarId2}, \text{FuncId1})$$

Any other variables that appear inside the function are also set to this scope as described in Section 6.2.2.3 below. Once the function definition is complete and the main program is reached, the *CurrentScope* is set to Null as below.

$$\text{CurrentScope}(\text{Null})$$

Any variables encountered within the main program are set to the Null scope as described in Section 6.2.2.3 below.

6.2.2.1 Scope of *ArrayVariables*

This situation is somewhat modified when considering *ArrayVariables*. All *ArrayVariables* belonging to a single array have the same scope and the scope is determined by where the *Array* itself is defined. The scope of the array is specified using the *HasArrayScope* predicate. The scope that is assigned for the *Array* applies to all *ArrayVariables* that are associated with the *Array*. This relationship is established by the first rule given in *Figure 6.9*.

For example, consider a case where an *Array* with id *ArrId1* is defined in the main program. Then, its scope is defined by the fact below.

HasArrayScope(ArrId1,Null)

Assume that this array contains two elements. Let the ids of the *Keys* corresponding to the two elements be *KeyId1* and *KeyId2* respectively. Let the ids of the corresponding *Variables* be *VarId1* and *VarId2* respectively. Then, the following facts are created.

HasVariableId(HasElement(ArrId1,KeyId1),VarId1)

HasVariableId(HasElement(ArrId1,KeyId2),VarId2)

Then, the first rule in *Figure 6.9* results in the scope of the two *ArrayVariables* being set, resulting in the following facts.

HasVariableScope(VarId1,Null)

HasVariableScope(VarId2,Null)

<p><i>HasVariableScope(varId1,funcId1)</i> ← <i>HasArrayScope(arrId1,funcId1) ∧</i> <i>HasVariableId(HasElement(arrId1,keyId1),varId1)</i></p> <p><i>HasVariableScope(varId1,funcId1)</i> ← <i>HasName(varId1,x) ∧ HasVariableScope(varId1,Null) ∧</i> <i>CurrentScope(funcId1) ∧ Global(x,funcId1)</i></p> <p><i>HasArrayScope(arrId1,funcId1)</i> ← <i>HasArrayName(arrId1,x) ∧ HasArrayScope(arrId1,Null) ∧</i> <i>CurrentScope(funcId1) ∧ Global(x,funcId1)</i></p>

Figure 6.9. Rules for handling variable scope.

6.2.2.2 Super-global and Global Variables

Some predefined *Variables* in PHP are super-globals. This means that these *Variables* are always in scope, no matter where in the program they are used. Although many such super-globals are beyond the scope of this thesis, a few are necessary for basis PHP programming. The main super-globals used in the scope of this thesis are actually super-global arrays, namely the `$_POST` and `$_GET` arrays described in Section 6.1. However, it is theoretically possible to model other super-global arrays in a similar manner.

PHP also uses global variables. These are variables that are defined in the main program but can be accessed from within a function. Before a global variable can be accessed within a PHP function, it must be declared to be global. An example of such a program is given in *Figure 6.10*. In this program, the variable \$y is defined within the main program. The 'global' keyword before the variable \$y in the function specifies that it is a global variable. This means that any reference to \$y within the function is a reference to the same variable as is defined in the main program.

```
function findTotal($num1)
{
    global $y;
    $tot=$num1+$y;
    return($tot);
}

$x=5;
$y=10;
$z=findTotal($x);
```

Figure 6.10. A PHP program with a global variable.

Both super-global and global variables and arrays are handled using a special predicate, *Global*. This takes two arguments, the name of the global variable and the id of the function where it is declared global. When this fact is present, the second and third rules in *Figure 6.9* are used to specify that these *Variables* are also in scope in the function that is currently being analysed.

Considering the example above, let the id of the *Variable* \$y be VarId2. Since it is defined within the main program, the following facts are created.

HasName(VarId2,'y')

HasVariableScope(VarId2,Null)

When the AST is being analysed, the function definition changes the *CurrentScope* to the scope of the function. Let the id of the *Function* be FuncId1. Then, the following fact is created.

CurrentScope(FuncId1)

Now, when the 'global' node is encountered in the AST, the following fact is created.

Global('y',FuncId1)

Now, the scope of the variable is also set to the function scope using the second rule in *Figure 6.9*. This results in the following fact.

HasVariableScope(VarId2,FuncId1)

A similar approach is used for super-global arrays and the third rule in *Figure 6.9* is used. A more detailed analysis for such a case is presented in Appendix F.

6.2.2.3 Extending Previous Rules and Actions

All rules and actions until this point did not consider the scope of variables. It was assumed that, once a variable was created, it was in scope and could therefore be accessed from anywhere. However, with the introduction of functions, it becomes necessary to consider the scope when dealing with variables. This means that this aspect needs to be incorporated into some of the rules and actions discussed in Chapter 4.

The rule to calculate the value of a *VariableExpr* in Section 4.4.1.1 ignored the fact that the *Variable* may not be in scope. The value of *VariableExpr* and pre and post fix expressions can only be found if the variable concerned is in scope. Therefore, these rules are extended to include this fact as shown in *Figure 6.11*.

The scope of variables needs to be taken into account in the *Assign* action as well. The existing *Variable* can only be updated if a *Variable* of the given name exists in the current scope. This is reflected in the modified version of the *Assign* action shown in *Figure 6.12*. The *AssignArrayVariable* action is also modified in a similar manner but in this case, the *HasArrayScope* predicate is used as shown in *Figure 6.13*. The *AssignAdd* action and the *AssignAddArrayVariable* actions are also modified by incorporating the scope. These and other combined assignment actions are given in Appendix C.

	$\text{ValueOf}(\text{variableExprId},v) \leftarrow \text{HasVariable}(\text{variableExprId},\text{variableId})$ $\quad \wedge \text{HasValue}(\text{variableId},v)$ $\quad \wedge \text{CurrentScope}(\text{funcId})$ $\quad \wedge \text{HasVariableScope}(\text{variableId},\text{funcId})$
C	$\text{ValueOf}(\text{preExprId},v) \leftarrow \text{HasPrePostVariable}(\text{preExprId},\text{varId})$ $\quad \wedge \text{HasValue}(\text{varId},v)$ $\quad \wedge \text{CurrentScope}(\text{funcId1})$

Action(Assign(x,expressionId),

PRECOND: \exists value ValueOf(expressionId,value) \wedge CurrentScope(funcId)

EFFECT: When \exists variableId (HasName(variableId,'x')

\wedge CurrentScope(funcId) \wedge HasVariableScope(variableId,funcId)):

HasValue(variableId,_) \leftarrow HasValue(variableId,value)

\wedge when $\neg \exists$ variableId(HasName(variableId,'x')

\wedge CurrentScope(funcId) \wedge HasVariableScope(variableId,funcId)):

Generate(newVariableId)

HasName(newVariableId,x)

HasVariableScope(newVariableId,funcId)

HasValue(newVariableId,value)

HasInitialValue(newVariableId,value)

Figure 6.12. Modified Assign action to include variable scope.

Action(AssignArrayVariable(x,y,expressionId),

PRECOND: \exists value ValueOf(expressionId,value)

EFFECT: when \exists varId,arrayed,keyId,exprId

(HasVariableId(HasElement(arrayId,keyId),varId)

\wedge HasKeyExpression(keyId,exprId)

\wedge ValueOf(exprId,y)

\wedge HasArrayName(arrayId,x)

6.2.3 Analysis of Programs that Use Functions

This section looks in more detail at how programs that use functions are analysed in the system.

6.2.3.1 Overall Goal Specification

As described in Section 4.4.2, an exercise specification contains an overall goal in order for the system to analyse potential solutions. When *PreDefinedFunctions* need to be accessed in the exercise, the overall goal specification is given in the same manner with the necessary facts. However, when the required program should contain *UserDefinedFunctions*, the overall goal specification becomes a bit more complex. The requirements of the *UserDefinedFunction* are given using a set of conditions of a sub-plan. If the conditions of the sub-plan are satisfied, a new fact, *FunctionOK(..)*, is created. This fact is included in the overall goal to ensure that a function conforming to the specifications is present (e.g. see first line in *Figure 6.16*). In order to describe this further, consider the example exercise given in *Figure 6.14*.

Write a PHP function called `findTotal` that takes in two parameters and returns their total. In the main program, call this function with the values stored in variables `$x` and `$y` and store the result into the variable `$z`. Note that the variables `$x` and `$y` already contain values when execution reaches the point where the code needs to be completed.

Figure 6.14. Example exercise for functions.

Here, the variables `$x` and `$y` contain values when the program needs to be written so the program has an initial state. The specification of the initial state is given in *Figure 6.15*. As before, symbolic values have been considered as the initial values of the variables `$x` and `$y`. Note that the scope has also been included in the initial state of the program.

The overall goal in this case consists of goals, constraints and a sub-plan as shown in *Figure 6.16*. The sub-plan defines the requirements for the function definition. Once the function definition node in the AST is encountered, the state should contain facts that are equal to the preconditions of the sub-plan. If not, an error is identified as the sub-plan cannot be satisfied. If any such facts are present, the function definition node of the AST node is walked through, creating relevant

facts as described in Section 4.5.3. Next, the available facts are checked to see whether the facts in the post-conditions of the sub-plan are present. If so, the sub-plan is taken to be satisfied and the *FunctionOK* fact is created.

```

HasName(VARID1,'x')
^ CurrentScope(Null)
^ HasValue(VARID1,val_x)
^ HasInitialValue(VARID1,val_x)
^ HasVariableScope(VARID1,Null)
^ HasName(VARID2,'y')
^ HasValue(VARID2,val_y)
^ HasInitialValue(VARID2,val_y)
^ HasVariableScope(VARID2,Null)

```

Figure 6.15. Initial state for example exercise for functions.

```

Goal :      FunctionOK(FUNCID1)
           ^ Add(val_x,val_y,VALUE)
           ^ HasValue(VARID3,VALUE)
Constraints : HasFunctionName(FUNCID1,'findTotal')
           ^ HasName(VARID3,'z')
           ^ HasFunctionCall(FUNCEXPID1,FUNCID1)
           ^ CallsFunction(FUNCCALLID1,FUNCID1)
           ^ HasParamExpression(FUNCCALLID1,1,EXPRID1)
           ^ ValueOf(EXPRID1,val_x)
           ^ HasParamExpression(FUNCCALLID1,2,EXPRID2)
           ^ ValueOf(EXPRID2,val_y)
           ^ HasReturnValue(FUNCCALLID1,VALUE)

Conditions of Subplan(FunctionOK(FUNCID1)):
  PRECOND : HasParameter(FUNCID1,1,VARID4)
           ^ HasParameter(FUNCID1,2,VARID5)
           ^ HasValue(VARID4, VALUEa)
           ^ HasValue(VARID5, VALUEb)
  POSTCOND: Add(VALUEa, VALUEb,VALUEc)
           ^ HasReturnExpression(FUNCID1, RETEXPRID1)
           ^ ValueOf(RETEXPRID1,VALUEc)

```

Figure 6.16. Overall goal specification for example exercise for functions.

The overall goal of the program is to assign the value of the total of variables \$x and \$y to the variable \$z. This is specified by the goal. The *FunctionOK* fact is

included in the goal since it is a requirement in this case that a function be used to achieve this. The name of the new *Variable* and *Function* are part of the constraints as explained in Section 4.4.2. The rest of the constraints in this case are used to specify that the assignment to the new variable should occur using the defined function and not using any other method. This is achieved through the *HasFunctionCall* and *CallsFunction* predicates. The *HasParamExpression* predicate and the values of the relevant expressions are used to ensure that the values of the parameters passed during the function call are correct. The *ValueOf* function expression ensures that the value returned by the function is the same one as is assigned to the variable.

6.2.3.2 Walking the AST

In order to study the process of walking the AST, consider the program given in *Figure 6.17* as a solution to the example exercise given in *Figure 6.14*. This is basically the same as the program in *Figure 6.8* except for the fact that \$x and \$y are not assigned values within the program but are assumed to have initial values.

```
function findTotal($num1,$num2)
{
    $tot=$num1+$num2;
    return($tot);
}

$z=findTotal($x,$y);
```

Figure 6.17. Solution to example exercise for functions.

The initial state of the program results in the following facts, assuming that the ids of the variables \$x and \$y are VarId1 and VarId2 respectively.

CurrentScope(Null)

HasName(VarId1,'x')

HasInitialValue(VarId1,val_x)

HasValue(VarId1,val_x)

HasVariableScope(VarId1,Null)

$HasName(VarId2, 'y')$
 $HasInitialValue(VarId2, val_y)$
 $HasValue(VarId2, val_y)$
 $HasVariableScope(VarId2, Null)$

The function definition is the first node of the AST to be processed and results in the following facts as described in Section 6.2.1.1.

$CurrentScope(FuncId1)$
 $HasFunctionName(FuncId1, 'findTotal')$
 $HasParameter(FuncId1, 1, ParamVarId1)$
 $HasName(ParamVarId1, 'num1')$
 $HasParameter(FuncId1, 2, ParamVarId2)$
 $HasName(ParamVarId2, 'num2')$

Since the *ParameterVariables* are only in scope within the function, new facts are created to indicate this.

$HasVariableScope(ParamVarId1, FuncId1)$
 $HasVariableScope(ParamVarId2, FuncId1)$

In order to see whether the *Function* behaves as it should, it is necessary for these *ParameterVariables* to be assigned values. However, there is no way to assign exact values to these *Variables* during function definition. The solution that is used here is to utilise the rule shown in *Figure 6.18* to assign the name of the *Variable* as the initial value of all *ParameterVariables*. These are then used as symbolic values which are used when analysing the statements within the function.

$HasValue(varId1, name1)$
 $\leftarrow HasParameter(funcId1, position1, varId1) \wedge HasName(varId1, name1)$
 $\wedge HasVariableScope(varId1, funcId) \wedge CurrentScope(funcId1)$

Figure 6.18. Rule to set initial value of parameter variables.

The resultant facts are given below.

$HasValue(ParamVarId1, 'num1')$

HasValue(ParamVarId2,'num2')

At this point, before processing the nodes in the function, a check is made to see whether the preconditions of a sub-plan are satisfied. When considering the overall goal specification in *Figure 6.16*, it can be seen that the precondition is satisfied when $\text{FUNCID1}=\text{FuncId1}$, $\text{VARID4}=\text{ParamVarId1}$, $\text{VARID5}=\text{ParamVarId2}$, $\text{VALUEa}=\text{'num1'}$ and $\text{VALUEb}=\text{'num2'}$. Therefore, the analysis process continues, analysing the AST nodes resulting from the statements within the function to create the relevant facts.

The first node corresponds to an assign statement with an *AddExpr* on the right hand side. Let the id of the *AddExpr* be *ExprId1* and the values of the *VarExprs* on either side of this expression be *VarExprId1* and *VarExprId2* respectively. Then, the following facts are created.

HasId(AddExpr(VarExprId1,VarExprId2),ExprId1)

HasVariable(VarExprId1,ParamVarId1)

HasVariable(VarExprId2,ParamVarId2)

The *ValueOf* of each of these sub-expressions is then found using the rules in *Figure 6.11*.

ValueOf(VarExprId1,'num1')

ValueOf(VarExprId2,'num2')

The *ValueOf* of the *AddExpr* is next found using the rules in *Figure 4.8*. Let the sum of 'num1' and 'num2' be *tot* so *Add('num1', 'num2',tot)*.

ValueOf(ExprId1,tot)

The value of this is assigned to a new variable, *\$tot* and the following facts are created as given in the *Assign* action in *Figure 6.12*. Let the id of the newly created *Variable* be *VarId1*.

HasName(VarId1,'tot')

HasValue(VarId1,tot)

HasInitialValue(VarId1,tot)

HasVariableScope(VarId1,FuncId1)

Next, the AST node corresponding to the return expression is analysed. Here, the return expression is actually a *VarExpr* returning the \$tot variable. This is used together with the rules to find the *ValueOf* the expression to create the following facts.

HasReturnExpression(FuncId1,RetExprId1)

HasVariable(RetExprId1,VarId1)

ValueOf(RetExprId1,tot)

Now, the function definition has been processed. At this point, a check is made to see whether the post-conditions of the sub-plan are satisfied. When comparing against the sub-plan in *Figure 6.16*, it can be seen that the post-conditions are satisfied when RETEXPRID1=RetExprId1 and VALUEc=tot. This means that the function definition matches the specifications, resulting in the following fact being created.

FunctionOK(FuncId1)

If at this point, the post-conditions are not satisfied, the program is identified as incorrect. Since the post-conditions are matched, the analysis process returns to the main program so the scope is altered as below.

CurrentScope(Null)

The main program contains an assignment so the *Assign* action is executed. In this case, the right hand side of the assignment is a function call, resulting in a *FunctionExpr*. Since this expression is identical to the one considered in Section 6.2.1.2, the following facts are created.

CallsFunction(FuncCallId1,FuncId1)

HasParamExpression(FuncCallId1,1,VarExprId1)

HasVariable(VarExprId1,VarId1)

HasParamExpression(FuncCallId1,2,VarExprId2)

HasVariable(VarExprId2,VarId2)

HasFunctionCall(FuncExprId1,FuncCallId1)

Using the rules in *Figure 6.11*, the *ValueOf* the parameter expressions are found as below.

$$\text{ValueOf}(\text{VarExprId1}, \text{val}_x)$$

$$\text{ValueOf}(\text{VarExprId2}, \text{val}_y)$$

Now, in order to find the output of the function, it is necessary to establish the values of the *ParameterVariables* are the values passed in as parameters. This is done using the rule shown in *Figure 6.19*. This results in the following facts being created.

$$\text{HasValue}(\text{ParamVarId1}, \text{val}_x)$$

$$\text{HasValue}(\text{ParamVarId2}, \text{val}_y)$$

Next, these are matched to the preconditions of the sub-plan that was satisfied and the corresponding post-conditions are created since the sub-plan is already known to be satisfied. This results in the following fact being created where *Add(val_x, val_y, value)*.

$$\text{ValueOf}(\text{RetExprId1}, \text{value})$$

Since the function call returns a value, the value of the return expression at this point is assigned to the return value of the function call, resulting in the following fact.

$$\text{HasReturnValue}(\text{FuncCallId1}, \text{value})$$

```

HasValue(varIdn, valn)
  ← CallsFunction(funcCallId1, funcId1)
    ∧ HasParamExpression(funcCallId1, n, paramExprIdn)
    ∧ ValueOf(paramExprIdn, valn)
    ∧ HasParameter(funcId1, n, varIdn)

```

Figure 6.19. Rule to calculate the *ValueOf* parameter variables.

In this case, it is necessary to find the *ValueOf* the function expression on the right hand side of the assignment expression in order to carry out the assignment. This is done using the rule shown in *Figure 6.20*. Using this rule, the following fact is created.

$$\text{ValueOf}(\text{FuncExprId1}, \text{value})$$

```

ValueOf(exprId1,value)
  ← HasFunctionCall(exprId1,funcCallId1)
    ∧ CallsFunction(funcCallId1,funcId1)
    ∧ HasReturnValue(funcCallId1,value)

```

Figure 6.20. Rule for calculating the *ValueOf FunctionExprs*.

Now, the assign action results in the following facts being created. Let the value of the new *Variable* be VarId3.

HasName(VarId3,'z')

HasInitialValue(VarId3,value)

HasValue(VarId3,value)

HasVariableScope(VarId3,Null)

These facts are included in the final state of the program. When comparing the final state against the goal specified in *Figure 6.16*, it can be seen that the goal is satisfied when FUNCID1=FuncId1, VALUE=value and VARID3=VarId3. Moreover, the constraints are satisfied when the above conditions are true and when FUNCCALLID1=FuncCallId1, EXPRID1=VarExprId1, EXPRID2=VarExprId2 and EXPRID3=RetExprId1. Therefore, the program is identified as correct.

This process is used to analyse different types of functions. If only the function definition is required in the specification, the second part of the analysis is unnecessary and the program is identified as correct as long as the sub-plan is satisfied. Several more examples of how functions are analysed can be found in Appendix F.

6.2.4 Pre-Defined Functions

As described in Section 6.2.1, facts defining a function are created when a function definition is encountered within the AST. However, in the case of *PreDefinedFunctions*, a function definition is never encountered when walking the AST. Only function calls for *PreDefinedFunctions* are embedded within the AST. When such a *FunctionCall* is encountered, the post-condition of the relevant sub-plan needs to be considered, in order to create the relevant facts. However, in this case, since the behaviour of the function is not part of the overall goal, no sub-plan is

included in the exercise specification. Therefore, it becomes impossible to create the facts that result from the execution of the function.

This problem is solved by storing relevant facts for *PreDefinedFunctions*. Whenever a *FunctionCall* is encountered, it is first checked to see whether a *PreDefinedFunction* of the same name exists. If so, the relevant facts for the function definition are created, based on data that is stored in the system. This data contains information regarding the function name, the number of parameters, the preconditions and the post-conditions. If the number of parameters in the function call does not match a function definition, an error in semantic analysis (as defined in the theory of compilers) is identified. Functions with optional parameters are handled by storing data for all possible numbers of parameters. Then, the relevant definition is selected based on the number of parameters in the function call. Once the relevant function definition is selected, the corresponding facts that result from the function definition and function execution are created based on data that is stored with respect to the *PreDefinedFunction*. If no *PreDefinedFunction* of the name is present, it is checked against the *UserDefinedFunctions* and processed as described in Section 6.2.3.2. If no *UserDefinedFunction* of the same name can be found, an error in semantic analysis is identified.

It should be noted that the number of *PreDefinedFunctions* that can be used in PHP is very large. Most of these functions are never encountered within basic PHP programs. Therefore, although the above modelling technique can theoretically be used to model any PHP function, only the ones that are used in the exercises have actually been modelled. The actual PHP functions modelled in this manner are `isset`, `intval` and `rand`.

6.2.5 Conditional Expressions Where the Condition is a *FunctionCall*

Sometimes, the conditional expression within a selection statement can be a *FunctionCall*. An example code is shown in *Figure 6.21*. In this program, the conditional expression within the *if* statement is a call to the function `isset` which returns True or False based on whether variable `$_POST['x']` has already been set or not. In such cases, it is necessary to determine the value returned by the function before a suitable conditional fact can be determined as described in Section 5.2. The rules in *Figure 6.22* are used for this purpose. The value returned by the function is set to be equal to the value of the *FunctionExpr*.


```

if(isset($_POST['x']))
{
    echo('The variable has a value');
}

```

Figure 6.21. PHP code that has a function expression as the condition within a selection statement.

```

EqualTo(value,True)
  ← HasId(FunctionExpr,exprId1)
    ∧ ValueOf(exprId1,True)
    ∧ HasFunctionCall(exprId1,funcCallId1)
    ∧ HasReturnValue(funcCallId1,value)

EqualTo(value,False)
  ← HasId(FunctionExpr,exprId1)
    ∧ ValueOf(exprId1,False)
    ∧ HasFunctionCall(exprId1,funcCallId1)
    ∧ HasReturnValue(funcCallId1,value)

```

Figure 6.22. Rules used to find conditional expressions for *FunctionExprs*

6.2.6 Unnecessary Statements in Functions

As described in Section 4.5.5, programs sometimes contain statements that are unnecessary to achieve the overall goal. Such statements in programs with functions are also handled using statuses. A new status is created each time a function definition is encountered. Any statuses created during the analysis of the function are linked to this initial function status. In addition to normal program statements, functions may also contain ‘return’ statements. A new status is created when a ‘return’ statement is encountered.

The flow of statuses in this case is slightly different from the earlier cases due to the use of sub-plans. As described in Section 6.2.3.2, a *FunctionOK* fact is created once it is established that a sub-plan is satisfied. A new status is created just before creating this *FunctionOK* fact. A link is maintained between this new status and the status where the sub-plan was satisfied in order to establish that this path is necessary to achieve the overall goal.

The flow of statuses for the example program in *Figure 6.14* is shown in *Figure 6.23*. The initial function definition creates the new status ‘Status 1’. The

assignment statement within the function results in a new status. This is linked to Status 1 for two reasons: it is part of the function definition as well as uses the values of the function parameters which are created within Status 1. The return statement results in another new status. This is linked to Status 1 because it is part of the function definition. It is also linked to Status 2 since the variable \$tot created within Status 2 is used here. At the end of the function definition, the sub-plan is satisfied as described in Section 6.2.3.2. Now, a new status is created before the *FunctionOK* fact is defined. This status is linked to Status 3 since this is the status where the sub-plan was satisfied. Another new status is created by the assignment statement in the main program. This status is linked to the initial status since it uses values from the variables \$x and \$y created during the initial status. It is also linked to Status 4 since *FunctionOK* forms a part of the overall goal. Now it can be seen that all the statuses are linked to the status where the overall goal is satisfied, i.e. Status 5. If any additional statements were present, they would not contain a path leading to this status and therefore, can be identified as unnecessary. If any unnecessary statements are present within the function, they would not lead to the status where the sub-plan is satisfied and would therefore be identified as unnecessary.

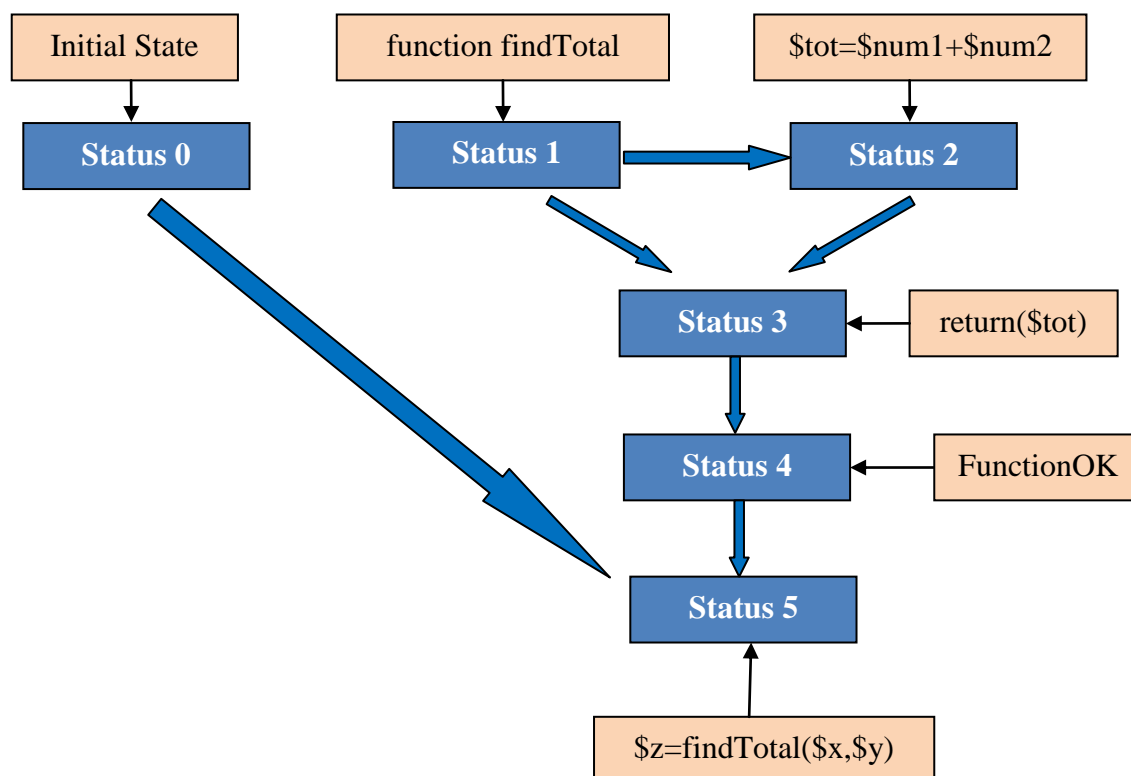


Figure 6.23. Flow of statuses for example program using functions.

6.3 FORMS

HTML forms are an integral part of dynamic web pages. Since little work has been done in teaching web programming using ITSs, knowledge bases to handle such forms have not been designed. HTML forms offer a major challenge in modelling since pages with such forms can be in one of two states: before submitting the form and after submitting the form. The variables that are available for manipulation depend on which of these states the form is in.

6.3.1 Form Definition

When modelling forms, it is necessary to consider the actual form and its elements as well as the values passed from this form. *Figure 6.24* shows the part of the ORM diagram that shows the various object types related to forms. The form itself is modelled as an object with a unique id. Each form has a method given by the *HasMethod* predicate. The method is either ‘GET’ or ‘POST’ depending on the method specified when creating the form.

A form has zero or more input elements, each with a unique id. The relationship between the form and its elements is shown using the *HasInputElement* predicate. Each input element has a type given by the *HasInputType* predicate. The types of input elements modelled in this case are ‘TEXT’, ‘SUBMIT’ and ‘SELECT’. Although other input element types are possible in HTML, these are outside the scope of this thesis. If the input type is ‘SELECT’, it also contains some options which are modelled as objects with unique ids. The options related to a particular select element are given by the *HasInputOption* predicate. The value of each option is modelled using the *HasOptionValue* predicate.

HTML allows defining input elements without names. However, in order to access the values stored in these elements using PHP, it is necessary to give each option a name. This is modelled using the *HasInputName* predicate.

An HTML form has an action that specifies the page onto which the form is being submitted. The values entered into the *InputElements* of the form are only accessible from within the page onto which the form is submitted. This research only considers forms that are submitted onto the same page, i.e. `action=""`. I.e. does not model forms where the form is submitted onto a different web page.

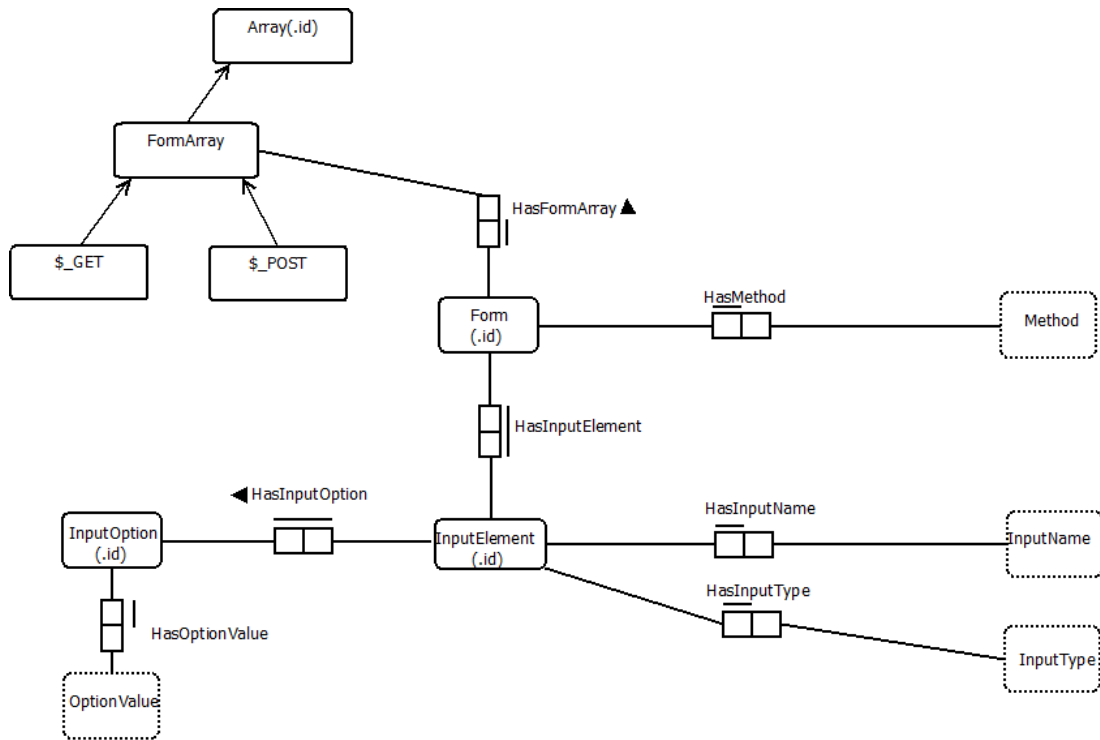


Figure 6.24. ORM diagram for forms.

In order to see how these predicates are used, consider the example exercise given in Figure 6.25. An example solution is given in Figure 6.26. The analysis of this program is described step by step since it contains many different aspects.

Write a PHP program that contains a form that uses the POST method and submits onto itself. The form should contain three input elements: a select list named 'item', a textbox named 'quantity' and a submit button named 'submit'. The select list should contain two items, 'paper' and 'pencil'. The names and values of these items should be the same. When the form is submitted, it should display the value entered in the 'quantity' textbox. The form should then be displayed again so that it can be used.

Figure 6.25. Example exercise for forms.

Although the first part of the program contains other statements, the first step in the analysis process is to create facts that are relevant to the form definition. Using the above description let the id of the created *Form* object be *FormId1*. The following fact is then created since the form uses the 'POST' method.

HasMethod(FormId1, 'POST')

```

<?php
if(isset($_POST['submit']))
{
    echo($_POST['quantity']);
}
?>
<form method=post action=''>
<select name=item>
<option name=paper value=paper>paper</option>
<option name=pencil value=pencil>pencil</option>
</select>
<input type=text name=quantity>
<input type=submit name=submit>
</form>

```

Figure 6.26. Example solution to exercise for forms.

Let ids of the three created *InputElement*s be *InputId1*, *InputId2* and *InputId3* respectively. Since the first *InputElement* is a select list, it also contains two options. Let the ids of the *Options* be *OptionId1* and *OptionId2* respectively. Then, the following facts are created.

HasInputElement(FormId1,InputId1)

HasInputName(InputId1,'item')

HasInputType(InputId1,'SELECT')

HasInputOption(InputId1,OptionId1)

HasOptionValue(OptionId1,'Paper')

HasInputOption(InputId1,OptionId2)

HasOptionValue(OptionId2,'Pencil')

The other two inputs do not contain any options and the following facts are created.

HasInputElement(FormId1,InputId2)

HasInputName(InputId2,'quantity')

HasInputType(InputId2,'TEXT')

HasInputElement(FormId1,InputId3)

HasInputName(InputId3,'submit')

HasInputType(InputId3,'SUBMIT')

6.3.2 Accessing Values Passed Through Forms

The value stored in an HTML input element is accessed using a super-global array. The name of the super-global array depends on the method used in the form. The values stored in forms submitted using the ‘GET’ method are stored in the array `$_GET` while the values stored in forms submitted using the ‘POST’ method are stored in the `$_POST` array. Whatever the method, it is necessary to create a *FormArray* object to hold the values upon submitting (for more information about the *FormArray* object type, see Section 6.1). When a form is created, the relevant subtype of form array, either `$_GET` or `$_POST` is created, based on the method used in the form.

Considering the example above, since the ‘POST’ method is used, the *FormArray* named `$_POST` is created and associated with the form as below. Let the id of the created `$_POST` array be *FormArrayId1*.

HasFormArray(FormId1,FormArrayId1)

When the form is submitted, this array is used to access the values stored in the *InputElements*. In PHP syntax, these values are accessed using the array with the *Key* containing the name of the *InputElement*. Therefore, facts are created to indicate that the array has the corresponding elements. This is done using the rule in *Figure 6.27*.

```
HasVariableId(HasElement(formArrayId1,keyId1,varId1)
^ HasKeyExpression(keyId1,exprId1)
^ HasLiteral(exprId1,litId1)
^ HasLitValue(litId1,inputName1)
←
HasInputElement(formId1,inputElementId1)
^ HasInputName(inputElementId1,inputName1)
^ HasFormArray(formId1,formArrayId1)
```

Figure 6.27. Rule to create array elements from form input elements.

The following facts are created for the above example using this rule. Let the ids of the corresponding *Variables* be *VarId1*, *VarId2* and *VarId3* respectively. Also

let the *Keys* of the corresponding elements be *KeyId1*, *KeyId2* and *KeyId3* and the ids of the *Expressions* corresponding to the keys be *ExprId1*, *ExprId2* and *ExprId3* respectively.

HasVariableId(HasElement(FormArrayId1,KeyId1),VarId1)

HasKeyExpression(KeyId1,ExprId1)

HasLiteral(ExprId1,LitId1)

HasLitValue(LitId1,'item')

HasVariableId(HasElement(FormArrayId1,KeyId2),VarId2)

HasKeyExpression(KeyId2,ExprId2)

HasLiteral(ExprId2,LitId2)

HasLitValue(LitId2,'quantity')

HasVariableId(HasElement(FormArrayId1,KeyId3),VarId3)

HasKeyExpression(KeyId3,ExprId3)

HasLiteral(ExprId3,LitId3)

HasLitValue(LitId3,'submit')

Although these array elements exist, they only contain values when the form is submitted. In PHP code, this is achieved by using an if condition with the ‘isset’ predefined function. The ‘isset’ function returns True if the variable passed in as its argument is set and is not null. Although it is theoretically possible to pass in any variable as the argument to this function, this research only considers the case where a variable corresponding to an element in a *FormArray* is passed in. Additionally, it is assumed that the ‘isset’ function is used to check whether the ‘submit’ button in the form is pressed throughout this research. Although it is possible to consider some other element of the form array, it is standard practice to check for the ‘submit’ button. Therefore, the standard form of program considered here is as shown in *Figure 6.26*.

The ‘isset’ function is a function call. Therefore, the following set of facts are created to handle the function call as described in Section 6.2.1.2. Since this is a *PreDefinedFunction*, the relevant facts are created based on the stored data as described in Section 6.2.4. Note that only the facts that are relevant to this analysis

are shown. Let the id of the created *Function* be *FuncId1* and the id of the *ParameterVariable* be *ParamVarId1*. Let the ids of the return expression and *FunctionCall* be *RetExprId1* and *FuncCallId1* respectively.

HasFunctionName(FuncId1,isset)
HasParameter(FuncId1,1,ParamVarId1)
HasReturnExpression(FuncId1,RetExprId1)
HasFunctionCall(FuncExprId1,FuncCallId1)
CallsFunction(FuncCallId1,FuncId1)
HasParamExpression(FuncCallId1,1,ParamExprId1)
HasVariable(ParamExprId1,VarId3)

The ‘isset’ function expression resides inside a if condition. Therefore, when considering the state inside the *if* condition, the value of the function expression is True. This means that the following predicate is valid inside the condition.

ValueOf(FuncExprId1,True)

The special rule defined in *Figure 6.28* is used to set the value of the variable passed into the ‘isset’ function to True if the value of the function expression is True. This rule is executed within the if condition, resulting in the following fact.

HasValue(VarId3,True)

```

HasValue(varId1,True)
←
HasFunctionCall(funcExprId1,funcCallId1)
∧ CallsFunction(funcCallId1,funcId1)
∧ HasFunctionName(funcId1,'isset')
∧ ValueOf(funcExprId1,True)
∧ HasParamExpression(funcCallId1,1,paramExprId1)
∧ HasVariable(paramExprId1,varId1)

```

Figure 6.28. Rule to set the value of the parameter variable when the value of a ‘isset’ function expression is True.

When considering the semantics of PHP form processing, it is obvious that once the submit button is pressed, all variables corresponding to input elements in the relevant form will be set. Therefore, the rule shown in *Figure 6.29* is used within

the bounds of the if statement to set initial values for these variables. Since it is not possible to set exact values for these variables, they are set to the names of the *InputElement*s. These form as a symbolic basis for program analysis.

In the example above the form contains two additional *InputElement*s of type SELECT and TEXT. Using the above rule, the following facts are now created within the if statement.

HasValue(VarId1,'item')

HasValue(VarId2,'quantity')

Therefore, all variables within the *FormArray* now have values within the if statement. This ensures that the submitted status of the form is accurately modelled.

```

HasValue(varId2,inputName2)
←
HasInputElement(formId1,inputElementId1)
∧ HasInputName(inputElement1,inputName1)
∧ HasInputType(inputElement1,'SUBMIT')
∧ HasFormArray(formId1,formArrayId1)
∧ HasVariableId(HasElement(formArrayId1,keyId1),varId1)
∧ HasKeyExpression(keyId1,exprId1)
∧ ValueOf(exprId1,inputName1)
∧ HasValue(varId1,True)
∧ HasVariableId(HasElement(formArrayId1,keyId2),varId2)
∧ HasKeyExpression(keyId2,exprId2)
∧ ValueOf(exprId2,inputName2)

```

Figure 6.29. Rule to set the values of all form variables once the form is submitted.

6.3.3 Handling Standard Form Definitions

The above analysis describes how form submissions are handled as long as all the elements of the form are known. However, as shown in *Figure 6.26*, it is standard practice to write the code for form submission before defining the actual form. This means that elements corresponding to the form have not been created by the time the AST walking process encounters the ‘isset’ function.

This problem is handled as in the case of HTML embedded within PHP described in Section 4.6.3. The AST is walked through several times. During the first round, any conditional statements are checked to see if the condition involves any `$_POST` or `$_GET` variables. If so, these nodes are not analysed. Once the end

of the AST is reached, any form definitions result in the relevant facts being formed. During the next round, the ignored AST nodes are walked through. Since the form definitions are now complete, the analysis can proceed as described in Section 6.3.

6.4 CHAPTER SUMMARY

This chapter looked at how some more advanced PHP topics are handled within the knowledge base. It discussed how arrays are modelled and how PHP syntax designed to make array definitions easier are handled. It also looked at how function definitions and function calls are handled within the knowledge base. The final section described how form processing is modelled. All these topics have received little focus in previous computerised learning systems.

The next chapter looks at another very often used type of program statement, loops. It explores how different types of loops are handled and discusses the limitations of the current knowledge base in handling loops. It is the final chapter on how program analysis is carried out within the PHP ITS.

Chapter 7: Loops

The previous chapter looked at how the PHP ITS analyses programs that contain arrays, functions and forms. This chapter looks at another common type of construct used in programming: loops. Several types of loops are used extensively to iterate through a set of statements as described below. However, the PHP ITS is not capable of analysing programs written using all these types of loops. Further, the PHP ITS does not allow recursive functions to be used as a way of implementing loops. Recursive functions are considered to be beyond the scope of this thesis which aims at teaching novice programmers. This chapter investigates the types of loops that can be analysed by the PHP ITS and the process followed during the analysis.

7.1 TYPES OF LOOPS

Loops are a common structure in any modern programming language. They can be classified using many different methods. One method is to classify them based on the syntactic construct used to create the loop (eg:- while, do, for etc). This method is quite useful when teaching the basics of programming. However, when designing a knowledge base to analyse loops, it is more useful to look at the logical model underlying the functionality of the loop and classify the loops accordingly. Again, loops can be classified using the logical model in many different ways. A main aim of this research project is to analyse student programs for correctness. A classification that lends itself to such analysis is given in *Figure 7.1* (personal communications, Reye, 2012).

In this classification, loops are classified based on whether they iterate through a collection of data items or not. Usually, loops are introduced in a collection independent manner in introductory programming courses. However, in most real-world applications, loops that iterate through a collection of data items are much more common (Stavely, 1993).

Collection independent loops are further classified into definite and indefinite loops. A *definite* loop is one that executes a number of times known in advance, before entering the loop. An *indefinite* loop is one where the number of iterations is not known in advance. Examples of definite and indefinite loops in PHP are shown

in Table 7.1. The first loop iterates exactly five times. Since the number of iterations is known before the loop iterates, it is a definite loop. On the other hand, the second program iterates until the variable \$found is true. If it is not true, a function (not defined here) operates on the variable \$x and changes it. Since the number of iterations depends on the return value of the function, the exact number of iterations cannot be determined beforehand. Therefore, this is an example of an indefinite loop.

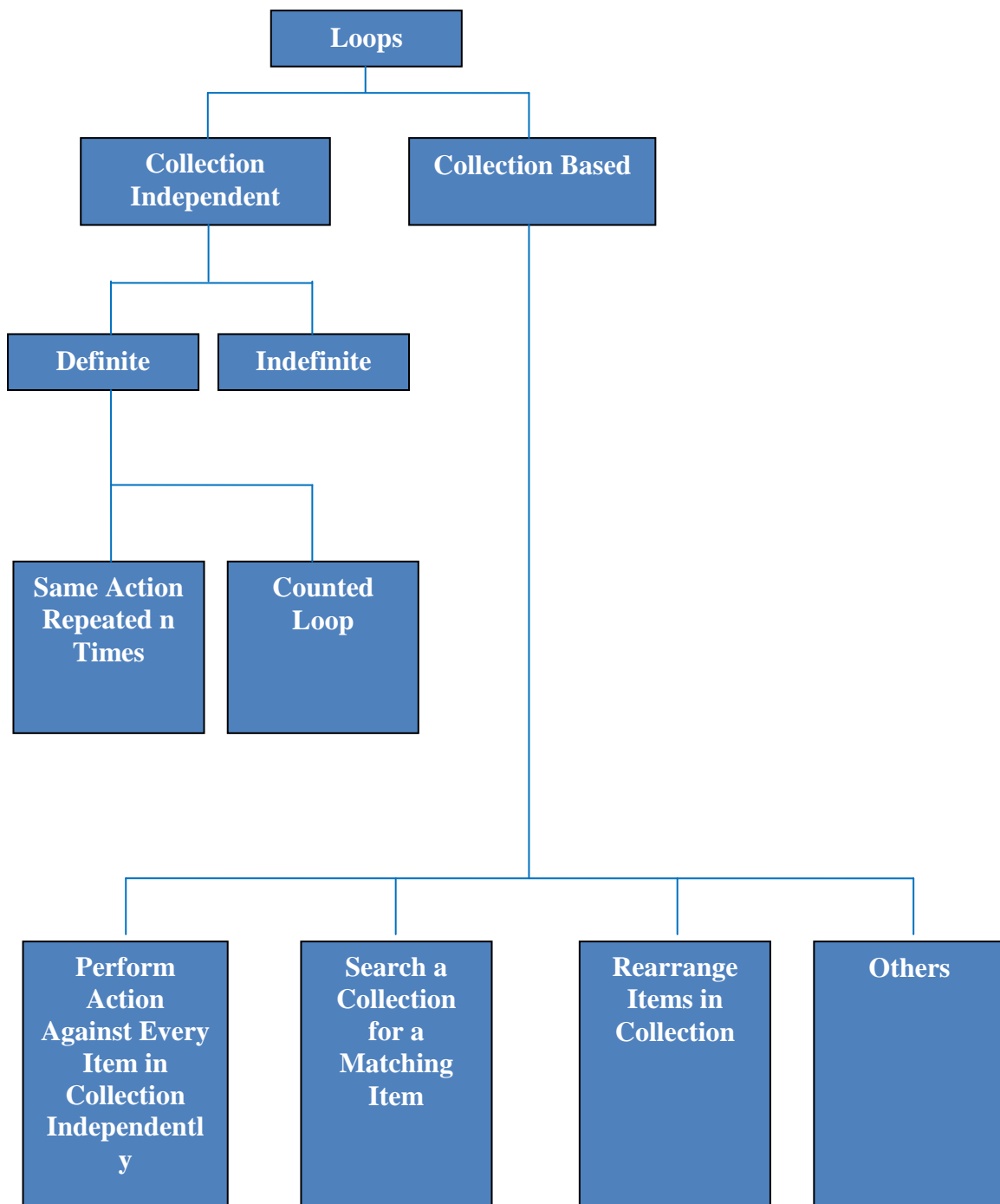


Figure 7.1. Classification of loops.

Table 7.1

Definite and Indefinite Loops in PHP

<i>Definite Loop</i>	<i>Indefinite Loop</i>
<pre>for(\$i=1;\$i<=5;\$i++) { echo("Hello"); }</pre>	<pre>\$found=false; while(!\$found) { if(\$x==5) { \$found=true; } else { \$x=doFunction(\$x); } }</pre>

Definite loops can be further classified into two types. The first type of definite loop is the most basic kind where a certain action is repeated, a given number of times. The definite loop shown in Table 7.1 belongs to this category. The second type of definite loop is the counted loop where a loop variable takes on certain integer values in a range and the value of the loop variable is used within the loop. Table 7.2 shows two examples of counted loops. In the first example, the loop variable, \$i takes successive values from 1 to 100 and each of these values are printed inside the loop. In the second example, the value of the loop variable is printed within the loop as in the previous case. However, the value of the counter variable does not take on all successive values from 1 to 100. It is incremented by 10 at each iteration and therefore changes according to an arithmetic sequence.

Table 7.2

Counted Loops in PHP

<i>Repeat for Successive Values of the Counter Variable</i>	<i>Repeat for Values of the Counter Variable Changing According to an Arithmetic Sequence</i>
<pre>for(\$i=10;\$i<=100;\$i++) { echo(\$i); }</pre>	<pre>for(\$i=10;\$i<=100;\$i+=10) { echo(\$i); }</pre>

Collection based loops can also be either definite or indefinite based on whether the number of iterations are known in advance. If the size of the collection is known in advance and every item in the collection needs to be processed, a definite loop is used. However, it is possible to use some algorithms in the same way whether or not the size of the collection is known in advance. Therefore, a different classification is considered for collection based loops.

Collection based loops can logically operate on any collection of items that are permitted by the programming language. Since only basic PHP is taught by the PHP ITS, the only collection that is considered in this thesis is the array. Such loops can be categorised into four main types as shown in *Figure 7.1*. The first type performs actions on each item of the collection independently. It is possible that each such item is updated as it is accessed, but the action performed in one iteration only accesses a single item in the collection. Two example of PHP programs belonging to this category are shown in Table 7.3. The first program simply accesses each array element and displays it on the screen. The second program goes a step further to summarise the array and find the maximum element.

Table 7.3

Perform Action Against Every Item in Collection Independently

<i>Access Every Item in Array without Summarising</i>	<i>Access Every Item in Array while Summarising</i>
<pre> \$array=array(10,20,30) for(\$i=0;\$i<3;\$i++) { echo(\$array[\$i]); } </pre>	<pre> \$array=array(10,20,30); \$i=0; \$found=false; \$max=\$array[0]; while(!found && \$i<3) { if(\$array[\$i]>\$max) { \$found=true; \$max=\$array[\$i]; } else { \$i++; } } </pre>

The second type of collection based loop searches for a matching item. Such loops perform one action if the current item matches a specific condition and some other action if it doesn't. The third type of collection based loop rearranges the items in a collection. A loop that sorts the items in an array in ascending order is a good example of this. Collection based loops that do not fall into any of the above categories are classified as 'Others'.

The knowledge base of the PHP ITS is not capable of handling all these types of loops. In its current form, it can handle all collection independent definite loops. The outcome of such collection independent definite loops varies based on what actually occurs within the loop. The actions performed by some loops are independent of the result of the same action performed during a previous iteration. In other loops, the result of one iteration depends on the results of a previous iteration. This is especially true in cases where a loop performs some form of aggregation of data such as adding to a variable defined outside the loop. Two examples of these different types of loops are shown in Table 7.4.

Table 7.4

Types of Loops Based on the Independence of Actions Performed Within the Loop

<i>Action Independent of Iteration</i>	<i>Action Dependent on Result of Previous Iteration</i>
<pre>for(\$i=1;\$i<=5;\$i++) { echo("Hello"); }</pre>	<pre>\$sum=0; for(\$i=1;\$i<=5;\$i++) { \$sum+=\$i; }</pre>

All loops containing independent actions can be handled by the PHP ITS. However, in the case of loops where the action within the loop is dependent on a previous iteration, the capabilities of the PHP ITS are limited. This sort of loop requires a special rule to be written for each new situation as described in Section 7.3.2 below. Since the rule is dependent on the specifications of the program, it is not possible to write an infinite number of rules to handle all situations. Therefore, rules have been added to the KB to only handle situations that are required by the specific set of exercises that are currently defined in the PHP ITS. Although it is

possible to extend the KB by adding similar rules based on the requirements of additional exercises, they have not been included in the present system.

The other type of loop that can be handled by the PHP ITS is collection based loops that perform some action against every item in the collection independently. It can handle both situations of such situations where the loops performs some summarisation or does not do so.

The PHP ITS in its present form is incapable of handling the other types of loops described here. More theoretical modelling needs to be carried out in order to identify how such loops can be handled using FOPL. However, a study by Stavely (1993) showed that over 50% of loops used in real-world programming belong to what he classified as for-each, and other definite loops. The type of loop he classified as for-each is synonymous to the collection based loops that perform some action against every item in the collection independently. Stavely's other definite loops are the same as collection independent definite loops described here. This shows that the PHP ITS is capable of handling a large percentage of loops that are encountered in practical situations.

In Computer Science theory [(Gries, 1981; Huth & Ryan, 2004), for example], two approaches are most common for analysing loops: (a) only covering while loops – and treating other kinds of loops as being equivalent to while loops – and reasoning about their loop invariants; and (b) converting each loop into an equivalent recursive formulation. Unfortunately, neither of these is really suitable for an ITS. While loop invariants have some nice aspects for proving program correctness, most people learn to program without ever knowing about such constructs. Trying to explain an error to a novice, in terms of a loop invariant is unlikely to succeed. Similarly, trying to explain an error in terms of a recursive reformulation is also unlikely to succeed. Neither is the approach that a human tutor would use with a novice. Therefore, the analysis process used in the PHP ITS does not consider either loop invariants or recursive formulations. The following sections describe the process used by the PHP ITS for this purpose in detail.

7.2 DEFINITE LOOPS

Definite loops form the basis for how the PHP ITS analyses all types of loops. Other types of loops are analysed by building on the analysis process for definite loops. This section describes how the KB analyses definite loops. Since these loops iterate a known number of times, they depend on a counter variable that changes its value for each iteration. Therefore, the most obvious PHP construct used for such loops is the *for* construct.

7.2.1 Predicate Definition

Figure 7.2 shows the set of predicates that are defined in the knowledge base to handle loops that depend on a counter variable. Any type of loop results in a *Loop* object with a new unique id being created. This knowledge base categorises loops into two main sub-types: *CountedLoops* and *ForEach*. Here *CountedLoops* are loops that use a counter variable to control the number of iterations of the loop. Of the types of loops described above definite loops and collection based loops, where some action is performed against every item in the collection, fall into this category. The other sub-type of Loop used here is a *ForEach* loop. Note that these loops are a special type of the collection based loops where some action is performed against every item in the collection as described in Section 7.1 and refer the use of the *foreach* construct in PHP. *ForEach* loops utilise different predicates which are not shown in *Figure 7.2* (for clarity) but are described later in Section 7.4.2. The rest of this section discusses the predicates that are used in handling *CountedLoops*. *CountedLoops* are again divided into two main sub-types *For* and *While* based on the syntactic construct used within the loop.

Consider how these types of loops are used to create predicates to handle definite loops. The number of iterations of such loops is defined using a starting value, an ending value and an increment. The relationships between these values and the actual loops are modelled using the *HasForStartValue*, *HasForEndValue* and *HasForIncrement* predicates respectively. The relationship between the *CountedLoop* and the counter *Variable* itself is established using the *HasLoopVariable* predicate. A loop of this type is terminated using some sort of condition. This condition is modelled as a *BooleanExpression* and the relationship is modelled using the *HasLoopCondition* predicate.

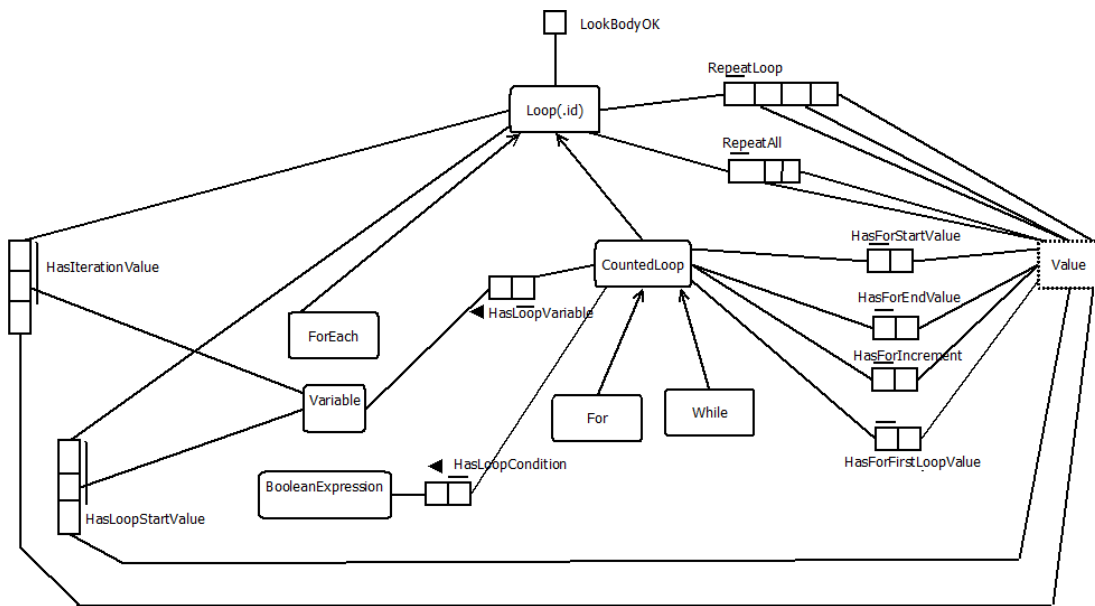


Figure 7.2. Predicates for handling loops with counters.

In order to understand how these predicates are used, consider the first *for* loop in Table 7.1. Let the id of the created *For* be ForId1. When analysing the *for* loop, a new variable \$i is encountered and is assigned a value 1. This results in the following facts being created as described in Section 4.5.3. Let the id of the created *Variable* be VarId1. Since scope does not play a part here as no functions are used, the facts related to scope are ignored here.

HasName(VarId1,'i')

HasValue(VarId1,1)

HasInitialValue(VarId1,10)

This *Variable* is the loop variable with a starting value of 1, an ending value of 5 and an increment of 1. The following facts are created to represent these details.

HasLoopVariable(ForId1,VarId1)

HasForStartValue(ForId1,1)

HasForEndValue(ForId1,5)

HasForIncrement(ForId1,1)

The condition in the *for* loop results in a *LessEqualExpr* object being created. Let the id of this expression be ExprId1. Let the id of the *VariableExpr* on the left

hand side of this expression be *VarExprId1* and the *LiteralExpr* on the right hand side be *LitExprId1*. Let the id of the *Literal* be *LitId1*. Then, the following facts relevant to this expression are created as explained in Section 4.4.1.1.

HasId(LessEqualExpr(VarExprId1,LitExprId1),ExprId1)

HasVariable(VarExprId1,VarId1)

HasLiteral(LitExprId1,LitId1)

HasLitValue(LitId1,5)

Since this expression is the condition of the *for* loop, the following fact is created.

HasLoopCondition(ForId1,ExprId1)

During the analysis process for loops, it becomes necessary to find the value of the loop variable at the end of the first iteration. This relationship is maintained using the *HasForFirstLoopValue* predicate. Another important consideration at this time is the range of actual values of the counter variable for which the loop iterates. This is modelled using the *RepeatLoop* predicate which relates the id of the loop with the start value, end value and increment of the counter variable. The *RepeatAll* predicate is used to specify that the loop repeats for all integer values of the counter variable between start value and end value. The starting value of each variable existing before the execution of the loop also becomes important during program analysis. This is maintained using the *HasLoopStartValue* predicate.

The values of variables very often change within loops. In such cases, it becomes necessary to assign a symbolic value for each variable as the starting value of that variable for each iteration. This relationship is maintained using the *HasIterationValue* predicate.

As in functions (Section 6.2.3.1), conditions of sub-plans are used to model the required results of execution of loops. The *LookBodyOK* predicate is used to indicate that the body of a loop performs the actions that it is supposed to, or in other words, the conditions of the sub-plan is satisfied.

7.2.2 Overall Goal Specification

In order to understand how loops containing counter variables are analysed within the PHP ITS, consider the example exercise given in *Figure 7.3*.

Write a PHP program to display the string “Hello” five times. Use a for loop.

Figure 7.3. Example exercise for simple counted loop.

As mentioned above, the overall goal for exercises containing loops is also specified using conditions of sub-plans. The conditions of the sub-plan define the results of each iteration of the loop while the overall goal specifies the combined outcome of the program. The overall goal also contains the predicate *LoopBodyOK* in order to ensure that the loop performs as it is supposed to.

The overall goal for the example program above is given in *Figure 7.4*. The goal itself specifies that for all values of *j* between 1 and 5 (i.e. for 5 iterations), an *OnPage* fact should be generated. Here, the value of *y* is immaterial since the order of the display does not matter. The constraints specify that a *For* loop should be used and that the loop should perform the necessary function. The first constraint could be removed if the exercise did not specify the type of loop. The second constraint is necessary to ensure that the output has been obtained by using a loop as expressed in the exercise. Otherwise, even if five consecutive echo statements were written, the program would be accepted as correct. The structural constraint of having to use a loop is controlled using the constraint.

The *LoopBodyOK* fact is only created if the sub-plan is satisfied. In this case, no pre-conditions are necessary for the loop to function properly. All that is necessary is that the string "Hello" is displayed within the loop. This is specified by the post-condition. Again, the value of *x* is immaterial since the order of display does not matter.

Goal : $\forall j [(1 \leq j \leq 5) \rightarrow \{ \text{OnPage}(\text{"Hello"}, Y) \}]$

Constraints : For(FORID1)
 \wedge LoopBodyOK(FORID1)

Conditions of Subplan(LookBodyOK(FORID1)):
 PRECOND :
 POSTCOND: OnPage("Hello",x)

Figure 7.4. Overall goal for example exercise for simple counted loop.

7.2.3 Program Analysis

In order to understand how such loops are analysed within the PHP ITS, consider the example solution for the above exercise given in the first *for* loop in Table 7.1. Since this program does not assume that any data is present before executing the program segment, no initial state is specified. The following facts are created as a result of the *for* loop as described in Section 7.2.1.

HasName(VarId1,'i')

HasValue(VarId1,1)

HasInitialValue(VarId1,1)

HasLoopVariable(ForId1,VarId1)

HasForStartValue(ForId1,1)

HasId(LessEqualExpr(VarExprId1,LitExprId1),ExprId1)

HasVariable(VarExprId1,VarId1)

HasLiteral(LitExprId1,LitId1)

HasLitValue(LitId1,5)

HasLoopCondition(ForId1,ExprId1)

Using the rules in *Figure 4.8*, the *ValueOf* the *LiteralExpr* is found, resulting in the following fact.

ValueOf(LitExprId1,5)

Note that the end value and increment of the loop are not created as facts at this point since they cannot directly be ascertained from the program statements. The end value of the loop depends on the type of expression that is used in the condition. A set of rules (as shown in *Figure 7.5*) are used to calculate the end value of the iteration. Note that this thesis only handles counted loops with a single condition. Therefore, the only possible types of expressions are *LessExpr*, *LessEqualExpr*, *GreaterExpr* and *GreaterEqualExpr*.

Using the second rule given here, the following fact is created.

HasForEndValue(ForId1,5)

It is also necessary to find the increment of the loop to analyse the program. The increment is again found using the rule specified in *Figure 7.6*. It can be seen that, before finding this value, it is necessary to find the value of the loop variable at the end of the first iteration. This is achieved by analysing the update condition of the *for* loop using the procedure described in Chapter 4.

In this case, the update condition is a post-increment statement. As described in Section 4.6.2, this results in an expression being created as well as an assignment operation being performed. This assignment results in the following fact since 1 is added to the current value of the variable. Note that only the facts pertinent to this analysis have been described here.

$$\text{HasValue}(\text{VarId1}, 2)$$

Since this is the value of the loop variable at the end of the first iteration, the following fact is created.

$$\text{HasForFirstLoopValue}(\text{ForId1}, 2)$$

Next, the rule in *Figure 7.6* is activated, resulting in the following fact.

$$\text{HasForIncrement}(\text{ForId1}, 1)$$

Once the facts relevant to the *CountedLoop* are obtained, it is necessary to analyse the loop itself. At this point, it becomes necessary to introduce a new notation to indicate the repetition of actions that occur within the loop. Assume that the overall actions that occur within the loop are given by *LoopActionEffects*. By the semantics of the counted loop, these actions are then repeated within the loop. The notation used within this thesis to specify this repetition is as below.

$$\text{repeat}(\text{LoopActionEffects}, \text{LoopId})$$

Using this notation, the effects of the overall loop in the example program can be specified as below.

$$\text{repeat}(\text{ForActionEffects}, \text{ForId1})$$

But *ForActionEffects* is the results of the analysis of what occurs inside the loop. Therefore, the program statements within the loop are next analysed by comparing it against the conditions of the sub-plan as in functions (Section 6.2.3.2). Before such analysis can be performed, it is necessary to understand that any existing

variables can change their values within the loop. Therefore, it is incorrect to consider the values that these variables currently have as the value that they will contain during execution of the program statements within the loop. For analysis purposes, symbolic values are given to any existing variables at this point. All variables are assumed to have this symbolic value during the execution of the loop. This is similar to the assignment of symbolic values to variables in the initial state (Section 4.5.1).

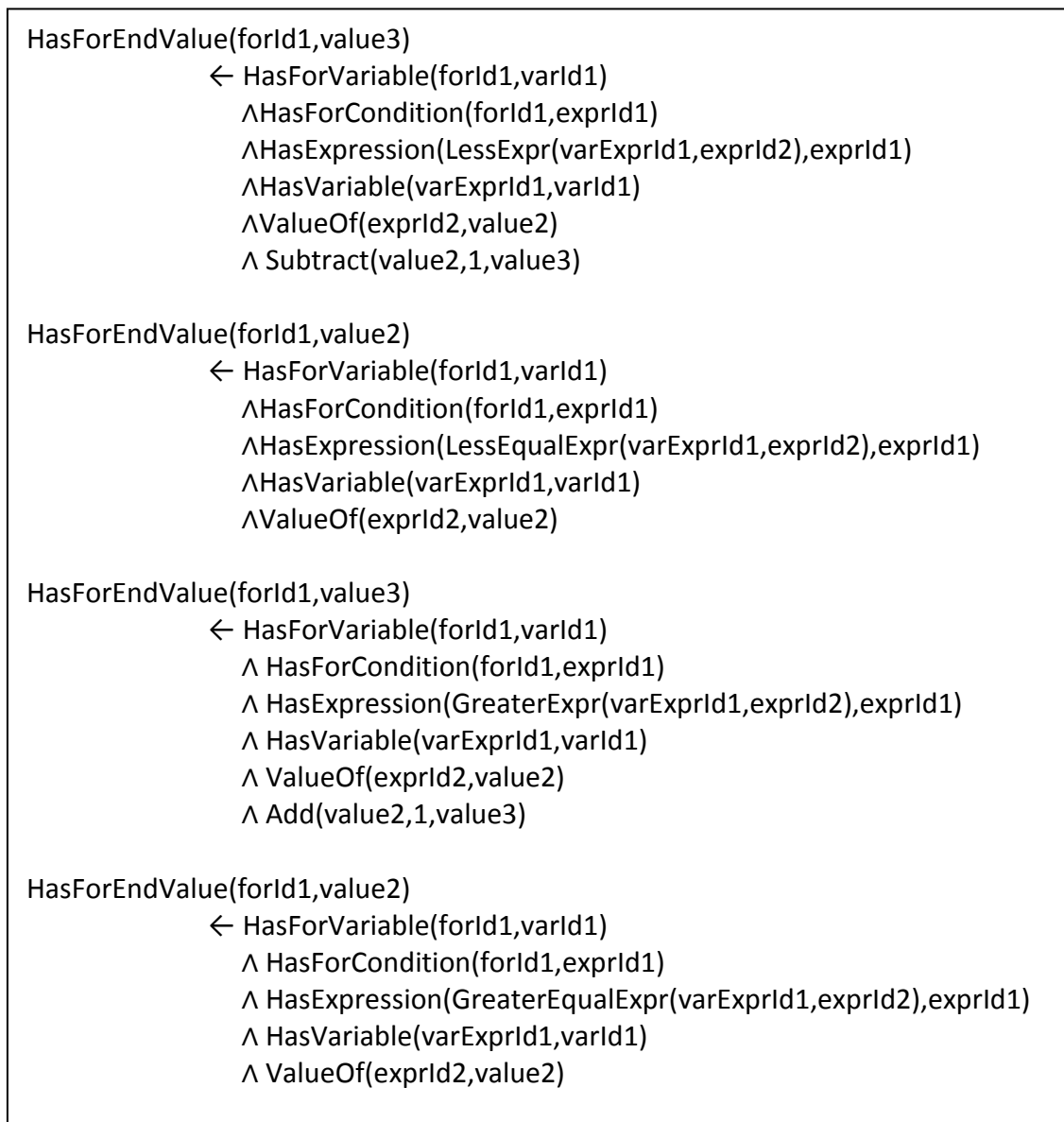


Figure 7.5. Rules for finding the end value of a *CountedLoop*

```

HasForIncrement(forId1,value3)
  ← HasForVariable(forId1,varId1)
    ∧ HasForStartValue(forId1,value1)
    ∧ HasForFirstLoopValue(forId1,value2)
    ∧ Subtract(value2,value1,value3)

```

Figure 7.6. Rule to find the increment of a *CountedLoop*

For the example program, only one variable, \$i, exists at this point. A symbolic value is assigned to this variable at this point, resulting in the following facts.

HasValue(VarId1,val_i)

HasIterationValue(ForId1,VarId1,val_i)

In this case, the sub-plan has no pre-conditions so this part of the conditions of the sub-plan is automatically satisfied.

The next step in the analysis process is to walk through the AST representing the actions performed by the loop. Here, this is just a simple echo statement resulting in a *Display* action. The result of this action is the following fact.

OnPage("Hello",1)

This is the state of the program at the end of execution of the rule. When comparing against the post-conditions of the sub-plan in *Figure 7.4*, it can be seen that it is satisfied when x=1. Therefore, the following fact is created.

LoopBodyOK(ForId1)

At this point, a few more rules are utilised to specify the fact that repeating a loop a given number of times results in the actions within the loop being performed for all values of a certain variable. These rules are given in *Figure 7.7*.

The first rule in this figure is activated at this time to create the following fact.

RepeatLoop(ForId1,1,5,1)

The second and third rules are activated only if the loop iterates through all integer values between a starting and ending value. In this case, the second rule is activated, resulting in the following fact.

RepeaAll(ForId1,1,5)


```

RepeatLoop(loopId1,startValue,endValue,incrementValue)
    ← HasForStartValue(loopId1,startValue)
      ∧ HasForEndValue(loopId1,endValue)
      ∧ HasForIncrement(loopId1,incrementValue)

RepeatAll(loopId1,startValue,n)← RepeatLoop(loopId1,startValue,n,1)

RepeatAll(loopId1,startValue,n)← RepeatLoop(loopId1,n,startValue,-1)

∀ value_i [(start≤value_i≤n) → ActionEffects]
    ← repeat(ActionEffects,loopId1)
      ∧ RepeatAll(loopId1,start,n)
      ∧ HasForVariable(loopId1,varId_i)
      ∧ HasValue(varId_i,value_i)

```

Figure 7.7. Rules to consolidate results of loop execution.

The final rule in *Figure 7.7* is a generalised rule that specifies that the effects of the loop are valid for all values of the counter variable. In this case, the *ActionEffects* is actually the result of a single action and is the *OnPage("Hello",count)* fact where count represents the value of the counter for displaying elements at the start of each loop. Therefore, the final rule results in the following facts.

$$\forall val_i [(1 \leq val_i \leq 5) \rightarrow OnPage("Hello",count)]$$

The resultant state is the final state of the system. When comparing this against the constraints in *Figure 7.4*, it can be seen that they are satisfied when FORID1=ForId1. Similarly, the goal is satisfied when j=val_i and Y=count. Therefore, this program is identified as correct.

7.2.4 Unnecessary Statements in Loops

As in the case of other constructs, it is necessary to ensure that programs that contain loops do not contain any unnecessary statements. Since loops are handled similar to functions, by using conditions of sub-plans, unnecessary statements are identified using a similar method to that described in Section 6.2.6. A new status is created each time a loop is encountered. Any statuses created during the analysis of the loop are linked to the initial state of the loop. Once the conditions of a sub-plan

are satisfied as explained in Section 7.2.3, a new status is created before the *LoopBodyOK* predicate is created. This status is linked to the status where the conditions of the sub-plan were satisfied. However, in the case of loops, there is an additional linking of statuses. As described in Section 7.2.3, several facts are created through rules after the *LoopBodyOK* predicate. These facts depend on previously created facts and therefore, the status at this point is also linked to any statuses that created the facts leading to these new facts.

The flow of statuses for the first program in Table 7.1 is shown in *Figure 7.8*. No initial status is present in this program since no program statements are supplied. A new status, Status 1, is created when the *for* loop is encountered. The first component of the *for* loop assigns a value to the counter variable. Since this is an assignment operation, a new status, Status 2 is created. This status is linked to the main status of the *for* loop, Status 1. Next, as described in Section 7.2.3, the increment operation is activated, resulting in a new status, Status 3. Since this action occurs on a previously created variable *\$i*, the state is which it was created, Status 2 is linked to the current status. This status is also linked to Status 1 since it is part of the loop. The echo statement within the loop results in another status, Status 4. Since this is also part of the loop, it is again linked to the main status of the *for* loop, Status 1. When the conditions of the sub-plan are satisfied, a new status, Status 5 is created. Since Status 4 is the status where the conditions of the sub-plan are satisfied, this is linked to Status 5. No other program statements are encountered during the analysis of the program so the final status, where the overall goal is satisfied is Status 5. However, more rules are activated during this status for consolidating the actions within the loop. As described in Section 7.2.3, the *RepeatLoop* predicate is created at this point. It can be seen from *Figure 7.7* that this depends on the *HasForIncrement* predicate which is created during Status 3. Therefore, a link is created between the current status, Status 5 and Status 3. By examining *Figure 7.8*, it can be seen that there is a path from all existing statuses to this final status, Status 5. This indicates that no unnecessary statements are present in the program.

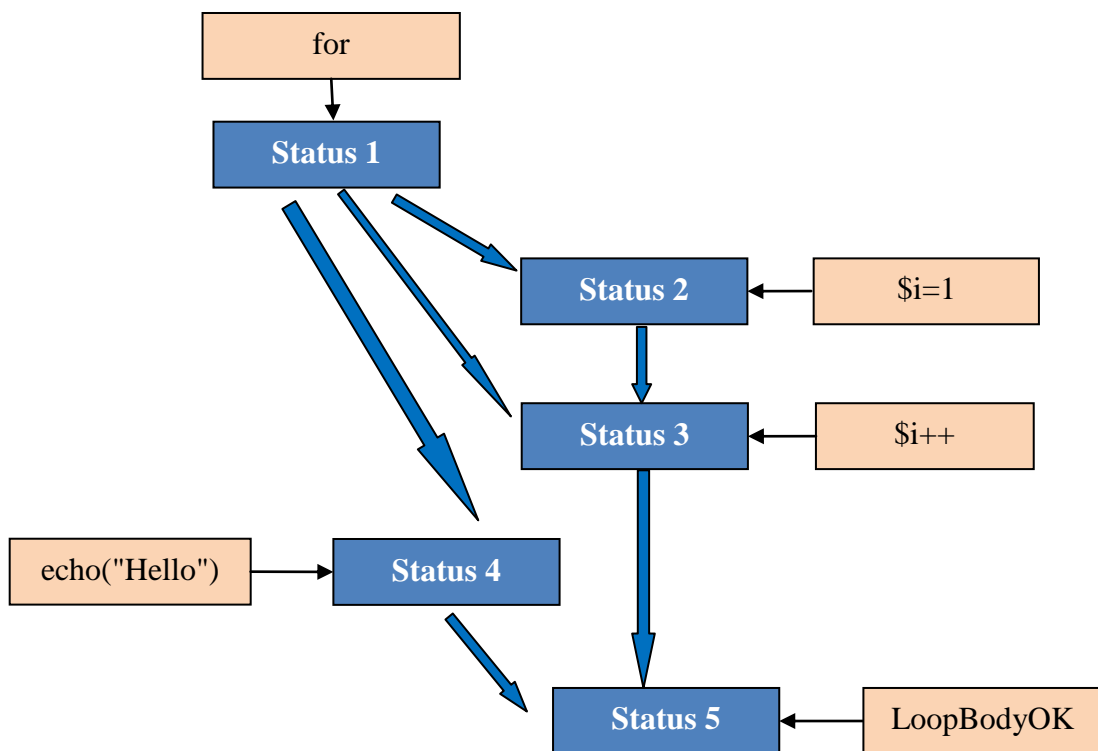


Figure 7.8. Flow of statuses example program for loops.

7.2.5 While Loops that Behave as For Loops

It is a known fact among programmers that any *for* loop can be converted into an equivalent *while* loop. For example, the first *for* loop in Table 7.1 can be written using an equivalent *while* loop as shown in Table 7.5. Therefore, the analysis of such *while* loops is similar to that of the equivalent *for* loop. The only difference occurs when walking the AST. When a *while* loop is encountered, it is first checked to see whether it is indeed a definite loop before analysis proceeds. In other words, it is checked to see whether it has a condition which refers to a variable contained within a *LessExpr*, *LessEqualExpr*, *GreaterExpr* or *GreaterEqualExpr*. It is next checked to see whether the variable within the condition contains an initial value before the *while* loop is reached. A final check is performed to see whether the same variable is changed within the loop in a manner where the change happens for all possible situations. If all these conditions are met, the system recognises that it can analyse the while loop. Analysis then proceeds as in Section 7.2.3. The initial value and the increment of the loop variable are set based on the statements identified during the earlier check. A more detailed analysis of a solution to the exercise in Figure 7.3 written using a while loop can be found in Appendix G.

Table 7.5

Equivalent For and While Loops

<i>For Loop</i>	<i>While Loop</i>
<pre>for(\$i=1;\$i<=5;\$i++) { echo("Hello"); }</pre>	<pre>\$i=1; while(\$i<=5) { echo("Hello"); \$i++; }</pre>

7.3 SPECIAL SITUATIONS

This section looks at some special situations that occur when analysing definite loops. Section 7.3.1 looks at loops that iterate for a pre-defined number of times but the value of the counter variable is changed according to an arithmetic sequence as described in Section 7.1. Section 7.3.2 looks at loops where the results of the execution of one iteration of the loop depend on the results of the previous iterations as described in Section 7.1.

7.3.1 Loops Where the Counter Variable Changes According to an Arithmetic Sequence

Sometimes, loops with counters are used to create loops that iterate for a fixed number of times, but the actual statements of concern within it do not execute for all integer values of the counter variable which are within the specified range. Beginners often use two methods to achieve this. Table 7.6 shows examples of using both these methods to display multiples of 10 between 1 and 100. The first program does this by incrementing the counter variable by 10 after each iteration. The second program increments the counter variable by 1 after each iteration but checks to see whether it is divisible by 10 before executing the program statements. The PHP ITS handles this type of situation by using rules to convert between these forms.

Figure 7.9 shows the overall goal for this program. As described in Section 7.2.2, the functionality of the loop is specified using conditions of a sub-plan. However, in this case, it can be seen that the functionality of the loop is different based on which method from Table 7.6 is used. In such a situation, it is possible to specify conditions for more than one sub-plan for the same *LoopBodyOK* predicate. In this case, conditions of two sub-plans have been given for the *LoopBodyOK(FORID1)* predicate. The first one refers to a situation similar to the

first program and the second one to a situation similar to the second program. If both the pre-conditions and post-conditions of one of the sub-plans are satisfied, the requirement for the loop is considered to be met. The *LoopBodyOK* predicate is then created.

Table 7.6

Examples of Loops That Do Not Execute for all Integer Values of the Counter Variable within the Specified Range

<i>Increment Other Than One</i>	<i>Modulus for Counter Variable</i>
<pre>for(\$i=10;\$i<=100;\$i+=10) { echo(\$i); }</pre>	<pre>for(\$i=1;\$i<=100;\$i++) { if(\$i%10==0) { echo(\$i); } }</pre>

Here, the value of the counter variable is used within the loop. Therefore, the fact that the counter variable already contains a value becomes important within the loop. This fact is expressed as a pre-condition for both the sub-plans of the loop.

Another important difference between these programs and the programs described previously is the fact that the order of the output is important. The multiples of 10 need to be displayed in ascending order. This requirement is captured in the overall goal by specifying that a variable RC has a value of COUNT_NEW and 1 is added to this value in order to find the new position for the *OnPage* predicate.

In this case, the left hand side of the implication in the goal contains two components. The first component specifies that the output should only occur if the value is divisible by 10 i.e. if the modulus of the value of the counter variable and 10 is 0. The second component specifies that the value should be between 10 and 100 (inclusive). This ensures that the goal specifies that only multiples of 10 between 10 and 100 are displayed.

7.3.1.1 Analysis of First Program

Consider how the first program in Table 7.6 is analysed by the system. As explained in Section 7.2.3 the following facts are created in the system when the *for*

loop is encountered. Again note that only facts pertinent to this analysis are presented here.

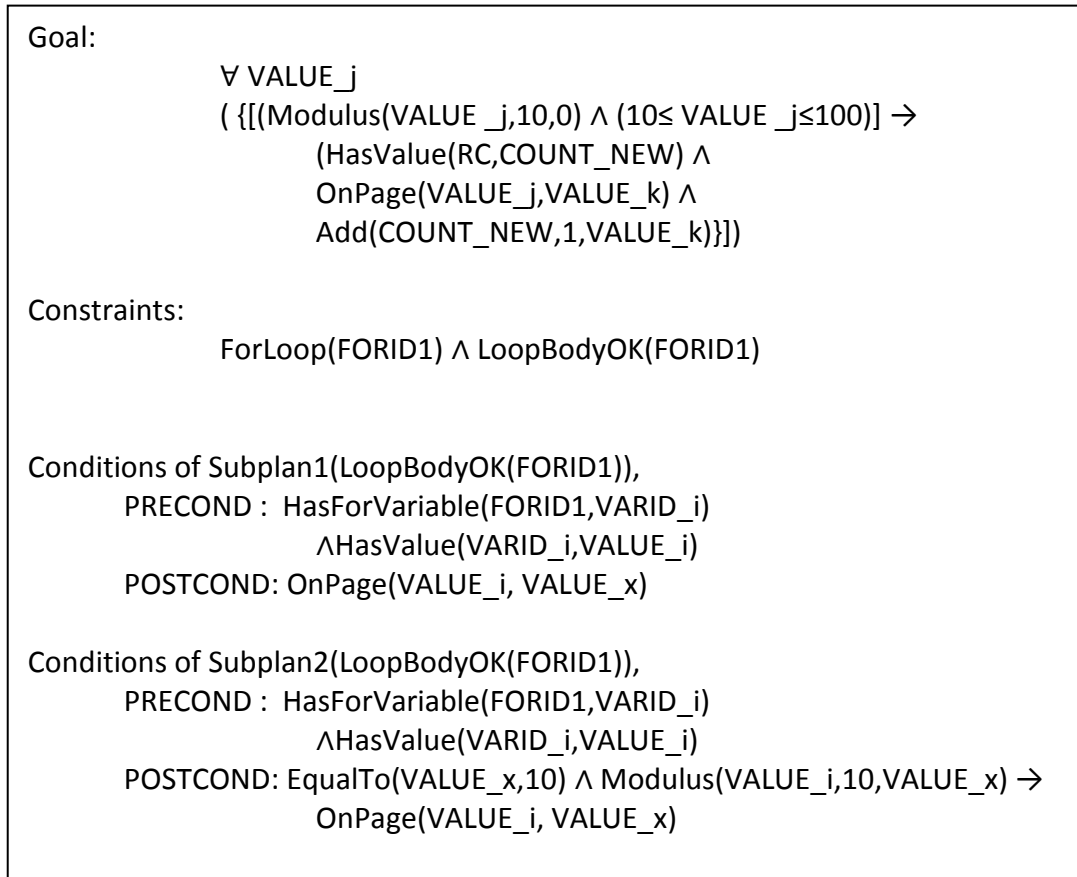


Figure 7.9. Overall goal for example program for loops that do not execute for all values of the counter variable.

HasName(VarId1, 'i')

HasValue(VarId1, 10)

HasInitialValue(VarId1, 10)

HasLoopVariable(ForId1, VarId1)

HasForStartValue(ForId1, 10)

HasId(LessEqualExpr(VarExprId1, LitExprId1), ExprId1)

HasVariable(VarExprId1, VarId1)

HasLiteral(LitExprId1, LitId1)

HasLitValue(LitId1, 100)

HasLoopCondition(ForId1, ExprId1)

Using the rules in *Figure 4.8*, the *ValueOf* the *LiteralExpr* is found, resulting in the following fact.

$$\textit{ValueOf}(\textit{LitExprId1},100)$$

Using the second rule in *Figure 7.5*, the following fact is created.

$$\textit{HasForEndValue}(\textit{ForId1},100)$$

Next, the update condition of the *for* loop is analysed using the procedure in 4.6.2. The resultant *Assign* action creates the following fact which is relevant to this analysis.

$$\textit{HasValue}(\textit{VarId1},20)$$

Since this is the value of the loop variable at the end of the first iteration, the following fact is created.

$$\textit{HasForFirstLoopValue}(\textit{ForId1},20)$$

Next, the rule in *Figure 7.6* is activated, resulting in the following fact.

$$\textit{HasForIncrement}(\textit{ForId1},10)$$

Now, the loop itself is analysed. Using the notation described in Section 7.2.3, the effect of the overall loop can be written as below.

$$\textit{repeat}(\textit{ForActionEffects},\textit{ForId1})$$

The program statements within the loop are next analysed against the conditions of the sub-plan. In order to analyse the statements within the loop it is first necessary to consider starting values for each loop iteration for all variables that already exist. This is achieved by assigning symbolic values to all existing variable at this point. Let the value of $\$i$ at the beginning of each iteration be $\textit{val_i}$. Let the id of the variable counting the display elements be $\textit{VarId_rc}$ and the value of this variable at the beginning of each iteration be $\textit{val_rc}$. Then, the following facts are created. Note that these facts are only true within the loop.

$$\textit{HasValue}(\textit{VarId1},\textit{val_i})$$
$$\textit{HasIterationValue}(\textit{ForId1},\textit{VarId1},\textit{val_i})$$
$$\textit{HasValue}(\textit{VarId_rc},\textit{val_rc})$$
$$\textit{HasIterationValue}(\textit{ForId1},\textit{VarId_rc},\textit{val_rc})$$

It can be seen that the pre-condition of both the sub-plans are satisfied when $FORID1=ForId1$, $VARID_i=VarId1$ and $VALUE_i=val_i$. Therefore, it is necessary to check the conditions for both the sub-plans to see whether the post-conditions are satisfied.

Now, a *Display* action occurs due to the echo statement. This results in the following facts.

$$OnPage(val_i,rc2) \text{ where } Add(rc,1,rc2)$$

When considering the post-condition of both the sub-plans it can be seen that it is satisfied for the first sub-plan when $VALUE_x=rc2$. Since the conditions of one of the sub-plans are satisfied, the following fact is created.

$$LoopBodyOK(ForId1)$$

Next, the rules specified in *Figure 7.7* are activated to create the following fact.

$$RepeatLoop(ForId1,10,100,10)$$

It is necessary to compare the existing facts into the for-all form in order to compare against the overall goal. A set of rules, as shown in *Figure 7.10* are used for this purpose.

Now, the first rule in this figure is used to create the following facts.

$$\forall val_i ([Modulus(val_i,10,0) \wedge (10 \leq val_i \leq 100)] \rightarrow ForActionEffects)$$

But in this case, the *ForActionEffects* is actually the result of a single action and is the *OnPage(val_i,rc2)* fact so the following fact is created.

$$\forall val_i ([Modulus(val_i,10,0) \wedge (10 \leq val_i \leq 100)] \rightarrow OnPage(val_i,rc2))$$

So it can be seen that the overall goal is satisfied when $FORID1=ForId1$ and the goal is satisfied when $VALUE_j=val_i$, $RC=VarId_rc$, $COUNT_NEW=rc$ and $VALUE_k=rc2$. Therefore, the program is identified as correct.

7.3.1.2 Analysis of Second Program

Next consider how the second program in *Table 7.6* is analysed. The *for* loop is analysed in the same way as above resulting in the following facts being created.

$\forall \text{value_i}$ $([\text{Modulus}(\text{value_i}, \text{incrementValue}, 0) \wedge (\text{start} \leq \text{value_i} \leq \text{n})] \rightarrow \text{ForActionEffects})$ $\leftarrow \text{repeat}(\text{ForActionEffects}, \text{loopId1})$ $\wedge \text{RepeatLoop}(\text{loopId1}, \text{startValue}, \text{n}, \text{incrementValue})$ $\wedge \text{HasForVariable}(\text{loopId1}, \text{varId_i})$ $\wedge \text{HasValue}(\text{loopId1}, \text{value_i})$ $\text{repeat}(\text{ActionEffects}, \text{loopId1}) \wedge \text{RepeatLoop}(\text{loopId1}, \text{newStart}, \text{newEnd}, \text{newInc}) \wedge$ $(\text{newStart} = \text{newInc}) \wedge \text{Modulus}(\text{end}, \text{newInc}, \text{x}) \wedge \text{Subtract}(\text{end}, \text{x}, \text{newEnd})$ $\leftarrow \text{repeat}([\text{EqualTo}(\text{Modulus}(\text{value_i}, \text{newInc}), 0) \rightarrow \text{ActionEffects}], \text{loopId1})$ $\wedge \text{HasForVariable}(\text{loopId1}, \text{varId_i})$ $\wedge \text{RepeatLoop}(\text{forId1}, \text{start}, \text{end}, \text{inc})$

Figure 7.10. Rules for consolidating loops that do not execute for all values of the counter variable.

HasName(VarId1, 'i')

HasValue(VarId1, 1)

HasInitialValue(VarId1, 1)

HasLoopVariable(ForId1, VarId1)

HasForStartValue(ForId1, 1)

HasId(LessEqualExpr(VarExprId1, LitExprId1), ExprId1)

HasVariable(VarExprId1, VarId1)

HasLiteral(LitExprId1, LitId1)

HasLitValue(LitId1, 100)

HasLoopCondition(ForId1, ExprId1)

ValueOf(LitExprId1, 100)

HasForEndValue(ForId1, 100)

HasForFirstLoopValue(ForId1, 2)

HasForIncrement(ForId1, 1)

Now, the loop itself is analysed. Using the notation described in Section 7.2.3, the effect of the overall loop can be written as below.

repeat(ForActionEffects,ForId1)

As before let the value of \$i at the beginning of each iteration be val_i. Let the id of the variable counting the display elements be VarId_rc and the value of this variable at the beginning of each iteration be val_rc. Then, the following facts are created. Note that these facts are only true within the loop.

HasValue(VarId1,val_i)

HasIterationValue(ForId1,VarId1,val_i)

HasValue(VarId_rc,val_rc)

HasIterationValue(ForId1,VarId_rc,val_rc)

It can be seen that the pre-condition of both the sub-plans are satisfied when FORID1=ForId1, VARID_i=VarId1 and VALUE_i=val_i. Therefore, it is necessary to check both the sub-plans to see whether the post-conditions are satisfied.

Next, a selection is encountered and is analysed as described in 5.2. Let the id of the created *EqualExpr* expression be ExprId2. Also let the id of the *ModulusExpr* on the left hand side of this expression be ModExprId1 and the id of the *LiteralExpr* on the right hand side be LitExprId2. Also let the id of the created Literal be LitId2. Then, the following facts are created.

HasId(EqualExpr(ModExprId1,LitExprId2),ExprId2)

HasLiteral(LitExprId2,LitId2)

HasLitValue(LitId2,0)

Let the ids of the two expressions on either side of the *ModulusExpr* be VarExprId3 and LitExprId3 respectively. Let the id of the corresponding Literal be LitId3. Then, the following facts are created.

HasId(ModulusExpr(VarExprId3,LitExprId3),ModExprId1)

HasLiteral(LitExprId3,LitId3)

HasLitValue(LitId3,10)

HasVariable(VarExprId3,VarId1)

Then, the *ValueOf* of the various expressions are found as below.

ValueOf(LitExprId2,0)

$ValueOf(LitExprId3,10)$

$ValueOf(VarExprId3, val_i)$

$ValueOf(ModExprId1, val_x)$ where $Modulus(val_i, 10, val_x)$

When the *if* condition is true, the following fact is created.

$ValueOf(ExprId2, True)$

Then, the rules in *Figure 5.4* results in the following predicate.

$EqualTo(val_x, 0)$

When this condition is satisfied, a *Display* action occurs due to the echo statement resulting in the following facts.

$OnPage(val_i, rc2)$ where $Add(rc, 1, rc2)$

Therefore, the overall result of the loop can be expressed as below.

$EqualTo(val_x, 0) \wedge Modulus(val_i, 10, val_x) \rightarrow OnPage(val_i, rc2)$

Therefore, the post-condition of the second sub-plan is satisfied when $VALUE_x=rc2$. Since the conditions of one of the sub-plans is satisfied, the following fact is created.

$LoopBodyOK(ForId1)$

In this case, the effect of the loop is actually the overall result given above so the repetition can be expressed as below.

$repeat(EqualTo(val_x, 0) \wedge Modulus(val_i, 10, val_x) \rightarrow OnPage(val_i, rc2), ForId1)$

Next, the rules specified in *Figure 7.7* are activated to create the following fact.

$RepeatLoop(ForId1, 1, 100, 1)$

Now, the second rule in *Figure 7.10* is activated, resulting in the following facts.

$repeat(OnPage(val_i, rc2), ForId1)$

$RepeatLoop(ForId1, 10, 100, 10)$

Next, the first rule in *Figure 7.10* is activated, resulting in the following facts.

$\forall val_i ([Modulus(val_i, 10, 0) \wedge (10 \leq val_i \leq 100)] \rightarrow ForActionEffects)$

But in this case, the *ForActionEffects* is actually the result of a single action and is the *OnPage(val_i,rc2)* fact so the following fact is created.

$$\forall val_i ([Modulus(val_i,10,0) \wedge (10 \leq val_i \leq 100)] \rightarrow OnPage(val_i,rc2))$$

So it can be seen that the overall goal is satisfied when FORID1=ForId1 and the goal is satisfied when VALUE_j=val_i, RC=VarId_rc, COUNT_NEW=rc and VALUE_k=rc2. Therefore, this program is also identified as correct.

7.3.2 Loop where the Execution of Statements Depends on the Results of Previous Iterations

As mentioned in Section 7.1, the execution of the statements within some loops depends on the result of the previous iteration of that loop. This section explores how such programs are analysed in the PHP ITS.

7.3.2.1 Factorial as Repeated Multiplication

In order to understand how such a loop is analysed, consider the PHP exercise defined in *Figure 7.11*. An example solution to this exercise is given in *Figure 7.12*.

Write a PHP code segment to find the factorial of a number and store the value into a new variable. Use a for loop to perform the calculation considering that a factorial of a number is the result of multiplying integers from 1 to that number. Note that when execution reaches the point where the code segment needs to be written, the variable \$num contains the number whose factorial needs to be found.

Figure 7.11. Example exercise for loops where execution depends on previous iterations.

```
$factorial=1;
for($i=1;$i<=$num;$i++)
{
    $factorial*=$i;
}
```

Figure 7.12. Example solution for factorial exercise.

Figure 7.13 shows the initial state and the overall goal for this exercise. The initial state specifies an initial symbolic value for the variable \$num as described in Section 4.5.1. The goal specifies that a variable with a value VALUE_f should exist

where VALUE_f is the factorial of the initial value of \$num. Note that *Factorial* here is a predicate similar to the *Add* predicate defined in Section 4.4.1.1. It is important to note that the goal specification in this case is different to the goal specification in 7.2.2 where it was specified as a \forall condition. In this case, the execution of one iteration depends on previous iterations and therefore, an aggregate is calculated. This means that the final outcome is in aggregate form as shown through the goal specification. This aggregate can be obtained in some other way, for example by directly calculating the factorial using a mathematical function. The constraints are used here to ensure that a loop of the appropriate form was used to perform the actual calculate. It specifies that a correctly functioning *for* loop should be used.

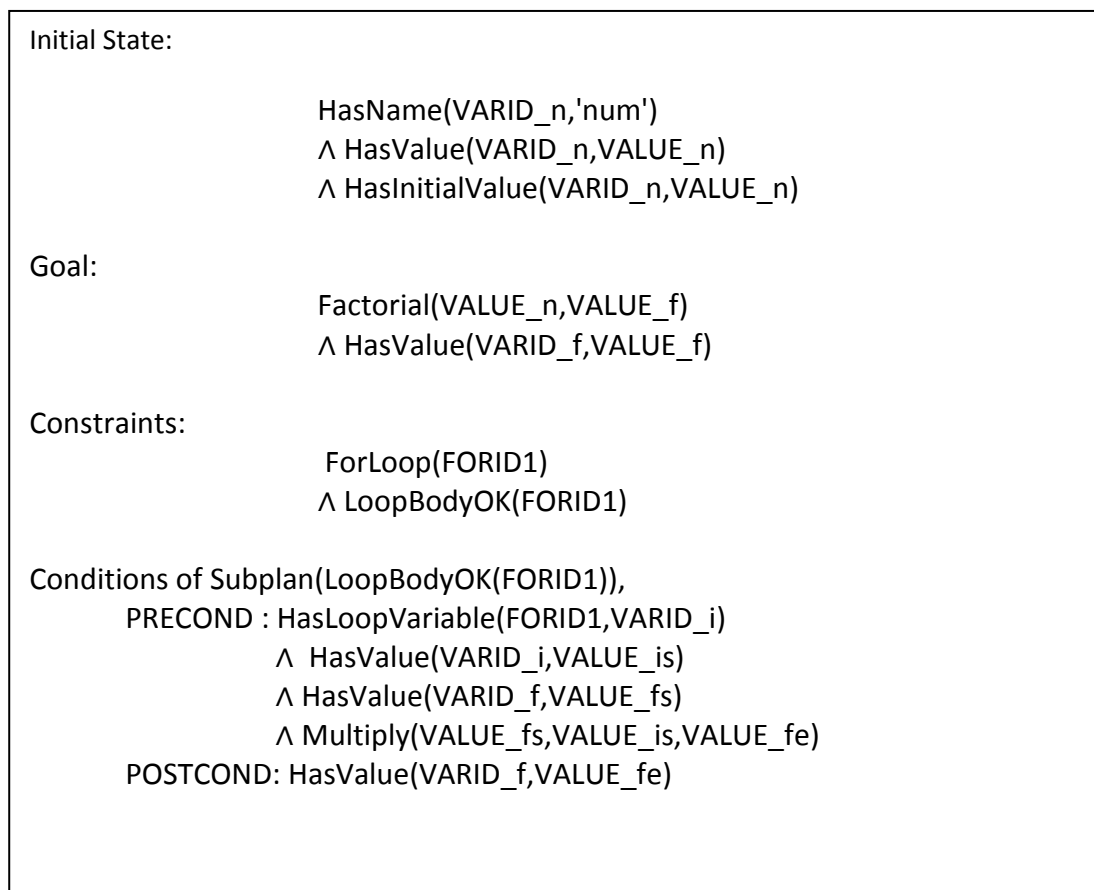


Figure 7.13. Initial state and overall goal for factorial exercise.

The conditions of the sub-plan specify what the loop should accomplish. The loop should multiply the loop variable with the variable that holds the result and store the new value to the result variable. In order to do this, both the loop variable and the variable holding the result should have a value at the beginning of the loop.

This is specified as the pre-condition of the loop. At the end of the execution of the loop, the variable holding the result should contain the multiplied value as described above. This fact is shown in the post-condition of the sub-plan.

Consider how the program in *Figure 7.12* is analysed. First, the initial state results in the following facts being created in the system. Let the id of the created *Variable* be *VarId1* and the symbolic value assigned to it be *val_n*. Note that only facts pertinent to this analysis are presented here.

HasName(VarId1,'num')

HasValue(VarId1,val_n)

HasInitialValue(VarId1,val_n)

The first assignment statement activates an *Assign* action. Let the id of the newly created *Variable* be *VarId2*. Let the id of the *LiteralExpr* on the right hand side be *LitExprId1* and the id of the *Literal* be *LitId1*. Then, the following facts are created.

HasLiteral(LitExprId1,LitId1)

HasLitValue(LitId1,1)

Then, the *ValueOf* of the *LiteralExpr* is found as below.

ValueOf(LitExprId1,1)

The *Assign* action then results in the following facts being created.

HasName(VarId2,'factorial')

HasValue(VarId2,1)

HasInitialValue(VarId2,1)

Next, the *for* loop is encountered and analysed as described in Section 7.2.3. Let the id of the created counter variable be *VarId3* and the id of the loop be *ForId1*. Also let the id of the *LessEqualExpr* be *ExprId1*. Let the ids of the *VariableExprs* on either side of this expression be *VarExprId1* and *VarExprId2* respectively. Then, the following facts are created as described above.

HasName(VarId3,'i')

HasValue(VarId3,1)

HasInitialValue(VarId3,1)

HasLoopVariable(ForId1,VarId3)

HasForStartValue(ForId1,1)

HasId(LessEqualExpr(VarExprId1, VarExprId2),ExprId1)

HasVariable(VarExprId1,VarId3)

HasVariable(VarExprId2,VarId1)

HasLoopCondition(ForId1,ExprId1)

Using the rules in *Figure 4.8*, the *ValueOf* VarExprId2 is found, resulting in the following fact.

ValueOf(VarExprId2,val_n)

Using the second rule in *Figure 7.5*, the following fact is created.

HasForEndValue(ForId1, val_n)

Next, the update condition of the *for* loop is analysed using the procedure in 4.6.2. The resultant *Assign* action creates the following fact which is relevant to this analysis.

HasValue(VarId3,2)

Since this is the value of the loop variable at the end of the first iteration, the following fact is created.

HasForFirstLoopValue(ForId1,2)

Next, the rule in *Figure 7.6* is activated, resulting in the following fact.

HasForIncrement(ForId1,1)

Now, the loop itself is analysed. Using the notation described in Section 7.2.3, the effect of the overall loop can be written as below.

repeat(ForActionEffects,ForId1)

Before the program statements within the loop can be analysed, all existing variables should be given symbolic values to specify what they contain at the beginning of each iteration as described in Section **Error! Reference source not found.** Let the value of \$i be val_i and the value of \$factorial be val_f at the

beginning of each iteration. Since the variable \$num does not change within the loop, a symbolic value for this variable is not required. Then, the following facts are created within the loop.

HasValue(VarId3, val_i)

HasIterationValue(ForId1, VarId1, val_i)

HasValue(VarId2, val_f)

HasIterationValue(ForId1, VarId2, val_f)

Next, it is necessary to check whether the pre-conditions in the sub-plan are satisfied. It can be seen that the existing facts satisfy the pre-conditions when FORID1=ForId1, VARID_i=VarId3, VALUE_is=val_i, VARID_f=VarId2 and VALUE_fs=val_f. The *Multiply(VALUE_fs, VALUE_i, VALUE_fe)* predicate is given as a pre-condition since it needs to be true for the assignment to occur. Since this is a mathematical fact, it will always be True. However, it is actually used to ensure that the correct value is assigned at the end of the loop.

Now, an *AssignMultiply* action is activated as described in Section 4.4.3. Since the variable on the left hand side already exists and is in scope, no new variable is created. However, it is assigned the value of the multiplication of its current value and the value of variable \$i, resulting in the following fact being created.

HasValue(VarId1, val_new) where *Multiply(val_f, val_i, val_new)*

It can be seen that the post-condition of the sub-plan is now satisfied when VALUE_fe=val_new, so the following fact is created.

LoopBodyOK(ForId1)

Next, the rules in *Figure 7.7* are executed to consolidate the actions performed by the loop, resulting in the following facts.

RepeatLoop(ForId1, 1, val_n, 1)

RepeatAll(ForId1, 1, val_n)

In this case, the *ActionEffects* is the result of the assignment which is the *HasValue(VarId1, val_new)* fact so the consolidated effect is as below.

$\forall val_i [(1 \leq val_i \leq val_n) \rightarrow HasValue(VarId1, val_new)]$

When considering the overall goal in *Figure 7.13*, it can be seen that, although the overall goal in the previous loops was specified in this manner, the overall goal here is specified using an aggregate form. In order to match these two states, it becomes necessary to use a specific rule to suit the current situation. In this case, the rule used is as shown in *Figure 7.14*.

By investigating the facts created in the system, it can be seen that this rule is now activated, resulting in the following fact.

HasValue(VarId1, val_fac) where *Factorial(val_n, val_fac)*

<pre> HasValue(varId_x,value_m) ← HasLoopStartValue(loopId,varId_x,1) ∧ HasIterationValue(loopId,varId_x,value_xf) ∧ Factorial(end,value_m) ∧ ∀ value_i [(start≤value_i≤end) → HasValue(varId_x,value_x) ∧ Multiply(value_xf,value_i,value_x)] </pre>

Figure 7.14. Rule to aggregate factorial as repeated multiplication.

When comparing this final state against the overall goal in *Figure 7.13*, it can be seen that it is satisfied when VALUE_f= val_fac, VARID_f=VarId2 and FORID1=ForId1. Therefore, the program segment is identified as correct.

7.3.2.2 *Multiplication as Repeated Addition*

In the analysis of the program above, it can be seen that the process of aggregating the for-all state to the necessary factorial state involved the use of a specific rule for this particular calculation (*Figure 7.14*). Therefore, although this same method of analysis can be used for other situations where such aggregations are performed, it becomes necessary to define specific rules for each such situation.

Such aggregations are usually used in cases where a mathematical definition involves such iteration. Another common example of such a situation where multiplication is treated as repeated addition. *Figure 7.15* shows the overall goal for an exercise where the student is required to write a program segment to multiply two numbers held in the variables \$a and \$b.

This goal specification is very similar to the one in *Figure 7.13*. In this case, the overall goal shows that the value stored in the variable should be the

multiplication of the initial values of the two variables. Here the program should add the value of one variable to a running variable and iterate the number of times of the other variable. It is possible to use the two variables provided in either direction, resulting in two possible sub-plans.

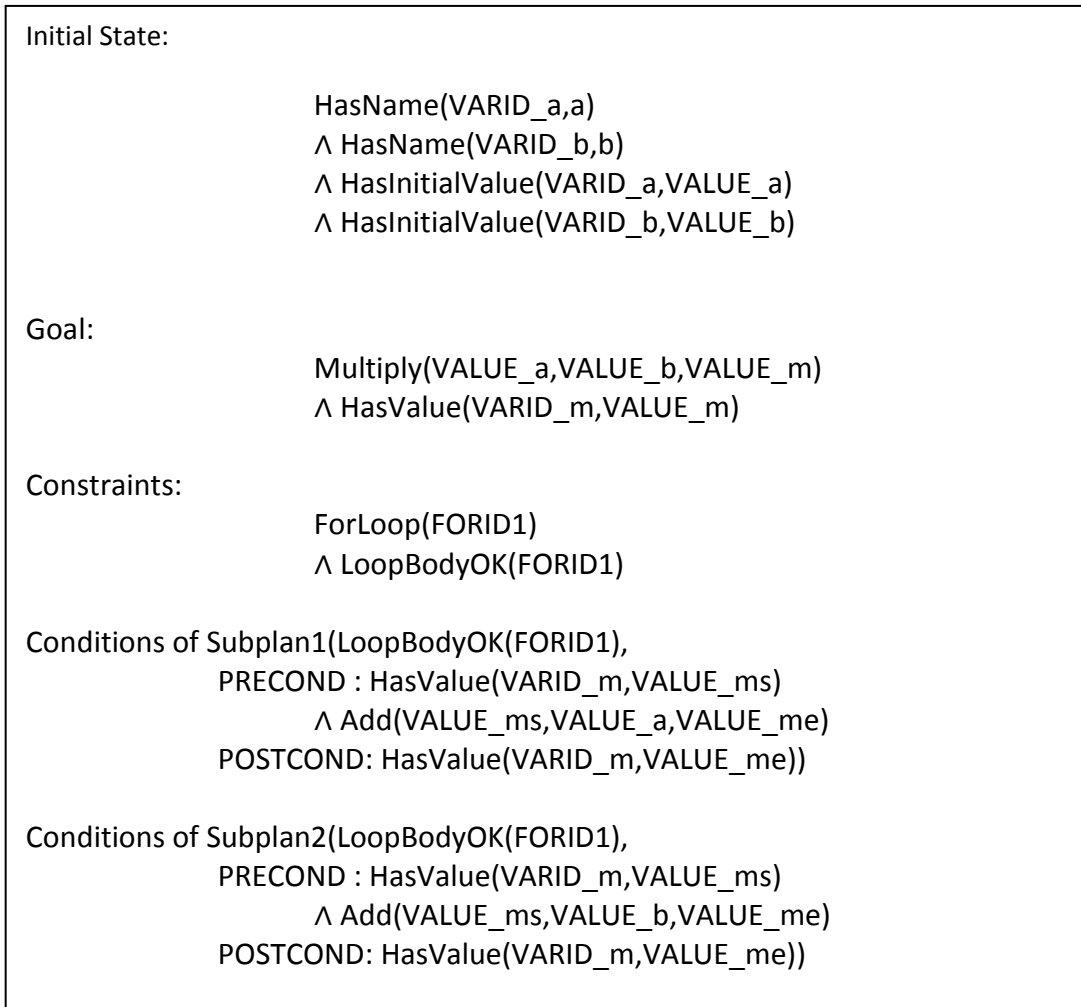


Figure 7.15. Initial state and overall goal for multiplication as repeated addition.

The detailed analysis of several solutions to this exercise can be found in Appendix G. One such solution is shown in Figure 7.16.

```

$multiply=0;
for($i=1;$i<=$b;$i++)
{
    $multiply+=$a;
}

```

Figure 7.16. Example solution for multiplication exercise.

In general, the analysis proceeds as described in Section 7.3.2.1 until it becomes necessary to aggregate the result. The rule used in this case is given in *Figure 7.17*. This is very similar to the rule in *Figure 7.14* with a few minor differences. The for-all part of the premises of the rule checks for an addition instead of a multiplication since repeated addition is being considered here. In this case, the number that is added repeatedly to the running variable is fixed and does not depend on the counter variable of the loop (*Figure 7.16*). Therefore, what matters is the number of times the loop is repeated and not the exact value of the counter variable of the loop. This is incorporated into the rule in *Figure 7.17* using mathematical predicates to specify that the resultant multiplication depends on the start and end values of the counter variable.

<pre> HasValue(varId_x,value_m) ← HasLoopStartValue(loopId,varId_x,0) ∧ HasIterationValue(loopId,varId_x,value_xf) ∧ Multiply(value_z,n,value_m) ∧ Subtract(end,start,n1) ∧ Add(n1,1,n) ∧ ∃ value_i [(start≤value_i≤end) → HasValue(varId_x,value_x) ∧ Add(value_xf,value_z,value_x)] </pre>
--

Figure 7.17. Rule to aggregate multiplication as repeated addition.

Similar rules can be written to handle many other mathematical functions that can be defined as repetitions. However, the exercises included in the PHP ITS only consider the cases where factorial is repeated multiplication and multiplication is repeated addition. Therefore, only rules to handle these two situations have been included in the KB.

7.4 COLLECTION BASED LOOPS THAT PERFORM SOME ACTION AGAINST EVERY ITEM IN THE COLLECTION INDEPENDENTLY WITHOUT SUMMARISING

The previous section discussed how the KB handled loops described as definite in Section 7.1. This section goes on to explain how the ideas used here are extended to handle one of the most common types of loops in real-world programming – collection based loop that perform some action against every item in the collection

without summarising (Stavely, 1993). As discussed in Section 7.1, such loops iterate through all the elements of a data structure. Since only basic PHP is taught by the PHP ITS, the only type of data structure considered during this thesis is an array. Therefore, this section describes how different statements that loop through the elements of an array are analysed.

Three types of constructs are typically used to iterate through the elements of an array in PHP: *for*, *while* and *foreach*. Both the *for* and *while* loops are similar to those used for definite loops. Therefore, the analysis process is similar to that described in Section 7.2.3. Such loops are discussed in Section 7.4.1. The other type of construct, the *foreach* construct is handled a little differently in the PHP ITS. The analysis process for such loops is described in Section 7.4.2.

7.4.1 For and While Constructs

The *for* and *while* constructs used to access array elements behave the same way as they do in definite loops. Therefore, the predicates used here are the same as those described in Section 7.2.1. However, very often the order of elements is important when dealing with arrays. In order to understand how this works, consider the example exercise given in *Figure 7.18*.

Write a PHP code segment to display all the elements of the \$myarray array in order. Note that when execution reaches the point where the code needs to be completed, the \$myarray array is initialised and contains three elements. Use a for loop to loop through these elements and display the contents.

Figure 7.18. Example exercise for for-each loop using the *for* construct.

Figure 7.19 shows the initial state and the overall goal for this exercise. In this case, the number of predicates is increased considerably since we are dealing with arrays and they require a large number of predicates to define the keys, elements and values. The somewhat lengthy initial state specifies that the array named ‘myarray’ contains three elements with indexes 0, 1 and 2 and values VALUE_1, VALUE_2 and VALUE_3.

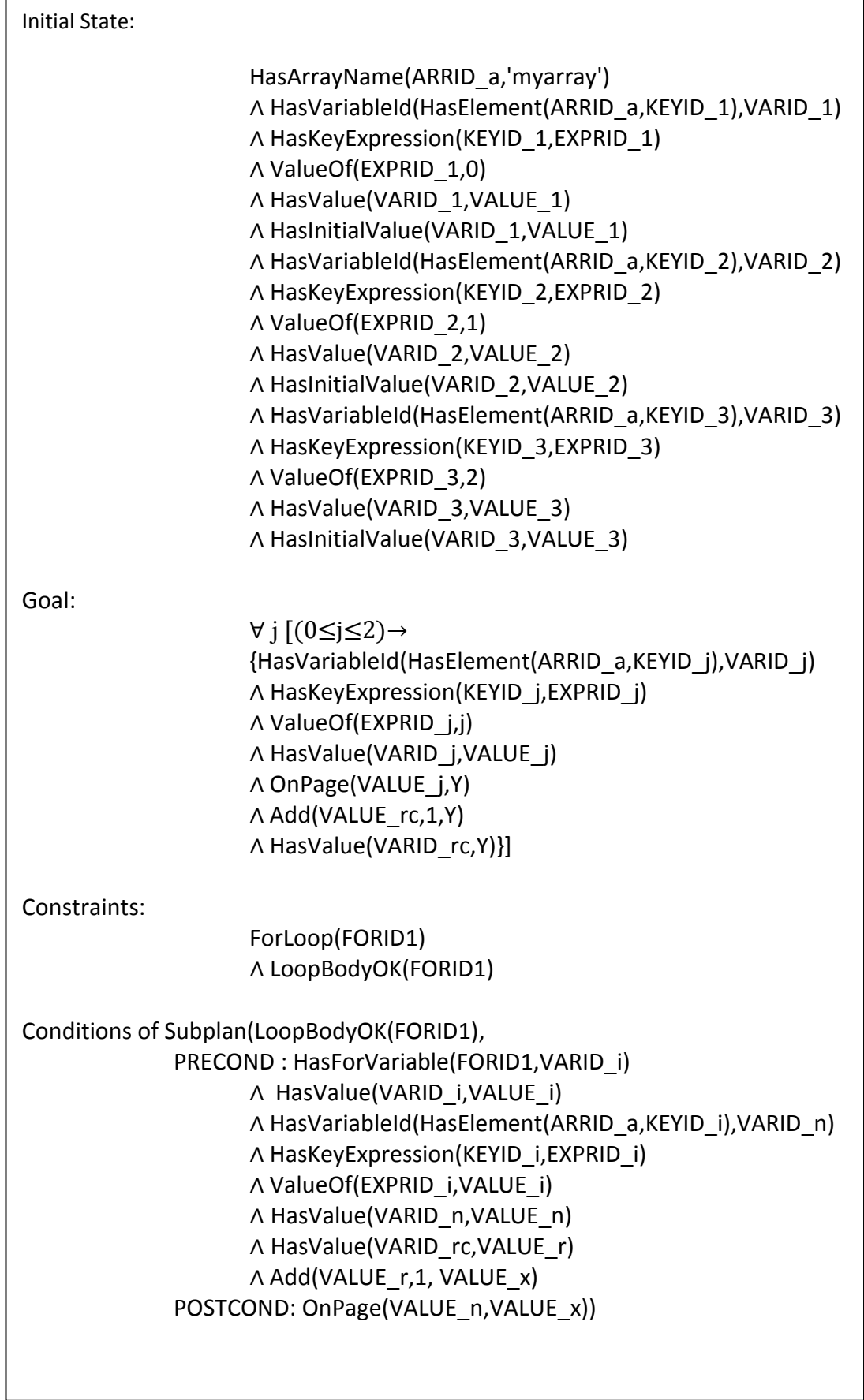


Figure 7.19. Initial state and overall goal for example exercise for for-each loop using *for* construct.

The goal in this case is very similar to that in *Figure 7.4* except for the fact that it contains the large number of predicates to handle arrays. The row counter variable that holds the current counter used in the *OnPage* predicate (Section 4.4.3) is used here to control the order. The goal specifies that 1 should be added to this before the relevant *OnPage* predicate is created.

The constraints and conditions of the sub-plan are also very similar to the previous case except for the row counter variable being included in the pre-conditions of the sub-plan.

Consider the solution to this exercise given in *Figure 7.20*. Before this program can be analysed, the facts relevant to the initial state are created as below. Let the id of the *Array* be *ArrId1*, the ids of the three *ArrayVariables* be *VarId1*, *VarId2* and *VarId3* and the ids of the relevant *Keys* be *KeyId1*, *KeyId2* and *KeyId3* respectively. Let the ids of the *Expressions* corresponding to these keys be *ExprId1*, *ExprId2* and *ExprId3* respectively. Let the symbolic values assigned to the three variables be *val_1*, *val_2* and *val_3* respectively.

```
for($i=0;$i<3;$i++)
{
    echo($myarray[$i]);
}
```

Figure 7.20. Example solution to exercise for for-each loop using for construct

HasArrayName(ArrId1,'myarray')

HasVariableId(HasElement(ArrId1,KeyId1),VarId1)

HasKeyExpression(KeyId1,ExprId1)

ValueOf(ExprId1,0)

HasValue(VarId1,val_1)

HasInitialValue(VarId1,val_1)

HasVariableId(HasElement(ArrId1,KeyId2),VarId2)

HasKeyExpression(KeyId2,ExprId2)

ValueOf(ExprId2,1)

$HasValue(VarId2, val_2)$
 $HasInitialValue(VarId2, val_2)$
 $HasVariableId(HasElement(ArrId1, KeyId3), VarId3)$
 $HasKeyExpression(KeyId3, ExprId3)$
 $ValueOf(ExprId3, 2)$
 $HasValue(VarId3, val_3)$
 $HasInitialValue(VarId3, val_2)$

Next, the *for* loop is analysed creating the following facts as explained in Section 7.2.3. Let the id of the loop variable be *VarId4* and the id of the loop be *ForId1*. Let the id of the *LessExpr* be *ExprId4* and the ids of the *VariableExpr* and the *LiteralExpr* on either side of it be *VarExprId1* and *LitExprId1* respectively. Also let the id of the created *Literal* be *LitId1*.

$HasName(VarId4, 'i')$
 $HasValue(VarId4, 0)$
 $HasInitialValue(VarId4, 0)$
 $HasLoopVariable(ForId1, VarId4)$
 $HasForStartValue(ForId1, 0)$
 $HasId(LessExpr(VarExprId1, LitExprId1), ExprId4)$
 $HasVariable(VarExprId1, VarId4)$
 $HasLiteral(LitExprId1, LitId1)$
 $HasLitValue(LitId1, 3)$
 $HasLoopCondition(ForId1, ExprId4)$

Using the rules in *Figure 4.8*, the *ValueOf* of the *LiteralExpr* is found, resulting in the following fact.

$ValueOf(LitExprId1, 3)$

Using the first rule in *Figure 7.5*, the following fact is created.

$HasForEndValue(ForId1, 2)$

Next, the update condition of the *for* loop is analysed using the procedure in 4.6.2. The resultant *Assign* action creates the following fact which is relevant to this analysis.

$$\text{HasValue}(\text{VarId4}, 1)$$

Since this is the value of the loop variable at the end of the first iteration, the following fact is created.

$$\text{HasForFirstLoopValue}(\text{ForId1}, 1)$$

Next, the rule in *Figure 7.6* is activated, resulting in the following fact.

$$\text{HasForIncrement}(\text{ForId1}, 1)$$

Now, the loop itself is analysed. Using the notation described in Section 7.2.3, the effect of the overall loop can be written as below.

$$\text{repeat}(\text{ForActionEffects}, \text{ForId1})$$

The program statements within the loop are next analysed against the conditions of the sub-plan. In order to analyse the statements within the loop it is first necessary to consider starting values for each loop iteration for all variables that already exist. Let the value of $\$i$ at the beginning of each iteration be val_i . Let the id of the variable counting the display elements be VarId_{rc} and the value of this variable at the beginning of each iteration be val_{rc} . Then, the following facts are created. Note that these facts are only true within the loop.

$$\text{HasValue}(\text{VarId4}, \text{val}_i)$$
$$\text{HasIterationValue}(\text{ForId1}, \text{VarId4}, \text{val}_i)$$
$$\text{HasValue}(\text{VarId}_{rc}, \text{val}_{rc})$$
$$\text{HasIterationValue}(\text{ForId1}, \text{VarId}_{rc}, \text{val}_{rc})$$

In this case, the loop analyses an array. At this point, it becomes necessary to analyse the statements within the loop to see whether the key corresponding to any array access is dependent on the loop variable or any other variable changed within the loop. Here, the key is dependent on the loop variable. Therefore, it becomes necessary to use the value of the loop variable to create a symbolic value for the key during each iteration. Let the id of the expression relevant to the key during each

iteration be ExprId_i. In this case, this is a variable expression referring to the loop variable so the following fact is created.

$$\text{HasVariable}(\text{ExprId}_i, \text{VarId}_4)$$

Next, the rule in *Figure 4.8* is used to calculate the *ValueOf* this expression, resulting in the following fact.

$$\text{ValueOf}(\text{ExprId}_i, \text{val}_i)$$

This is the expression that is linked to the key for each iteration. Let the id of the key for each iteration be KeyId_i. Then, the following fact is created.

$$\text{HasKeyExpression}(\text{KeyId}_i, \text{ExprId}_i)$$

As described in Section 6.1, the relationship between an array and a key is reified to create an *ArrayVariable*. Therefore, the link between the key for each iteration and the array is reified to create a new *ArrayVariable* with id VarId_n as below.

$$\text{HasVariableId}(\text{HasElement}(\text{ArrId}_1, \text{KeyId}_i), \text{VarId}_n)$$

For analysis purposes, it becomes necessary to consider the value of this variable at the beginning of each iteration as described in Section 7.2.3. Let the symbolic value assigned to this variable be val_n. Then, the following facts are created.

$$\text{HasValue}(\text{VarId}_n, \text{val}_n)$$

$$\text{HasIterationValue}(\text{ForId}_1, \text{VarId}_n, \text{val}_n)$$

It can be seen that the pre-condition of the sub-plan is satisfied when FORID1=ForId1, VARID_i=VarId4, VALUE_i=val_i, ARRID_a=ArrId1, KEYID_i=KeyId_i, EXPRID_i=ExprId_i, VARID_n=VarId_n, VALUE_n=val_n, VARID_rc=VarId_rc and VALUE_r=val_rc.

Now, a *Display* action occurs due to the echo statement. This results in the following facts.

$$\text{OnPage}(\text{val}_n, \text{rc}_2) \text{ where } \text{Add}(\text{rc}, 1, \text{rc}_2)$$

$$\text{HasValue}(\text{VarId}_{rc}, \text{rc}_2)$$

When considering the post-condition of the sub-plan it can be seen that it is satisfied when VALUE_x=rc2. Therefore the following fact is created.

LoopBodyOK(ForId1)

Next, the rules specified in *Figure 7.7* are activated to create the following fact.

RepeatLoop(ForId1,0,2,1)

RepeaAll(ForId1,0,2)

The final rule in *Figure 7.7* is next activated to result in the final rule results in the following facts. The *ForActionEffects* in this case are a combination of facts that lead to the *Display* action.

$\forall val_i [(0 \leq val_i \leq 2)$
 $\rightarrow HasVariableId(HasElement(ArrId1, KeyId_i), VarId_n)$
 $\wedge HasKeyExpression(KeyId_i, ExprId_i)$
 $\wedge ValueOf(ExprId_i, val_i)$
 $\wedge HasValue(VarId4, val_i)$
 $\wedge OnPage(val_n, rc2)$
 $\wedge Add(rc, 1, rc2)$
 $\wedge HasValue(VarId_rc, rc2)]$

The resultant state is the final state of the system. When comparing this against the overall goal in *Figure 7.19*, it can be seen that it is satisfied when FORID1=ForId1, j=val_i, ARRID_a=ArrId1, KEYID_j=KeyId_i, VARID_j=VarId_n, EXPRID_j=ExprId_i, VALUE_j=val_i, VARID_j=VarId4, VALUE_j=val_n, Y=rc2, VALUE_rc=rc and VARID_rc=VarId_rc. Therefore, this program is identified as correct.

While constructs are handled in a similar manner, as described in Section 7.2.5. The only difference is again the need to use facts related to arrays and to create facts relevant to these arrays at the beginning of the analysis of the loop.

7.4.2 Foreach Construct

As described above the elements of a collection can be accessed in PHP using the *foreach* construct. Although such loops behave in the same manner as the previously described loops logically, their different syntax makes it necessary to define a set of new predicates to handle them.

7.4.2.1 Predicate Definition

The predicates used for handling the *foreach* construct are shown in *Figure 7.21*. Every *foreach* loop iterates through an *Array*. The relationship between the loop and the array is maintained using the *HasForEachArray* predicate. The *foreach* loop refers to the element at the current position of the array. Since array elements are defined as a sub-type of *Variable* as described in Section 6.1, the *HasForEachVariable* predicate is used to model the relationship between the loop and the *ArrayVariable*. Sometimes, it is possible for *foreach* loops to refer to the value of the key of the current array variable. This relationship is established using the *HasForEachKey* predicate. The key used in a *foreach* construct is always a variable. Therefore, this is maintained as a *VariableExpr*. The *DoForEach* predicate is similar to the *RepeatLoop* predicate described in Section 7.2.1. It maintains a link between the loop itself and the value of the key and the element that each iteration accesses.

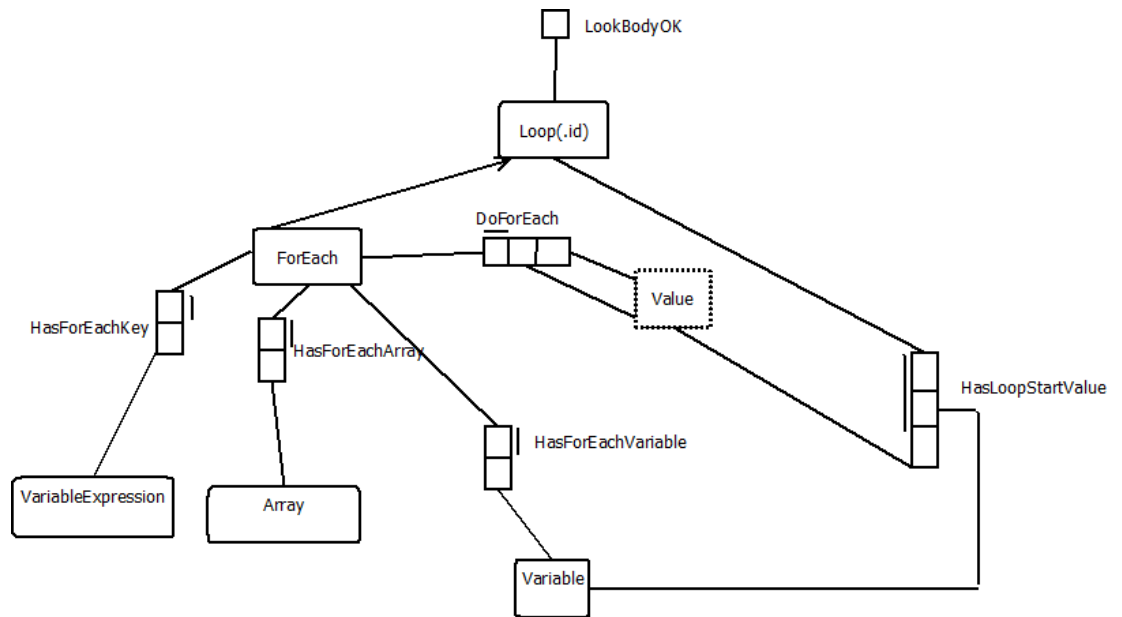


Figure 7.21. Predicates for handling the *foreach* construct.

7.4.2.2 Program Analysis

In order to understand how these predicates work, consider the example exercise shown in *Figure 7.18*. Assume that this exercise has been extended to require the position in the array to be displayed, along with the value for each array element. *Figure 7.22* shows an example solution to this exercise.

```

foreach($myarray as $key=>$value)
{
    echo($key)
    echo($myarray[$i]);
}

```

Figure 7.22. Example program for *foreach* construct.

In this case, the initial state is the same as that shown in *Figure 7.19*. The overall goal in this case needs to be expressed differently in order to handle that the *foreach* construct is used. The number of iterations depends on the number of elements in the array. Although it is possible to give an exact number in the overall goal since the number of elements are known, this causes a problem during program analysis. There is no possibility to write a generalised rule similar to *Figure 7.7* since a counter variable doesn't exist. Therefore, this knowledge base is only capable of handling programs that specifically require the student to use a *foreach* construct and a similar program written using any other construct is not accepted. The overall goal for this exercise is shown in *Figure 7.23*.

HasVariableId(HasElement(ArrId1,KeyId3),VarId3)

HasInitialValue(VarId2,val_2)

As before, the initial state results in the following facts.

HasArrayName(ArrId1,'myarray')

HasVariableId(HasElement(ArrId1,KeyId1),VarId1)

HasKeyExpression(KeyId1,ExprId1)

ValueOf(ExprId1,0)

HasValue(VarId1,val_1)

HasInitialValue(VarId1,val_1)

HasVariableId(HasElement(ArrId1,KeyId2),VarId2)

HasKeyExpression(KeyId2,ExprId2)

ValueOf(ExprId2,1)

HasValue(VarId2,val_2)

HasKeyExpression(KeyId3,ExprId3)

ValueOf(ExprId3,2)

HasValue(VarId3,val_3)

HasInitialValue(VarId3,val_3)

HasForEachArray(ForEachId1,ArrId1)

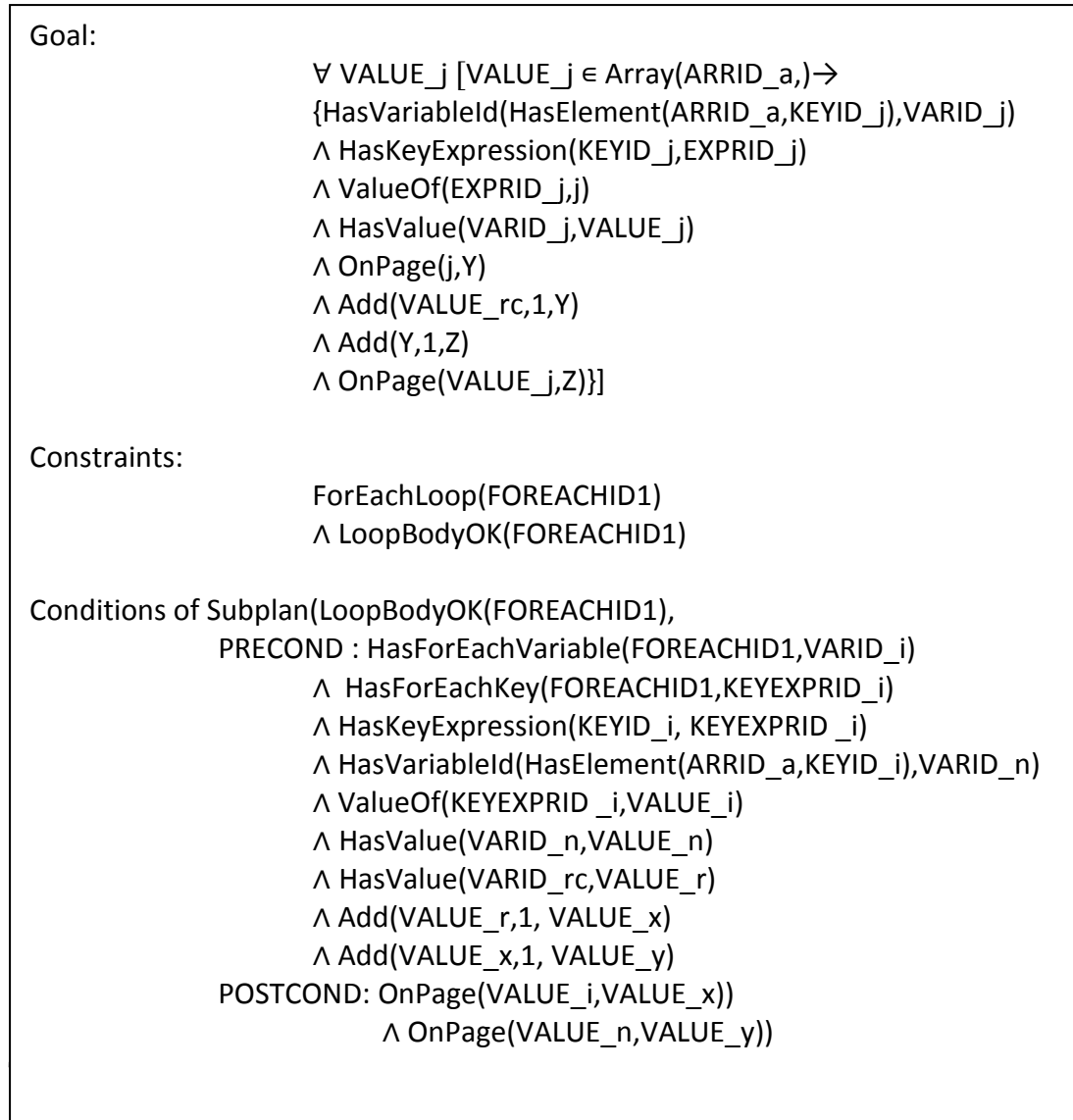


Figure 7.23. Overall goal for example exercise for *foreach* construct.

The *foreach* construct here uses both a key and a value. If no key is specified in the syntax, a symbolic key is automatically created. The key in any *foreach* construct is a *VariableExpr* related to a *Variable*. Let the ids of the *VariableExpr*

and *Variable* be *VariableExprId1* and *VarId4* respectively. Let the id of the *Key* be *KeyId1*. Then, the following facts are created.

HasForEachKey(ForEachId1,VarExprId1)

HasVariable(VarExprId1,VarId4)

This key expression is associated with a *Key* as described in Section 6.1. Let the id of the relevant *Key* be *KeyId1*. Then, the following fact is created.

HasKeyExpression(KeyId1,VarExprId1)

Now, the relationship between the *Array* and *Key* accessed by the *foreach* construct is reified into an *ArrayVariable* as described in Section 6.1. Let the id of the created *ArrayVariable* be *VarId5*. Then, the following fact is created.

HasVariableId(HasElement(ArrId1,KeyId1),VarId5)

But this is the variable that is accessed within the loop so the following fact is created.

HasForEachVariable(ForEachId1,VarId5)

Normally, an array variable is not assigned a name. However, in the case of the *foreach* construct, the values within the array are accessed using a variable name specified. Therefore, in this case, the specified name is assigned to the *ArrayVariable* as below.

HasName(VarId5,'value')

In order to analyse the loop, it is necessary to define values for all existing variables at the beginning of the loop. Here, the variables of concern within the loop are those referring to the key, the *ArrayVariable* and the counter used in the *Display* actions. For simplicity, only facts relevant to the initial value of these variables are given here.

HasValue(VarId4,val_i)

HasIterationValue(ForEachId1,VarId4,val_i)

HasValue(VarId5,val_n)

HasIterationValue(ForEachId1,VarId5,val_n)

HasValue(VarId_rc,rc)

HasIterationValue(ForEachId1,VarId_rc,rc)

Then, using the rules in *Figure 4.8*, the following fact is created.

ValueOf(VarExprId1,val_i)

Next, the loop itself is analysed. The results of the repetition of the loop can be expressed as below.

repeat(ForEachActionEffects,ForEachId1)

When considering the current state of the program, it can be seen that the pre-conditions of the sub-plan are satisfied when FOREACHID1=ForEachId1, VARID_i=VarId4, KEYEXPRID_i=VarExprId1, KEYID_i=KeyId1, ARRID_a=ArrId1, VARID_n=VarId5, VALUE_i=val_i, VALUE_n=val_n, VARID_rc=VarId_rc and VALUE_r=rc.

Here, the *ForEachActionEffects* are two echo statements. The first echo statement activates a *Display* action resulting in the following facts.

OnPage(val_i,rc2) where Add(rc,1,rc2)

Similarly, the second *Display* action results in the following facts.

OnPage(val_n,rc3) where Add(rc2,1,rc3)

So it can be seen that the post-conditions of the sub-plan are satisfied when VALUE_x=rc2 and VALUE_y=rc3. Therefore the sub-plan is satisfied and the following fact is created.

LoopBodyOK(ForEachId1)

In the case of the *foreach* construct, rules similar to those in *Figure 7.7* are used to consolidate the function of the loop. These rules are shown in *Figure 7.24*.

The following fact is created using the first rule in this figure.

DoForEach(ForEachId1,val_i,val_n)

Using the second rule in the figure, the following fact is created.

$\forall val_n [(val_n \in ArrId1) \rightarrow ForEachActionEffects]$

```

DoForEach(forEachId1,keyValue,elementValue)
  ← HasForEachArray(forEachId1,arrId1)
  ∧ HasForEachVariable(forEachId1,varId1)
  ∧ HasForEachKey(forEachId1,exprId1)
  ∧ HasKeyExpression(keyId1,exprId1)
  ∧ ValueOf(exprId1,keyValue)
  ∧ HasVariableId(HasElement(arrId1,keyId1),varId1)
  ∧ HasValue(varId1,elementValue)

∀ elementValue [(elementValue ∈ arrId1) → ForEachActionEffects]
  ← repeat(ForEachActionEffects,forEachId1)
  ∧ DoForEach(forEachId1,keyValue,elementValue)
  ∧ HasVariableId(HasElement(arrId1,keyId1),varId1)
  ∧ HasValue(varId1,elementValue)

```

Figure 7.24. Rules for consolidating *foreach* constructs.

But in this case, *ForEachActionEffects* is the combined results of the two *Display* actions so the final state can be written as below. Note that other facts that exist in the system are also included in this representation.

$$\begin{aligned}
& \forall val_n [(val_n \in ArrId1) \rightarrow \\
& \quad \wedge HasVariableId(HasElement(ArrId1,KeyId1),VarId5) \\
& \quad \wedge HasKeyExpression(KeyId1,VarExprId1) \\
& \quad \wedge ValueOf(VarExprId1,val_i) \\
& \quad \wedge HasValue(VarId5,val_n) \\
& \quad \wedge OnPage(val_i,rc2) \\
& \quad \wedge Add(rc,1,rc2) \\
& \quad \wedge Add(rc2,1,rc3) \\
& \quad \wedge OnPage(val_n,rc3)]
\end{aligned}$$

When comparing against the overall goal in *Figure 7.23*, it can be seen that it is satisfied when $VALUE_j=val_n$, $ARRID_a=ArrId1$, $KEYID_j=KeyId1$, $VARID_j=VarId5$, $EXPRID_j=VarExprId1$, $j=val_i$, $VALUE_rc=rc$, $Y=rc2$ and $Z=rc3$. Therefore, the student's program is identified as correct.

7.5 COLLECTION BASED LOOPS THAT PERFORM SOME ACTION AGAINST EVERY ITEM IN THE COLLECTION INDEPENDENTLY WHILE SUMMARISING

This section investigates how the KB in the PHP ITS handles summarising the data in an array while accessing every element of it as described in Section 7.1. In order to understand how such loops are analysed, consider the example exercise given in *Figure 7.25*. The initial state for this exercise is given in *Figure 7.26*.

Write a PHP code segment to find the maximum of the elements stored in an array \$marks and store it into a variable named 'max'. Use a for loop to perform the search. Note that when execution reaches the point where the code needs to be completed, the array is initialised and contains four elements.

Figure 7.25. Example exercise for a search loop.

```
HasArrayName(ARRID_m,'marks')
^ HasVariableId(HasElement(ARRID_m,KEYID_1),VARID_1)
^ HasKeyExpression(KEYID_1,EXPRID_1)
^ ValueOf(EXPRID_1,1)
^ HasValue(VARID_1,VALUE_1)
^ HasInitialValue(VARID_1, VALUE_1)
^ HasVariableId(HasElement(ARRID_m,KEYID_2),VARID_2)
^ HasKeyExpression(KEYID_2,EXPRID_2)
^ ValueOf(EXPRID_2,2)
^ HasValue(VARID_2,VALUE_2)
^ HasInitialValue(VARID_2, VALUE_2)
^ HasVariableId(HasElement(ARRID_m,KEYID_3),VARID_3)
^ HasKeyExpression(KEYID_3,EXPRID_3)
^ ValueOf(EXPRID_3,3)
^ HasValue(VARID_3,VALUE_3)
^ HasInitialValue(VARID_3, VALUE_3)
^ HasVariableId(HasElement(ARRID_m,KEYID_4),VARID_4)
^ HasKeyExpression(KEYID_4,EXPRID_4)
^ ValueOf(EXPRID_4,4)
^ HasValue(VARID_4,VALUE_4)
^ HasInitialValue(VARID_4, VALUE_4)
```

Figure 7.26. Initial state for example exercise for collection based loops that perform some action against every item in the collection.

Two common algorithms are used when developing solutions to exercises of this form. Table 7.7 shows example solutions for this exercise written using these two algorithms. The first program uses what is known as the indirect method. Here the position of the currently selected element is stored in a variable. Then each element of the array is accessed and compared against the element at the stored position to see if a certain criterion is satisfied. In this case, since the aim is to find the maximum, the criterion is to check if the element at the current array position is larger than the element at the stored position. If the criterion is satisfied, the current position replaces the stored position. Once all the elements have been processed, the element at the stored position is taken to be the desired element. In the direct method shown in the second program, the array element itself, and not its position is stored initially. Inside the loop, the element, and not its position replaces the stored value when the criterion is satisfied. This means that the required element is stored when all the elements have been accessed.

Table 7.7

Solutions to Example Exercise for Collection Based Loops the Perform Some Action on Every Element while Summarising using Indirect and Direct Methods of Array Access

<i>Searching using Indirect Method</i>	<i>Searching using Direct Method</i>
<pre> \$maxpos=1; for(\$i=2;\$i<5;\$i++) { if(\$array[\$i]>\$array[\$maxpos]) { \$maxpos=\$i; } } \$max=\$array[\$maxpos]; </pre>	<pre> \$max=\$array[1]; for(\$i=2;\$i<5;\$i++) { if(\$array[\$i]>\$max) { \$max=\$array[\$i]; } } </pre>

7.5.1 Overall Goal Specification

Figure 7.27 shows the overall goal for this example exercise. It is quite similar to the goal for for-each loops but contains some noteworthy characteristics. First of all, the goal here is also specified using a for-all term. In this case, it specifies that for all values of *j* between 1 and 4 (inclusive), the element at the given position in the array should be less than or equal to the value stored in a given variable. The constraints go on to specify that the name of this variable is 'max'. Upon careful consideration, it can be seen that if the variable 'max' holds the maximum value in

the array, this condition is always true. The constraints go on to ensure that a *for* loop is used and the body of the loop functions appropriately.

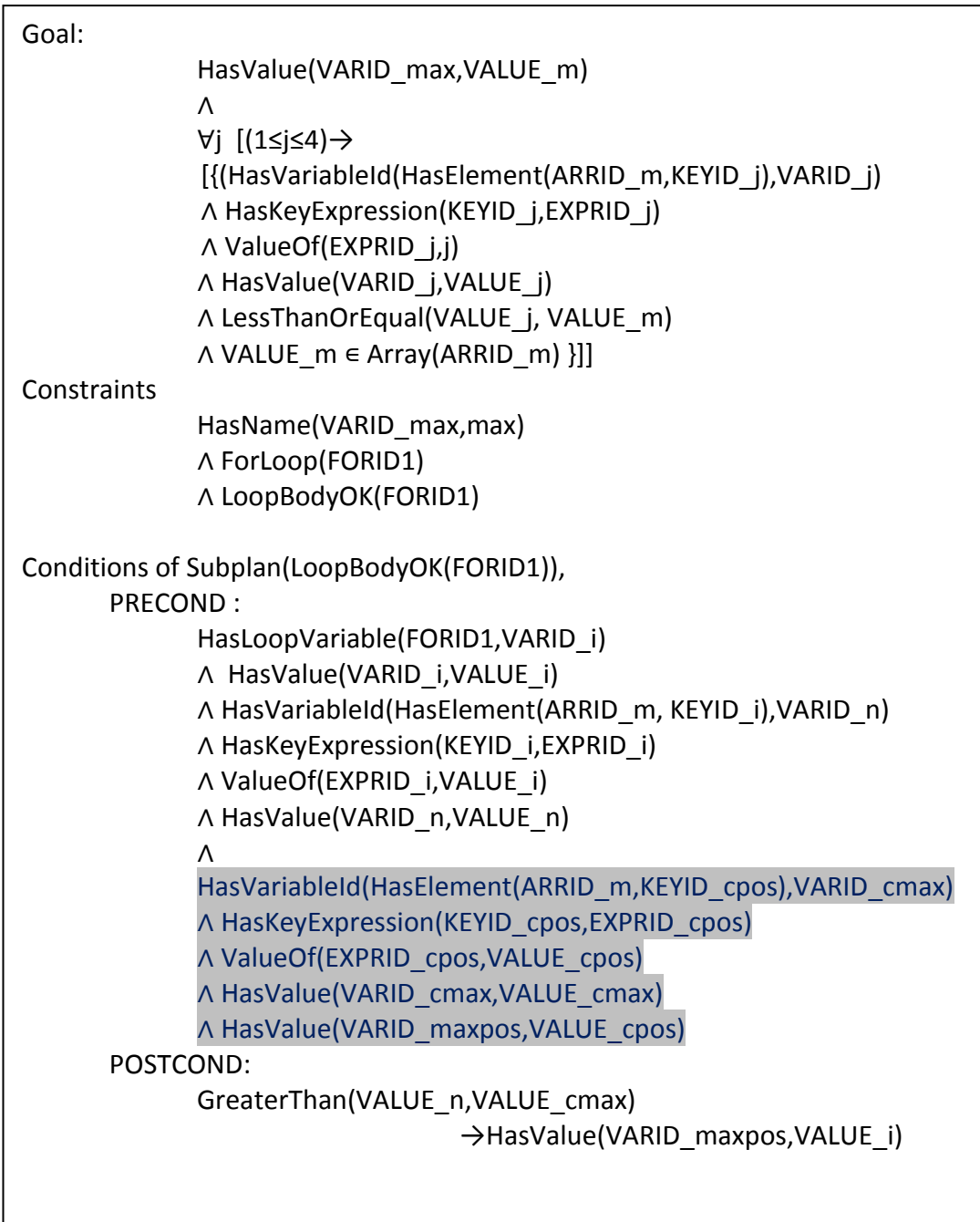


Figure 7.27. Overall goal for example exercise for collection based loops that perform some action against every item in the collection.

As described above, two common methods, the direct and indirect methods, are used for this type of searching. The function of the body of the loop needs to be different based on which algorithm is being used for the search. This means that two alternative conditions for sub-plans can be specified for the loop as discussed in

Section 7.3.1. However, in this case, the format of the conditions of one sub-plan can easily be obtained from the other. The exact method of doing this is discussed later in Section 7.5.1.2. Therefore only a single set of conditions of a sub-plan are specified in the overall goal. The pre-conditions of the sub-plan are divided into two sections – the unchangeable pre-conditions and the changeable preconditions. The unchangeable pre-conditions remain the same for both sets of conditions of sub-plans. The changeable pre-conditions and the post-condition are automatically generated for the conditions of the second sub-plan as described later in Section 7.5.1.2. Note that part of the pre-condition in the figure is highlighted. This is the part known as the changeable pre-condition and is automatically replaced with other predicates if the sub-plan for the indirect method is not satisfied.

The pre-conditions for the sub-plan for searching arrays is specified assuming that the indirect method of array access will be used. This is because the pre and post-conditions for the direct method can easily be derived from those for the direct method. In *Figure 7.27*, the pre-conditions specify that, if the element in the current array position is greater than the element in the array position stored in the variable indicating the current maximum, the current maximum position is updated to the current position. In other words, this is the indirect mode of access.

7.5.1.1 Program Analysis for Indirect Method

First consider how this overall goal specification is used to analyse a solution to the exercise written using the indirect method. Such a solution is given in the first program in *Table 7.7*. The following facts are created as a result of the initial state.

HasArrayName(*ArrId1*, 'marks')

HasVariableId(*HasElement*(*ArrId1*, *KeyId1*), *VarId1*)

HasKeyExpression(*KeyId1*, *ExprId1*)

ValueOf(*ExprId1*, 1)

HasValue(*VarId1*, *val_1*)

HasInitialValue(*VarId1*, *val_1*)

HasVariableId(*HasElement*(*ArrId1*, *KeyId2*), *VarId2*)

HasKeyExpression(*KeyId2*, *ExprId2*)

ValueOf(ExprId2,2)

HasValue(VarId2,val_2)

HasInitialValue(VarId2,val_2)

HasVariableId(HasElement(ArrId1,KeyId3),VarId3)

HasKeyExpression(KeyId3,ExprId3)

ValueOf(ExprId3,3)

HasValue(VarId3,val_3)

HasInitialValue(VarId3,val_3)

HasVariableId(HasElement(ArrId1,KeyId4),VarId4)

HasKeyExpression(KeyId4,ExprId4)

ValueOf(ExprId4,4)

HasValue(VarId4,val_4)

HasInitialValue(VarId4,val_4)

The first statement encountered during program analysis is an assignment. Let the id of the created variable be *VarId_{mp}*. Then, the following facts are created as a result of the *Assign* action. Note that only the facts relevant to this analysis are presented here.

HasName(VarId_{mp}, 'maxpos')

HasValue(VarId_{mp},1)

HasInitialValue(VarId_{mp},1)

Next, a *for* loop is encountered. Let the id of the created loop be *ForId1*. Let the id of the loop variable be *VarId_i* and the id of the conditional expression be *ExprId1*. Let the ids of the two expressions on either side of the conditional expression be *VarExprId1* and *LitExprId1* respectively. Let the id of the Literal related to the expression be *LitId1*. Then, the following facts are created as described in Section 7.2.3.

HasName(VarId_i, 'i')

HasValue(VarId_i,2)

HasInitialValue(VarId_i,2)
HasLoopVariable(ForId1,VarId_i)
HasForStartValue(ForId1,2)
HasId(LessExpr(VarExprId1,LitExprId1),ExprId1)
HasVariable(VarExprId1,VarId_i)
HasLiteral(LitExprId1,LitId1)
HasLitValue(LitId1,5)
HasLoopCondition(ForId1,ExprId1)

Using the rules in *Figure 4.8*, the *ValueOf* the *LiteralExpr* is found, resulting in the following fact.

ValueOf(LitExprId1,5)

Using the rule in *Figure 7.5*, the following fact is created.

HasForEndValue(ForId1,4)

Next, the update condition of the *for* loop is analysed using the procedure in 4.6.2. The resultant *Assign* action creates the following fact which is relevant to this analysis.

HasValue(VarId_i,3)

Since this is the value of the loop variable at the end of the first iteration, the following fact is created.

HasForFirstLoopValue(ForId1,3)

Next, the rule in *Figure 7.6* is activated, resulting in the following fact.

HasForIncrement(ForId1,1)

Now, the loop itself is analysed. Using the notation described in Section 7.2.3, the effect of the overall loop can be written as below.

repeat(ForActionEffects,ForId1)

The program statements within the loop are next analysed against the sub-plan. Let the starting value of *VarId_i* for each iteration be *val_i*. Let the starting value of *VarId_mp* be *val_mp*. Then, the following facts are created.

HasValue(VarId_i, val_i)

HasIterationValue(ForId1, VarId_i, val_i)

HasValue(VarId_mp, val_mp)

HasIterationValue(ForId1, VarId_mp, val_mp)

In this case, the loop accesses two array variables that have key values that are changed within the loop. Let the keys and expressions related to these keys have ids KeyId1, KeyId2, KeyExprId1 and KeyExprId2 respectively. Then, the following facts are created.

HasKeyExpression(KeyId1, KeyExprId1)

HasKeyExpression(KeyId2, KeyExprId2)

But the key expressions are actually variable expressions that access existing variables so the following facts are created.

HasVariable(KeyExprId1, VarId_i)

HasVariable(KeyExprId2, VarId_mp)

Using the rules in *Figure 4.8*, the following facts are created.

ValueOf(KeyExprId1, val_i)

ValueOf(KeyExprId2, val_mp)

But these relationships between the array and the keys are reified to create *ArrayVariables*. Let the ids of the two created *ArrayVariables* be VarId5 and VarId6 respectively. Then, the following facts are created.

HasVariableId(HasElement(ArrId1, KeyId1), VarId5)

HasVariableId(HasElement(ArrId1, KeyId2), VarId6)

For the purpose of analysing the loop, these variables need to be assigned symbolic values for their initial values during each iteration of the loop. Let the corresponding values be val_a and val_b respectively. Then, the following facts are created.

HasValue(VarId5, val_a)

HasIterationValue(ForId1, VarId5, val_a)

HasValue(VarId6, val_b)

HasIterationValue(ForId1, VarId6, val_b)

When comparing the existing facts against the pre-conditions of the sub-plan, it can be seen that they are satisfied when FORID1=ForId1, VARID_i=Varid_i, VALUE_i=val_i, ARRID_m=ArrId1, KEYID_i=KeyId1, VARID_n=VarId5, EXPRID_i=KeyExprId1, VALUE_n=val_a, KEYID_cpos=KeyId2, VARID_cmax=VarId5, EXPRID_cpos=KeyExprId2, VALUE_cpos=val_mp, VARID_cmax=VarId6, VALUE_cmax=val_b and VARID_maxpos=VarId_mp.

Next, the statements within the loop are analysed. The first statement within the loop is an *if* construct which is analysed as described in Section 5.2. Let the id of the conditional expression within the construct be ExprId2. Let the ids of the two *VariableExprs* on either side of the conditional expression be VarExprId2 and VarExprId3 respectively. Then, the following fact is created.

HasId(GreaterExpr(VarExprId2, VarExprId3), ExprId2)

Since the *VariableExprs* on either side of this expression refer to *ArrayVariables*, it is necessary to find the corresponding ids. VarExprId2 refers to the *ArrayVariable* connecting the array to the loop variable \$i. From above, it can be seen that this corresponds to the key expression KeyExprId1 which in turn corresponds to the key KeyId1. The variable connecting the array and KeyId1 is VarId5 so this is the variable that VarExprId2 refers to. Similarly, VarExprId3 refers to the variable VarId6 so the following facts are created.

HasVariable(VarExprId2, VarId5)

HasVariable(VarExprId3, VarId6)

Now, the *ValueOf* these expressions are calculated using the rules in *Figure 4.8*, resulting in the following facts.

ValueOf(VarExprId2, val_a)

ValueOf(VarExprId3, val_b)

Within the *if* statement, the conditional expression is true so the following fact is created.

ValueOf(ExprId2, True)

When this condition is true, the following fact is created, using the rules in *Figure 5.4*.

$$\text{GreaterThan}(\text{val}_a, \text{val}_b)$$

The assignment statement occurs if this condition is satisfied, resulting in an *Assign* action. The following predicate is then created.

$$\text{HasValue}(\text{VarId}_{mp}, \text{val}_i)$$

Therefore, the effects of the for action can be written as below.

$$\text{GreaterThan}(\text{val}_a, \text{val}_b) \rightarrow \text{HasValue}(\text{VarId}_{mp}, \text{val}_i)$$

So the post-condition of the sub-plan is satisfied and the following fact is created.

$$\text{LoopBodyOK}(\text{ForId1})$$

Now, the results of the loop are consolidated using the set of rules given in *Figure 7.7*.

$$\text{RepeatLoop}(\text{ForId1}, 2, 4, 1)$$

$$\text{RepeatAll}(\text{ForId1}, 2, 4)$$

$$\forall \text{val}_i [(2 \leq \text{val}_i \leq 4) \rightarrow$$

$$(\text{HasVariableId}(\text{HasElement}(\text{ArrId1}, \text{KeyId1}), \text{VarId5})$$

$$\wedge \text{HasKeyExpression}(\text{KeyId1}, \text{KeyExprId1})$$

$$\wedge \text{ValueOf}(\text{KeyExprId1}, \text{val}_i)$$

$$\wedge \text{HasValue}(\text{VarId5}, \text{val}_a)$$

$$\wedge \text{HasVariableId}(\text{HasElement}(\text{ArrId1}, \text{KeyId2}), \text{VarId6})$$

$$\wedge \text{HasKeyExpression}(\text{KeyId2}, \text{KeyExprId2})$$

$$\wedge \text{ValueOf}(\text{KeyExprId2}, \text{val}_{mp})$$

$$\wedge \text{HasValue}(\text{VarId6}, \text{val}_b)$$

$$\wedge \{\text{GreaterThan}(\text{val}_a, \text{val}_b) \rightarrow \text{HasValue}(\text{VarId}_{mp}, \text{val}_i)\}]$$

At this point, several new rules need to be introduced in order to consolidate this into the required form. As happens very often in search loops, the first program in *Table 7.7* assigns the first value relevant to the array (in this case the key of the first element) to a variable and then loops through the rest of the array, ignoring the

first element. This is the same as performing the function for all elements of the array and is similar to loop unrolling for the first element. Since this happens very often in practical programming, a special rule is included to specify that these two forms are equivalent. A similar rule is included to handle storing the data relevant to the last element of the array and looping through the rest of the elements backwards. Both these rules are shown in *Figure 7.28*.



Figure 7.28. Rules for handling loop unrolling of the first or last element of the array.

he

first section containing the premises defines an *ArrayVariable* with an index of 1 has the value *val1*. The next part specifies some variable should exist with a value of either 1 or *val1* at the beginning of the execution of the loop. The next part specifies that the loop should iterate over the elements of an array starting at index 2 and cause a certain *ActionEffect*. When these premises are satisfied, the rule is activated to specify that this is the same as iterating over the elements of the array starting at 1 and causing the same *ActionEffect*. The second rule in this figure can be explained in the same way except that the starting value of the variable is connected to the last element of the array.

Since the above program resulted in consolidating the array from the second to the last element of the array, the first rule in *Figure 7.28* is activated, resulting in the following fact.

$$\begin{aligned}
 & \forall \text{val_i} [(1 \leq \text{val_i} \leq 4) \rightarrow \\
 & \quad (\text{HasVariableId}(\text{HasElement}(\text{ArrId1}, \text{KeyId1}), \text{VarId5}) \\
 & \quad \wedge \text{HasKeyExpression}(\text{KeyId1}, \text{KeyExprId1}) \\
 & \quad \wedge \text{ValueOf}(\text{KeyExprId1}, \text{val_i}) \\
 & \quad \wedge \text{HasValue}(\text{VarId5}, \text{val_a}) \\
 & \quad \wedge \text{HasVariableId}(\text{HasElement}(\text{ArrId1}, \text{KeyId2}), \text{VarId6}) \\
 & \quad \wedge \text{HasKeyExpression}(\text{KeyId2}, \text{KeyExprId2}) \\
 & \quad \wedge \text{ValueOf}(\text{KeyExprId2}, \text{val_mp}) \\
 & \quad \wedge \text{HasValue}(\text{VarId6}, \text{val_b}) \\
 & \quad \wedge \{\text{GreaterThan}(\text{val_a}, \text{val_b}) \rightarrow \text{HasValue}(\text{VarId_mp}, \text{val_i})\}]
 \end{aligned}$$

This state is in a form that specifies what happens within the loop and repeats it for all values of the loop counter. However, in the case of loops that summarise collections, what is of interest is that the selected value is a member of the array and relates to all elements of the array based on some criterion. Several new facts and a rule are needed to convert this given representation into a form that specifies the necessary result. These facts and rule are shown in *Figure 7.29*.

```

Opposite(GreaterThan,LessThanOrEqual)
Opposite(LessThan,GreaterThanOrEqual)
Opposite(GreaterThanOrEqual,LessThan)
Opposite(LessThanOrEqual,GreaterThan)

HasValue(varId_m,value_m)
 $\wedge$ 
 $\forall$  value_i [(start $\leq$ value_i $\leq$ n)  $\rightarrow$ 
    [HasVariableId(HasElement(arrId_m,keyId_y),varId_z)
 $\wedge$  HasKeyExpression(keyId_y,exprId_y)
 $\wedge$  ValueOf(exprId_y,value_i)
 $\wedge$  HasValue(varId_z,value_z)
 $\wedge$  HasVariableId(HasElement(arrId_m,keyId_a),varId_b)
 $\wedge$  HasKeyExpression(keyId_a,exprId_a)
 $\wedge$  ValueOf(exprId_a,value_m)
 $\wedge$  HasValue(varId_b,value_b)
 $\wedge$  BooleanExpression2(value_z,value_b)
 $\wedge$  value_m  $\in$  Array(arrId1)]

 $\leftarrow$ 
 $\forall$  value_i [(start $\leq$ value_i $\leq$ n)  $\rightarrow$ 
    [HasVariableId(HasElement(arrId_m,keyId_y),varId_z)
 $\wedge$  HasKeyExpression(keyId_y,exprId_y)
 $\wedge$  ValueOf(exprId_y,value_i)
 $\wedge$  HasValue(varId_z,value_z)
 $\wedge$  HasVariableId(HasElement(arrId_m,keyId_a),varId_b)
 $\wedge$  HasKeyExpression(keyId_a,exprId_a)
 $\wedge$  ValueOf(exprId_a,value_a)
 $\wedge$  HasValue(varId_b,value_b)
 $\wedge$  Opposite(BooleanExpression1,BooleanExpression2)
 $\wedge$  BooleanExpression1(value_z,value_b) $\rightarrow$  HasValue(varId_m,value_i)]

```

Figure 7.29. Facts and rules for finding search results.

The *Opposite* predicate is used to define *BooleanExprs* that are logically opposite to each other. For example, the opposite of ($\$x > 10$) is ($\$x \leq 10$) so the opposite of *GreaterThan* is *LessThanOrEqual*. The four facts given here define all possible combinations of opposite for the four comparison expressions *GreaterThan*, *LessThan*, *GreaterThanOrEqual* and *LessThanOrEqual*. These facts are then utilised in a rule to describe the search result as mentioned earlier. The first part of the premise of the rule specifies that the facts should repeat for all values of the counter starting from 1. The next part specifies the value and index relevant to the

ArrayVariable for each value of the counter. The next part specifies the same details for another *ArrayVariable* in the array. The final part of the premise depicts the effects of the repetition. This effect is based on a condition. If the value of the *ArrayVariable* at the index indicated by the loop counter is related using a particular *BooleanExpr* type with the value of another *ArrayVariable* of the same array, another variable is assigned the value of the counter variable. When these premises are satisfied, the rule is fired. The result again uses the two *ArrayVariables* described above. Also, all the values in the array now take on the opposite relationship to the *BooleanExpr* considered in the loop effects. For example, as the check here was whether each value in the array was larger than the value at the currently largest position, the check was for the maximum of the array. Therefore, the greater-than check within the loop results in all elements being less than or equal to the selected value. The selected value obtained in this way is stored within the variable whose value is implied by the selection within the loop. Since this value is stored in an *ArrayVariable* related to the array, it is obviously a member of the array.

Upon comparing against the state of the analysis above, it can be seen that this rule is now fired. Since the expression corresponding to *BooleanExpression1* is a *GreaterThan*, the expression corresponding to *BooleanExpression2* is a *LessThanOrEqual* so the following facts are created.

$$\begin{aligned}
& \text{HasValue}(\text{VarId}_{mp}, \text{val}_{mp}) \\
& \wedge \\
& \forall \text{val}_i [(1 \leq \text{val}_i \leq 4) \rightarrow \\
& \quad (\text{HasVariableId}(\text{HasElement}(\text{ArrId1}, \text{KeyId1}), \text{VarId5}) \\
& \quad \wedge \text{HasKeyExpression}(\text{KeyId1}, \text{KeyExprId1}) \\
& \quad \wedge \text{ValueOf}(\text{KeyExprId1}, \text{val}_i) \\
& \quad \wedge \text{HasValue}(\text{VarId5}, \text{val}_a) \\
& \quad \wedge \text{HasVariableId}(\text{HasElement}(\text{ArrId1}, \text{KeyId2}), \text{VarId6}) \\
& \quad \wedge \text{HasKeyExpression}(\text{KeyId2}, \text{KeyExprId2}) \\
& \quad \wedge \text{ValueOf}(\text{KeyExprId2}, \text{val}_{mp}) \\
& \quad \wedge \text{HasValue}(\text{VarId6}, \text{val}_b) \\
& \quad \wedge \text{LessThanOrEqual}(\text{val}_a, \text{val}_b) \\
& \quad \wedge \text{val}_b \in \text{Array}(\text{ArrId1})])
\end{aligned}$$

Now, the last statement in the program is analysed. This is an assignment using the *Assign* action. The right hand side of this action is a *VariableExpr* referring to the *ArrayVariable* that is the connection between the given array and the \$maxpos variable. Therefore, the key expression related to the *ArrayVariable* is a *VariableExpr* as well. Let the id of this be VarExprId_k. Then, the following fact is created.

$$\text{HasVariable}(\text{VarExprId}_k, \text{VarId}_{mp})$$

Now, the *ValueOf* this expression is calculated as below.

$$\text{ValueOf}(\text{VarExprId}_k, \text{val}_{mp})$$

Let the key relevant to this *ArrayVariable* be KeyId_k and the id of the variable be VarId_x. Then, the following facts are created.

$$\text{HasVariableId}(\text{HasElement}(\text{ArrId1}, \text{KeyId}_k), \text{VarId}_x)$$

$$\text{HasKeyExpression}(\text{KeyId}_k, \text{VarExprId}_k)$$

The values of the variables VarId_mp and VarId6 change during the iteration of the loop. Therefore, in order to analyse the rest of the program, it becomes necessary to assign a symbolic value to it at the end of the loop. This is similar to assigning a symbolic value to each variable at the beginning of the iteration. Let the values of VarId_mp and VarId6 be val_mf and val_bf at end of the execution of the loop. Then, the following fact is created.

$$\text{HasValue}(\text{VarId}_x, \text{val}_{bf})$$

Now, the *Assign* action results in the following facts being created.

$$\text{HasName}(\text{VarId}_m, \text{'max'})$$

$$\text{HasValue}(\text{VarId}_m, \text{val}_{bf})$$

So the final state of the program is as below.

$$\begin{aligned}
& \text{HasName}(\text{VarId}_m, 'max') \\
& \wedge \text{HasValue}(\text{VarId}_m, \text{val}_{bf}) \\
& \wedge \\
& \forall \text{val}_i [(1 \leq \text{val}_i \leq 4) \rightarrow \\
& \quad (\text{HasVariableId}(\text{HasElement}(\text{ArrId1}, \text{KeyId1}), \text{VarId5}) \\
& \quad \wedge \text{HasKeyExpression}(\text{KeyId1}, \text{KeyExprId1}) \\
& \quad \wedge \text{ValueOf}(\text{KeyExprId1}, \text{val}_i) \\
& \quad \wedge \text{HasValue}(\text{VarId5}, \text{val}_a) \\
& \quad \wedge \text{HasVariableId}(\text{HasElement}(\text{ArrId1}, \text{KeyId2}), \text{VarId6}) \\
& \quad \wedge \text{HasKeyExpression}(\text{KeyId2}, \text{KeyExprId2}) \\
& \quad \wedge \text{ValueOf}(\text{KeyExprId2}, \text{val}_{mf}) \\
& \quad \wedge \text{HasValue}(\text{VarId6}, \text{val}_{bf}) \\
& \quad \wedge \text{LessThanOrEqual}(\text{val}_a, \text{val}_{bf}) \\
& \quad \wedge \text{val}_b \in \text{Array}(\text{ArrId1})])
\end{aligned}$$

When comparing against the overall goal in *Figure 7.27*, it can be seen that it is satisfied when $\text{VARID}_{max} = \text{VarId}_m$, $\text{VALUE}_m = \text{val}_{bf}$, $j = \text{val}_i$, $\text{ARRID}_m = \text{ArrId1}$, $\text{KEYID}_j = \text{KeyId1}$, $\text{VARID}_j = \text{VarId5}$, $\text{VALUE}_j = \text{val}_a$ and $\text{FORID1} = \text{ForId1}$. Therefore, the program is identified as correct.

7.5.1.2 Program Analysis for Direct Method

The previous section discussed how searching using the indirect method is handled within the PHP ITS. Although programs written using the direct method can be handled by specifying an alternate set of conditions for the sub-plan, this is not necessary. Since the format is the same, this alternate set of conditions is automatically generated within the PHP ITS. This section describes how this process is carried out.

In order to understand how such programs are analysed, consider the second program in *Table 7.7*, which is written using the direct method. The initial state is the same as before, resulting in the following facts.

$$\begin{aligned}
& \text{HasArrayName}(\text{ArrId1}, 'marks') \\
& \text{HasVariableId}(\text{HasElement}(\text{ArrId1}, \text{KeyId1}), \text{VarId1}) \\
& \text{HasKeyExpression}(\text{KeyId1}, \text{ExprId1})
\end{aligned}$$

ValueOf(ExprId1,1)
HasValue(VarId1,val_1)
HasInitialValue(VarId1,val_1)
HasVariableId(HasElement(ArrId1,KeyId2),VarId2)
HasKeyExpression(KeyId2,ExprId2)
ValueOf(ExprId2,2)
HasValue(VarId2,val_2)
HasInitialValue(VarId2,val_2)
HasVariableId(HasElement(ArrId1,KeyId3),VarId3)
HasKeyExpression(KeyId3,ExprId3)
ValueOf(ExprId3,3)
HasValue(VarId3,val_3)
HasInitialValue(VarId3,val_3)
HasVariableId(HasElement(ArrId1,KeyId4),VarId4)
HasKeyExpression(KeyId4,ExprId4)
ValueOf(ExprId4,4)
HasValue(VarId4,val_4)
HasInitialValue(VarId4,val_4)

Here, the first step in the program is an assignment statement but it assigns the value stored in the array and not 1 as in the previous case. From the initial state, it can be seen that the value on the right hand side of the assignment is val_1 so the following facts are created.

HasName(VarId_m,'max')
HasValue(VarId_m,val_1)

Now, a *for* loop similar to the indirect method is encountered resulting in the following facts as described in Section 7.5.1.1.

HasName(VarId_i,'i')

HasValue(VarId_i,2)
HasInitialValue(VarId_i,2)
HasLoopVariable(ForId1,VarId_i)
HasForStartValue(ForId1,2)
HasId(LessExpr(VarExprId1,LitExprId1),ExprId1)
HasVariable(VarExprId1,VarId_i)
HasLiteral(LitExprId1,LitId1)
HasLitValue(LitId1,5)
HasLoopCondition(ForId1,ExprId1)
ValueOf(LitExprId1,5)
HasForEndValue(ForId1,4)
HasValue(VarId_i,3)
HasForFirstLoopValue(ForId1,3)
HasForIncrement(ForId1,1)

Now, the loop itself is analysed. Using the notation described in Section 7.2.3, the effect of the overall loop can be written as below.

repeat(ForActionEffects,ForId1)

As described in Section 7.5.1.1 many new facts and objects need to be considered when analysing the loop itself. The following facts are created as a result of this as described previously. The only difference is that a starting value is now considered for the variable *VarId_m* and only the *ArrayVariable* corresponding to the current loop counter is considered since this is the only *ArrayVariable* accessed within the loop.

HasValue(VarId_i,val_i)
HasIterationValue(ForId1,VarId_i,val_i)
HasValue(VarId_m,val_ms)
HasIterationValue(ForId1,VarId_m,val_ms)
HasKeyExpression(KeyId1,KeyExprId1)

HasVariable(KeyExprId1,VarId_i)

ValueOf(KeyExprId1,val_i)

HasVariableId(HasElement(ArrId1,KeyId1),VarId5)

HasValue(VarId5,val_a)

HasIterationValue(ForId1,VarId5,val_a)

Next, it is necessary to ascertain whether the pre-conditions of the sub-plan, as defined in *Figure 7.27*, are satisfied. It can be seen that although facts corresponding to the unchangeable pre-condition are present in the present state, facts corresponding to the unchangeable pre-condition are not present.

In the case of loops that contain a changeable pre-condition, the analysis process deviates from the usual at this point. An alternate set of conditions for a sub-plan is generated by changing the predicates in the changeable precondition. All the predicates in this part of the pre-condition are replaced by a single predicate *HasValue(VARID_max,VALUE_cmax)*.

Now, it can be seen that the pre-conditions of this newly generated sub-plan are satisfied by the current program when *FORID1=ForId1*, *VARID_i=Varid_i*, *VALUE_i=val_i*, *ARRID_m=ArrId1*, *KEYID_i=KeyId1*, *VARID_n=VarId5*, *EXPRID_i=KeyExprId1*, *VALUE_n=val_a*, *VARID_max=VarId_m* and *VALUE_cmax=val_ms*.

Next, the statements within the loop are analysed. The first statement within the loop is an *if* construct which is analysed as described in Section 5.2. Let the id of the conditional expression within the construct be *ExprId2*. Let the ids of the two *VariableExprs* on either side of the conditional expression be *VarExprId2* and *VarExprId3* respectively. Then, the following fact is created.

HasId(GreaterExpr(VarExprId2,VarExprId3),ExprId2)

Since the *VariableExprs* on the left hand side of this expression refer to *ArrayVariables*, it is necessary to find the corresponding ids. *VarExprId2* refers to the *ArrayVariable* connecting the array to the loop variable *\$i*. From above, it can be seen that this corresponds to the key expression *KeyExprId1* which in turn corresponds to the key *KeyId1*. The variable connecting the array and *KeyId1* is

VarId5 so this is the variable that VarExprId2 refers to. The *VariableExpr* on the right hand side refers to a *SimpleVariable* so the following facts are created.

$$\text{HasVariable}(\text{VarExprId2}, \text{VarId5})$$

$$\text{HasVariable}(\text{VarExprId3}, \text{VarId}_m)$$

Now, the *ValueOf* these expressions are calculated using the rules in *Figure 4.8*, resulting in the following facts.

$$\text{ValueOf}(\text{VarExprId2}, \text{val}_a)$$

$$\text{ValueOf}(\text{VarExprId3}, \text{val}_ms)$$

Within the *if* statement, the conditional expression is true so the following fact is created.

$$\text{ValueOf}(\text{ExprId2}, \text{True})$$

When this condition is true, the following fact is created, using the rules in *Figure 5.4*.

$$\text{GreaterThan}(\text{val}_a, \text{val}_ms)$$

The assignment statement occurs if this condition is satisfied, resulting in an *Assign* action. The expression on the right hand side of the assignment is a *VariableExpr* referring to the value of the variable at the loop counter position of the array. From above, it can be seen that this value is *val_a* so the following fact is created.

$$\text{HasValue}(\text{VarId}_m, \text{val}_a)$$

Therefore, the effects of the for action can be written as below.

$$\text{GreaterThan}(\text{val}_a, \text{val}_ms) \rightarrow \text{HasValue}(\text{VarId}_m, \text{val}_a)$$

Now, it is necessary to see if the post-condition of the sub-plan is satisfied. However, in this case, we are dealing with a set of conditions of a sub-plan generated by the system. When generating such a set of conditions, not only the changeable pre-condition but also the right hand side of the post-condition is changed. The generated post-condition is as below.

$$\begin{aligned} &\text{GreaterThan}(\text{VALUE}_n, \text{VALUE}_c\text{max}) \\ &\quad \rightarrow \text{HasValue}(\text{VARID}_\text{max}, \text{VALUE}_n) \end{aligned}$$

Comparing the current state of the program against this post-condition, it can be seen that it is satisfied so the following fact is created.

LoopBodyOK(ForId1)

Now, the results of the loop are consolidated using the set of rules given in *Figure 7.7*.

RepeatLoop(ForId1,2,4,1)

RepeatAll(ForId1,2,4)

$\forall val_i [(2 \leq val_i \leq 4) \rightarrow$
 $(HasVariableId(HasElement(ArrId1,KeyId1),VarId5)$
 $\wedge HasKeyExpression(KeyId1,KeyExprId1)$
 $\wedge ValueOf(KeyExprId1,val_i)$
 $\wedge HasValue(VarId5,val_a)$
 $\wedge \{Greater\Than(val_a,val_ms) \rightarrow \neg HasValue(VarId_m,val_a)\})]$

Again, it can be seen that there is a loop unrolling situation for the first element of the array. The rule in *Figure 7.28* is now activated to result in the following fact.

$\forall val_i [(1 \leq val_i \leq 4) \rightarrow$
 $(HasVariableId(HasElement(ArrId1,KeyId1),VarId5)$
 $\wedge HasKeyExpression(KeyId1,KeyExprId1)$
 $\wedge ValueOf(KeyExprId1,val_i)$
 $\wedge HasValue(VarId5,val_a)$
 $\wedge \{Greater\Than(val_a,val_ms) \rightarrow \neg HasValue(VarId_m,val_a)\})]$

A rule similar to that in *Figure 7.29* is specified for handling the direct search mechanism as well. This rule is shown in *Figure 7.30*.

In this case, the value of the variable \$max is changed within the loop and its final value cannot be ascertained directly. For analysis purposes, it is assigned a value after iteration of the loop, similar to *HasIterationValue*. Let the value of the variable after execution of the loop be val_mf.

$$\begin{array}{l}
\text{HasValue}(\text{varId}_m, \text{value}_m) \\
\wedge \\
\forall \text{value}_i \ [(\text{start} \leq \text{value}_i \leq n) \rightarrow \\
\quad \wedge [\text{HasVariableId}(\text{HasElement}(\text{arrId}_m, \text{keyId}_y), \text{varId}_z) \\
\quad \wedge \text{HasKeyExpression}(\text{keyId}_y, \text{exprId}_y) \\
\quad \wedge \text{ValueOf}(\text{exprId}_y, \text{value}_i) \\
\quad \wedge \text{HasValue}(\text{varId}_z, \text{value}_z) \\
\quad \wedge \text{BooleanExpression2}(\text{value}_z, \text{value}_m) \\
\quad \wedge \text{value}_m \in \text{Array}(\text{arrId}_m)] \\
\leftarrow \\
\forall \text{value}_i \ [(\text{start} \leq \text{value}_i \leq n) \rightarrow \\
\quad [\text{HasVariableId}(\text{HasElement}(\text{arrId}_m, \text{keyId}_y), \text{varId}_z) \\
\quad \wedge \text{HasKeyExpression}(\text{keyId}_y, \text{exprId}_y) \\
\quad \wedge \text{ValueOf}(\text{exprId}_y, \text{value}_i) \\
\quad \wedge \text{HasValue}(\text{varId}_z, \text{value}_z) \\
\quad \wedge \text{HasIterationValue}(\text{loopId}_1, \text{varId}_m, r) \\
\quad \wedge \text{Opposite}(\text{BooleanExpression1}, \text{BooleanExpression2}) \\
\quad \wedge \text{BooleanExpression1}(\text{value}_z, r) \rightarrow \text{HasValue}(\text{varId}_m, \text{value}_z)]
\end{array}$$

Figure 7.30. Rule for handling direct method of array access in search loops.

Then the above rule is activated to create the following facts.

$$\begin{array}{l}
\text{HasValue}(\text{VarId}_m, \text{val}_{mf}) \\
\wedge \\
\forall \text{val}_i \ [(1 \leq \text{val}_i \leq 4) \rightarrow \\
\quad (\text{HasVariableId}(\text{HasElement}(\text{ArrId1}, \text{KeyId1}), \text{VarId5}) \\
\quad \wedge \text{HasKeyExpression}(\text{KeyId1}, \text{KeyExprId1}) \\
\quad \wedge \text{ValueOf}(\text{KeyExprId1}, \text{val}_i) \\
\quad \wedge \text{HasValue}(\text{VarId5}, \text{val}_a) \\
\quad \wedge \text{LessThanOrEqual}(\text{val}_a, \text{val}_{mf}) \\
\quad \wedge \text{val}_{mf} \in \text{Array}(\text{ArrId1})]
\end{array}$$

When comparing against the overall goal in Figure 7.27, it can be seen that it is satisfied when $\text{VARID}_m = \text{VarId}_m$, $\text{VALUE}_m = \text{val}_{mf}$, $j = \text{val}_i$, $\text{ARRID}_m = \text{ArrId1}$, $\text{KEYID}_j = \text{KeyId1}$, $\text{VARID}_j = \text{VarId5}$, $\text{EXPRID}_j = \text{KeyExprId1}$ and $\text{VALUE}_j = \text{val}_a$. So the program is identified as correct.

This method of generating alternate sets of conditions of sub-plans to handle direct and indirect array access can be used to analyse programs that summarise an array to find the maximum or minimum.

7.6 CHAPTER SUMMARY

This chapter looked at how the knowledge base of the PHP ITS is designed to handle different types of loops. Loops used in PHP programming can be categorised based on their underlying logical model. The PHP ITS is not capable of handling all possible types of loops but it can analyse many of the types of loops used commonly in practical programming. This chapter first looked at how basic definite loops are analysed. It went on to investigate how these ideas were extended to handle more generalised loops.

This is the final chapter describing how the PHP ITS handles program analysis. The next chapter looks at the user interfaces of the ITS and how the student and teaching modules are designed. It also looks at some implementation details of the system.

Chapter 8: Implementation of the PHP Intelligent Tutoring System

As discussed in Section 2.2.2, an ITS consists of four main modules : the domain module, the student module, the teaching module and the communications module. Chapter 4, Chapter 5, Chapter 6 and Chapter 7 described the domain module used in the PHP Intelligent Tutoring System. This chapter describes the communications, student and teaching modules used in the PHP Intelligent Tutoring System. It details the actual implementation of the system, including the GUI seen by the users. The implementation of the system is the third phase of the research project as described in the research design (Chapter 3). Section 8.1 describes the user interfaces of the system. Section 8.2 discusses the design of the student module and Section 8.3 covers the teaching module in detail. Section 8.4 describes how the various available software and tools were used to create the actual system.

8.1 THE PHP INTELLIGENT TUTORING SYSTEM

The PHP Intelligent Tutoring System (PHP ITS) is a completely web based system that can be accessed through a web browser. In order to use the system, each student must create a user name. They then login to the system using this user name and the relevant password. When a student logs in for the first time, he/she is required to complete a pre-test to gauge their current knowledge of PHP. The pre-test is a set of multiple choice questions, each of which the student can leave blank if they do not know the answer to the question. It is even possible to not answer any questions if the student has no relevant knowledge.

Once a student has completed the pre-test, s/he is directed to the exercise selection page. This page is also directly displayed on each subsequent login since the pre-test is only permitted once per student. The student selects an exercise to attempt and then enters PHP code for that exercise. When requested, the system provides appropriate feedback. The student is also permitted to abandon the current exercise and return to the exercise selection page at any time.

The system displays a banner across the top of most of its pages as shown in *Figure 8.1*. This banner allows the student to select from several options. S/he can

logout of the system or change password. The ‘Help’ link brings up some help pages on how to use the system. The ‘Skillometer’ (*Figure 8.2*) allows the student to bring up another page that displays his/her current knowledge of the topics covered by the PHP ITS as gauged by the system.

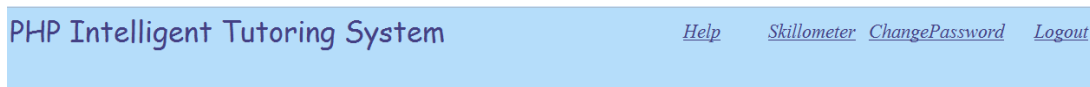


Figure 8.1. Banner.

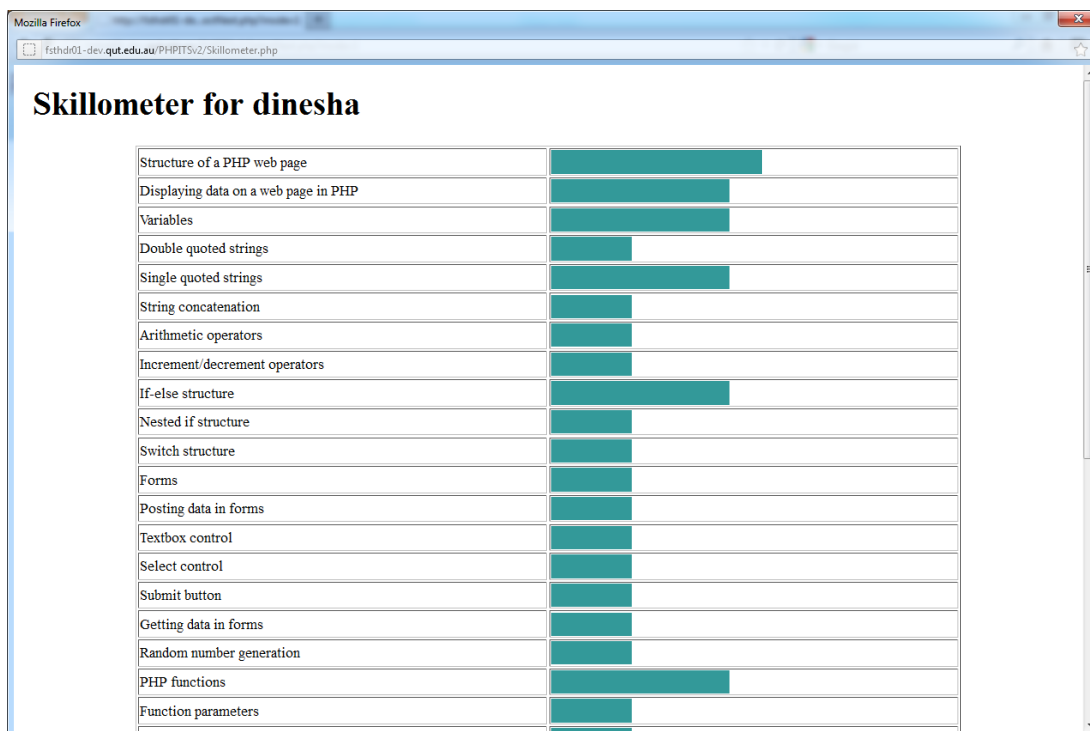


Figure 8.2. Skillometer.

The security of the system is handled through password protection. Each student can set up a password and also enter the answers to two security questions when the user name is created. A logged in student can change the password of the current account. If the password is forgotten, it is possible to reset it using the security questions. If this is selected, the password is reset to a default value and the student is asked to change the password during the next login session.

The main advantage of this system is that it is web based. The system has currently been tested in Internet Explorer, Mozilla Firefox, Opera and Google

Chrome browsers. Therefore, it makes it possible for a multitude of users to access the ITS from different platforms.

8.1.1 Exercise Selection

One main advantage of the PHP ITS is that it guides each student towards topics that are most suitable for his or her current level of knowledge. This guidance is done through the list of exercises. The system shows a list of exercises that it thinks are most suitable for the logged in student. The exercises are shown in decreasing order of suitability with prominence being given to the most suitable exercise. This is also the exercise that is selected by default. The student may decide to attempt another exercise from the list if s/he wishes to. The Exercise Selection page of the PHP ITS is shown in *Figure 8.3*.

PHP Intelligent Tutoring System [Help](#) [Skillometer](#) [ChangePassword](#) [Logout](#)

Let the ITS suggest exercises
 Search for exercises

Select the next exercise that you would like to attempt and then click the 'Proceed to exercise' button

The exercise that is most suitable for you

PHP012 NEW 2012-02-29
 You have not yet attempted this exercise

```

Display the text "Hello World" on a web page. Use PHP instead of HTML.
<html>
<body>
<?php
enter your code here
?>
</body>
</html>
  
```

Other exercises in order of decreasing suitability

Select	Exercise Code	Already Attempted	Successfully Completed	Exercise
<input type="radio"/>	PHP013	No		Store your name into a variable named "myname". Display the string "Hello! My name is" followed by the contents of the above variable using a double quoted string. NEW 2012-02-29
<input type="radio"/>	PHP014	No		Store your name into a variable called "myname". Display the string "Hello! My name is" followed by the contents of the above variable using a single quoted string and the PHP concatenation operator. NEW 2012-02-29

Proceed to exercise

Figure 8.3. Exercise selection page.

Although the ITS suggests exercises based on its measure of the subject knowledge of each student, some students may want to be in charge of selecting their next exercises. In such cases, they can select a different exercise from the list but since this list could be very long, they may find it difficult to find what they want. A

search option is provided for this purpose. If the student decides to search for an exercise, the page shown in *Figure 8.4* is displayed. The student can now select which topic(s) need to be covered by the exercises s/he wants to attempt. It is also possible to choose whether to display exercises that have already been attempted/not attempted or both and successfully completed/not completed or both.

<p>Select the topics to be included</p> <ul style="list-style-type: none"> <input type="checkbox"/> Structure of a PHP web page <input type="checkbox"/> Displaying data on a web page in PHP <input type="checkbox"/> Variables <input type="checkbox"/> Double quoted strings <input type="checkbox"/> Single quoted strings <input type="checkbox"/> String concatenation <input type="checkbox"/> Arithmetic operators <input type="checkbox"/> Increment/decrement operators <input type="checkbox"/> If-else structure <input type="checkbox"/> Nested if structure <input type="checkbox"/> Switch structure <input type="checkbox"/> Forms <input type="checkbox"/> Posting data in forms <input type="checkbox"/> Textbox control <input type="checkbox"/> Select control <input type="checkbox"/> Submit button <input type="checkbox"/> Getting data in forms <input type="checkbox"/> Style sheets <input type="checkbox"/> Hidden fields <input type="checkbox"/> Conditional display of elements on a web page <input type="checkbox"/> Boolean variables <input type="checkbox"/> Random number generation <input type="checkbox"/> PHP functions <input type="checkbox"/> Function parameters <input type="checkbox"/> Returning values from functions <input type="checkbox"/> For construct <input type="checkbox"/> Nested for loops <input type="checkbox"/> HTML Tables <input type="checkbox"/> HTML borders <input type="checkbox"/> Dynamically adding rows to a table <input type="checkbox"/> While construct <input type="checkbox"/> Explicitly assigning data to an array <input type="checkbox"/> Accessing array elements <input type="checkbox"/> Array construct <input type="checkbox"/> Associative arrays <input type="checkbox"/> Foreach construct <input type="checkbox"/> Comparison operators <input type="checkbox"/> Assignment <input type="checkbox"/> Accessing MySQL databases 	<p>Show exercises that are</p> <ul style="list-style-type: none"> <input type="radio"/> Already Attempted <input type="radio"/> Not Attempted <input checked="" type="radio"/> Both
	<p>Show exercises that you have</p> <ul style="list-style-type: none"> <input type="radio"/> Not Completed Successfully <input type="radio"/> Completed Successfully <input checked="" type="radio"/> Both
	<p>Show exercises that are</p> <ul style="list-style-type: none"> <input type="radio"/> New <input type="radio"/> Not New <input checked="" type="radio"/> Both

Figure 8.4. Exercise search page

The system allows releasing the exercises in batches to the students. This is useful because too many exercises at once may be too much for some students. Each exercise can be assigned a date of release. Exercises which have been released, but which have a date of release greater than a specified date, are displayed as new exercises. The student may also select to display exercises that are new, not new or both. Once the student selects the necessary search criteria, s/he can return to the

exercise selection page. Now, this page displays a list of exercises that match the search criteria. The student can select the exercise that s/he wants to attempt.

Once a student has chosen whether the next exercise should be suggested by the system or searched for, this mode remains active for the current login session unless the student explicitly changes it. This makes it easier for each student to work based on his preference without having to choose the mode over and over again.

8.1.2 Solving an Exercise

Once an exercise is selected, the Exercise Solution page of the ITS is displayed. This page is illustrated in *Figure 8.5*. The text of the selected exercise is displayed on the top of the page. The left hand pane of the page contains the area where the answer is to be entered and the right hand pane provides feedback to the student. The bottom section of the page is used to display the page generated by the student's code once it does not contain any syntax errors.

The answer area is divided into three sections. The darker sections on either side contain any code that is supplied by the exercise when the exercise is a gap exercise. The student enters code into the lighter area in the middle. All code is analysed in conjunction with whatever is supplied by the system. If no code is supplied by the system, the darker areas are left blank and the student needs to write complete PHP programs.

While entering code, the student has many options. He/she may choose to save whatever is already typed into the area for later use. The program is then saved onto a predefined file in the server and can be reloaded into the answer area. One answer per exercise can be saved and reloaded in this manner. The student can also erase everything in the answer area and restart the exercise from the beginning.

When the student has entered some valid PHP code, he/she can choose to view its output. As stated above, the output from the student's code is displayed in the area at the bottom. The code is not analysed by the ITS and no feedback is provided. Therefore, the student is responsible for deciding whether the program is correct or not. In order for the system to analyse the solution, the student must click 'Check My Answer'. When this button is used, the ITS analyses the student's answer as described in Chapter 4.

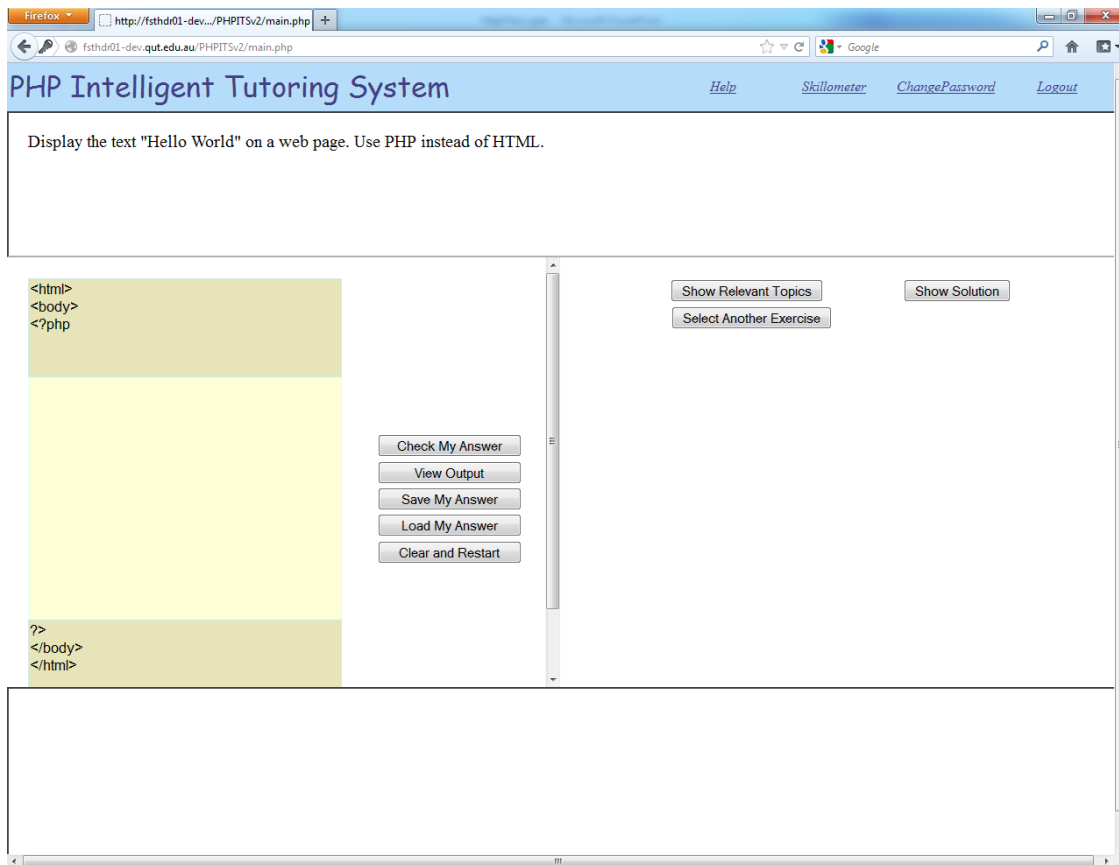


Figure 8.5. The solution page.

If the analysis results in a syntax error being identified, the position of the syntax error is determined as described in Section 4.5.2. The node containing the syntax error is then highlighted with an error message saying that the code contains a syntax error in the highlighted position. The student can double click on the highlighted node to obtain additional information about the error. This displays an error message based on the type of error returned by the grammar. It should be noted here that errors in semantic analysis are handled in a similar manner since they are considered to be lightweight errors that are similar to syntax errors. For example, if the student calls a non-existent function, or if the number of parameters in a function call does not match the number of parameters in the corresponding function definition, it is treated in a similar manner to a syntax error.

If no syntax errors are found, the program analysis continues as described in Chapter 4. If any logical errors are found, it displays the error message ‘Your program is incorrect’. At this point, no further information about the error is displayed. However, further feedback can be requested by the student if s/he requires it.

Next, the student can choose to either correct his answer by himself or ask for hints. Hints are provided for two questions ‘What is Wrong’ and ‘How Do I Solve It’. Each of these questions has two levels of hints and they can be accessed if the student wants.

The system works on only one error at a time. This is determined by the order of the sub-goals in the overall goal described in Section 4.4.2. During the analysis process, these sub-goals are tested one by one in the order given in the exercise specification as described earlier. The moment one of these sub-goals is not satisfied, the system indicates that the program contains an error. Any further information refers to this specific error. Other sub-goals are only analysed ones this particular sub-goal is satisfied by the program. Links to web pages that are relevant to that specific error are also displayed at this time.

In addition to this analysis process, the system also has two other levels of support for students. The student can ask to view the entire solution to the exercise. The student can also decide to display all the web links that are relevant to the given exercise. This displays a list of links that are relevant to all the topics covered by the exercise.

The PHP ITS is a new system. Therefore, it is possible that there are bugs in the program code. Additionally, it is possible that a correct solution submitted by a student is out of the scope of the thesis. In such situations, it is possible that the system will identify a correct solution as incorrect. If the student is convinced that his or her answer is correct but the system refuses to accept it, the student is given the opportunity to register his concern regarding this matter. An email is then generated to the administrator indicating the concern. These concerns can then be handled by the administrator. If an error is present in the student’s program, an explanation of the error can be provided via email. If the student’s concern proves to be accurate, the administrator then takes action to correct the error in the system. This is useful in order to develop the system further.

Although this is theoretically the process followed by the system, a problem occurred during the implementation. Due to the server used being administrated by QUT’s IT Services section and the development team having little control over it, it proved impossible to actually send email through this server. Although email messages were initially generated, they were not actually sent but were automatically

saved to a folder on the server. This meant that the administrator could not receive such email messages. Therefore, the email generation feature was disabled in the final system but the student's concern was recorded in the database. The administrator then responded to such concerns by manually checking for such entries in the database.

8.2 STUDENT MODULE

Section 8.1 describes the user interface, the main focus of the communications module, of the PHP ITS. In order to suggest exercises that are most suitable for the student as described in Section 8.1.1, it is necessary for the system to maintain a model of each student. This is done in the student module. This section outlines the design of the student module in the PHP ITS.

The PHP Intelligent Tutoring System uses the concept of knowledge tracing for the student module. As the student works with the tutoring system, it is expected that his or her knowledge regarding relevant subject matter will change. Knowledge tracing attempts to model this changing state of knowledge (Corbett & Anderson, 1995). In order to do this, it is necessary to break down the subject matter into some knowledge components (KCs). The KCs selected in this research are the specific topics of PHP programming as outlined in Table 8.1.

The actual student model used is an indication of the knowledge level of each student regarding each of these topics. The knowledge level is maintained as a probability. This is necessary because it is impossible to decide whether a student definitely has or does not have knowledge about a certain KC. The probability accounts for this uncertainty. A knowledge level of 1 indicates that the student has mastered the topic while a knowledge level of 0 indicates that the student has absolutely no idea of the topic.

8.2.1 Equations for Updating the Student Model

As mentioned previously, the knowledge level of the student regarding a topic is expected to change as learning occurs through interaction with the PHP ITS. The student model needs to be updated to reflect this change. This section describes the process used to initialise and update the student model.

In order to update the knowledge level of a student based on the interaction with the tutor, it is necessary to have a method to map the interactions to the KCs. This is done through the overall goal of an exercise as described in Section 4.4.2. The overall goal is considered as a set of sub-goals that need to be achieved for the program to be correct. Each sub-goal can be one or more facts in the exercise specification. Each sub-goal is mapped to one or more of the topics listed in Table 8.1. When a student attempts an exercise and the final state of their solution matches a specific sub-goal in the overall goal, the probability that the topics related to this sub-goal are known by the student increases. Similarly, if a sub-goal is not matched, i.e. the facts in the sub-goal are not present in the final state, the probability that the topics related to this sub-goal are known reduces. The student model is updated in this manner, each time a student decides to check the answer as described in Section 8.1.2.

The process used for updating the student model in the PHP ITS is a simplified version of the model proposed by Reye (2004). It is based on the theory of Bayesian Belief Networks (BBN). This method of modelling is necessary because the fact that the student's program matched a certain sub-goal cannot be taken as a certain indication of the student's knowledge of the topic. It is possible that a student made a lucky guess. Similarly, a student can make an inadvertent slip and not match a sub-goal although s/he knows the relevant KCs. However, given some evidence of the prior knowledge level of a student about a particular KC, it is easier to gauge whether such guesses or inadvertent slips were made. A probabilistic estimate as to the actual knowledge of the student can be made based on the system's knowledge about the prior knowledge level of the student. BBNs are a very useful method of modelling such unreliable pieces of information. This model specifies that an interaction provides clues about two distinct pieces of information. The first piece of information is the knowledge level of the student of that KC before the relevant interaction. This becomes important during two occasions: before the very first interaction and when the student knowledge changes as a result of something outside the system. This is reflected in the first phase of updating of the student model. The second piece of information is the knowledge level of the student of the particular KC after the relevant interaction. The second piece of information is reflected in the second phase of the update process.

Table 8.1

List of Knowledge Components

Structure of a PHP web page
Displaying data on a web page in PHP
Variables
Double quoted strings
Single quoted strings
String concatenation
Arithmetic operators
Increment/decrement operators
If-else structure
Nested if structure
Switch structure
Forms
Posting data in forms
Textbox control
Select control
Submit button
Getting data in forms
Random number generation
PHP functions
Function parameters
Returning values from functions
For construct
Nested for loops
HTML Tables
HTML borders
Dynamically adding rows to a table
While construct
Explicitly assigning data to an array
Accessing array elements
Array construct

Figure 8.6 shows the equations used for the first phase of updating the student model for the n^{th} interaction. $p(L_{n-1})$ is the system's belief that the student already knows the relevant KC prior to the interaction. O_n is an element in the set of possible outcomes of the interaction. $p(O_n | L_{n-1})$ represents the system's belief that outcome O_n will occur when the student already knows the KC while $p(O_n | \neg L_{n-1})$ represents the system's belief that this outcome will occur when the student does not know the KC under consideration. Therefore, this equation results in the calculation of the probability about the knowledge level of the student before the n^{th} interaction, given an outcome.

$$p(L_{n-1} | O_n) = \frac{\gamma(O_n)p(L_{n-1})}{1 + [\gamma(O_n) - 1]p(L_{n-1})}$$

$$\gamma(O_n) = \frac{p(O_n | L_{n-1})}{p(O_n | \neg L_{n-1})}$$

Figure 8.6. Equations for first phase of updating the student model.

The second phase reflects the knowledge level of the student after the interaction. This is affected by both the state of knowledge before the interaction and the outcome of the interaction. The final knowledge level of the student after the interaction is a combination of the first and second phases of updating the student model. The equations that show the combined effect of the update process are shown in Figure 8.7. Based on the functionality, $\rho(O_n)$ represents the rate of remembering and $\lambda(O_n)$ represents the rate of learning.

8.2.2 Assumptions

The above description is a generalised process for updating the student model for any architecture that uses probability theory for student modelling (Reye, 1998). However, many simplifications are made considering the particular usage in the PHP

ITS and some empirical results. In the ITS, a particular sub-goal is taken to be either correct or incorrect as mentioned in Section 8.2.1. Therefore, there are only two possible outcomes of an interaction: either correct or incorrect. Let C_n represent the correct outcome and $\neg C_n$ represent the incorrect outcome. It is also assumed that the student will not forget something he or she already knows as a result of interacting with the ITS. Therefore, the rate of remembering, $\rho(O_n)$ is always assumed to be 1, i.e. no forgetting occurs. It is assumed that the probability that the student will make the transition from the unlearned state to the learned state is independent of the outcome.

As in any subject, the different topics have pre-requisite relationships between them. In other words, some topics need to be learned before others can be studied. The method of handling such pre-requisites using Dynamic Belief Networks is described by Reye (1998). Although it would have been ideal to model these pre-requisites, the effort required for this makes it impossible to achieve within the time constraints of the PhD. Therefore, it has been assumed that no prerequisite relationships exist between the different KCs. This means that there are no conditional probabilities linking the different rules. Under these conditions, the equations in *Figure 8.7* are simplified further. Substituting the two possible values for O_n in the resultant equation gives the two equations shown in *Figure 8.8*.

$$p(L_n | O_n) = \frac{\lambda(O_n) + [\rho(O_n)\gamma(O_n) - \lambda(O_n)]p(L_{n-1})}{1 + [\gamma(O_n) - 1]p(L_{n-1})}$$

$$\rho(O_n) = p(L_n | L_{n-1}, O_n)$$

$$\lambda(O_n) = p(L_n | \neg L_{n-1}, O_n)$$

Figure 8.7. Equations for calculating combined effect of two phase updating of the student model.

$$p(L_n | C_n) = p(L_{n-1} | C_n) + \lambda(C_n)(1 - p(L_{n-1} | C_n))$$

$$p(L_n | \neg C_n) = p(L_{n-1} | \neg C_n) + \lambda(\neg C_n)(1 - p(L_{n-1} | \neg C_n))$$

Figure 8.8. Modified equations for two phase update of the student model.

As shown by Reye (2004), these are actually the equations used to update the student model in the ACT programming tutors (Corbett & Anderson, 1992, 1995) although the notation used is slightly different. Corbett and Anderson (1992) also used some estimates for certain parameters in this model based on their previous empirical results. These parameter estimates, translated into the notation used here, is shown in Figure 8.9. Here, $p(L_0)$ refers to the initial probability of a student knowing a topic, before the student uses the system for the first time. Setting this to 0.5 indicates that there is an equal chance of a student knowing or not knowing each topic. As described above, the value of λ is taken to be a constant and is taken to be independent of both the outcome and the interaction number. Similarly, the probabilities $p(C_n | \neg L_n)$ and $p(\neg C_n | L_n) = 0.2$ are considered to be independent of the interaction number. By substituting these values, it can be seen that, under these simplifications $\gamma(C_n) = 4$ and $\gamma(\neg C_n) = 0.25$.

$$p(L_0) = 0.5$$

$$\lambda(C_n) = \lambda(\neg C_n) = 0.4$$

$$p(C_n | \neg L_n) = 0.2$$

$$p(\neg C_n | L_n) = 0.2$$

Figure 8.9. Empirical parameter values.

Using these empirical values the equation for the two phase update of the student model in Figure 8.8 is translated into the equations shown in Figure 8.10. These are the equations that are used for the actual updating of the student model in the PHP ITS.

$$p(L_n | C_n) = \frac{0.4 + 3.6p(L_{n-1})}{1 + 3p(L_{n-1})}$$

$$p(L_n | \neg C_n) = \frac{0.4 - 0.15p(L_{n-1})}{1 + 0.75p(L_{n-1})}$$

Figure 8.10. Final equations for two phase updating of student model.

8.2.3 Updating the Student Model in the PHP ITS

The previous sections described the formulation of the equations of the equations used for updating the student model in the PHP ITS. This section discusses the actual process used in more detail.

When a new student starts to use the PHP ITS, a student model is created for that student. At this point, it is assumed that the probability that he or she knows each and every topic is 0.5 as described in Section 8.2.2. Before the student can proceed to use the system, the student model is updated to try to more accurately reflect the knowledge of the student. This is done by getting each student to complete a pre-test before he or she can proceed with any other interaction with the system. As stated in Section 8.1, the pre-test consists of a set of multiple choice questions. Each question is linked to one or more of the topics given in Table 8.1. The student is permitted to leave the answers blank, indicating that they do not know the answer to that particular question.

Once the student submits the answers, they are analysed to check whether they are correct. Based on whether the answer is correct or not, the corresponding topics in the student model are updated to reflect the system's belief about the student's knowledge of those topics. This is done using the single phase updating of the student model since the system does not provide any feedback on which answers are correct and does not provide any instruction at this time. The second phase accounts for any learning gained by the interaction with the system, and is therefore irrelevant for the pre-test.

Since this is the first interaction of the student with the system, the equations in *Figure 8.6* can be re-written as shown in *Figure 8.11*. Substituting the values of the parameters given in Section 8.2.2 and enumerating for the possible outcomes, this equation reduces to the equations shown in *Figure 8.12*. Therefore, if the student answered a pre-test question correctly, the probability that the student knows all related topics is updated to 0.8. Similarly, if the student answered the question incorrectly, the probability that the student knows all relevant topics is updated to 0.2.

Once the initial student model is established in this manner, the student is allowed to interact with the rest of the system. From this point on, the student model is updated using the two phase approach and the equations given in *Figure 8.10*. Each time a student provides a solution to an exercise, the system checks whether sub-goals are met as described in Section 8.2.1. If the sub-goal is met, the system uses the first equation in *Figure 8.10* to update its belief about all relevant topics. On the other hand, if a certain sub-goal is not met, the system uses the second equation in *Figure 8.10* for the update.

$$p(\mathbf{L}_0 | \mathbf{O}_1) = \frac{\gamma(\mathbf{O}_1) p(\mathbf{L}_0)}{1 + [\gamma(\mathbf{O}_1) - 1] p(\mathbf{L}_0)}$$

Figure 8.11. Single phase update of the student model for the first interaction.

$$\begin{aligned} p(\mathbf{L}_0 | \mathbf{C}_1) &= 0.8 \\ p(\mathbf{L}_0 | \neg \mathbf{C}_1) &= 0.2 \end{aligned}$$

Figure 8.12. Final equations for updating the student model based on the pre-test.

8.3 TEACHING MODULE

The main purpose of a tutoring system is to increase the knowledge level of the students that use the system. The teaching module is the component that is directly concerned with this aspect of the system. It uses information from the domain module to understand a student's interaction with the system and combines it with

information from the student module to decide on the best instruction to provide the student at any given time.

The teaching module in the PHP ITS is based on methods that were utilised in previous ITSs. No new teaching approaches have been introduced during the course of this research project. However, existing methods have been analysed to decide on an approach that is most suitable for the PHP ITS. The teaching module used here consists of two main components. The first component provides assistance to students when solving programming exercises. The second component provides assistance to students when selecting the most suitable exercise to attempt at any given time. The next two sections discuss each of these components.

8.3.1 Assistance for Solving Exercises

8.3.1.1 Viewing Web Pages

The PHP ITS provides many forms of support for students to help them solve exercises. One such form of support is the ability to display relevant web pages. Although many ITSs exist to provide problem solving practice, few provide conceptual and procedural information, leaving this task to be predominantly performed by teachers. In accord with the approach suggested by Gong, Beck and Heffernan (2012), the PHP ITS addresses this problem by utilising resources that are readily available on the Internet and providing links to these pages as and when appropriate. This is accomplished by considering the different sub-goals that need to be achieved in order to correctly solve an exercise. As mentioned in Section 8.2.1, each sub-goal for the exercise is mapped to one or more topics. The system also stores a list of web pages that contain information that is relevant to PHP programming. Each topic is linked to one or more of these web pages. A student can ask the system to show topics that are relevant to a particular exercise. The system then finds a list of all topics that are covered by the exercise, and thereby a list of web pages that are relevant to the exercise and displays links to these pages. This makes it easier for the student to sort out which web pages he should be reading to gain the relevant knowledge.

Asking the system to show relevant web pages also indicates that the student is unfamiliar with the topics covered by the exercise. Therefore, the student model is again updated as explained in Section 8.3.3.

The links mentioned here are also used to provide feedback when a student submits an incorrect solution to an exercise. As described in Section **Error! Reference source not found.**, the final step of program analysis is to check whether the overall goal has been achieved. If not, there is an error in the student's program. The overall goal is considered as a set of sub-goals. The relative priority of these sub-goals is specified by the order in which they are listed. This means that sub-goals that are listed earlier are considered to be more important than sub-goals that are specified later. During goal checking, the sub-goals are checked in the order specified. When a particular sub-goal is not achieved, the system finds topics that are relevant to that specific sub-goal and thereby finds web pages with relevant information. Links to these web pages are then displayed so that the student can read information that is immediately related to what he got wrong.

8.3.1.2 Accessing Feedback

An important consideration in ITS research is whether a student should be provided feedback proactively by the system or whether the system should wait for the student to request feedback. Although many different ideas have been presented in the literature, this research is based on the idea that a student should only be provided detailed feedback if he or she desires it. Therefore, when a student submits a solution to be analysed, the only form of feedback initially provided by the system is whether the solution is correct or incorrect.

Section 8.1.2 gives an overview of this analysis. The following provides more details. The analysis is done in two steps. First, the solution is checked to see if it contains any syntax errors. If it does, these errors are highlighted. This syntax error analysis is implemented through the support of the PHP and HTML grammars described in Section 4.5.2. The grammars detect the position of any program code that it cannot match against any of its tokens. The position and error type generated by the grammar are matched against a suitable error message. This error message is displayed if the student double clicks on the highlighted error to obtain further information. This syntax error analysis process is sometimes not very reliable since it depends on the type and position of the error identified by the grammar. This information is highly dependent on the actual implementation of the grammar and can result in erroneous identification of error locations. For example, if an opening quote exists but no corresponding closing quote, the grammar is sometimes unable to

pinpoint the location of the error. This results in the highlight being in a wrong location or no highlight appearing at all. Although a relevant message is displayed in the system, this process is a shortcoming that needs to be taken in to account in the future.

Once the student's solution is syntax error free, the second step in the analysis process takes place. At this point, any sub-goals that are not satisfied are identified as described in Section 8.2.1. Again, the order of priority is set by the order in which the sub-goals are listed. Analysis stops as soon as one sub-goal is not met and error messages are only displayed for this specific sub-goal. At this point, no detailed error messages are displayed. The displayed message only indicates whether the solution is correct or incorrect. It is at the student's discretion to decide whether he or she wants more feedback or not.

Each sub-goal is linked to four error messages, two levels for messages on what is wrong and two on how to solve the issue. The first level in each type of message contains a general description while the second level is more detailed and refers to the exact error. For example, Table 8.2 shows the error messages for a sub-goal to check for a condition if the value entered in a textbox is greater than 10. Assume that after submitting the form, the value has been stored in the variable \$x.

Table 8.2

Feedback Messages for Checking if a Value Entered in a Textbox is Greater than 10

<i>Message Type</i>	<i>Level</i>	<i>Feedback Message</i>
What is wrong?	1	Your program does not check for the necessary condition.
What is wrong?	2	Your program does not contain a check to see whether the value in the textbox is greater than 10.
How to solve?	1	Include a conditional statement to check whether the value of \$x is greater than 10.
How to solve?	2	Include the conditional statement <code>if(\$x>10)</code>

When the system informs a student that the program contains an error s/he can opt to see either what is wrong with his program or how to solve the problem. Depending on the type of message that is requested, the appropriate first level message is displayed. The student can then request the same type of message again

to display the second level message. S/he can also request the other type of message if it is considered more appropriate.

In addition to errors that are caused by the program not satisfying certain sub-goals, the PHP ITS also identifies unnecessary program statements in the code as described in Section 4.5.5. In this case, the error messages depend on the type of extra program statement. Table 8.3 shows the error messages shown by the system for a program that contains an unnecessary assignment statement in line number 10.

Table 8.3

Feedback Messages for an Unnecessary Assignment Statement in Line 10

<i>Message Type</i>	<i>Level</i>	<i>Feedback Message</i>
What is wrong?	1	Your program contains some unnecessary code.
What is wrong?	2	Your program contains an unnecessary assignment statement.
How to solve?	1	Delete the unnecessary assignment statement.
How to solve?	2	Delete the unnecessary assignment statement in line 10.

In addition to displaying error messages, the program analysis process also results in the student model being updated to indicate whether the student knows or does not know the topics covered by the various sub-goals in the exercise.

8.3.2 Assistance for Selecting Next Exercise

An Intelligent Tutoring must be capable of varying its interaction based on the current knowledge of the student. In the PHP ITS, this is accomplished by assisting students to select the next best exercise that is suitable for them. The next best exercise is selected based on the topics covered by each exercise and the probability that the current student knows each of these topics.

In order to do this, it is necessary to find a method of identifying whether a topic has been learnt or not. This is done by setting a threshold probability above which the topic is taken to be in the learned state. The probability used here is 0.85. This value is taken with the intention that a student does not have to have 100% (a probability of 1) proof that s/he has learned the topic. Very often, students make mistakes or slips, even though they know the subject matter. 0.85 is taken to be a considerably high enough value to consider the topic to be in the learned state.

The most suitable exercise for a student at any given time is taken to be the exercise which covers the least number of topics that are not in the learned state. If there is more than one problem with the same number of topics not in the learned state, these problems are ordered randomly. The reason for selecting the problem with the least number of topics that are unknown is that this ensures that not too much new material is included in the exercise. This allows the students to gradually build up their knowledge of the topics without working on an exercise that has so many new topics that it is extremely challenging. This concept is based on Vygotsky's (1978) work on the Zone of Proximal Development (ZPD). The ZPD is the area where a student can comfortably learn. It is slightly higher than the student's current level of knowledge but not too high.

All the available exercises are formed into a list in the manner described above. The exercises at the end of the list contain a large number of unlearned topics while the ones at the beginning of the list have fewer unlearned topics. This makes it easier for the student to decide which exercise s/he should attempt next.

8.3.3 Viewing the Suggested Solution

If all other forms of support fail, students may wish to view the entire solution to the exercise. This is achieved by storing an ideal solution against each exercise. However, it should be noted that many other alternative solutions to the exercise are also accepted as correct as explained earlier. The ideal solution is only stored in order to provide a student with a possible solution if it is required.

When a student requests to view a solution this provides evidence that the student is not familiar with at least some of the topics covered by this exercise. While the evidence is a bit vague, the approach adopted here is that the student model is updated to indicate that the student did not achieve any of the sub-goals of the exercise specification.

8.4 IMPLEMENTATION DETAILS

The above sections describe the user interfaces of the PHP ITS and the theory behind the student and teaching modules. This section discusses the implementation of the system. Section 8.4.1 describes the software architecture of the system. It looks at the programming languages and tools that integrate to result in the PHP Intelligent Tutoring System. Section 8.4.2 discusses the structure of the database that

has been used in the system. Finally, Section **Error! Reference source not found.** goes on to discuss some issues that arise when implementing the models described above using the selected software tools and how these issues were overcome.

8.4.1 Software and Tools

Figure 8.13 shows the software architecture used during the development of the PHP ITS. As described in Section 8.1, the ITS is a web based system and therefore, web development languages were used to create it. HTML was used to create web pages and CSS was used to maintain consistent styles across the system. Javascript was used for client side scripting, mainly for validating data. The dynamic aspect of the web pages was developed using PHP ("PHP Hypertext Processor," 2011). This language was selected for many reasons. It is free and is easily downloadable from the web. It is also very versatile. Another benefit of using PHP is that this is the language taught by the ITS. When PHP was used for the development of the system as well, it was possible to execute the results of the students' answers without having to include any external simulators. The PHP interpreter used to interpret the system code was used to interpret the students' code as well.

Two external tools were used for the analysis of computer programs written by students. As described in Section 4.5.2, grammar files are used during one part of program analysis. These grammar files were developed using ANTLR which is a tool that supports the creation of grammar files. ANTLR also allows the creation of ASTs from the code supplied as a text. Although ANTLR creates outputs that are suitable for program analysis, it is not possible to access ANTLR through PHP. Therefore, an intermediate language was necessary to communicate between PHP and ANTLR. The C language is known to integrate very easily with PHP and it is also possible to access ANTLR through C. Therefore, this was selected as the intermediary language. The ANTLR C Runtime library was used to access ANTLR from a normal C program, so that this program could analyse the ASTs generated by ANTLR.

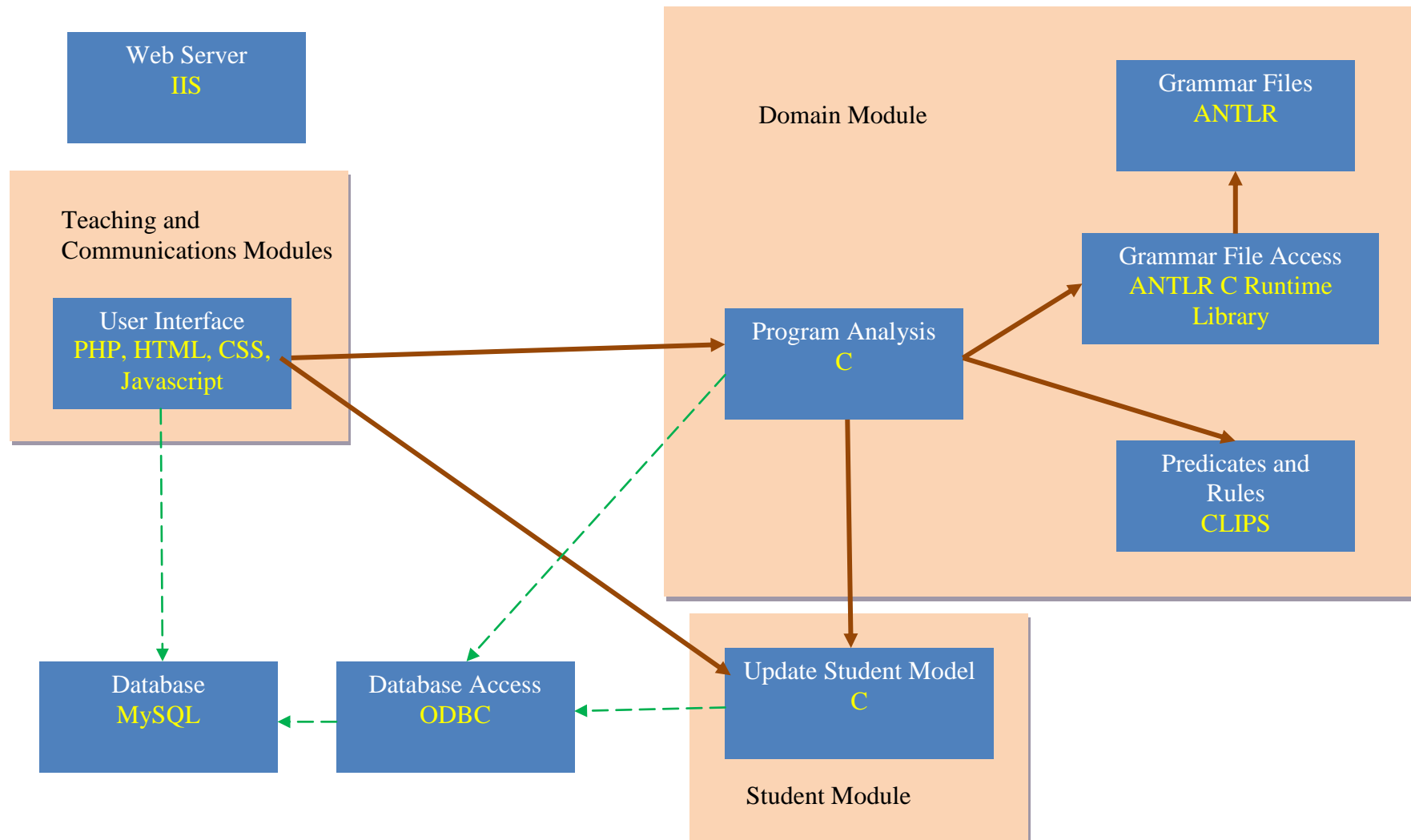


Figure 8.13. Software architecture used in the PHP ITS.

This meant that, as explained in Section 4.5.3 it was necessary to work with predicates and rules using this C program. The C language does not contain methods for logic programming. CLIPS (Riley, 2011) is a tool developed with the explicit purpose of handling logic programming through the C language. This tool was used to handle predicates and rules during program analysis. Since it can directly be accessed using C, no intermediate software was necessary.

In addition to program analysis, another important part of the PHP ITS is the updating of the student model. As explained in Section 8.2, this is done at the time of the pre-test and whenever the student clicks ‘Check My Answer’. It is also updated when other buttons in the interface are pressed, as explained in Section 8.3.1. This meant that the program to update the student model needed to be called both from the interface (PHP) and from the program analysis code (C). In order to make it easy to access this program from both these languages, the program to update the student model was written in C.

Several C language compilers are available and their functions slightly differ from each other. The Integrated Development Environment (IDE) used for the development was Eclipse ("Eclipse," 2011). The main reason for this is that it allows working with many of the programming languages and tools used in this project at the same time. It is possible to open PHP, HTML, Javascript, C and ANTLR files and work with them in a single environment. The C compiler that is most suitable for this environment is MinGW ("MinGW - Minimalist GNU for Windows,") so this was the compiler used during the development process.

A database is an imperative part of any considerable software system. The database management system used in the PHP ITS was MySQL ("MySQL,"). The main reason for this is that it is also free and is the database that is used most often in PHP applications. PHP includes a native interface to connect directly to MySQL databases. Therefore, the database was accessed directly from the PHP interface. However, it was also necessary to access the database through the C programs for analysing the students’ code as well as for updating the student model. MySQL provides a client library for directly connecting to a C program known as Connector/C or libmysql. Although this allowed direct connection, it caused problems when using this library with the ANTLR C runtime library. Many clashes in definitions occurred when trying to use these libraries together. Therefore, a

different method of accessing the MySQL database from a C program was necessary. In this case, the selected method was to access the MySQL database through an Open Database Connectivity (ODBC) connection using the MySQL ODBC connector ("Connector/ODBC, "). It should be noted that the connector used here was the 32 bit version.

Since the PHP ITS is a web application, it needs to be deployed on a web server. The web server used during development was Apache ("Apache, "). The main reason for this is that the development environment is set up using XAMPP ("XAMPP, ") on a Windows 7 PC. XAMPP is an integrated package which makes it easy to install PHP, MySQL, the Apache server and several other software products using a single installation. However, the deployment environment was somewhat different to this. It used the IIS server installed on a Windows Server 2008 operating system. This was necessary due to practical issues existing at the Queensland University of Technology where this research was carried out. The server provided for deployment was preinstalled with Windows Server 2008 and a PHP system using IIS was already running on it. The decision to use the existing web server was taken in order to avoid conflicts and also to make the URLs easier for the students. The PHP ITS worked seamlessly in both the development and deployment environments.

8.4.2 Database Structure

The database used in the PHP ITS has a fairly complex structure as shown in *Figure 8.14* and *Figure 8.15*. Note that these figures need to be read in conjunction with the lines reaching the right edge of *Figure 8.14* joining those in the same vertical position on the left edge of *Figure 8.15*. The *problem_mst* table is the main table that contains exercise information. The *problem_mst* table is linked to the *problem_sol_ref* table that contains a row each for each sub-goal in the final goal. These sub-goals are matched to function names that are defined in CLIPS. The relevant CLIPS function is called to check whether each sub-goal is satisfied. The *problem_message_ref* table contains message references for each of the four possible types of messages for each sub-goal as described in Section 8.3.1.2. The actual text of the messages is stored in the *message_mst* table. The *goal_topic_ref* table contains references to the topics that are linked to each sub-goal as described in Section 8.2.1. The *topic_mst* table contains the actual topics. The

topic_webpage_ref table links each topic with web pages that are defined in the *webpage_mst* table.

A separate set of tables is maintained in order to handle sub-plans. The *prob_subplan_ref* table creates the link between different sub-plans that are possible for a single problem. The *subplan_pre_post_ref* table contains the CLIPS function names for each sub-goal in the sub-plan while the *subplan_message_ref* table contains links to all the possible types of messages when a sub-goal in the sub-plan is not satisfied. The *subplan_topic_ref* table contains references to the topics that are linked to each sub-plan sub-goal.

The next set of important tables is formed around the *user_mst* table which contains details for each user. The *user_topic_ref* table contains the student model, showing the knowledge level for each topic of the user. The *sntx_error_dat* and the *user_error_ref* tables contain references to errors identified during program analysis. The *user_pretest_ref* and the *user_posttest_result* tables contain the pre and post test data for each user.

8.4.3 Implementation Issues

Chapter 4, Chapter 5 and Chapter 7 described the theoretical procedure for analysing PHP programs that is used by the PHP ITS. However, some of these theoretical aspects result in challenging situations when implemented using the technologies described in Section 8.4.1. This section gives a brief description of some of the more important issues and how they are solved.

During the analysis of PHP selection statements, it can be seen that the facts that exist in the *if* section are different from those that exist in the *else* section (Chapter 5). When handling functions (Section 6.2) or loops (Chapter 7), the facts that exist within the sub-plan are different from those that exist outside. During implementation, facts are handled using the CLIPS tool (Section 8.4.1). This tool does not provide a means of separating facts into groups. Therefore, this situation is handled by maintaining different set of facts in separate CLIPS sessions known as environments. In other words, implications are handled by having the main set of facts in one CLIPS environment and the implied set of facts in another CLIPS environment. The *clips_env_dat*, *env_fact_ref* and *loop_env_ref* tables in the database (Section 8.4.2) are used for the purpose of maintaining these links.

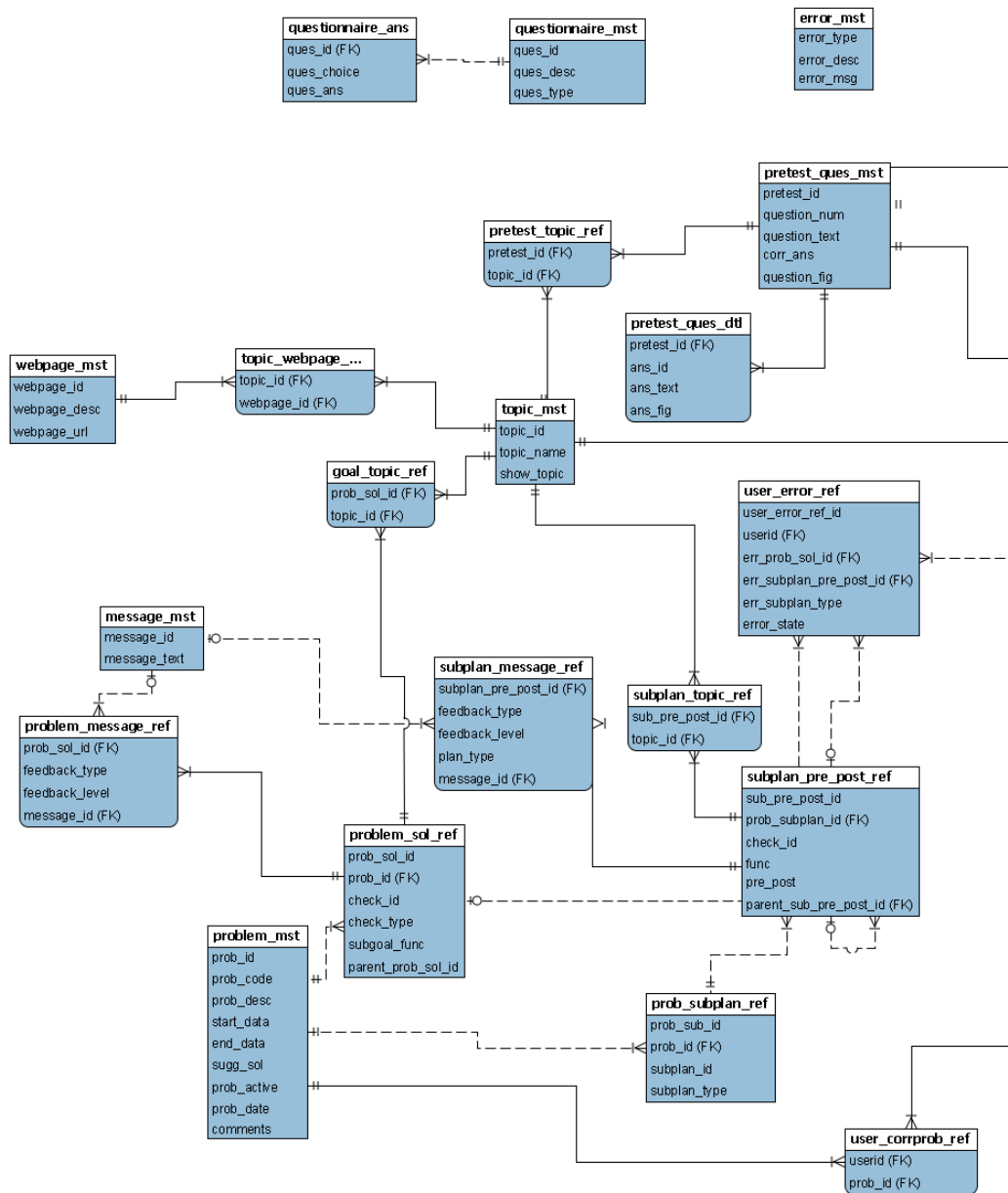


Figure 8.14. Database model of the PHP ITS – 1.

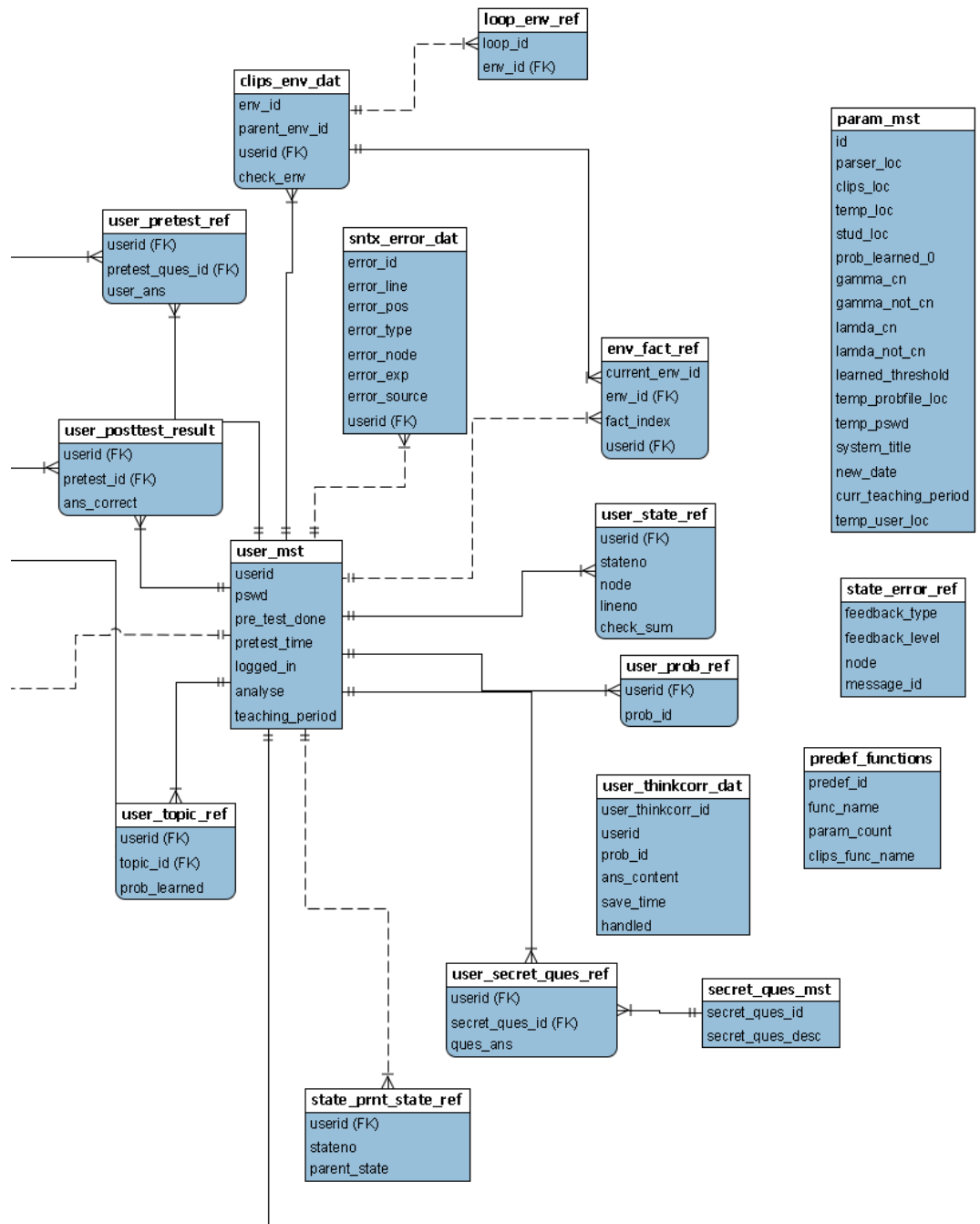


Figure 8.15. Database model of the PHP ITS – 2.

Another important consideration is the implementation of mathematical facts such as $Add(x,y,z)$. As explained in Section 4.4.1.1, this predicate becomes true if z is the sum of x and y . An infinite number of such facts need to be created in order to handle all the possible mathematical calculations. Since this is impractical, these facts have been implemented in the form of a symbolic calculator. The symbolic calculator finds the sum of x and y and, thereby, the value of z . Symbolic calculations are necessary as the initial values of some variables are given in symbolic form as described in Section 4.4.2.

As described in Section 4.5.5, a set of statuses are maintained in order to identify unnecessary program statements. This is implemented using a *CurrentState* predicate to hold a pointer to the current status. When a new status is created, the pointer to the previous one is lost. This means that, if unnecessary program statements are found at the end of a program, a new status is created before the extra statements are identified, thereby destroying the link to the status when the goal was actually achieved. In order to avoid this problem, the overall goal is checked to see whether it is satisfied just before a new status is created. Then, if it is satisfied, the statement that is creating the current status is identified as an unnecessary program statement.

As described above, the facts relevant to the *if* and *else* states are maintained in separate CLIPS environments. This means that, if a new variable is created in both these environments, it should be accessible in any environment that corresponds to subsequent program statements. Therefore, at the end of a selection statement, a check is made to see if any new variables were created in all the corresponding sub states. If so, a new status is created and a corresponding variable is also created.

This section discussed some important implementation issues faced when implementing the PHP ITSs theory as an actual program. However, many other minor incompatibilities also needed to be overcome. A brief account of some of these issues can be found in Appendix H.

8.5 CHAPTER SUMMARY

This chapter discussed the implementation of the PHP Intelligent Tutoring System. It discussed the features and functionality of the system and how they relate to the theoretical aspects explained in previous chapters. It discussed the design of

both the student and teaching modules used in the system. It also looked at how different programming languages and software tools were used to develop the system. It briefly examined the database structure and also discussed some situations where the implementation differed from the theoretical viewpoint for technical reasons.

Chapter 9: System Evaluation

This chapter discusses how the PHP Intelligent Tutoring System was evaluated to see whether it achieved its objectives. Section 9.1 explains the evaluation process in detail. It describes how the participants were selected and also discusses the procedures and instruments that were used during the evaluation process. Section 9.2 describes the different versions of the PHP ITS and how they were used for the purpose of analysis. Section 9.3 discusses the results of the evaluation and Section 9.4 summarises the chapter.

9.1 EVALUATION PROCESS OF THE PHP INTELLIGENT TUTORING SYSTEM

As described in Section 3.2.4, the PHP ITS was evaluated using empirical methods. The evaluation process addressed such aspects as the usability of the system, improvement in student knowledge due to use of the system, the appropriateness of the subject matter taught, the effectiveness of the teaching module and the validity of the student module. Both qualitative and quantitative methods were used to carry out this evaluation as outlined below.

9.1.1 Participants

The participants in the evaluation process were postgraduate students of the Queensland University of Technology (QUT) who wanted to study web development using PHP. The students taking the unit had no prior background in programming using PHP. The unit was an optional advanced reading module and offered in both the first and second semesters of 2012. It was administered during a typical 13 week semester. The first six weeks of the unit consisted of the students using the PHP Intelligent Tutoring System to study introductory material by themselves. They needed to work through a set of exercises that were released each week. No lectures or tutorials were provided by a human tutor. The ITS was used as a stand-alone education system with links to web pages containing relevant reading material. In the second half of the semester, the students followed a carefully selected textbook to learn more advanced features of PHP development. They were not required to use

the PHP ITS during this period. All the material was studied during the students' own time and no fixed class times were administered.

As mentioned in Section 3.2.4, two versions of the system were evaluated. The students in the first semester worked on the first version of the system while the students in the second semester worked on the second version. Students were recruited by open invitation for postgraduate students undertaking the courses IT43 (Master of Information Technology) and IT44 (Master of Information Technology (Advanced)). They needed to fulfil several requirements to be selected for the course. They should have completed at least 48 credit points of postgraduate level IT units. They also needed to have knowledge in basic HTML. It was also stressed that the material was intended for students with no existing knowledge of PHP. Prior knowledge of programming and database concepts was not required. These requirements were used as the PHP ITS was aimed at students learning PHP for the first time with or without prior programming experience. Based on these requirements, 19 students worked with the first version of the system and 15 worked with the second version. Although it would have been better to have more students, only this number showed an interest and satisfied the necessary requirements for participation.

9.1.2 Procedures and Instruments

The participants were required to undergo a pre-test and a post-test. The pre-test was administered when they initially started using the PHP ITS, before they used any of its tutoring functions. This was a multiple choice test with 19 questions (Appendix I). The test was delivered over the web and they could do it in their own time. At the end of the six weeks of using the PHP ITS, the students were required to undergo a post-test. The post-test was administered as part of the mid-semester examination for the unit. The examination contained the same 19 questions from the pre-test, but in a different order, as well as some additional multiple choice questions integrated into the examination. Only the 19 questions from the pre-test were considered as constituting the post-test. The post-test was again done through the web and administered via Blackboard. However, unlike the pre-test, the examination had to be taken on a fixed day and time. The PHP ITS was not available to the students during the examination.

During the students' interaction with the system, their actions were recorded in a database. The recorded data included the date and time of the interaction, the type of interaction (i.e. login, logout, check a solution etc.) and also the actual answer submitted by the student as well as any errors that were identified. Their knowledge level in each topic after each interaction was also recorded.

The students were no longer required to use the PHP ITS after the mid semester examination. A questionnaire was then opened to the students, again using the web. It remained open for two weeks so that students could complete it in their own time. The questionnaire was anonymous. It contained a set of multiple choice and free-answer questions. The multiple choice questions were of two groups. The first group required them to grade their prior knowledge of relevant subject matter on a scale of 1 to 5, 5 being the highest. The second group required them to rate various aspects of the PHP ITS on a 5 point Likert-type scale. The final set of questions in the questionnaire was free-answer questions where the students were free to write anything. The questionnaire is included as Appendix J. Thirteen responses to the questionnaire were received for the first version of the system while six were received for the second version.

The answers to the questionnaires were then analysed to find areas where the PHP ITS could be improved. In order to get a better understanding of the weaknesses identified by the questionnaire, a focus group discussion was conducted during the first iteration only. This was a one hour session during which the participants came together in one room to discuss issues. The participants were given the opportunity to say anything they liked about the overall system. Then, a set of fixed questions (Appendix K) were asked. The discussion was recorded in order to facilitate data collection. The students were assured that their responses would not affect their final grades in any manner.

All participants were required to complete the pre and post-tests. They were also all given the opportunity to complete the questionnaire although it was not compulsory. Only three students decided to participate in the focus group due to the difficulties of finding a time convenient to everyone, and that most of the students would have had to make a special trip to come, to attend at that time.

9.2 DIFFERENT VERSIONS OF THE PHP INTELLIGENT TUTORING SYSTEM

The qualitative data gathered during system evaluation, as well as weaknesses identified by the administrator were used to improve the PHP ITS across multiple versions as described in Chapter 3. The first two versions of the system were developed, evaluated as described in Section 9.1, and identified improvements were carried out – see below. These improvements were identified mainly based on the feedback obtained through the questionnaire and the focus group. The final version of the system, version 3, that fixed issues emerging from the previous evaluations, is the one that is described in this thesis. The following sections outline how the different versions differ across particular aspects of the system.

9.2.1 Feedback to Students' Solutions

The first version of the system had limited functionality as described below.

- It did not have the ability to identify unnecessary program statements as described in Section 4.5.5. Even if the student's solution contained unnecessary code, it was analysed as correct if the statements necessary to achieve the objective were present.
- When a student asked that the solution be analysed, the system immediately displayed an error message if an error was identified.
- There was only one level of error message for each error.

Student feedback on the first version indicated that they were not too happy with the error messages provided by the system. In order to handle this problem, the second version contained improved error messages. When a student asked that their solution be checked, the system first indicated only whether the solution was correct or not. The student was then allowed to ask for additional messages using the mechanisms described in Section 8.3.1.2. This version also provided several levels of messages for each error.

Unfortunately, subsequent student feedback for this version was similar to that for version 1 - students were still not too satisfied with the error messages provided by the system.

In order to improve this further, it was felt that it would be useful if the system could identify unnecessary program statements in students' code. The third version of the system incorporates this change.

9.2.2 Selecting the Next Exercise

The first version of the PHP ITS contained only one way for students to select the next exercise – the list of exercises suggested by the system. Feedback received from students indicated that this list was sometimes cumbersome and some students preferred to just select exercises based on criteria. Additionally, exercises were removed from the list when the system was satisfied that the student knew all the topics covered by the exercises. This was because several exercises covered the same topics and students could therefore gain knowledge about the topics without attempting all of them. However, it seemed that some students still preferred to work on all the exercises in the system and wanted methods of accessing these exercises. In order to handle this, the two modes of exercise selection described in Section 8.3.2 were introduced in version 2.

It appeared that some students also got confused when some exercises were removed from the list of suggested exercises as described above. In order to reduce the confusion and also to give the students an indication of what the system knew about their knowledge, the Skillometer was introduced into the second version of the system.

9.2.3 Handling Students' Doubt Regarding Program Analysis

Student feedback on the first version also indicated that sometimes students felt the system did not analyse their solutions properly, as described in Section 8.1.2. Since this student feedback occurred long after the use of the first version, it was impossible to ascertain whether there was an actual bug in the system or whether the student was simply unable to identify their own error(s). The facility to record their concern over such a program (Section 8.1.2) was incorporated into the second version so that students could communicate such errors allowing the administrator to ascertain whether the problem was in the system or in the student's code and take necessary action. Upon analysis of errors reported by students using this method, it was clear that the majority of time, the students were unable to identify their own

errors. Action was taken to correct any errors in the system the few times that such errors actually existed.

9.2.4 User Interface

In addition to these functional changes, many students using the first version indicated that they were unhappy about the colours, images and fonts in the user interface. Discussion at the focus group revealed that they felt that a very simple colour combination with light colours would be preferred. Therefore, the interface of the second version was changed accordingly to include minimum colours. However, several students using the second version commented that they were unhappy with the fact that the interface was dull and did not have enough colours and pictures. Some also indicated that they were unhappy about the separate frames in which the material was provided, while others indicated that the way the frames were structured made it easier for them to understand the material. Some students welcomed the simplistic and uncluttered interface with an easily navigate-able main menu while others indicated dissatisfaction about the lack of use of complicated interface elements like Flash. It appears that this is very dependent on personal preference and it would be very difficult to achieve a theme that is liked by all.

9.3 RESULTS AND DISCUSSION

The analysis of this data focused on both the educational impact and the affective responses of the students who use the system. As described in Section 9.1.2, the data gathered consisted of both qualitative and quantitative data. The next sections describe how the data was analysed to answer the research questions described in Section 2.7.

9.3.1 Effectiveness of the System

The main measure of the educational impact of the system was the pre and post-test results. Table 9.1 shows these results for the first version of the system while Table 9.2 shows these results for second version. For the first version, the average test score increased from 6.58 to 13.58 from the pre- to the post-test, while the standard deviation reduced from 4.83 to 2.36. For the second version, the mean test score increased from 4.80 to 13.27 while the standard deviation changed from 5.53 to 3.92. *Figure 9.1* shows a graph of the average pre- and post-test scores

achieved by students for the two versions of the system. It can be seen that there was an increase in the average score of the students after using the PHP ITS.

Table 9.1

Pre and Post-test Results for Version 1

<i>Student</i>	<i>Pre-test Score (out of 19)</i>	<i>Post-test Score (out of 19)</i>
1	0	13
2	14	17
3	0	13
4	9	11
5	4	12
6	9	17
7	7	10
8	14	15
9	3	12
10	0	15
11	2	14
12	9	17
13	0	14
14	10	10
15	14	17
16	7	13
17	7	12
18	6	15
19	10	11

A paired t-test with a 95% confidence interval was used to test whether the increase in test score was significant. The test used was a one-tailed t-test since the results showed that the test-score increased as a result of use of the system. The null hypothesis was:

There is no difference between the pre-test and post-test scores of the students who used the system.

The SPSS statistical package gave p-values less than 0.001 for paired t-tests on both versions of the system. This signifies that it is extremely likely that the null hypothesis is false and therefore, there is a significant positive difference between the pre-test and post-test scores of the students who used the system. In other words, the post-test scores are significantly higher the pre-test scores. As described in Section

3.2.4, it was not possible to have a control group for ethical reasons. But the PHP ITS was the only means of learning-by-doing provided to the students. Therefore, it is reasonable to conclude that the increase in test scores was a direct result of using the PHP ITS. This indicates that the PHP ITS was effective in teaching the subject matter to the students.

Table 9.2
Pre and Post-Test Results for Version 2

<i>Student</i>	<i>Pre-test Score (out of 19)</i>	<i>Post-test Score (out of 19)</i>
1	11	14
2	0	10
3	15	19
4	5	14
5	11	16
6	0	17
7	0	13
8	0	8
9	0	9
10	10	14
11	0	15
12	1	4
13	10	15
14	0	16
15	9	15

The learning gain of the PHP ITS was then compared against the learning gain of the JITS (E. R. Sykes, 2006) system. Although the JITS system also works in the programming domain, it teaches the Java language which is different to the PHP language. The average learning gain for JITS was 28.62% while the average learning gain for the PHP ITS was 40.25%. This shows that the PHP ITS is at least as effective as JITS when considering the learning achieved by students.

Another test that was used to measure the effectiveness of the system was a paired t-test comparing the initial and final probabilities that the subject matter was learned. In this case, the student model contains data for the learned probability per student per topic. In order to perform the t-test, the average learned probability across all topics per student was considered. Table 9.3 shows the initial and final

average learned probabilities for each student for the first version while Table 9.4 shows these figures for the second version of the system.

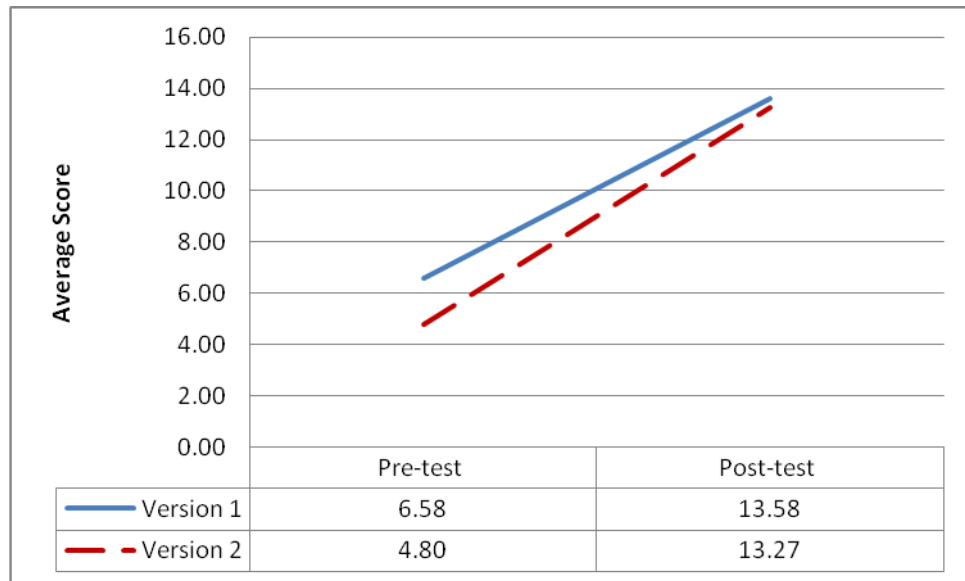


Figure 9.1. Average pre and post-test score.

Table 9.3

Initial and Final Average Learned Probabilities for Version 1

<i>Student</i>	<i>Initial Average Learned Probability</i>	<i>Final Average Learned Probability</i>
1	0.26	0.52
2	0.59	0.80
3	0.26	0.64
4	0.47	0.80
5	0.33	0.54
6	0.41	0.63
7	0.50	0.55
8	0.61	0.73
9	0.31	0.83
10	0.26	0.54
11	0.29	0.47
12	0.47	0.72
13	0.26	0.58
14	0.53	0.55
15	0.50	0.80
16	0.26	0.30
17	0.50	0.69
18	0.39	0.83
19	0.50	0.72

Table 9.4

Initial and Final Average Learned Probabilities for Version 2

<i>Student</i>	<i>Initial Average Learned Probability</i>	<i>Final Average Learned Probability</i>
1	0.50	0.50
2	0.49	0.74
3	0.26	0.64
4	0.61	0.77
5	0.40	0.75
6	0.46	0.54
7	0.26	0.56
8	0.26	0.62
9	0.05	0.50
10	0.26	0.69
11	0.46	0.64
12	0.26	0.60
13	0.27	0.33
14	0.44	0.44
15	0.50	0.51
16	0.45	0.67

A one-tailed paired t-test with a confidence level of 0.95 was carried out using this data. The null hypothesis in this case was as below.

There is no difference between the initial and final learned probabilities of the students who used the system.

Again the SPSS statistics package returned p-values less than 0.001 for both versions, indicating that there is ample evidence the null hypothesis is false. This means that there is a significant positive difference between the initial and final learned probabilities. This result consolidates the fact that the students learned the subject matter after using the system.

A paired sample t-test was carried out between the percentages of students who got each question of the post-test correct in the two versions. The aim of this test was to check whether there was a significant difference between the effects of the two versions of the system. The test carried out in this case was two-tailed since the direction of any variation could not be guessed. The confidence interval used was 0.95. The null hypothesis in this case was as below.

There is no significant difference between the results for each question for the two versions of the system.

The test results showed a p-value of 0.643. This meant that the evidence of the test was not strong enough to reject the null hypothesis and a significant difference could not be concluded.

A correlation was calculated to see whether the amount of help obtained by the student when solving exercises had a significant impact on the improvement in test scores. The null hypothesis tested for this correlation was as below.

There is no significant difference in the improvement of test results with the amount of help obtained when solving exercises.

The improvement in test score was calculated as the difference between the post-test and pre-test scores. The amount of help obtained was found by adding the total number of help requests by the student. Table 9.5 shows the amount of help requested by each student for both versions of the system. In this case, the types of help considered were checking the solution, asking what is wrong, asking how to fix it, showing relevant topics and showing the solution entirely. The number of times the student requested to show their program's output was not considered here since this can anyway be accomplished with a standard PHP Integrated Development Environment (IDE) and was not a type of help offered by the system but merely a way for the students to check the output of their code. A correlation was then calculated between the number of help requests and the improvement in test score. The results of the test are shown in Table 9.6.

It can be seen that these results are significant at the 0.05 level. This means that there is good evidence to reject the null hypothesis. In other words, it can be seen that there is significant difference in improvement in test score based on the number of help requests.

Table 9.5

Number of Help Requests for Each Student

<i>Student</i>	<i>Number of Help Requests</i>
1	59
2	53
3	137
4	155
5	68
6	34
7	58
8	70
9	141
10	86
11	92
12	73
13	98
14	43
15	73
16	46
17	97
18	84
19	66
20	48
21	68
22	46
23	114
24	24
25	58
26	68
27	13
28	158
29	79
30	178
31	13
32	57
33	105
34	76

In order to further test this relationship, a linear regression was carried out between the improvement in test score and the number of help requests. The results of this analysis indicate an R value of 0.364 with a p-value of 0.034. Since the p-value is below 0.05, the number of help requests is significant with regards to the improvement in test score although the regression coefficient is not that large. The normal probability plot (*Figure 9.2*) resulting from this analysis is close to a straight

line, indicating that the error terms are normally distributed. This validates the fundamental assumption in linear regression that the errors are normally distributed.

Table 9.6

Correlations Results for Improvement in Test Score and Number of Help Requests

		Test Improvement	Number of Help Requests
Test Improvement	Pearson Correlation	1	.364*
	Sig. (2-tailed)		.034
	N	34	34
Number of Help Requests	Pearson Correlation	.364*	1
	Sig. (2-tailed)	.034	
	N	34	34

* Correlation is significant at the 0.05 level (2-tailed)

Another correlation was calculated to test whether the duration of the usage of the system had a significant effect on the improvement in test scores. The null hypothesis of the test was as below.

There is no significant difference in the improvement of test results with the duration of usage of the system.

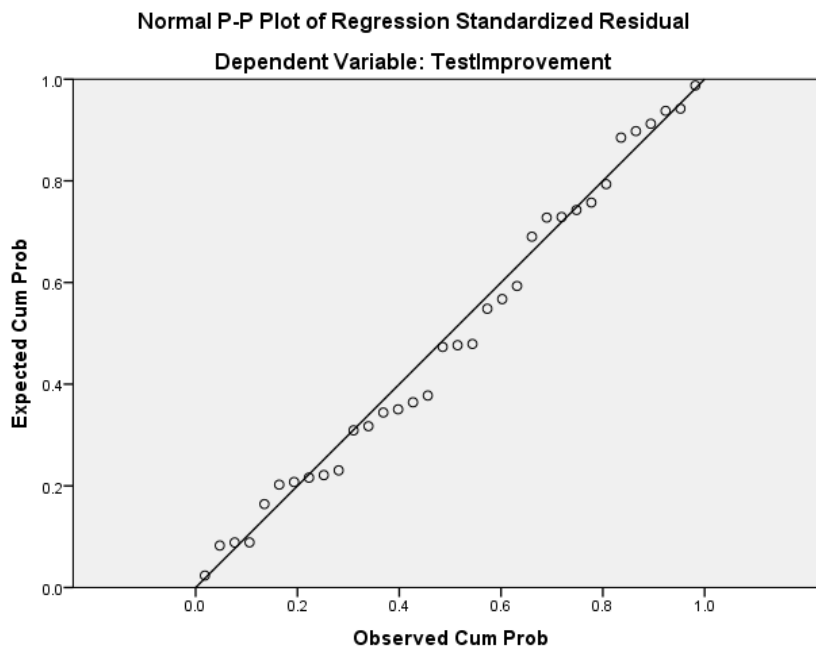


Figure 9.2. Normal probability plot for regression analysis.

Several calculations were carried out in order to obtain the relevant data. From the system usage information recorded in the system, the total duration of use of the system by each student was calculated (Table 9.7). This was done by finding the difference between each login and the subsequent logout¹. The improvement in test score was again calculated as described above. A correlation was then calculated between the duration of usage and the improvement in test score. The results of this test are shown in Table 9.8. It can be seen that this correlation was not significant indicating that there was no reason to reject the null hypothesis. In other words, it was not possible to say that there was a significant difference in the improvement of test results based on the duration of usage of the system. A possible explanation for this is that the students did not utilise the features of the system, the entire time they were logged on. They may have spent some of this time learning the subject matter using web resources, textbooks and other study aides. Therefore, the duration when the students were logged on may not have been an accurate reflection of the time they actually used the system.

A correlation test was also carried out to see whether the number of problems attempted and the number of problems correctly completed had any significant effect on the improvement in test score. Table 9.9 shows the number of problems attempted and the number of problems correct for each student. These figures were used to correlate against the improvement in test scores. The three null hypotheses are as below.

There is no significant difference in number of problems attempted with the number of problems correct.

There is no significant difference in the improvement of test results with the number of problems attempted.

¹Due to a development bug, some students managed to close the system without cleanly logging out, especially during the initial usage of the system. In such cases a forced logout was carried out either by the administrator, or later by the students themselves. This happened after a significant time delay, thereby making the duration of usage unrealistic. In order to account for this problem, any time durations of greater than 10 hours were ignored during the analysis. It should be noted that only a very few data items were ignored in this manner and it is therefore felt that this is a reasonable estimation based on the available data.

There is no significant difference in the improvement of test results with the number of problems correct.

Table 9.7

Total Duration of System Use for Each Student

<i>Student</i>	<i>Total Duration Used</i>
1	5:23:06
2	5:01:16
3	9:07:45
4	14:54:34
5	8:17:26
6	12:57:18
7	9:49:45
8	4:49:21
9	11:09:57
10	3:35:59
11	0:42:51
12	19:33:50
13	10:46:57
14	0:50:34
15	11:57:09
16	21:38:27
17	5:09:30
18	20:26:04
19	8:55:17
20	6:10:22
21	2:48:33
22	10:12:46
23	9:04:08
24	3:09:51
25	6:51:45
26	3:03:00
27	3:15:20
28	10:36:17
29	5:05:55
30	12:05:12
31	7:46:32
32	10:22:58
33	9:38:07
34	14:15:20

Table 9.8.

Correlation Results for Minutes Used and Improvement in Test Score

		<i>Minutes Used</i>	<i>Test Improvement</i>
<i>Minutes Used</i>	Pearson Correlation	1	.019
	Sig. (2-tailed)		.914
	N	34	34
<i>Test Improvement</i>	Pearson Correlation	.019	1
	Sig. (2-tailed)	.914	
	N	34	34

The results of this correlation test are shown in Table 9.10. It can be seen that the only result that is significant here is the correlation between the number of problems attempted and the number of problems correct. This result is extremely significant, allowing the first null hypothesis above to be rejected, meaning that there is a significant difference between the number of problems attempted with the number of problems correctly completed. This can easily be explained since it is quite reasonable that the number of problems correct is related to the number of problems attempted. However, all the other correlations are not significant, indicating that the two other null hypotheses cannot be rejected. In other words, it cannot be shown that there is a significant difference in the improvement in test scores with either the number of problems attempted or the number of problems correct.

The results of the above tests as a whole indicate that although the improvement in test score does not seem to be affected by the duration of usage of the system, the number of problems attempted, or the number of problems correct, it is significantly affected by the number of help requests that the student issues.

Table 9.9.

The Number of Problems Attempted and the Number of Problems Correct for Each Student

<i>Student</i>	<i>No. of Problems Attempted</i>	<i>No. of Problems Correct</i>
1	28	20
2	29	23
3	33	25
4	26	11
5	32	29
6	26	20
7	32	21
8	19	8
9	23	8
10	31	18
11	31	22
12	27	18
13	32	25
14	32	29
15	30	20
16	26	23
17	29	21
18	29	23
19	32	23
20	31	30
21	28	23
22	31	30
23	29	26
24	14	4
25	31	30
26	28	22
27	7	2
28	31	30
29	31	30
30	31	30
31	15	1
32	31	0
33	31	6
34	30	27

Table 9.10.

Correlation Results for Number of Problems Attempted, Number of Problems Correct and Improvement in Test Score

		No. Attempted	No. Correct	Test Improvem ent
<i>No. Attempted</i>	Pearson Correlation	1	.705**	.202
	Sig. (2-tailed)		.000	.252
	N	34	34	34
<i>No. Correct</i>	Pearson Correlation	.705**	1	.131
	Sig. (2-tailed)	.000		.459
	N	34	34	34
<i>Test Improvement</i>	Pearson Correlation	.202	.131	1
	Sig. (2-tailed)	.252	.459	
	N	34	34	34

** Correlation is significant at the 0.01 level (2-tailed)

9.3.2 Validity of the Student Model

As explained in Section 8.2, the student model consisted of a set of probabilities of each student knowing a topic. Each question in the post-test was linked to one or more of these topics. A prediction was made as to whether a student would or would not get the post-test questions correct, based on their knowledge of the relevant topics before the post-test. In order to make this prediction, it was necessary to determine a threshold value to decide that a student was indeed knowledgeable in a topic. The threshold value used here was 0.85, the same value used in the PHP ITS. When more than a question tested more than one topic, the average probability across those topics was considered. If the final average probability that a particular student knew the topics covered by a post-test question was above the threshold value, it was predicted that the student would get the answer correct.

This prediction was used to calculate a predicated post-test score for each student. The predictions obtained in this manner were correlated using Pearson's correlation against the actual post-test scores for the students. The predicted and actual post-test scores for version 1 are shown in Table 9.11 while those for version 2 are shown in

Table 9.12. A two-tailed test was considered in this case as the direction of any change could not be estimated. Table 9.13 shows the results of performing this analysis using SPSS for version 1 and

Table 9.14 shows the corresponding results for version 2. It can be seen that there is no positive correlation between the post-test score and the predicted post-test score in the case of version 1. However, the situation is different in the case of version 2. Here, a strong positive correlation of 0.660 exists with a significance level of 0.007 so the correlation is significant at the 0.01 level.

Table 9.11

Predicted and Actual Post Test Scores for Each Student for Version 1

<i>Student</i>	<i>Predicted Post Test Score</i>	<i>Actual Post Test Score</i>
1	10	13
2	15	17
3	12	13
4	15	11
5	9	12
6	11	17
7	8	10
8	13	15
9	14	12
10	9	15
11	10	14
12	12	17
13	10	14
14	9	10
15	15	17
16	0	13
17	10	12
18	14	15
19	12	11

Therefore, it is possible to conclude that although the probabilities in the student model in version 1 do not accurately reflect the students' knowledge, those in version 2 provides a better estimate of the students' knowledge.

Table 9.12

Predicted and Actual Post Test Scores for Each Student for Version 2

<i>Student</i>	<i>Predicted Post Test Score</i>	<i>Actual Post Test Score</i>
1	12	14
2	11	10
3	16	19
4	10	14
5	10	16
6	10	17
7	10	13
8	0	8
9	10	9
10	9	14
11	11	15
12	0	4
13	2	15
14	9	16
15	10	15

Table 9.13

Correlation results of Post-test Score and Predicted Post-test Score for Version 1

		<i>Post-test score</i>	<i>Predicted post-test score</i>
<i>Post-test score</i>	Pearson Correlation	1	.307
	Sig. (2-tailed)		.201
	N	19	19
<i>Predicted post-test score</i>	Pearson Correlation	.307	1
	Sig. (2-tailed)	.201	
	N	19	19

9.3.3 System Usage

The PHP ITS provides multiple forms of support for students as described in Section 8.3.1. An analysis of the logged usage data was carried out to see which help features provided by the students were most used by students. A summary of the results showing the percentages of the number of requests for each type of help are shown in *Figure 9.3*.

Table 9.14

Correlation results of Post-test Score and Predicted Post-test Score for Version 2

		Post-test score	Predicted post-test score
Post-test score	Pearson Correlation	1	.660**
	Sig. (2-tailed)		.007
	N	15	15
Predicted post-test score	Pearson Correlation	.660**	1
	Sig. (2-tailed)	.007	
	N	15	15

** Correlation is significant at the 0.01 level (2-tailed)

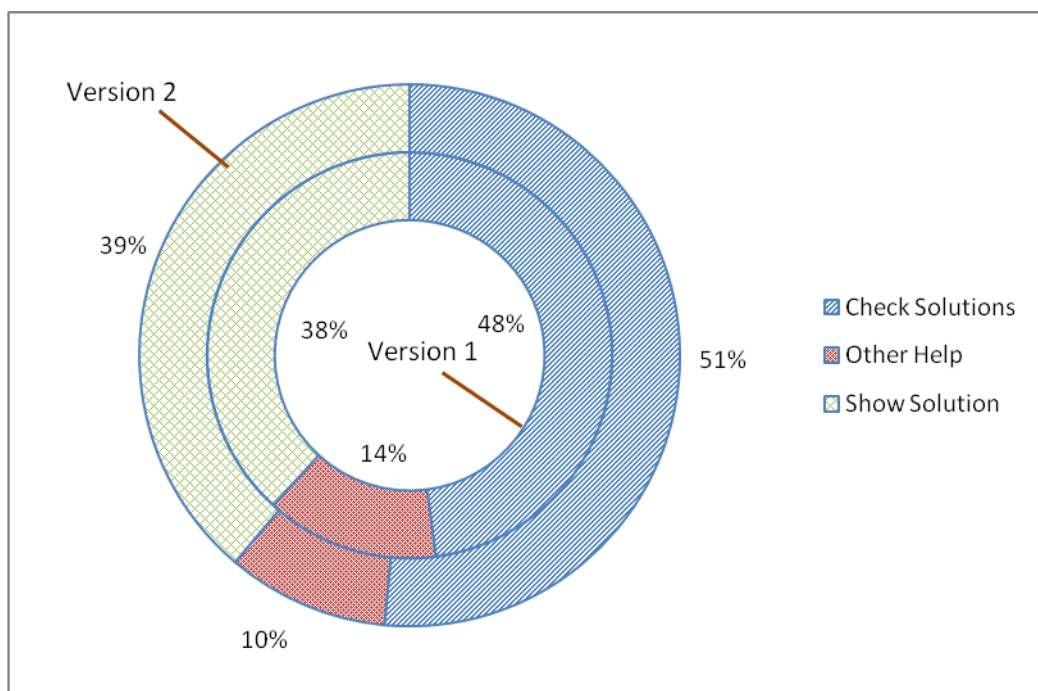


Figure 9.3. Types of help used by students.

It can be seen that there is very little difference between the percentage use of different help features between the two versions. Of the different types of help provided, close to 50% of the interactions were for checking the solution of their code. However, few students seem to have requested further help as indicated by the low percentage using Other Help. Here, Other Help refers to the total of displaying

relevant topics and asking for further help on errors using either the ‘What is Wrong’ or ‘How do I Solve It’ buttons as described in Section 8.3.1. However, note that of these three types of Other Help, only displaying relevant topics was available in version 1 (Section 9.2). The slight increase in use of Other Help in version 2 could be due to the introduction of the new features to request for further help. It can be seen that many students also chose to display the entire solution. This indicates that students seem to find it highly useful to see what the system thinks is a correct solution. A possible reason for this is that they want to learn by comparing their solution with the solution provided by the system.

As described in Section 8.3.2, the PHP ITS provides two modes of selecting the next exercise: the student can either select the next exercise based on specific search criteria, or allow the system to suggest the next exercise. This selection was only available in version 2 of the system so only the data from this version are analysed here. *Figure 9.4* shows a stacked bar chart of the number of times each student selected each mode of exercise selection. It can be seen that few students were happy to allow the system to suggest their next exercise without ever choosing to search for an exercise. A majority of the others allowed the system to suggest the exercise more than 50% of the time. A few preferred mostly to search for exercises on their own.

An additional feature added to the second version of the system was the Skillometer. In order to see whether this feature was utilised by students, a histogram showing the number of students that used the Skillometer a given number of times was prepared. The resultant chart is shown in *Figure 9.5*. This shows that many students did not use the Skillometer at all and very few used it more than twice during the entire unit. In the PHP ITS, the Skillometer was only available by clicking on a link at the top of the screen and its use was not explicitly pointed out to the students except for a mention in the help files. It seems that the Skillometer needs to be made more visible for students to gain maximum benefit.

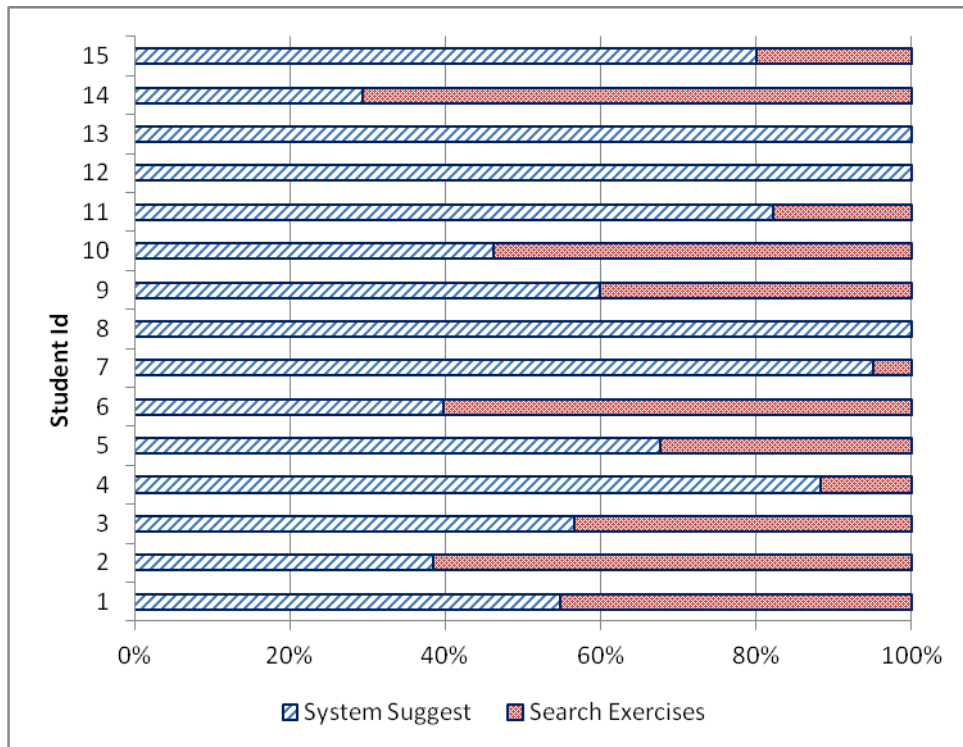


Figure 9.4. Exercise selection mode used by each student.

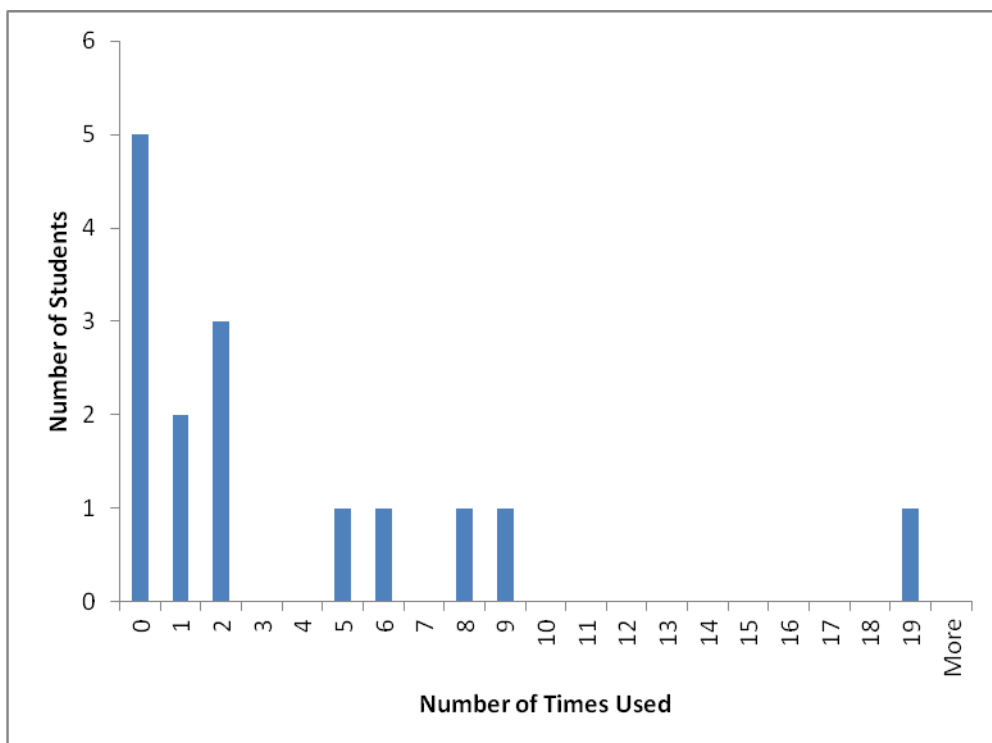


Figure 9.5. Frequency of Skillometer usage by students.

9.3.4 Satisfaction

In addition to the effectiveness of the system, another major component when deciding the usability of a system for practical use is user satisfaction. The user satisfaction of the PHP ITS was measured using both the feedback questionnaire and the focus group as described in Section 9.1.2. The responses to the Likert questions in the questionnaire (Appendix J) were used to create charts to gauge how the students rated the various aspects of the system. Since it was not compulsory for the students to provide feedback in this manner, only 13 out of the 19 students in version 1 and 6 out of the 15 students in version 2 provided feedback. The response to version 2 proved disappointing, providing insufficient evidence to compare the two versions of the system. Therefore, most of the analysis in this section was carried out using the combined data from both versions. Another possible weakness of this analysis was the fact that the students were aware that the system was built as a result of this research, and may therefore have not wished to offend the researchers with their answers.

Figure 9.6 shows how the students rated the system overall. It can be seen that although only 5% rated the system as excellent, more than half the students had an overall impression that the system was good. No students felt it was very poor. Therefore, it can be seen that the majority were satisfied with the overall impression of the system.

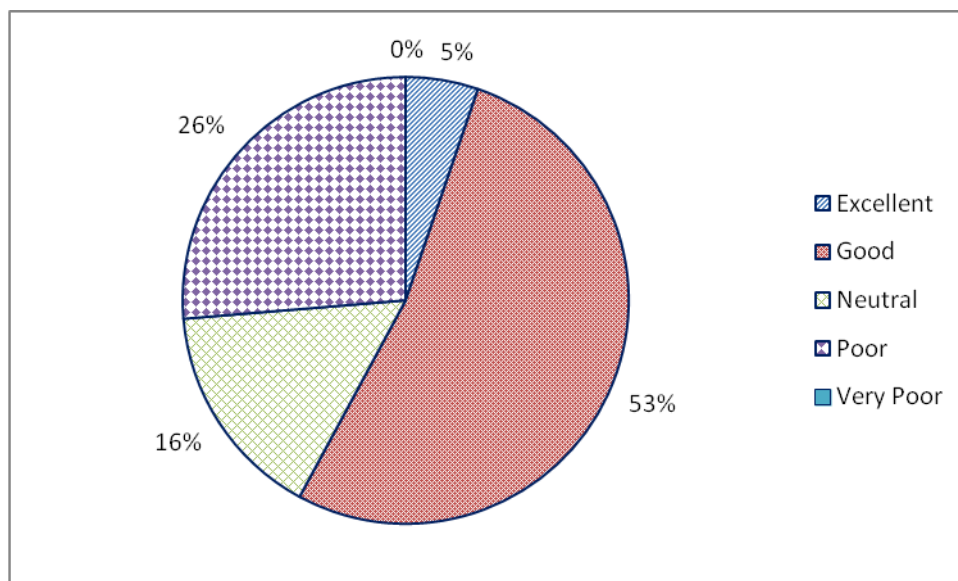


Figure 9.6. Overall impression of the system.

The ease of use of the system was another aspect that was rated by students. *Figure 9.7* shows the ratings chosen by the students for this parameter. In this case, it can be seen that 10% of the students thought the system was very easy to use while a further 37% felt that it was relatively easy to use. Again, no students rated this as very poor.

The students also rated how they felt about the quality of the programming exercises provided by the system. The overall distribution in this case is shown in *Figure 9.8*. Again, more than half the students rated the exercises as either excellent or good and none rated them very poor.

It is extremely unlikely that students would use a computerised system that would take unacceptable time periods to respond. Therefore, the speed of response of the system is an important indicator as to its usability. The ratings provided by the students for this parameter (*Figure 9.9*) were very similar to those for the programming exercises.

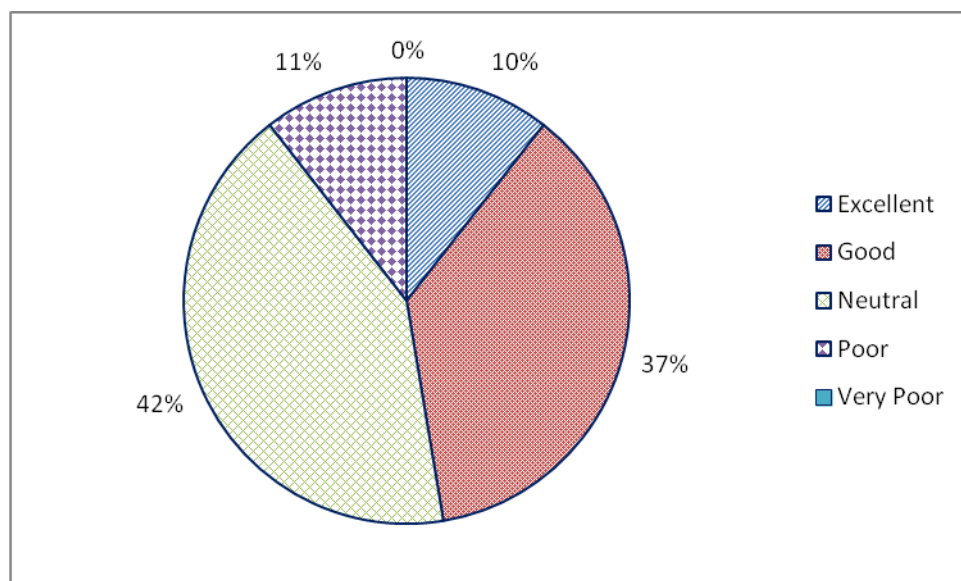


Figure 9.7. Ease of use of the system.

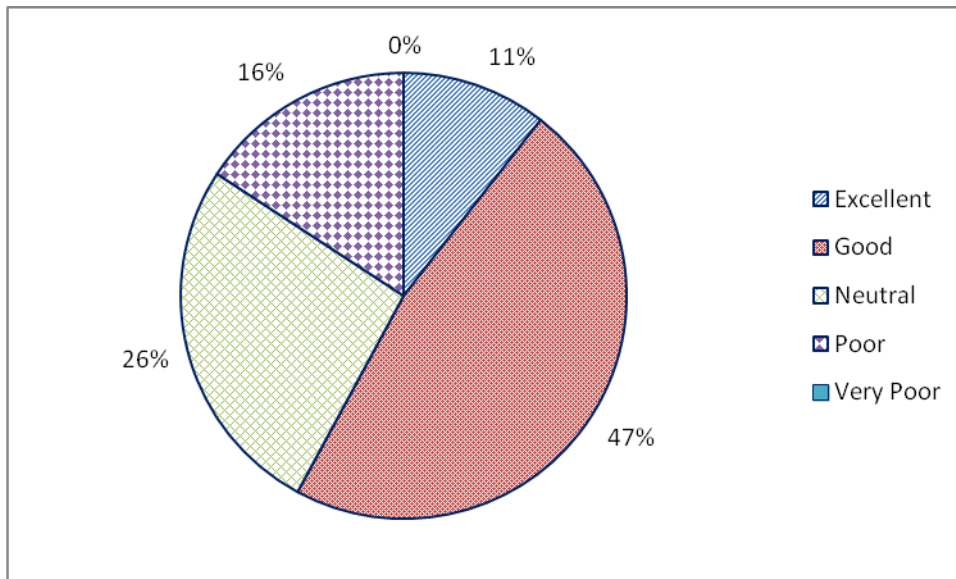


Figure 9.8. Programming exercises.

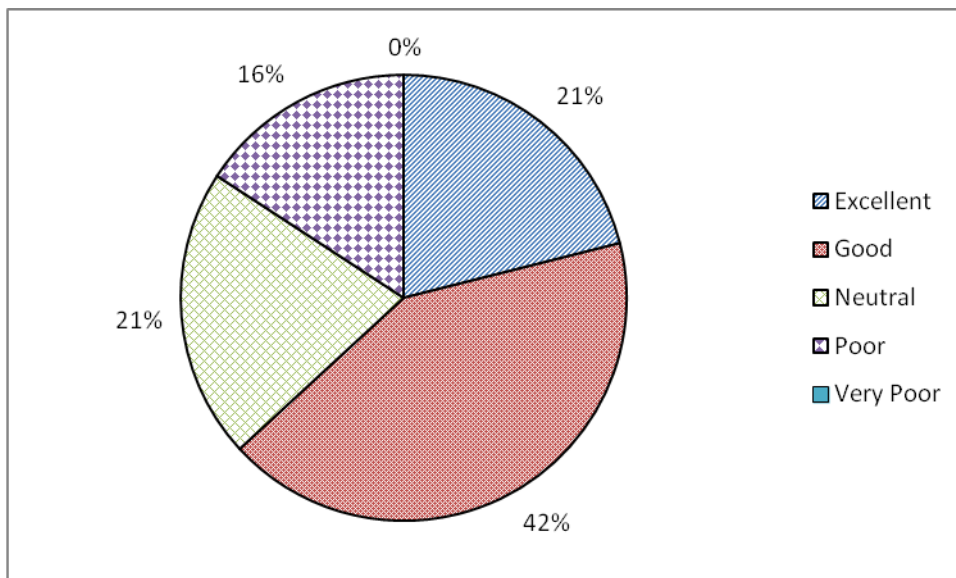


Figure 9.9. Speed of response of the system.

All the above ratings were summaries of those provided for both versions of the system. However, as described in Section 9.2, some aspects of the second version varied considerably from those of the first version. Therefore, it was more appropriate to compare the ratings for the two versions for some of the questions.

A major change between the versions was how feedback was provided. The first version immediately displayed error messages based on any identified errors while the second version indicated that there was an error but waited for the students to ask for further information. The ratings of the students for feedback messages across the two versions are shown in *Figure 9.10*. It can be seen that about the same percentage of students in each version felt that the feedback messages were excellent.

However, more students felt that the feedback messages were good in version 2 than did those in version 1. 15% of students using version 1 actually felt that the feedback messages were very poor while none using version 2 felt that the messages were very poor. Even though some improvement between the two versions is apparent, this seems to be an area where further improvements should ideally be made. Given that the students learned PHP and the overall impression of the system was good, it is a possibility that students felt there wasn't sufficient information as to how they should fix their problems. This is an area that needs to be looked into in future developments of the system.

The next analysis was performed to gauge whether students felt the different versions of the PHP ITS contributed to their success in gaining knowledge and understanding. A donut chart of the ratings given by the student is shown in *Figure 9.11*. In this figure, it can be seen that around 50% of users in both versions felt that their success in gaining knowledge and understanding was either excellent or good. More students using version 1 rated this as poor than did the percentage of students using version 2. No student using either version rated their success in gaining knowledge and understanding as very poor.

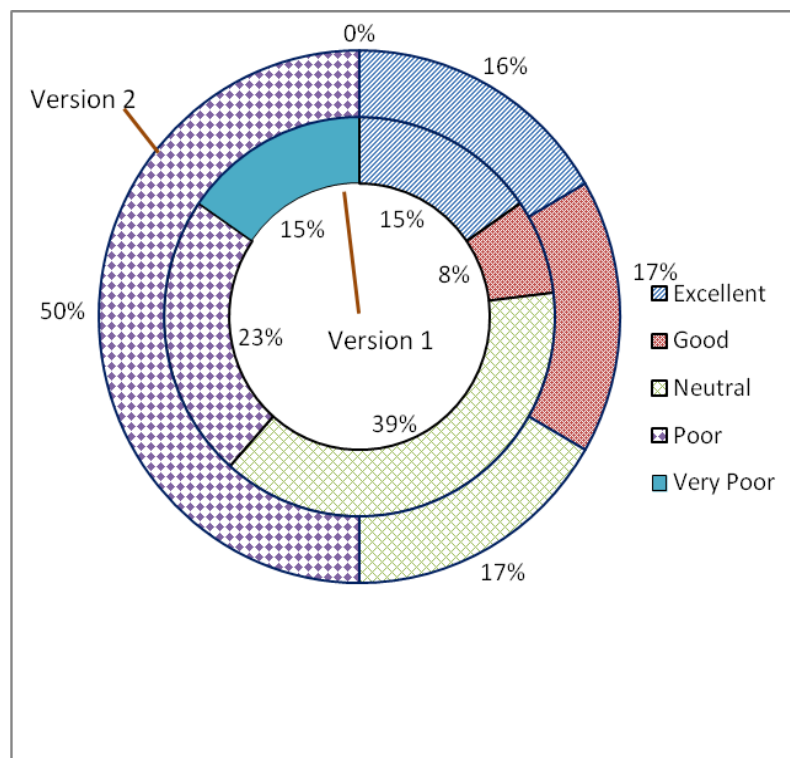


Figure 9.10. Feedback messages.

As mentioned in Section 9.2, the look and feel of the system was changed considerably from the first version to the second. *Figure 9.12* shows a donut graph of the students rating for the look and feel across the two versions. It can be seen that very few students rated the look and feel as excellent or good for either version although a considerable number rated it as neutral. It is of some concern that many students have rated it as very poor for both versions, indicating that more work needs to be done in this area.

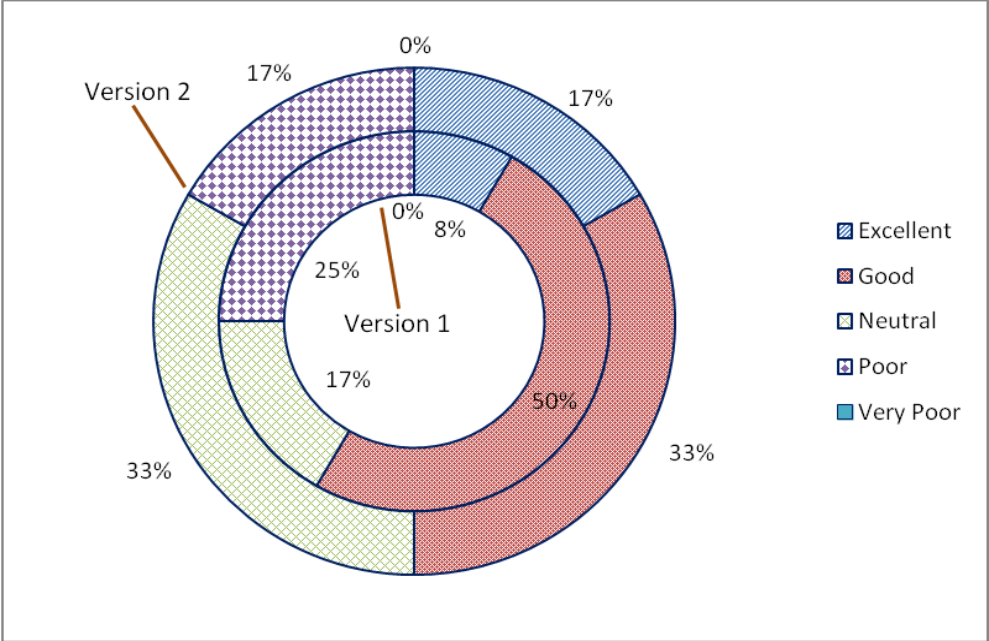


Figure 9.11. Success in gaining student knowledge and understanding.

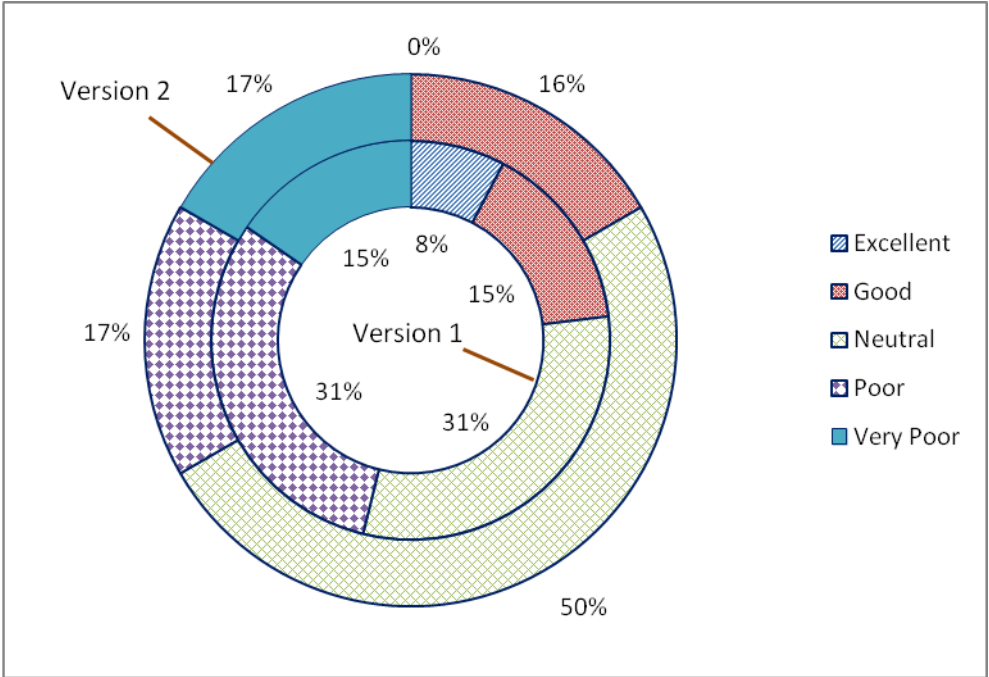


Figure 9.12. Look and feel.

The responses to the free-answer questions in the questionnaire (Appendix J) and also the students' responses during the focus group discussion provided some insight into what other improvements the students would like to see. Many students responded that they would like to use similar systems in other domains in the future, as well as recommending the PHP ITS to others. Some were of the opinion that the system should provide theoretical instruction within it, without directing the user to relevant external websites. Several suggested that the system be made accessible through mobile devices. Another suggestion was that the interface be improved to include syntax highlighting and auto-completion of keywords like in many traditional IDEs.

The following were some additional comments that were given by students that used the PHP ITS.

- I enjoy (sic) the self paced learning and the availability of the system. Essentially it taught me basic PHP, rather than just reading about it.
- It is good because it provides the development environment, links to relevant info and progressively more difficult exercises that use repetition with small variation to increase retention.
- I enjoy (sic) the ITS, with a few improvements it will just keep getting stronger.

These comments show that overall, the system has many positive attributes which students feel are useful in learning introductory programming using PHP.

9.4 CHAPTER SUMMARY

This chapter described the process used to evaluate the PHP Intelligent Tutoring System under practical use. It described the empirical evaluation and the results of the evaluation. The next chapter analyses these results and concludes how the PHP ITS has answered the problem of teaching introductory web development using PHP to novices.

Chapter 10: Conclusions

Teaching introductory programming is a major challenge to educators for many reasons. Although many methods have been suggested to overcome this challenge, it continues to be a major problem. In particular, little research has been carried out on methods of teaching web programming to beginners.

It is an accepted fact that students taught on a one-to-one basis learn any subject much better than those taught using traditional classroom situations. However, using human tutoring to do this is not very suitable to teach web programming to beginners since it requires an enormous amount of resources. The solution to this problem, as suggested by this research, is to use Intelligent Tutoring Systems for this purpose. The PHP ITS is such a system that focuses on teaching the basics of PHP programming to beginners in web development. It provides exercises to students based on their specific requirements in order to maximise their learning. The student's solutions are then analysed and appropriate feedback is given. The feedback relates to the specific error made by the student. Four levels of feedback are available at the student's request. This ensures that each student can obtain feedback at his or her own level, thereby maximising their learning.

The major achievement in the PHP ITS is its capability of identifying a large number of alternative solutions to a single programming exercise. It can recognise programs written using many combinations of conditional statements as semantically equivalent. It is capable of accepting many types or combinations of expressions for the right hand side of assignment statements. It can handle several types of loops that function in the same manner. It permits the use of PHP statements embedded within HTML and vice versa.

The PHP ITS analyses not only basic procedural programming concepts but also concepts related to creating, submitting and processing data using web forms. This functionality is peculiar to web development and no ITS in the available literature is capable of doing this.

10.1 RESEARCH CONTRIBUTIONS

This section discusses more specific details of the contributions of this study to the research community in light of the research aims and objectives described in Section 1.3. The research problem addressed by this study is reiterated as below.

Is it possible to create an Intelligent Tutoring System to effectively teach web development using PHP?

During the course of this study, a working Intelligent Tutoring System to teach introductory web development using PHP was developed and successfully implemented in a PHP unit at the Queensland University of Technology. An evaluation was conducted to test the effectiveness of the system. The results of the evaluation showed that the test scores of the students significantly improved after using the system.

In addition to the increase in test results, the students showed a positive attitude towards many features of the PHP ITS such as the ease of use of the system and the response time of the system. The fact that it is web enabled ensured that the students were free to use it during their own time.

These overall results indicate that the system was successful in teaching the subject matter effectively to the students, thereby achieving the primary goal of the study.

In order to address this research problem, three main research questions as described in Section 1.3 and repeated below were addressed.

1. *What is the best method of knowledge representation that can be used to model the subject matter necessary to effectively teach basic PHP programming while achieving the following?*
 - a. *Analysing alternative solutions to a given programming problem, both correct and incorrect*
 - b. *Providing feedback based on the specific errors made by the student*
1. *What is a suitable student model for the above system?*
2. *What methods of feedback and individualised interactions are useful to teach the above subject matter effectively through an ITS?*

The rest of this section investigates how the PHP ITS addresses each of these three research questions.

10.1.1 Knowledge Representation

Programming is a practical subject and therefore, any course designed to teach programming must include practical exercises. An ITS to teach programming should be capable of analysing example solutions to such exercises. A major challenge encountered here is that a single programming exercise can have many correct solutions. The PHP ITS concentrates on analysing PHP programming solutions to exercises that are suitable for a beginner in PHP web page development and handling such alternate solutions. It uses theories of Artificial Intelligence to model computer programs written using combinations of PHP and HTML, and analyses them for correctness. It covers display and assignment statements, selection structures, arrays, HTML forms, PHP functions and some looping constructs that are considered to be the most common constructs used by beginners.

The methods used by the PHP ITS to analyse such programs are explained in detail in Chapter 4, Chapter 5, Chapter 6 and Chapter 7. The examples provided in these chapters describe how the PHP ITS is capable of handling alternative solutions to a given exercise using many commonly used PHP constructs. Section 8.3 explains how the results of the analysis are used to provide feedback messages that are specific to an error made by the student.

Therefore, it can be seen that this research has established a theoretical framework for analysing basic computer programs written in PHP and HTML and identifying alternative solutions and errors.

Therefore, the first research question above, regarding the method of knowledge representation to handle alternative solutions and provide appropriate feedback is answered in this research project.

10.1.2 Student Model

Typical students in a beginning web programming course vary widely in their prior knowledge of relevant subject matter. In order to maximise the learning, it is important to support each student within their own Zone of Proximal Development (see Section 2.4.1). In the PHP ITS, this support is given by showing the student the next best exercise for their current level of knowledge. In order to do this, it is

necessary for the system to maintain a model of the current level of knowledge of each student. This is achieved by dividing the subject matter into topics and maintaining a probabilistic estimate as to the current level of knowledge for each student for each topic. A more detailed description of how the student model is designed and updated is given in Section 8.2.

In order for the selected exercises to be appropriate to the current student, the student model needs to be accurate. The results of the evaluation of the PHP ITS showed that the model used in the improved version of the system estimated the knowledge level of each student quite accurately. Therefore, the second research question regarding an appropriate student model is answered in this thesis.

However, it should be noted that, it is not claimed that the student model is always highly accurate due to several reasons. Sometimes, students deliberately make mistakes in their code in order to either test out theories or even to test out the system. In doing so, they indicate to the system that they do not have knowledge about certain topics, even if they actually do. No student modelling system is capable of identifying such intentional errors. The student model can only be as accurate as the evidence provided by the students.

10.1.3 Feedback and Individualised Instruction

Analysing a solution for correctness is insufficient for students to learn the subject matter effectively. Appropriate forms of support such as feedback on errors in the program and methods of accessing relevant factual data should be provided for this purpose. The PHP ITS gives students the option of viewing feedback messages regarding their errors. This feedback is provided at four levels, allowing the students to select the level that is most suitable for them. The ITS also displays links to web pages that contain material relevant to the current error or for solving the current exercise. A detailed description of the support provided by the system to solve exercises is given in Section 8.3.

The evaluation process suggested some shortcomings in the feedback provided by the PHP ITS. Overall, the students did not seem to be satisfied by the feedback messages. Although additional functionality for obtaining more detailed feedback was incorporated into the second version of the system (Section 9.2) the students did not seem to use these functions to a great degree.

The individualised instruction in the PHP ITS was provided by the system suggesting exercises for the students (as mentioned above). The results of the evaluation showed that many students were happy to use this feature, indicating that they found the suggestions by the system useful to enhance their knowledge.

Therefore, it seems that although the methods of individualised instruction provided by the PHP ITS were useful to teach the subject matter effectively, the feedback was not of sufficient use. Therefore, more work needs to be carried out in order to answer the third research question more thoroughly. However, the fact that the overall ITS was effective in teaching the subject matter effectively suggests that the feedback also proved useful at least to a certain degree.

10.1.4 Publications and Talks

The following publications and talks are a direct result of this project.

10.1.4.1 Peer-reviewed Conferences

Weragama D., & Reye, J. (2013). *The PHP Intelligent Tutoring System*, 16th International Conference on Artificial Intelligence in Education, Memphis, USA

Weragama, D., & Reye, J. (2012). Designing the Knowledge Base for a PHP Tutor. In S. Cerri, W. Clancey, G. Papadourakis & K. Panourgia (Eds.), 11th International Conference on Intelligent Tutoring Systems (Vol. 7315, pp. 628-629). Chania, Greece: Springer Berlin Heidelberg.

Weragama, D., & Reye, J. (2012). Design of a Knowledge Base to Teach Programming. In S. Cerri, W. Clancey, G. Papadourakis & K. Panourgia (Eds.), 11th International Conference on Intelligent Tutoring Systems (Vol. 7315, pp. 600-602). Chania, Greece: Springer Berlin Heidelberg.

10.1.4.2 Other Talks

Weragama D. (2012), *Developing Intelligent Tutoring Systems to Assist Students Learning Programming*, Talk presented at the Queensland Computing Education Conventicle 2012, Brisbane, Australia

Weragama D. (2010), *Intelligent Tutoring System for Dynamic Web Development using PHP and MySQL*, Talk presented at the Higher Degree Research Student Consortium of the Computer Science Discipline 2010, Brisbane, Australia

Weragama D. (2012), *Intelligent Tutoring System to Teach Programming*, Talk presented at the Three Minute Thesis Competition of the Queensland University of Technology 2012, Brisbane, Australia

10.2 LESSONS LEARNED

This section looks back critically at the lessons learned during the design, development and implementation of the PHP ITS. Section 10.2.1 highlights the pros and cons of the system design while Section 10.2.2 looks at issues related to the evaluation of the system.

10.2.1 System Design

As described in Section 2.3.2 many representations have been used by previous researchers to represent the knowledge base in Intelligent Tutoring Systems designed to teach programming. Each of these representations has many advantages but also certain shortcomings. Therefore, an entirely new approach was utilised during the design of the PHP ITS. The main requirement of the representation was that it be capable of supporting logical reasoning about the structure of programs written by students and providing appropriate feedback based on the specific errors. Artificial Intelligence techniques seemed like a very reasonable means of achieving this objective. Of the many formalisms available in AI, FOPL is a fairly simple yet powerful representation. Therefore, it seemed like a good candidate to use for this purpose, although no previous work seems to have looked at this possibility.

PHP is a language that is used in conjunction with HTML. Therefore, both these languages needed to be considered when designing a system to analyse PHP programs. The possible constructs in both these languages are numerous and it was practically impossible to handle all of them during the time limitations of a PhD. Therefore, only a subset of both these languages, that were deemed suitable for a beginning programmer were considered. This meant that more advanced PHP topics such as Object Oriented Programming and recursion were ignored. The subset of PHP that is covered by the representation here is highlighted in Appendix B.

The main advantage of the formal representation of PHP programs used in this thesis is that it is capable of identifying alternative solutions to a single programming exercise. Since the modelling proposed here looks at the various possible programming constructs and not at a particular set of exercises, this gives it the flexibility to handle many more exercises than those actually implemented. The fact that the overall goal can be broken down into a set of sub-goals allows the system to identify the exact sub-goals that are not satisfied by a program. This gives the

possibility to provide feedback based on the specific errors made by the student. It also allows more accurate updating of the student model since the specific sub-goals can be linked to specific topics as described in Section 8.2.3. Another advantage is that web pages that are directly related to the specific error made can be suggested based on these sub-goals (Section 8.3.1.1).

The formalism used here to represent PHP programs does have certain disadvantages. Its main weakness lies in the fact that it is incapable of handling all types of loops, as described in Section 7.1. It can handle collection independent definite loops and collection based loops that perform some action against every item in a collection independently. This is a fair percentage of loops that are encountered in practical situations. Where definite loops are concerned, the system can analyse situations where a certain part of the loop has been unrolled. The analysis process has some limitations even for the types of loops that can be analysed in this manner. For example, special rules are needed in cases where some form of summarising is done by the loop. This makes it necessary to write new rules for each such exercise, thereby reducing the flexibility of the system (Section 7.3.2). When considering array access, the PHP ITS can handle both direct and indirect access of array elements. However, the searching capabilities are limited to finding a maximum or minimum array element.

In addition to the limitations in processing loops, there are several other situations encountered in basic PHP programs that prove difficult for the given formalism to handle. One such issue arises when ‘&&’ and ‘||’ operators are used in conditional expressions. These expressions consist of two expressions on either side which must have a Boolean value. Although the PHP ITS can handle situations where ‘&&’ expressions are true and ‘||’ expressions are false as explained in Section 5.4.2, other such expressions present a problem. Another issue encountered is when students use arbitrary functions. Although the system has been designed to handle user-defined functions, the analysis process involves the validation of sub-plans that correspond to these functions (Section 6.2). Since such sub-plans are only included when functions are required by the specification, the analysis process fails when unexpected functions are encountered.

When considering the PHP ITS itself, one main limitation is that it does not tailor its feedback to the needs of the individual student. Although this is a desirable

aspect for an ITS, it was beyond the scope of this thesis due to time constraints. Additionally, better feedback and user interface design would be advantageous improvements.

10.2.2 Evaluation

The evaluation of the PHP Intelligent Tutoring System proved very challenging since it was conducted within a postgraduate unit which counted towards the students' GPA. This meant that it was impossible to obtain ethical clearance for a study of sufficient duration to contain a control group. Although the lack of a control group was a major impediment to the evaluation process, the results still showed that the ITS contributed to increasing the students' knowledge of the subject matter. Another issue encountered was that the number of students was limited due to the nature of the unit. A larger group of students would have provided a more accurate measure as to the usefulness and usability of the system.

The number of exercises included in the system was also not that large. A larger set of exercises would give a more accurate result, especially of whether the system is good at selecting exercises which are appropriate for the student.

When considering the instruments used, it would have been useful to include more data in the questionnaire. Although details about the students' programming background were included, the questionnaire could not be linked to the students and therefore, these details could not provide any meaningful gauge about the usefulness of the system to students of different knowledge levels.

Although these limitations existed in the evaluation of the PHP ITS, the results obtained are still useful for showing that it is a useful tool for students learning beginning PHP.

10.3 FUTURE DIRECTIONS

Section 10.1 discusses how the results of the evaluation show that the PHP ITS answers the research questions and the research problem to a great degree. An important outcome of the evaluation process is that it identified some areas in which future developments to the PHP ITS could be beneficial. The following are the areas that have been identified for future improvement in this manner based on both my own thoughts and student feedback.

1. *Include more PHP domain knowledge in the knowledge base.*

The knowledge base of the PHP ITS in its current form handles only the PHP topics that are considered suitable for a beginning web programmer. Future versions of the system could be developed to handle the other forms of loops (Section 7.1), more HTML elements such as hidden inputs and more advanced PHP concepts such as accessing MySQL databases.

2. *Include prioritising of sub-goals for feedback*

The current program analysis method compares the sub-goals in the overall goal against the final state in the order specified during the exercise specification. Once a single sub-goal is identified as not matched, the analysis process is terminated and feedback is provided for that sub-goal. It may be better to continue the analysis process until all sub-goals are checked and then prioritise the order of mismatched sub-goals for which feedback should be provided, based on criteria such as the student's current knowledge on the topics covered by each mismatched sub-goal.

3. *Include pre-requisite relationships for topics in the student module.*

The current student module assumes that each topic can be studied independently of the others. In practice, certain topics are pre-requisites for studying other topics. Such pre-requisite relationships could be included in the Bayesian Networks that models the student knowledge in order to obtain a more accurate student model.

4. *Investigate the students' actions after viewing the skillometer.*

The results show that the students did not use the skillometer of the PHP ITS as much as expected. In order to understand the reason for this, it is possible to investigate the students' actions after viewing the skillometer. This would enable us to understand their reasons for viewing the skillometer and to find methods for improving its use.

5. *Include theories of pedagogy and education in the teaching module.*

The current teaching module does not use the information from the student model to customise feedback messages based on the abilities of the student. Future versions of the PHP ITS could be developed to utilise such knowledge and also include more theories from education and pedagogy in order to maximise the students' learning. For example, the Zone of Proximal Development (Vygotsky, 1978) can be coupled with the current level of student knowledge to automatically customise the level of feedback provided to the student.

6. *Include more theories of UI design in the PHP ITS.*

The current user interface of the system could be improved, utilising theories of UI design.

7. *Compare the PHP ITS against standard non-adaptive tutorials.*

The current study only compares the learning gains of the PHP ITS. It does not investigate whether it is better than a standard non-adaptive tutorial in terms of either learning gains or learning time. A study comparing the PHP ITS against several freely available non-adaptive tutorials would be a valuable addition to validate its capability and utility.

8. *Extend the concepts of the domain module to handle other programming languages.*

One of the main outcomes of this research project is a theoretical framework to analyse semantically equivalent programs written in PHP. A future area of research could be to see if these concepts could be extended to handle the analysis of programs written in other programming languages.

Although these enhancements would make the PHP ITS a stronger system, the evaluation results prove that the ITS in its present form is of sufficient standard to teach PHP to beginning programmers and has achieved the predominant goal of this thesis of analysing alternative solutions to a given programming exercise to a great degree.

Bibliography

- Adam, A., & Laurent, J.-P. (1980). LAURA, a system to debug student programs *Artificial Intelligence*, 15(1-2), 75-122. doi: [http://dx.doi.org/10.1016/0004-3702\(80\)90019-3](http://dx.doi.org/10.1016/0004-3702(80)90019-3)
- Al-Imamy, S., Alizadeh, J., & Nour, M. A. (2006). On the development of a programming teaching tool: The effect of teaching by templates on the learning process. *Journal of Information Technology Education*, 5, 271-283. <http://jite.org/documents/Vol5/v5p271-283Al-Imamy115.pdf>
- Aleven, V., Sewall, B. M. M. J., & Koedinger, K. R. (2006). The Cognitive Tutor Authoring Tools (CTAT): Preliminary evaluation of efficiency gains. In M. Ikeda, K. D. Ashley & T. W. Chan (Eds.), 8th International Conference on Intelligent Tutoring Systems Lecture Notes in Computer Science (Vol. 4053, pp. 61-70). Berlin Heidelberg: Springer-Verlag. Retrieved from <http://www.cs.cmu.edu/~bmclaren/pubs/AlevenEtAl-CTAT-ITS2006.pdf>. doi: 10.1007/11774303
- Anderson, J. R. (1993). *Rules of the mind*: Lawrence Erlbaum.
- Anderson, J. R. (1996). *The architecture of cognition* Retrieved from http://books.google.com.au/books?id=BHda99HmRpsC&printsec=frontcover&dq=%22the+architecture+of+cognition%22&source=bl&ots=kiQVeJb2dD&sig=QnOKor6KgleDyA2MZSIgz1o8ov4&hl=en&ei=6DbGTIy8CYe3cK6ApbMO&sa=X&oi=book_result&ct=result&resnum=1&ved=0CB0Q6AEwAA#v=onepage&q&f=false
- Anderson, J. R., Corbett, A. T., Koedinger, K. R., & Pelletier, R. (1995). Cognitive tutors:Lessons learned. *The Journal of Learning Sciences*, 4(2), 167-207. doi:10.1207/s15327809jls0402_2
- . Apache (Version 2.0). Retrieved from <http://www.apache.org/>
- Beck, J. E., Chang, K.-m., Mostow, J., & Corbett, A. (2008). Does help help? Introducing the Bayesian evaluation and assessment methodology. In B. P. Woolf, E. Aïmeur, R. Nkambou & S. Lajoie (Eds.), *9th International Conference on Intelligent Tutoring Systems* (Vol. 5091, pp. 383-394). Berlin Heidelberg: Springer-Verlag.
- Bloom, B. S. (1984). The 2 sigma problem: The search for methods of group instruction as effective as one-to-one tutoring. *Educational Researcher*, 13(6), 4-16. <http://web.mit.edu/bosworth/MacData/afs.course/5/5.95/readings/bloom-two-sigma.pdf>
- Brna, P., Bundy, A., Dodd, T., Eisenstadt, M., Looi, C. K., Pain, H., . . . Someren, M. (1991). Prolog programming techniques. *Instructional Science*, 20(2), 111-133.
- Bull, S. (2012). Preferred Features of Open Learner Models for University Students. In S. Cerri, W. Clancey, G. Papadourakis & K. Panourgia (Eds.), *11th International Conference on Intelligent Tutoring Systems* (Vol. 7315, pp. 411-421). Berlin Heidelberg: Springer-Verlag.
- Chee, Y. S. (1994). SMALLTALKER: A Cognitive Apprenticeship Multimedia Environment for Learning Smalltalk Programming. *Educational Multimedia and Hypermedia, 1994. Proceedings of ED-MEDIA 94--World Conference*

- on Educational Multimedia and Hypermedia*
<http://www.eric.ed.gov/ERICWebPortal/detail?accno=ED388291>
- . Connector/ODBC (Version 5.1.11). Retrieved from
<http://www.mysql.com/downloads/connector/odbc/>
- Corbett, A. T. (2000). Cognitive Mastery Learning in the ACT Programming Tutor. *AAAI Technical Report SS-00-01*. Cognitive Mastery Learning in the ACT Programming Tutor
- Corbett, A. T. (2001). Cognitive computer tutors: Solving the two-sigma problem. In M. Bauer, P. J. Gmytrasiewicz & J. Vassileva (Eds.), 8th International Conference on User Modeling Lecture Notes in Computer Science, UM 2001 (Vol. 2109, pp. 137-147). Sonthofen, Germany: Springer Verlag. doi: 10.1007/3-540-44566-8
- Corbett, A. T., & Anderson, J. R. (1992). Student modeling and mastery learning in a computer-based programming tutor. In C. Frasson, G. Gauthier & G. I. McCalla (Eds.), 2nd International Conference on Intelligent Tutoring Systems (Vol. 608). Berlin Heidelberg: Springer-Verlag. Retrieved from <http://repository.cmu.edu/cgi/viewcontent.cgi?article=1088&context=psychology>.
- Corbett, A. T., & Anderson, J. R. (1995). Knowledge tracing: Modeling the acquisition of student knowledge. *User Modeling and User - Adapted Interaction*, 4, 253-278. <http://act-r.psy.cmu.edu/papers/893/CorbettAnderson1995.pdf>
- Delgado, K. V., & Barros, L. N. d. (2004). ProPAT: A programming ITS based on pedagogical patterns. In J. C. Lester, R. M. Vicari & F. Paraguacu (Eds.), 7th International Conference on Intelligent Tutoring Systems (Vol. 3220, pp. 812-814). Berlin Heidelberg: Springer-Verlag.
- Ebrahimi, A. (1994). Novice programmer errors: Language constructs and plan composition. *International Journal of Human Computer Studies*, 41(4), 457-480. doi:10.1006/ijhc.1994.1069
- Ebrahimi, A., & Schweikert, C. (2006). Empirical study of novice programming with plans and objects. *Working group reports on ITiCSE on Innovation and technology in computer science education*, 52-54. doi:10.1145/1189215.1189169
- . Eclipse (Version Indigo). (2011). Retrieved from <http://www.eclipse.org/>
- Eliot, C., Williams, K., & Woolf, B. (1996). An intelligent learning environment for advanced cardiac life support. In M. D. James J. Cimino (Ed.), *Proceedings of the AMIA annual fall symposium* (pp. 7-11): American Medical Informatics Association.
- Galvez, J., Guzman, E., Conejo, R., & Millan, E. (2009). Student knowledge diagnosis using Item Response Theory and Constraint-Based Modeling. In V. Dimitrova, R. Mizoguchi, B. Du Boulay & A. C. Graesser (Eds.), 14th International Conference on Artificial Intelligence in Education (Vol. 200, pp. 291-298). Brighton, UK: IOS Press.
- Garner, S. (2007). An exploration of how a technology-facilitated part-complete solution method supports the learning of computer programming. (Technical report). *Issues in Informing Science & Information Technology*, 4, 491. <http://proceedings.informingscience.org/InSITE2007/IISITv4p491-501Garn260.pdf>
- Gegg-Harrison, T. S. (1991). Learning Prolog in a schema-based environment. *Instructional Science*, 20(2), 173-192. doi:10.1007/BF00120881

- Gomes, A., & Mendes, A. J. (2007). Learning to program - difficulties and solutions. *International Conference on Engineering Education – ICEE*.
<http://www.ineer.org/Events/ICEE2007/papers/411.pdf>
- Gong, Y., Beck, J., & Heffernan, N. (2012). WEBSistments: Enabling an Intelligent Tutoring System to Excel at Explaining Rather Than Coaching. In S. Cerri, W. Clancey, G. Papadourakis & K. Panourgia (Eds.), *11th International Conference on Intelligent Tutoring Systems* (Vol. 7315, pp. 268-273). Berlin Heidelberg: Springer-Verlag.
- Gries, D. (1981). *The science of programming* (Vol. 198). New York: Springer-Verlag.
- Halpin, T. A., & Morgan, T. (2008). *Information modeling and relational databases*. Burlington, MA: Elsevier/Morgan Kaufman Publishers.
- Hatzilygeroudis, I., & Prentzas, J. (2004). Knowledge representation requirements for Intelligent Tutoring Systems. In J. C. Lester, R. M. Vicari & F. Paraguacu (Eds.), *7th International Conference on Intelligent Tutoring Systems* (Vol. 3220, pp. 87-97). Berlin Heidelberg: Springer-Verlag.
- Heffernan, N. Ms. Lindquist : The tutor. Retrieved February 16, 2011, from <http://www.cs.cmu.edu/~neil/>
- Holland, J., Mitrovic, A., & Martin, B. (2009). J-LATTE: a Constraint-based Tutor for Java. In S. C. Kong, H. Ogata, H. C. Arnseth, C. K. K. Chan, T. Hirashima, F. Klett, J. H. M. Lee, C. C. Liu, C. K. Looi, M. Milrad, A. Mitrovic, K. Nakabayashi, S. L. Wong & S. J. H. Yang (Eds.), *17th International Conference on Computers in Education* (pp. 142-146). Hong Kong: Asia-Pacific Society for Computers in Education.
- Hong, J. (2004). Guided programming and automated error analysis in an intelligent Prolog tutor. *International Journal of Human-Computer Studies*, 61(4), 505-534. doi:10.1016/j.ijhcs.2004.02.001
- Huth, M., & Ryan, M. (2004). *Logic in Computer Science - Modelling and reasoning about systems* (Second Edition ed.). Cambridge, UK: Cambridge University Press.
- Jin, W., Barnes, T., Stamper, J., Eagle, M., Johnson, M., & Lehmann, L. (2012). Program Representation for Automatic Hint Generation for a Data-Driven Novice Programming Tutor. In S. Cerri, W. Clancey, G. Papadourakis & K. Panourgia (Eds.), *11th International Conference on Intelligent Tutoring Systems* (Vol. 7315, pp. 304-309). Berlin Heidelberg: Springer-Verlag.
- Johns, J., Mahadevan, S., & Woolf, B. (2006). Estimating student proficiency using an item response theory model. In M. Ikeda, K. D. Ashley & T. W. Chan (Eds.), *8th International Conference on Intelligent Tutoring Systems* (Vol. 4053, pp. 473-480). Berlin Heidelberg: Springer-Verlag
- Johnson, W. L. (1985). *Intention based diagnosis of errors in novice programs*. (PhD), Yale University.
- Johnson, W. L. (1990). Understanding and debugging novice programs. *Artificial Intelligence*, 42(1), 51-97. doi:10.1016/0004-3702(90)90094-G
- Johnson, W. L., & Soloway, E. (1985). PROUST: Knowledge-based program understanding. *IEEE Transactions on Software Engineering*, SE-11(3), 267 - 275 doi:10.1109/TSE.1985.232210
- Kelleher, C., & Pausch, R. (2005). Lowering the barriers to programming : A taxonomy of programming environments and languages for novice programmers. *ACM Computing Surveys*, 37(2), 83-137. doi:10.1145/1089733.1089734

- Kemp, R., Kemp, E., & Todd, E. (2009). Self-regulated fading in on-line learning. In V. Dimitrova, R. Mizoguchi, B. Du Boulay & A. C. Graesser (Eds.), *14th International Conference on Artificial Intelligence in Education* (Vol. 200, pp. 449-456). Brighton, UK: IOS Press.
- Koedinger, K. R., Anderson, J. R., Hadley, W. H., & Mark, M. A. (1997). Intelligent tutoring goes to school in the big city. *International Journal of Artificial Intelligence in Education*, 8(1), 30-43.
http://www.ijaied.org/pub/1041/file/1041_paper.pdf
- Koedinger, K. R., & Sueker, E. L. F. (1996). PAT goes to college: evaluating a cognitive tutor for developmental mathematics. In D. C. Edelson & E. A. Domeshek (Eds.), *Proceedings of the 1996 international conference on Learning sciences* (pp. 180-187). Evanston, USA: International Society of the Learning Sciences.
- Kolling, M. (2010). The Greenfoot Programming Environment. *ACM Transactions on Computing Education*, 10(4), 1-21. doi:10.1145/1868358.1868361
- Kuruwila, S. (2009). phpparser. Retrieved from
<http://code.google.com/p/phpparser/downloads/detail?name=Php.g&can=2&q=>
- Long, Y., & Aleven, V. (2011). Students' understanding of their student model. In G. Biswas, S. Bull, J. Kay & A. Mitrovic (Eds.), *15th International Conference on Artificial Intelligence in Education* (Vol. 6738, pp. 179-186). Berlin Heidelberg: Springer-Verlag.
- Looi, C. K. (1991). Automatic debugging of Prolog programs in a Prolog Intelligent Tutoring System. *Instructional Science*, 20(2-3), 215-263.
 doi:10.1007/BF00120883
- Mark, M., & Greer, J. (1993). Evaluation methodologies for intelligent tutoring systems. *Journal of Artificial Intelligence in Education*, 4, 129-129.
<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.52.6842&rep=rep1&type=pdf>
- Mason, B. J., & Bruning, R. (2001). Providing feedback in computer-based instruction: What the research tells us (Vol. 6, pp. 2004): Center for Instructional Innovation, University of Nebraska–Lincoln.
- Mayo, M., & Mitrovic, A. (2001). Optimising ITS behaviour with Bayesian networks and Decision Theory. *International Journal of Artificial Intelligence in Education*, 12, 124-153. http://www.ijaied.org/pub/961/file/961_paper.pdf
- Mayo, M., Mitrovic, A., & McKenzie, J. (2000). CAPIT: An intelligent tutoring system for capitalisation and punctuation. In J. C. Kinshuk & O. T (Eds.), *Advanced Learning Technology: Design and Development Issues* (pp. 151-154). Los Alamitos, CA: IEEE Computer Society
- Miliszewska, I., & Tan, G. (2007). Befriending computer programming: a proposed approach to teaching introductory programming. *Issues in Informing Science & Information Technology*, 4, 277.
<http://proceedings.informingscience.org/InSITE2007/IISITv4p277-289Mili310.pdf>
- . MinGW - Minimalist GNU for Windows. Retrieved from <http://www.mingw.org/>
- Mitrovic, A. (1998). A knowledge-based teaching system for SQL. *Proceedings of ED-MEDIA, 1998, World Conference on Educational Multimedia, Hypermedia & Telecommunications*, 98, 1027-1032.
<http://www.cosc.canterbury.ac.nz/tanja.mitrovic/702.pdf>

- Mitrovic, A., Suraweera, P., Martin, B., & Weerasinghe, A. (2004). DB-Suite: Experiences with three intelligent, web-based database tutors. *Journal of Interactive Learning Research*, 15(4), 409.
- Mow, I. T. C. (2008). *Issues and difficulties in teaching novice computer programming Innovative techniques in instruction technology, e-learning, e-assessment and education* (pp. 199-204). Retrieved from EBL Ebook Library database doi:10.1007/978-1-4020-8739-4_36
- Murray, T. (1999). Authoring intelligent tutoring systems: An analysis of the state of the art. *International Journal of Artificial Intelligence in Education*, 10(1), 98-129. http://www.ijaied.org/pub/991/file/991_paper.pdf
- . MySQL (Version 5.5.16). Retrieved from <http://www.mysql.com/>
- Naser, S. S. A. (2008). Developing an Intelligent Tutoring System For Students Learning To Program in C++. *Information Technology Journal*, 7(7), 1055-1060. <http://docsdrive.com/pdfs/ansinet/itj/2008/1055-1060.pdf>
- Nunamaker Jr., J. F., Chen, M., & Purdin, T. D. M. (1990). Systems development in information systems research. *Journal of Management Information Systems*, 7(3), 89-106.
- Ohlsson, S., & Mitrovic, A. (2006). Constraint-based knowledge representation for individualized instruction. *Computer Science and Information Systems*, 3(1), 1-22. <http://www.cosc.canterbury.ac.nz/tanja.mitrovic/comsis.pdf>
- Ohlsson, S., & Mitrovic, A. (2007). Fidelity and efficiency of knowledge representations for intelligent tutoring systems.
- Parr, T. (2007). *The Definitive ANTLR Reference - Building Domain-Specific Languages*. Raleigh-North Caroline, Dallas-Texas: The Pragmatic Bookshelf.
- Parr, T. (2011). ANTLR v3 (Version 3.4). Retrieved from <http://www.antlr.org>
- . PHP Hypertext Processor (Version 5.3.8). (2011). Retrieved from <http://www.php.net/>
- PHP Manual. (September 7, 2012). Retrieved September 13, 2012, from <http://php.net/manual/en/index.php>
- PHP Tutorial. (undated). Retrieved May 13, 2010, from <http://www.w3schools.com/php/default.asp>
- PHP tutorial - free. Retrieved May 13, 2010, from <http://www.learnphp-tutorial.com/>
- PHP/MySQL Tutorial. Retrieved May 13, 2010, from <http://www.freewebmasterhelp.com/tutorials/phpmysql>
- PHP: A simple tutorial - manual. Retrieved May 13, 2010, from <http://php.net/manual/en/tutorial.php>
- Pillay, N. (2003). Developing intelligent programming tutors for novice programmers. *ACM SIGCSE Bulletin*, 35(2), 78-82.
- Razzaq, L., & Heffernan, N. (2006). Scaffolding vs. hints in the Assistment system. In M. Ikeda, K. D. Ashley & T. W. Chan (Eds.), *8th International Conference on Intelligent Tutoring Systems* (Vol. 4053, pp. 635-644). Berlin Heidelberg: Springer-Verlag.
- Reye, J. (1998). Two-phase updating of student models based on dynamic belief networks. In B. P. Goettl, H. M. Half, C. L. Redfield & V. J. Shute (Eds.), *4th International Conference on Intelligent Tutoring Systems* (Vol. 1452, pp. 274-283). Berlin Heidelberg: Springer-Verlag.
- Reye, J. (2004). Student modelling based on belief networks. *International Journal of Artificial Intelligence in Education*, 14(1), 63-96. http://www.ijaied.org/pub/956/file/956_Reye04.pdf

- Riley, G. (2011). CLIPS - A Tool for Building Expert Systems (Version 6.3). Retrieved from clipsrules.sourceforge.net
- Risco, S., & Reye, J. (2009). Personal Access Tutor - helping students to learn MS Access. In V. Dimitrova, R. Mizoguchi, B. Du Boulay & A. C. Graesser (Eds.), *14th International Conference on Artificial Intelligence in Education* (Vol. 200, pp. 541-548). Brighton, UK: IOS Press.
- Rivers, K., & Koedinger, K. (2012). A Canonicalizing Model for Building Programming Tutors. In S. Cerri, W. Clancey, G. Papadourakis & K. Panourgia (Eds.), *11th International Conference on Intelligent Tutoring Systems* (Vol. 7315, pp. 591-593). Berlin Heidelberg: Springer-Verlag.
- Robins, A., Rountree, J., & Rountree, N. (2003). Learning and teaching programming: A review and discussion. *Computer Science Education*, 13(2), 137-172. doi:10.1076/csed.13.2.137.14200
- Russell, S. J., & Norvig, P. (2010). *Artificial intelligence*. Upper Saddle River, N.J: Prentice Hall.
- Sack, W., Soloway, E., & Weingrad, P. (1992). From PROUST to CHIRON: ITS Design as Iterative Engineering: Intermediate Results are Important. *Computer-Assisted Instruction and Intelligent Tutoring Systems: Shared Goals and Complementary Approaches*. Lawrence Erlbaum Associates, Hillsdale, NJ, 239-274.
- Self, J. (1990). Bypassing the intractable problem of student modelling. In C. Frasson & G. Gauthier (Eds.), *Intelligent tutoring systems: At the crossroads of artificial intelligence and education* (pp. 107-123). Norwood, N.J, USA: Ablex. Retrieved from <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.55.2809&rep=rep1&type=pdf>.
- Singh, R., Saleem, M., Pradhan, P., Heffernan, C., Heffernan, N., Razzaq, L., . . . Mulcahy, C. (2011). Feedback during web-based homework: The role of hints. In G. Biswas, S. Bull, J. Kay & A. Mitrovic (Eds.), *15th International Conference on Artificial Intelligence in Education* (Vol. 6738, pp. 328-336). Berlin Heidelberg: Springer-Verlag.
- Song, J. S., Hahn, S. H., Tak, K. Y., & Kim, J. H. (1997). An intelligent tutoring system for introductory C language course. *Computers & Education*, 28(2), 93-102. doi:10.1016/s0360-1315(97)00003-1
- Spohrer, J. C., & Soloway, E. (1986). Novice mistakes: Are the folk wisdoms correct? *Communications of the ACM*, 29(7), 624-632. doi:10.1145/6138.6145
- Staveland, A. M. (1993). An empirical-study of iteration in applications software. *Journal of Systems and Software*, 22(3), 167-177.
- Suarez, M., & Sison, R. (2008). Automatic construction of a bug library for object-oriented novice Java programmer errors. In B. P. Woolf, E. Aïmeur, R. Nkambou & S. Lajoie (Eds.), *9th International Conference on Intelligent Tutoring Systems* (Vol. 5091, pp. 184-193). Berlin Heidelberg: Springer-Verlag.
- Suraweera, P., & Mitrovic, A. (2002). KERMIT: A constraint-based tutor for database modeling. In S. Cerri, G. Gouardères & F. Paraguaçu (Eds.), *6th International Conference on Intelligent Tutoring Systems* (Vol. 2363, pp. 377-387). Berlin Heidelberg: Springer-Verlag.
- Sykes, E. (2007). Developmental process model for the Java Intelligent Tutoring System. *Journal of Interactive Learning Research*, 18(3), 399-410.

- Sykes, E., & Franek, F. (2004). Presenting JECA: A java error correcting algorithm for the java intelligent tutoring system. *IASTED International Conference on Advances in Computer Science and Technology*.
<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.137.303&rep=rep1&type=pdf>
- Sykes, E. R. (2006). *Design, development and assessment of the Java Intelligent Tutoring System*. (Ph.D. NR18018), Brock University (Canada), Canada. Available from ProQuest Dissertations and Theses database.
- TIOBE Programming Community Index for December 2012. Retrieved December 18, 2012, from
<http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html>
- Truong, N. (2007). *A web-based programming environment for novice programmers*. (Doctor of Philosophy), Queensland University of Technology, Brisbane. Retrieved from http://eprints.qut.edu.au/16471/1/Nghi_Truong_Thesis.pdf
- Truong, N., Bancroft, P., & Roe, P. (2003). A web based environment for learning to program. *Proceedings of the 26th Australasian Computer Science Conference, 16*, 255-264. <http://crpit.com/confpapers/CRPITV16Truong.pdf>
- VanLehn, K., Lynch, C., Schulze, K., Shapiro, J. A., Shelby, R., Taylor, L., . . . Wintersgill, M. (2005). The Andes physics tutoring system: Lessons learned *International Journal of Artificial Intelligence in Education, 15*(1), 147-204. <http://www.ijaied.org/pub/1135/file/VanLehn05.pdf>
- VanLehn, K., Siler, S., Murray, C., & Baggett, W. (1998). What makes a tutorial event effective. In M. A. Gernsbacher & S. Derry (Eds.), *Proceedings of the Twenty-First Annual Conference of the Cognitive Science Society* (pp. 1084-1089). Hillsdale, NJ, USA: Erlbaum. Retrieved from citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.115.1180&rep=rep1&type=pdf.
- Vygotsky, L. (1978). Interaction between learning and development. *Readings on the Development of Children, 34-41*.
- Wang, X. (2006). A practical way to teach web programming in computer science. *J. Comput. Small Coll., 22*(1), 211-220.
- Weber, G. (1996). Individual selection of examples in an intelligent learning environment. *Journal of Artificial Intelligence in Education, 7*(1), 3-31.
- Weber, G., & Brusilovsky, P. (2001). ELM-ART: An adaptive versatile system for web-based instruction. *International Journal of Artificial Intelligence in Education, 12*(4), 351-384. http://www.ijaied.org/pub/965/file/965_paper.pdf
- Weber, G., & Möllenberg, A. (1995). ELM programming environment: A tutoring system for LISP beginners. In K. F. Wender, F. Schmalhofer & H.-D. Böcker (Eds.), *Cognition and computer programming* (pp. 373-408). Norwood, New Jersey, USA: Ablex Publishing Corporation. Retrieved from http://books.google.com.au/books?hl=en&lr=&id=nfysCfUVs3UC&oi=fnd&pg=PA373&dq=ELM-programming+environment&ots=ghdQjUqxSq&sig=U7cfBw35uAa_XPVXWe8qzsy8Ngk#v=onepage&q=ELM-programming%20environment&f=false.
- Wenger, E. (1987). *Artificial intelligence and tutoring systems*. Los Altos, CA: Morgan Kaufman.
- Weragama, D., & Reye, J. (2012). Design of a Knowledge Base to Teach Programming. In S. Cerri, W. Clancey, G. Papadourakis & K. Panourgia

- (Eds.), *11th International Conference on Intelligent Tutoring Systems* (Vol. 7315, pp. 600-602). Berlin Heidelberg: Springer-Verlag.
- Winskel, G. (1993). *The Formal Semantics of Programming Languages - An Introduction*. Cambridge, MA: MIT Press.
- Woolf, B. P. (2009). *Building intelligent interactive tutors*. Burlington, MA: Morgan Kaufman.
- . XAMPP. Retrieved from <http://sourceforge.net/projects/xampp>

Appendices

Appendix A Introduction to Bayesian Belief Networks

Bayesian Belief Networks (BBNs) are an important method of representation used for student modelling in Intelligent Tutoring Systems. This appendix describes the basic theory of BBNs. The description here is based on the book “Artificial Intelligence a Modern Approach” (Russell & Norvig, 2010) and the interested reader is referred to Chapter 14 of this book for further information.

Sometimes problem solving agents in AI need to handle uncertainty. This uncertainty is usually quantified using probability theory. Probabilities that refer to a degree of belief in propositions in the absence of any other information are called **unconditional** or **prior probabilities**. Probabilities that refer to a degree of belief after certain information is obtained are called **conditional** or **posterior probabilities**.

Probabilistic assertions are about possible worlds. A possible world is represented using a set of variable/value pairs. Such variables used in probability theory are called **random variables**. A **probability distribution** specifies all possible values of a random variable in vector form. The probabilities of all combinations of the values of two or more random variables are usually given in a table known as the **joint probability distribution**. The joint probability distribution of all possible random variables is called the **full joint probability distribution** and the probability model is entirely determined by this.

As the number of random variables become higher, the full joint probability distribution gets more and more complex. Very often, many of these random variables are independent from each other and therefore, the joint probability distribution contains a lot of unnecessary data. In such cases, the dependencies among random variables in a probability distribution can be represented using a data structure called a **Bayesian Belief Network (BBN)**. This uses nodes to represent the random variables and a set of directed links to show the relationships. Each node has a conditional probability distribution that specifies the effect of the

parent on the node. This means that probability distributions need to be maintained only for the random variables that are inter-related.

Figure A1 shows a Bayesian Belief Network for how a student learns a topic. L_{n-1} is the knowledge state of the student before using the topic to solve some problem. This influences the outcome when the student demonstrates usage of the topic by answering a question since the outcome would depend on the student's current knowledge. Assuming that the outcome contributes to learning (i.e. some form of feedback is provided based on whether the answer was correct or not), the learned state after this depends on both the outcome and the previous level of knowledge. These relationships are shown by the arrows in the BBN.

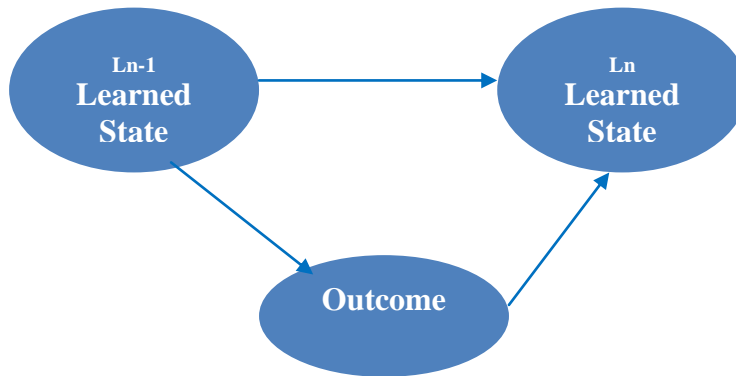


Figure A1. A Bayesian Belief Network for pre-requisite topics.

Given the knowledge level of the student before demonstrating the topic as well as the outcome of the demonstration, together with the probability distributions at each of the nodes, the posterior probability that the student learned the topic can be calculated. Although the actual calculation can be quite tedious, different algorithms have been found for this purpose.

Appendix B PHP Grammar

This is the grammar used for parsing the PHP component written by student programs. The shaded parts indicate the sections of the PHP language that are not covered by the PHP ITS.

```
grammar php;

tokens{
    SemiColon = ';';
    Comma = ',';
    OpenBrace = '(';
    CloseBrace = ')';
    OpenSquareBrace = '[';
    CloseSquareBrace = ']';
    OpenCurlyBrace = '{';
    CloseCurlyBrace = '}';
    ArrayAssign = '=>';
    LogicalOr = '||';
    LogicalAnd = '&&';
    ClassMember = '::';
    InstanceMember = '->';
    SuppressWarnings = '@';
    QuestionMark = '?';
    Dollar = '$';
    Colon = ':';
    Dot = '.';
    Ampersand = '&';
    Pipe = '|';
    Bang = '!';
    Plus = '+';
    Minus = '-';
    Asterisk = '*';
    Percent = '%';
    Forwardslash = '/';
    Tilde = '~';
    Equals = '=';
    New = 'new';
    Clone = 'clone';
    Echo = 'echo';
    If = 'if';
    Else = 'else';
    ElseIf = 'elseif';
    For = 'for';
    Foreach = 'foreach';
    While = 'while';
    Do = 'do';
    Switch = 'switch';
    Case = 'case';
    Default = 'default';
    Function = 'function';
    Break = 'break';
    Continue = 'continue';
    Goto = 'goto';
    Return = 'return';
}
```

```

Global = 'global';
Static = 'static';
And = 'and';
Or = 'or';
Xor = 'xor';
Instanceof = 'instanceof';

Class = 'class';
Interface = 'interface';
Extends = 'extends';
Implements = 'implements';
Abstract = 'abstract';
Var = 'var';
Const = 'const';
Modifiers;
ClassDefinition;

Block;
Params;
Apply;
Member;
Reference;
Empty;
Prefix;
Postfix;
IfExpression;
Label;
Cast;
ForInit;
ForCondition;
ForUpdate;
Field;
Method;
}

prog
:statement*;

statement
: simpleStatement? BodyString
  | '{' statement '}' -> statement
  | bracketedBlock
  | UnquotedString Colon statement -> ^(Label UnquotedString statement)
  | classDefinition
  | interfaceDefinition
  | complexStatement
  | simpleStatement ';'!
;

bracketedBlock
: '{' stmts=statement* '}' -> ^(Block statement*)
;

interfaceDefinition

```



```

: Interface interfaceName=UnquotedString interfaceExtends?
  OpenCurlyBrace
  interfaceMember*
  CloseCurlyBrace
  -> ^(Interface $interfaceName interfaceExtends? interfaceMember*)
;

interfaceExtends
: Extends^ UnquotedString (Comma! UnquotedString)*
;

interfaceMember
: Const UnquotedString (Equals atom)? ';'
  -> ^(Const UnquotedString atom?)
  | fieldModifier* Function UnquotedString

parametersDefinition ';'
  -> ^(Method ^(Modifiers fieldModifier*) UnquotedString
parametersDefinition)
;

classDefinition
: classModifier?
  Class className=UnquotedString
  (Extends extendsclass=UnquotedString)?
  classImplements?
  OpenCurlyBrace
  classMember*
  CloseCurlyBrace
  -> ^(Class ^(Modifiers classModifier?) $className ^(Extends
$extendsclass)? classImplements?
  classMember*
  )
;

classImplements
: Implements^ (UnquotedString (Comma! UnquotedString)*)
;

classMember
: fieldModifier* Function UnquotedString parametersDefinition
  (bracketedBlock | ';')
  -> ^(Method ^(Modifiers fieldModifier*) UnquotedString
parametersDefinition bracketedBlock?)
  | Var Dollar UnquotedString (Equals atom)? ';'
  -> ^(Var ^(Dollar UnquotedString) atom?)
  | Const UnquotedString (Equals atom)? ';'
  -> ^(Const UnquotedString atom?)
  | fieldModifier* (Dollar UnquotedString) (Equals atom)? ';'
  -> ^(Field ^(Modifiers fieldModifier*) ^(Dollar UnquotedString)
atom?)
;

fieldDefinition
: Dollar UnquotedString (Equals atom)? ';' -> ^(Field ^(Dollar
UnquotedString) atom?)
;

classModifier
: 'abstract';

```

```

fieldModifier
: AccessModifier | 'abstract' | 'static'
;

complexStatement
: If '(' ifCondition=expression ')' ifTrue=statement conditional?
  -> ^('if' expression $ifTrue conditional?)
| For '(' forInit forCondition forUpdate ')' statement -> ^(For
forInit forCondition forUpdate statement)
| Foreach '(' variable 'as' arrayEntry ')' statement -> ^(Foreach
variable arrayEntry statement)
| While '(' whileCondition=expression? ')' statement -> ^(While
$whileCondition statement)
| Do statement While '(' doCondition=expression ')' ';' -> ^(Do
statement $doCondition)
| Switch '(' expression ')' '{cases}' -> ^(Switch expression cases)
| functionDefinition
;

simpleStatement
: Echo^ commaList
| Global^ name (',! name)*
| Static^ variable Equals! atom
| Break^ Integer?
| Continue^ Integer?
| Goto^ UnquotedString
| Return^ expression?
| RequireOperator^ expression
| expression
;

conditional
: ElseIf '(' ifCondition=expression ')' ifTrue=statement conditional?
-> ^(If $ifCondition $ifTrue conditional?)
| Else statement -> statement
;

forInit
: commaList? ';' -> ^(ForInit commaList?)
;

forCondition
: commaList? ';' -> ^(ForCondition commaList?)
;

forUpdate
: commaList? -> ^(ForUpdate commaList?)
;

cases
: casestatement* defaultcase?
;

casestatement
: Case^ expression ':'! statement*
;

```

```

defaultcase
  : (Default^ ':'! statement*)
  ;

functionDefinition
  : Function UnquotedString parametersDefinition bracketedBlock ->
    ^(Function UnquotedString parametersDefinition bracketedBlock)
  ;

parametersDefinition
  : OpenBrace (paramDef (Comma paramDef)*)? CloseBrace -> ^(Params
paramDef*)
  ;

paramDef
  : paramName (Equals^ atom)?
  ;

paramName
  : Dollar^ UnquotedString
  | Ampersand Dollar UnquotedString -> ^(Ampersand ^(Dollar
UnquotedString))
  ;

commalist
  : expression (','! expression)*
  ;

expression
  : weakLogicalOr
  ;

weakLogicalOr
  : weakLogicalXor (Or^ weakLogicalXor)*
  ;

weakLogicalXor
  : weakLogicalAnd (Xor^ weakLogicalAnd)*
  ;

weakLogicalAnd
  : assignment (And^ assignment)*
  ;

assignment
  : name ((Equals | AssignmentOperator)^ assignment)
  | ternary
  ;

ternary
  : logicalOr QuestionMark expression Colon expression -> ^(IfExpression
logicalOr expression*)
  | logicalOr
  ;

```

```

logicalOr
  : logicalAnd (LogicalOr^ logicalAnd)*
  ;

logicalAnd
  : bitwiseOr (LogicalAnd^ bitwiseOr)*
  ;

bitwiseOr
  : bitWiseAnd (Pipe^ bitWiseAnd)*
  ;

bitWiseAnd
  : equalityCheck (Ampersand^ equalityCheck)*
  ;

equalityCheck
  : comparisionCheck (EqualityOperator^ comparisionCheck)?
  ;

comparisionCheck
  : bitWiseShift (ComparisionOperator^ bitWiseShift)?
  ;

bitWiseShift
  : addition (ShiftOperator^ addition)*
  ;

addition
  : multiplication ((Plus | Minus | Dot)^ multiplication)*
  ;

multiplication
  : logicalNot ((Asterisk | Forwardslash | Percent)^ logicalNot)*
  ;

logicalNot
  : Bang^ logicalNot
  | instanceOf
  ;

instanceOf
  : negateOrCast (Instanceof^ negateOrCast)?
  ;

negateOrCast
  : (Tilde | Minus | SuppressWarnings)^ increment
  | OpenBrace PrimitiveType CloseBrace increment -> ^(Cast PrimitiveType
increment)
  | OpenBrace! weakLogicalAnd CloseBrace!
  | increment
  ;

increment
  : IncrementOperator name -> ^(Prefix IncrementOperator name)
  | name IncrementOperator -> ^(Postfix IncrementOperator name)
  | newOrClone

```

```

;
newOrClone
: New^ nameOrFunctionCall
| Clone^ name
| atomOrReference
;

atomOrReference
: atom
| reference
;

arrayDeclaration
: Array OpenBrace (arrayEntry (Comma arrayEntry)*)? CloseBrace ->
^(Array arrayEntry*)
;

arrayEntry
: (keyValuePair | expression)
;

keyValuePair
: (expression ArrayAssign expression) -> ^(ArrayAssign expression+)
;

atom: SingleQuotedString | DoubleQuotedString | HereDoc | Integer | Real |
Boolean | arrayDeclaration
;

reference
: Ampersand^ nameOrFunctionCall
| nameOrFunctionCall
;

nameOrFunctionCall
: name OpenBrace (expression (Comma expression)*)? CloseBrace ->
^(Apply name expression*)
| name
;

name: staticMemberAccess
| memberAccess
| variable
;

staticMemberAccess
: UnquotedString '::'^ variable
;

memberAccess
: variable
( OpenSquareBrace^ expression CloseSquareBrace!
| '->'^ UnquotedString)*
;

variable
: Dollar^ variable
| UnquotedString

```

```

;

BodyString
: '?>' ;

MultilineComment
: '/*' (('*' ~ '/')=>'*' | ~ '*')* '*/' ;

SinglelineComment
: '//' (('?' ~ '>')=>'?' | ~('\n'|'?'))* ;

UnixComment
: '#' (('?' ~ '>')=>'?' | ~('\n'|'?'))*
;

Array
: ('a'|'A')('r'|'R')('r'|'R')('a'|'A')('y'|'Y')
;

RequireOperator
: 'require' | 'require_once' | 'include' | 'include_once'
;

PrimitiveType
: 'int' | 'float' | 'string' | 'array' | 'object' | 'bool'
;

AccessModifier
: 'public' | 'private' | 'protected'
;

fragment
Decimal
: ('1'..'9' ('0'..'9')*)|'0'
;

fragment
Hexadecimal
: '0'('x'|'X')('0'..'9'|'a'..'f'|'A'..'F')+
;

fragment
Octal
: '0'('0'..'7')+
;

//Minus sign added to handle negative numbers singly
Integer
: '-'? (Octal|Decimal|Hexadecimal)
;

fragment
Digits
: '0'..'9'+
;

```

```

fragment
DNum
    : (('.' Digits)=>('.' Digits)|(Digits '.' Digits?))
    ;

```

```

fragment
Exponent_DNum
    : ((Digits|DNum)('e'|'E')('+''-')?Digits)
    ;

```

//Minus sign added to handle negative numbers singly

```

Real
    : '-'? (DNum|Exponent_DNum)
    ;

```

```

Boolean
    : 'true' | 'false'
    ;

```

```

SingleQuotedString
    : '\\' (('\\' '\\')=>'\\' '\\')
    |   ('\\' '\\')=>'\\' '\\'
    |   '\\'| ~ ('\\'| '\\'))*
    '\\'
    ;

```

```

fragment
EscapeCharector
    : 'n' | 'r' | 't' | '\\\| '$' | '"' | Digits | 'x'
    ;

```

```

DoubleQuotedString
    : '"' ( '\\\| EscapeCharector)=> '\\\| EscapeCharector
    | '\\\|
    | ~('\\\|'"') )*
    '"'
    ;

```

```

HereDoc
    : '<<<' HereDocContents
    ;

```

```

UnquotedString
    : ('a'..'z' | 'A'..'Z' | '_' ) ('a'..'z' | 'A'..'Z' | '0'..'9' | '_' )*
    ;

```

```

HereDocContents
    : ;
    (UnquotedString|Eol)+
    {
        int consumed;
        pANTLR3_STRING thisString;

        consumed=0;
        if($UnquotedString!=NULL)
        {

```

```

    thisString=$UnquotedString.text;
    if(number==1)
    {
        hereDocName=thisString;
    }
    else
    {
        printf("heredoc %s\n",hereDocName->chars);
        printf("thisstring %s\n",thisString->chars);
        if(strcmp(thisString->chars,hereDocName->chars)!=0)
        {
            CONSUME();
            consumed=1;
        }
        else
        {
            //Need to break out of rule
            return;
        }
    }
}else
{
    CONSUME();
}
if(consumed==0)
{
    CONSUME();
}
};

```

AssignmentOperator

```

: '+' | '-' | '*' | '/' | '.' | '%' | '&' | '|' | '^' | '<<=' | '>>='
;

```

EqualityOperator

```

: '==' | '!=' | '===' | '!=='
;

```

ComparisonOperator

```

: '<' | '<=' | '>' | '>=' | '<>'
;

```

ShiftOperator

```

: '<<' | '>>'
;

```

IncrementOperator

```

: '--' | '++'
;

```

fragment

```

Eol : '\n'
;

```

WhiteSpace

```

: (' ' | '\t' | '\n' | '\r')*
;

```


Appendix C Combined Assign Actions

Note that the actions in this appendix take the scope of variables into account. Therefore, these are the actions that have been extended with predicates to handle variable and array scope (Section 6.2.2.3)

$\text{AssignAdd}(x, \text{expressionId}) \subset \text{Assign}(x, \text{expressionId})$

Action(AssignAdd(x, expressionId),

PRECOND:

EFFECT: When \exists variableId (HasName(variableId, 'x')

\wedge HasValue(variableId, value2) \wedge Add(value2, value, value1)

\wedge CurrentScope(funcId) \wedge HasVariableScope(variableId, funcId)):

HasValue(variableId, value2) \leftarrow HasValue(variableId, value1)

$\text{AssignSubtract}(x, \text{expressionId}) \subset \text{Assign}(x, \text{expressionId})$

Action(AssignSubtract(x, expressionId),

PRECOND:

EFFECT: When \exists variableId (HasName(variableId, 'x')

\wedge HasValue(variableId, value2) \wedge Subtract(value2, value, value1)

\wedge CurrentScope(funcId) \wedge HasVariableScope(variableId, funcId)):

HasValue(variableId, value2) \leftarrow HasValue(variableId, value1)

$\text{AssignMultiply}(x, \text{expressionId}) \subset \text{Assign}(x, \text{expressionId})$

Action(AssignMultiply(x, expressionId),

PRECOND:

EFFECT: When \exists variableId (HasName(variableId, 'x')

\wedge HasValue(variableId, value2) \wedge Multiply(value2, value, value1)

\wedge CurrentScope(funcId) \wedge HasVariableScope(variableId, funcId)):

HasValue(variableId, value2) \leftarrow HasValue(variableId, value1)

$\text{AssignDivide}(x, \text{expressionId}) \subset \text{Assign}(x, \text{expressionId})$

$\text{Action}(\text{AssignDivide}(x, \text{expressionId}),$

PRECOND:

EFFECT: When $\exists \text{variableId}$ ($\text{HasName}(\text{variableId}, 'x')$
 $\wedge \text{HasValue}(\text{variableId}, \text{value2}) \wedge \text{Divide}(\text{value2}, \text{value}, \text{value1})$
 $\wedge \text{CurrentScope}(\text{funcId}) \wedge \text{HasVariableScope}(\text{variableId}, \text{funcId})$):
 $\text{HasValue}(\text{variableId}, \text{value2}) \leftarrow \text{HasValue}(\text{variableId}, \text{value1})$)

$\text{AssignModulus}(x, \text{expressionId}) \subset \text{Assign}(x, \text{expressionId})$

$\text{Action}(\text{AssignModulus}(x, \text{expressionId}),$

PRECOND:

EFFECT: When $\exists \text{variableId}$ ($\text{HasName}(\text{variableId}, 'x')$
 $\wedge \text{HasValue}(\text{variableId}, \text{value2}) \wedge (\text{value2}, \text{value}, \text{value1})$
 $\wedge \text{CurrentScope}(\text{funcId}) \wedge \text{HasVariableScope}(\text{variableId}, \text{funcId})$):
 $\text{HasValue}(\text{variableId}, \text{value2}) \leftarrow \text{HasValue}(\text{variableId}, \text{value1})$)

$\text{AssignAddArrayVariable}(x, y, \text{exprId}) \subset \text{AssignArrayVariable}(x, y, \text{exprId})$

$\text{Action}(\text{AssignAddArrayVariable}(x, y, \text{exprId}),$

PRECOND: $\exists \text{value}$ $\text{ValueOf}(\text{exprId}, \text{value})$

EFFECT: when $\exists \text{varId}, \text{arrayId}, \text{keyId}, \text{exprId}$
 $(\text{HasVariableId}(\text{HasElement}(\text{arrayId}, \text{keyId}), \text{varId})$
 $\wedge \text{HasKeyExpression}(\text{keyId}, \text{exprId})$
 $\wedge \text{ValueOf}(\text{exprId}, y)$
 $\wedge \text{HasArrayName}(\text{arrayId}, 'x')$
 $\wedge \text{HasValue}(\text{varId}, \text{value2}) \wedge \text{Add}(\text{value2}, \text{value}, \text{value1})$
 $\wedge \text{CurrentScope}(\text{funcId}) \wedge \text{HasVariableScope}(\text{variableId}, \text{funcId})$):
 $\text{HasValue}(\text{varId}, _) \leftarrow \text{HasValue}(\text{varId}, \text{value1})$)

$\text{AssignSubtractArrayVariable}(x,y,\text{exprId}) \subset \text{AssignArrayVariable}(x,y,\text{exprId})$

Action(AssignSubtractArrayVariable(x,y,exprId))

PRECOND: $\exists \text{value ValueOf}(\text{exprId},\text{value})$

EFFECT: when $\exists \text{varId,arrayed,keyId,exprId}$

$(\text{HasVariableId}(\text{HasElement}(\text{arrayId},\text{keyId}),\text{varId})$

$\wedge \text{HasKeyExpression}(\text{keyId},\text{exprId})$

$\wedge \text{ValueOf}(\text{exprId},y)$

$\wedge \text{HasArrayName}(\text{arrayId},'x')$

$\wedge \text{HasValue}(\text{varId},\text{value2}) \wedge \text{Subtract}(\text{value2},\text{value},\text{value1})$

$\wedge \text{CurrentScope}(\text{funcId}) \wedge \text{HasVariableScope}(\text{variableId},\text{funcId}):$

$\text{HasValue}(\text{varId},_) \leftarrow \text{HasValue}(\text{varId},\text{value1})$

$\text{AssignMultiplyArrayVariable}(x,y,\text{exprId}) \subset \text{AssignArrayVariable}(x,y,\text{exprId})$

Action(AssignMultiplyArrayVariable(x,y,exprId))

PRECOND: $\exists \text{value ValueOf}(\text{exprId},\text{value})$

EFFECT: when $\exists \text{varId,arrayed,keyId,exprId}$

$(\text{HasVariableId}(\text{HasElement}(\text{arrayId},\text{keyId}),\text{varId})$

$\wedge \text{HasKeyExpression}(\text{keyId},\text{exprId})$

$\wedge \text{ValueOf}(\text{exprId},y)$

$\wedge \text{HasArrayName}(\text{arrayId},'x')$

$\wedge \text{HasValue}(\text{varId},\text{value2}) \wedge \text{Multiply}(\text{value2},\text{value},\text{value1})$

$\wedge \text{CurrentScope}(\text{funcId}) \wedge \text{HasVariableScope}(\text{variableId},\text{funcId}):$

$\text{HasValue}(\text{varId},_) \leftarrow \text{HasValue}(\text{varId},\text{value1})$

$\text{AssignDivideArrayVariable}(x,y,\text{exprId}) \subset \text{AssignArrayVariable}(x,y,\text{exprId})$

Action(AssignDivideArrayVariable(x,y,exprId))

PRECOND: $\exists \text{value ValueOf}(\text{exprId},\text{value})$

EFFECT: when $\exists \text{varId,arrayed,keyId,exprId}$

$(\text{HasVariableId}(\text{HasElement}(\text{arrayId},\text{keyId}),\text{varId})$

$\wedge \text{HasKeyExpression}(\text{keyId},\text{exprId})$

$\wedge \text{ValueOf}(\text{exprId},y)$

$\wedge \text{HasArrayName}(\text{arrayId},'x')$

$\wedge \text{HasValue}(\text{varId},\text{value2}) \wedge \text{Divide}(\text{value2},\text{value},\text{value1})$

$\wedge \text{CurrentScope}(\text{funcId}) \wedge \text{HasVariableScope}(\text{variableId},\text{funcId})):$

$\text{HasValue}(\text{varId},_) \leftarrow \text{HasValue}(\text{varId},\text{value1})$

$\text{AssignModulusArrayVariable}(x,y,\text{exprId}) \subset \text{AssignArrayVariable}(x,y,\text{exprId})$

Action(AssignModulusArrayVariable(x,y,exprId))

PRECOND: $\exists \text{value ValueOf}(\text{exprId},\text{value})$

EFFECT: when $\exists \text{varId,arrayed,keyId,exprId}$

$(\text{HasVariableId}(\text{HasElement}(\text{arrayId},\text{keyId}),\text{varId})$

$\wedge \text{HasKeyExpression}(\text{keyId},\text{exprId})$

$\wedge \text{ValueOf}(\text{exprId},y)$

$\wedge \text{HasArrayName}(\text{arrayId},'x')$

$\wedge \text{HasValue}(\text{varId},\text{value2}) \wedge \text{Modulus}(\text{value2},\text{value},\text{value1})$

$\wedge \text{CurrentScope}(\text{funcId}) \wedge \text{HasVariableScope}(\text{variableId},\text{funcId})):$

$\text{HasValue}(\text{varId},_) \leftarrow \text{HasValue}(\text{varId},\text{value1})$

Appendix D HTML Grammar

```
grammar html;

options {
    language = C;
    output = AST;
    ASTLabelType=pANTLR3_BASE_TREE;
}

tokens{
    DOCUMENT;
    HEAD;
    TITLE;
    BODY;
    HEADING;
    OLIST;
    ULIST;
    DLIST;
    DLITEM;
    TABLE;
    TROW;
    LINK;
    FORM;
    INPUTC;
    SELECT;
    OPTION;
    TEXTAREA;
    BUTTON;
    TEXT;
    ATTRIB;
    CC='>';
    CSINGLE='>';
    DQ='\"';
    EQ='=';
    PHP='PHP';
}

@parser::members
{
#include <string.h>

    ANTLR3_MARKER start;
    pANTLR3_INT_STREAM inputst;

    //check if string contains only attributes allowable for the relevant
tag
int retattr(char *mystring, char *attributes[],int len)
{
    char *token;
    char attr[100];
    int num,i,found;

    mystring=mystring+1;
    token=strtok(mystring,"=");
    found=1;
}
```

```

while(found==1 && token!=NULL)
{
    i=0;
    num=0;
    while(num==0 && i<len)
    {
        strcpy(attr,attributes[i]);
        if(strcmp(token,attr)==0)
            num=1;
        i++;
    }
    if(num==0)
    {
        found=0;
    }
    else
    {
        token=strtok(NULL," ");
        token=strtok(NULL,"=");
    }
}
return num;
}

void handleAttributes(char *text,char *attributes[],phtmlParser ctx,int
size)
{
    int t;
    pANTLR3_COMMON_TOKEN temp;
    SQLHANDLE stmt;
    SQLCHAR *query;
    char errortext[50];
    char sql[100];

    t=retattr(text,attributes,size);
    //If unrecognized attributes, generate a new syntax error
    if(t==0)
    {
        errcount++;
        temp=LT(-2);
        strcpy(errortext,temp->getText(temp)->chars);
        storeSyntaxError(temp->getLine(temp),temp-
>getCharPositionInLine(temp),ATTRIB_ERROR,temp-
>getText(temp),"",PARSER_ERR);
    }
}

//Main HTML Document
document
@before
{
    ind1=0;
    ind2=0;
}
: OHTML headstring? bodystring CHTML ->^(DOCUMENT headstring?
bodystring)
| headstring? bodystring ->^(DOCUMENT headstring? bodystring);

```

```

headstring
  : OHEAD headcontent* CHEAD ->^(HEAD headcontent*);

headcontent
  : title
  | block;

title : OTITLE text CTITLE ->^(TITLE text);

//Body elements
bodystring : phpbody? obody heading? block* CBODY -> phpbody? ^(BODY
obody heading? block*)
  | heading? block* -> ^(BODY heading? block*);
obody: OBODY STRING? CC
  {
    if($STRING!=NULL)
    {
      //Allowable attributes for obody
      char
*allowedattr[]={ "id", "style", "class", "onclick", "ondblclick", "onload", "onun
load", "onmousedown", "onmouseup", "onkeydown", "onkeypress", "onkeyup" };
      int len=sizeof(allowedattr)/sizeof(*allowedattr);
      handleAttributes($STRING.text->chars, allowedattr, ctx, len);
    }
  } ->^(ATTRIB STRING)?;

//Headings
heading : (h1|h2|h3|h4|h5|h6)+;
h1 : OH1 text* CH1 ->^(HEADING text*);
h2 : OH2 text* CH2 ->^(HEADING text*);
h3 : OH3 text* CH3 ->^(HEADING text*);
h4 : OH4 text* CH4 ->^(HEADING text*);
h5 : OH5 text* CH5 ->^(HEADING text*);
h6 : OH6 text* CH6 ->^(HEADING text*);

//Ordered list
olist : OOLIST litem+ COLIST ->^(OLIST litem+);
ulist : OULIST litem+ CULIST ->^(ULIST litem+);
litem : OLITEM text+ CLITEM -> text+;

dlist : ODLIST dlitem+ CDLIST ->^(DLIST dlitem+);
dterm : ODTERM text+ CDTERM ->text+ ;
ddef : ODDEF text+ CDDEF ->text+ ;
dlitem : dterm ddef ->^(DLITEM dterm ddef);

//Tables
table : otable (thead)? (tfoot)? (tbody) CTABLE ->^(TABLE otable thead?
tfoot? tbody);
otable: OTABLE STRING? CC
  {
    if($STRING!=NULL)
    {
      //Allowable attributes for otable
      char
*allowedattr[]={ "id", "border", "cellpadding", "cellspacing", "frame", "width",
"style", "class", "onclick", "ondblclick", "onmousedown", "onmouseup", "onkeydow
n", "onkeypress", "onkeyup" };

```

```

        int len=sizeof(allowedattr)/sizeof(*allowedattr);
        handleAttributes($STRING.text->chars,allowedattr,ctx,len);
    }
} ->^(ATTRIB STRING)?;

thead : othead trow* CTHEAD ->othead trow*;
othead: OTHEAD STRING? CC
{
    if($STRING!=NULL)
    {
        //Allowable attributes for othead
        char
*allowedattr[]={ "id","align","valign","style","class","onclick","ondblcl
ic k","onfocus","onmousedown","onmouseup","onkeydown","onkeypress","onkeyu
p"}
;
        int len=sizeof(allowedattr)/sizeof(*allowedattr);
        handleAttributes($STRING.text->chars,allowedattr,ctx,len);
    }
} ->^(ATTRIB STRING)?;
tfoot : otfoot trow* CTFOOT ->otfoot trow*;
tfoot: OTFOOT STRING? CC
{
    if($STRING!=NULL)
    {
        //Allowable attributes for otfoot
        char
*allowedattr[]={ "id","align","valign","style","class","onclick","ondblcl
ic k","onfocus","onmousedown","onmouseup","onkeydown","onkeypress","onkeyu
p"}
;
        int len=sizeof(allowedattr)/sizeof(*allowedattr);
        handleAttributes($STRING.text->chars,allowedattr,ctx,len);
    }
} ->^(ATTRIB STRING)?;
tbody : otbody trow* CTBODY ->otbody trow*
|trow* ->trow*;
tbody: OTBODY STRING? CC
{
    if($STRING!=NULL)
    {
        //Allowable attributes for otbody
        char
*allowedattr[]={ "id","align","valign","style","class","onclick","ondblcl
ic k","onfocus","onmousedown","onmouseup","onkeydown","onkeypress","onkeyu
p"}
;
        int len=sizeof(allowedattr)/sizeof(*allowedattr);
        handleAttributes($STRING.text->chars,allowedattr,ctx,len);
    }
} ->^(ATTRIB STRING)?;
trow : otrow trowcontent* CTROW ->(TROW otrow trowcontent*) ;
otrow : OTROW STRING? CC
{
    if($STRING!=NULL)
    {
        //Allowable attributes for otrow
        char
*allowedattr[]={ "id","align","valign","style","class","onclick","ondblcl
ic k","onfocus","onmousedown","onmouseup","onkeydown","onkeypress","onkeyu
p"}
;
        int len=sizeof(allowedattr)/sizeof(*allowedattr);

```



```

        handleAttributes($STRING.text->chars,allowedattr,ctx,len);
    }
} ->^(ATTRIB STRING)?;
trowcontent
: rcell
| hcell;
rcell : otcell block CTCELL -> otcell block;
otcell: OTCELL STRING? CC
{
    if($STRING!=NULL)
    {
        //Allowable attributes for otcell
        char
*allowedattr[]={"id","align","colspan","rowspan","width","valign","style",
"class","onclick","ondblclick","onfocus","onmousedown","onmouseup","onkeyd
own","onkeypress","onkeyup"};
        int len=sizeof(allowedattr)/sizeof(*allowedattr);
        handleAttributes($STRING.text->chars,allowedattr,ctx,len);
    }
} ->^(ATTRIB STRING)?;
hcell : othcell block CTHCELL -> othcell block;
othcell: OTHCELL STRING? CC
{
    if($STRING!=NULL)
    {
        //Allowable attributes for othcell
        char
*allowedattr[]={"id","align","colspan","rowspan","width","valign","style",
"class","onclick","ondblclick","onfocus","onmousedown","onmouseup","onkeyd
own","onkeypress","onkeyup"};
        int len=sizeof(allowedattr)/sizeof(*allowedattr);
        handleAttributes($STRING.text->chars,allowedattr,ctx,len);
    }
} ->^(ATTRIB STRING)?;

//Text formatting
abbr : OABBR forttext* CABBR ->forttext*;
acrn : OACRY forttext* CACRY ->forttext*;
addr : OADDR forttext* CADDR ->forttext*;
bold : OBOLD forttext* CBOLD ->forttext*;
big : OBIG forttext* CBIG ->forttext*;
bquote : OBQUOTE forttext* CBQUOTE ->forttext*;
cite : OCITE forttext* CCITE ->forttext*;
code : OCODE forttext* CCODE ->forttext*;
dfn : ODFN forttext* CDFN ->forttext*;
em : OEM forttext* CEM ->forttext*;
itl : OITL forttext* CITL ->forttext*;
kbd : OKBD forttext* CKBD ->forttext*;
quote : OQUOT forttext* CQUOT ->forttext*;
smallt : OSMALL forttext* CSMALL ->forttext*;
strong : OSTRONG forttext* CSTRONG ->forttext*;
sub : OSUB forttext* CSUB ->forttext*;
sup : OSUP forttext* CSUP ->forttext*;
tt : OTT forttext* CTT ->forttext*;
pre : OPRE forttext* CPRE->forttext*;
com : OCOM forttext* CCOM->;

//Hyperlinks
link : olink text? CA ->^(LINK olink text?);

```

```

olink
  : OA STRING? CC
  {
    if($STRING!=NULL)
    {
      //Allowable attributes for olink
      char
*allowedattr[]={ "id", "name", "href", "target", "rel", "style", "class", "onclick",
"ondblclick", "onfocus", "onmousedown", "onmouseup", "onkeydown", "onkeypress",
"onkeyup" };
      int len=sizeof(allowedattr)/sizeof(*allowedattr);
      handleAttributes($STRING.text->chars, allowedattr, ctx, len);
    }
  } ->^(ATTRIB STRING)?;

//Forms
form : oform formcontent* CFORM ->^(FORM oform formcontent*);
oform : OFORM STRING? CC
  {
    if($STRING!=NULL)
    {
      //Allowable attributes for oform
      char
*allowedattr[]={ "id", "name", "action", "target", "method", "style", "class", "on
click", "ondblclick", "onreset", "onsubmit", "onmousedown", "onmouseup", "onkeyd
own", "onkeypress", "onkeyup" };
      int len=sizeof(allowedattr)/sizeof(*allowedattr);
      handleAttributes($STRING.text->chars, allowedattr, ctx, len);
    }
  } ->^(ATTRIB STRING)?;
formcontent
  : ielement|text;
ielement: label? (input|selectlist|button|textarea);
input : oinput CINPUT? ->^(INPUTC oinput);
oinput : ((b=OINPUT a=STRING? CC)|(OINPUT a=STRING? CSINGLE))
  {
    if($a!=NULL)
    {
      //Allowable attributes for oinput
      char
*allowedattr[]={ "id", "name", "type", "value", "align", "size", "checked", "disab
led", "maxlength", "readonly", "src", "tabindex", "style", "class", "onclick", "on
dblclick", "onfocus", "onmousedown", "onmouseup", "onkeydown", "onkeypress", "on
keyup", "onselect", "onchange" };
      int len=sizeof(allowedattr)/sizeof(*allowedattr);
      handleAttributes($a.text->chars, allowedattr, ctx, len);
    }
  } ->^(ATTRIB STRING)?;

option : option text* COPTION ->^(OPTION option text*);
option : OOPTION STRING? CC
  {
    if($STRING!=NULL)
    {
      //Allowable attributes for option
      char
*allowedattr[]={ "id", "disabled", "selected", "value", "label", "style", "class",
"onclick", "ondblclick", "onmousedown", "onmouseup", "onkeydown", "onkeypress",
"onkeyup" };

```

```

        int len=sizeof(allowedattr)/sizeof(*allowedattr);
        handleAttributes($STRING.text->chars,allowedattr,ctx,len);
    }
} ->^(ATTRIB STRING)?;
optgroup : (oopgroup option* COPTGROUP) -> oopgroup option*;
oopgroup: OOPTGROUP STRING? CC
{
    if($STRING!=NULL)
    {
        //Allowable attributes for ooptgroup
        char
*allowedattr[]={ "id","disabled","label","style","class","onclick","ondblcl
ick","onmousedown","onmouseup","onkeydown","onkeypress","onkeyup"};
        int len=sizeof(allowedattr)/sizeof(*allowedattr);
        handleAttributes($STRING.text->chars,allowedattr,ctx,len);
    }
} ->^(ATTRIB STRING)?;
selectlist : oselect opcontent* CSELECT ->^(SELECT oselect opcontent*);
oselect : OSELECT STRING? CC
{
    if($STRING!=NULL)
    {
        //Allowable attributes for oselect
        char
*allowedattr[]={ "id","name","disabled","multiple","size","style","class","
onclick","ondblclick","onmousedown","onmouseup","onkeydown","onkeypress","
onkeyup","onchange"};
        int len=sizeof(allowedattr)/sizeof(*allowedattr);
        handleAttributes($STRING.text->chars,allowedattr,ctx,len);
    }
} ->^(ATTRIB STRING)?;
opcontent
: ogroup|option;
button : obutton text* CBUTTON ->^(BUTTON obutton);
obutton : OBUTTON STRING? CC
{
    if($STRING!=NULL)
    {
        //Allowable attributes for obutton
        char
*allowedattr[]={ "id","name","disabled","type","value","style","class","onc
lick","ondblclick","onmousedown","onmouseup","onkeydown","onkeypress","onk
eyup","onfocus"};
        int len=sizeof(allowedattr)/sizeof(*allowedattr);
        handleAttributes($STRING.text->chars,allowedattr,ctx,len);
    }
} ->^(ATTRIB STRING)?;
textarea: otext text* CTEXT ->^(TEXTAREA otext text*);
otext : OTEXT STRING? CC
{
    if($STRING!=NULL)
    {
        //Allowable attributes for otextarea
        char
*allowedattr[]={ "id","name","cols","rows","disabled","readonly","style","c
lass","onclick","ondblclick","onmousedown","onmouseup","onkeydown","onkeyp
ress","onkeyup","onfocus","onselect"};
        int len=sizeof(allowedattr)/sizeof(*allowedattr);
        handleAttributes($STRING.text->chars,allowedattr,ctx,len);
    }
}

```

```

    }
    } ->^(ATTRIB STRING)?;
label : olabel text* CLABEL;
olabel : OLABEL STRING? CC
    {
        if($STRING!=NULL)
        {
            //Allowable attributes for olabel
            char
*allowedattr[]={ "id", "for", "style", "class", "onclick", "ondblclick", "onmouse
down", "onmouseup", "onkeydown", "onkeypress", "onkeyup", "onfocus" };
            int len=sizeof(allowedattr)/sizeof(*allowedattr);
            handleAttributes($STRING.text->chars, allowedattr, ctx, len);
        }
    } ->^(ATTRIB STRING)?;

//General text rule
block:(olist|ulist|dlist|table|link|form|HR|com|phpbody|para|text);
para : opara text+ CPARA ;
opara : OPARA STRING? CC
    {
        if($STRING!=NULL)
        {
            //Allowable attributes for opara
            char
*allowedattr[]={ "id", "align", "style", "class", "onclick", "ondblclick", "onmou
sedown", "onmouseup", "onkeydown", "onkeypress", "onkeyup" };
            int len=sizeof(allowedattr)/sizeof(*allowedattr);
            handleAttributes($STRING.text->chars, allowedattr, ctx, len);
        }
    } ->^(ATTRIB STRING)?;
fortext :
(abbr|acrn|addr|bold|big|bquote|cite|code|dfn|em|it1|kbd|quote|smallt|stro
ng|sub|sup|tt|pre|btext|BR);
text: fortext ->^(TEXT fortext);
btext: STRING;

//attribstring:(STRING|phpbody)*;

//php
phpbody
@after
{
ind2++;
}
:PHP ->^(PHP {myphptree[ind2]});

//Main HTML Document

OHTML
: '<HTML>' | '<html>';
CHTML: '</HTML>' | '</html>';
OHEAD
: '<HEAD>' | '<head>';
CHEAD
: '</HEAD>' | '</head>';

```

```

OBODY
  : '<BODY'|'<body>';
CBODY
  : '</BODY>'|'</body>';

//Header rules

OTITLE : '<TITLE>'|'<title>';
CTITLE : '</TITLE>'|'</title>';

//Headings
OH1 : '<H1>'|'<h1>';
OH2 : '<H2>'|'<h2>';
OH3 : '<H3>'|'<h3>';
OH4 : '<H4>'|'<h4>';
OH5 : '<H5>'|'<h5>';
OH6 : '<H6>'|'<h6>';

CH1 : '</H1>'|'</h1>';
CH2 : '</H2>'|'</h2>';
CH3 : '</H3>'|'</h3>';
CH4 : '</H4>'|'</h4>';
CH5 : '</H5>'|'</h5>';
CH6 : '</H6>'|'</h6>';

//Paragraphs and lists
OPARA : '<P>'|'<p>';
CPARA : '</P>'|'</p>';

OOLIST : '<OL>'|'<ol>';
OULIST : '<UL>'|'<ul>';
ODLIST : '<DL>'|'<dl>';

COLIST : '</OL>'|'</ol>';
CULIST : '</UL>'|'</ul>';
CDLIST : '</DL>'|'</dl>';

OLITEM : '<LI>'|'<li>';
CLITEM : '</LI>'|'</li>';
ODTERM : '<DT>'|'<dt>';
CDTERM : '</DT>'|'</dt>';
ODDEF : '<DD>'|'<dd>';
CDDEF : '</DD>'|'</dd>';

//Tables
OTABLE : '<TABLE>'|'<table>';
OTHEAD : '<THEAD>'|'<thead>';
OTHCELL : '<TH>'|'<th>';
OTROW : '<TR>'|'<tr>';
OTCELL : '<TD>'|'<td>';
OTFOOT : '<TFOOT>'|'<tfoot>';
OTBODY : '<TBODY>'|'<tbody>';

CTABLE : '</TABLE>'|'</table>';
CTHEAD : '</THEAD>'|'</thead>';
CTHCELL : '</TH>'|'</th>';
CTROW : '</TR>'|'</tr>';

```

```

CTCELL : '</TD>'|'</td>';
CTFOOT : '</TFOOT>'|'</tfoot>';
CTBODY : '</TBODY>'|'</tbody>';

//Text formatting
OABBR : '<ABBR>'|'<abbr>';
OACRY : '<ACRONYM>'|'<acronym>';
OADDR : '<ADDRESS>'|'<address>';
OBOLD : '<B>'|'<b>';
OBIG : '<BIG>'|'<big>';
OBQUOTE : '<BLOCKQUOTE>'|'<blockquote>';
OCITE : '<CITE>'|'<cite>';
OCODE : '<CODE>'|'<code>';
ODFN : '<DFN>'|'<dfn>';
OEM : '<EM>'|'<em>';
OITL : '<I>'|'<i>';
OKBD : '<KDB>'|'<kbd>';
OQUOT : '<Q>'|'<q>';
OSMALL : '<SMALL>'|'<small>';
OSTRONG : '<STRONG>'|'<strong>';
OSUB : '<SUB>'|'<sub>';
OSUP : '<SUP>'|'<sup>';
OTT : '<TT>'|'<tt>';
OPRE : '<PRE>'|'<pre>';

CABBR : '</ABBR>'|'</abbr>';
CACRY : '</ACRONYM>'|'</acronym>';
CADDR : '</ADDRESS>'|'</address>';
CBOLD : '</B>'|'</b>';
CBIG : '</BIG>'|'</big>';
CBQUOTE : '</BLOCKQUOTE>'|'</blockquote>';
CCITE : '</CITE>'|'</cite>';
CCODE : '</CODE>'|'</code>';
CDFN : '</DFN>'|'</dfn>';
CEM : '</EM>'|'</em>';
CITL : '</I>'|'</i>';
CKBD : '</KDB>'|'</kbd>';
CQUOT : '</Q>'|'</q>';
CSMALL : '</SMALL>'|'</small>';
CSTRONG : '</STRONG>'|'</strong>';
CSUB : '</SUB>'|'</sub>';
CSUP : '</SUP>'|'</sup>';
CTT : '</TT>'|'</tt>';
CPRE : '</PRE>'|'</pre>';

//Lines and Comments
HR : '<HR>'|'<hr>';
BR : '<BR>'|'<br>';
OCOM : '<!-->';
CCOM : '-->';

//Hyperlinks
OA : '<A>'|'<a>';
CA : '</A>'|'</a>';

//Forms
OFORM : '<FORM>'|'<form>';
OINPUT : '<INPUT>'|'<input>';
OSELECT : '<SELECT>'|'<select>';

```

```

OOPTGROUP
  : '<OPTGROUP'|'<optgroup';
OOPTION : '<OPTION'|'<option';
OBUTTON : '<BUTTON'|'<button';
OTEXT  : '<TEXTAREA'|'<textarea';
OLABEL  : '<LABEL'|'<label';

CFORM : '</FORM>'|'</form>';
CINPUT: '</INPUT>'|'</input>';
CSELECT : '</SELECT>'|'</select>';
COPTGROUP
  : '</OPTGROUP>'|'</optgroup>';
COPTION : '</OPTION>'|'</option>';
CBUTTON : '</BUTTON>'|'</button>';
CTEXT  : '</TEXTAREA>'|'</textarea>';
CLABEL  : '</LABEL>'|'</label>';

//Begin PHP
BEGIN
@declarations
{
  pANTLR3_COMMON_TOKEN mytoken;
  pANTLR3_INPUT_STREAM  in;
  pANTLR3_STRING input_string;
}
: '<?php'{
  callPhp(INPUT);
  $channel=PHP_CHANNEL;
  input_string = (pANTLR3_STRING)"PHP";
  in = antlr3NewAsciiStringInPlaceStream(input_string,
strlen(input_string), "input text stream");
  PUSHSTREAM(in);
};

//DEFINITIONS

WS      : (' '|'\t'|\n'|\r')+{$channel=HIDDEN;};

STRING
  : (~('<'|'>'|\r'|\n'|'/'))+;

fragment
WORD  : LETTER+;

fragment
LETTER : ('a'..'z')|('A'..'Z');

```

Appendix E

Examples of Analysis of Selection Structures

Exercise 1

Exercise : Write a program to set the value of variable y to 1 if variable x is greater than 10 and to 0 otherwise.

Initial State: *HasName(VarId1,'x')*
 HasValue(VarId1,val_x)
 HasInitialValue(VarId1,val_x)

Goal: (*GreaterThan(val_x,10) → HasValue(VARID2,0)*)
 \wedge (*LessThanOrEqualTo(val_x,10) → HasValue(VARID2,1)*)

Constraints: *HasName(VARID2,'y')*

Solution 1a

```
if($x<=10)
{
    $y=1;
}
else
{
    $y=0;
}
```

AST :

```
(DOCUMENT (BODY (PHP (If (<= ($ x) 10) (= ($ y) 1) (= ($ y) 0))))))
```

Analysis:

The first part of the AST to be analysed is the if structure. Let the *BooleanExpression* created at this point have id ExprId1. Let the ids of the *VariableExpr* and the *LiteralExpr* that make up the *BooleanExpression* be VarExprId1 and LitExprId1 respectively. Let the id of the created literal be LitId1. Then, the following facts are created.

HasId(LessEqualExpr(VarExprId1,LitExprId1),ExprId1)

HasVariable(VarExprId1,VarId1)

HasLiteral(LitExprId1,LitId1)

HasLitValue(LitId1,10)

Finding the value of these expressions as explained in Section 4.4.1.1 results in the following facts being created.

ValueOf(VarExprId1,val_x)

ValueOf(LitExprId1,10)

Considering the section of the AST where the condition is true, the following fact is created.

ValueOf(ExprId1,True)

This fact translates to the following fact using the rules defined in *Figure 5.4*.

LessThanOrEqualTo(val_x,10)

When this condition is satisfied, the assign action comes into effect resulting in the following set of facts.

HasName(VarId2,'y')

HasValue(VarId2,1)

HasInitialValue(VarId2,1)

Since this only happens when the if condition is satisfied and therefore, the *LessThanOrEqualTo(val_x,10)* fact is true, the state of the program can now be written as below.

LessThanOrEqualTo(val_x,10) → HasValue(VarId2,1)

Similarly, considering the else part of the if statement, the *BooleanExpression* is false resulting in the following fact.

ValueOf(ExprId1,False)

This fact translates to the following fact using the rules in *Figure 5.4*.

GreaterThan(val_x,10)

The next part of the if statement is satisfied when this condition is met, resulting in the following facts.

HasName(VarId2,'y')

HasValue(VarId2,0)

HasInitialValue(VarId2,0)

Since this only happens when the *BooleanExpression* is false, it can be written as an implication as below.

GreaterThan(val_x,10) → HasValue(VarId2,0)

Therefore, the final state of the program contains the following facts.

HasName(VarId2,y)

\wedge (*LessThanOrEqual(val_x,10) → HasValue(VarId2,1)*)

\wedge (*GreaterThan(val_x,10) → HasValue(VarId2,0)*)

Although the order is different, it can be seen that this is identical to the overall goal of the exercise when VARID2=VarId2. Therefore, this program is identified as correct.

Solution 1b

```
if($x>10)
{
    $y=1;
}
else
{
    $y=0;
}
```

AST :

(DOCUMENT (BODY (PHP (If (> (\$ x) 10) (= (\$ y) 1) (= (\$ y) 0))))))

Analysis:

The first part of the AST to be analysed is the if structure. Let the *BooleanExpression* created at this point have id ExprId1. Let the ids of the *VariableExpr* and the *LiteralExpr* that make up the *BooleanExpression* be

VarExprId1 and LitExprId1 respective. Let the id of the created literal be LitId1. Then, the following facts are created.

HasId(GreaterExpr(VarExprId1,LitExprId1),ExprId1)

HasVariable(VarExprId1,VarId1)

HasLiteral(LitExprId1,LitId1)

HasLitValue(LitId1,10)

Finding the value of these expressions as explained in Section 4.4.1.1 results in the following facts being created.

ValueOf(VarExprId1,val_x)

ValueOf(LitExprId1,10)

Considering the section of the AST where the condition is true, the following fact is created.

ValueOf(ExprId1,True)

This fact translates to the following fact using the rules defined in *Figure 5.4*.

GreaterThan(val_x,10)

When this condition is satisfied, the assign action comes into effect resulting in the following set of facts.

HasName(VarId2,'y')

HasValue(VarId2,1)

HasInitialValue(VarId2,1)

Since this only happens when the if condition is satisfied and therefore, the *LessThanOrEqualTo(val_x,10)* fact is true, the state of the program can now be written as below.

GreaterThan(val_x,10) → HasValue(VarId2,1)

Similarly, considering the else part of the if statement, the *BooleanExpression* is False resulting in the following fact.

ValueOf(ExprId1,False)

This fact translates to the following fact using the rules in *Figure 5.4*.

LessThanOrEqual(val_x,10)

The next part of the if statement is satisfied when this condition is met, resulting in the following facts.

HasName(VarId2,'y')

HasValue(VarId2,0)

HasInitialValue(VarId2,0)

Since this only happens when the *BooleanExpression* is false, it can be written as an implication as below.

LessThanOrEqual (val_x,10) → HasValue(VarId2,0)

Therefore, the final state of the program contains the following facts.

HasName(VarId2,y)

\wedge (*GreaterThan(val_x,10) → HasValue(VarId2,1)*)

\wedge (*LessThanOrEqual(val_x,10) → HasValue(VarId2,0)*)

When comparing this final state against the overall goal, it can be seen that the goal is not met since the values of the variables are assigned for the wrong conditions. Therefore, this program is identified as incorrect.

Solution 1c

```
if($x>10)
{
    $y=1;
}
```

AST :

```
(DOCUMENT (BODY (PHP (If (> ($ x) 10) (= ($ y) 1) ))))
```

Analysis:

The first part of the AST to be analysed is the if structure. Let the *BooleanExpression* created at this point have id ExprId1. Let the ids of the *VariableExpr* and the *LiteralExpr* that make up the *BooleanExpression* be

VarExprId1 and LitExprId1 respective. Let the id of the created literal be LitId1. Then, the following facts are created.

HasId(GreaterExpr(VarExprId1,LitExprId1),ExprId1)

HasVariable(VarExprId1,VarId1)

HasLiteral(LitExprId1,LitId1)

HasLitValue(LitId1,10)

Finding the value of these expressions as explained in Section 4.4.1.1 results in the following facts being created.

ValueOf(VarExprId1,val_x)

ValueOf(LitExprId1,10)

Considering the section of the AST where the condition is true, the following fact is created.

ValueOf(ExprId1,True)

This fact translates to the following fact using the rules defined in *Figure 5.4*.

GreaterThan(val_x,10)

When this condition is satisfied, the assign action comes into effect resulting in the following set of facts.

HasName(VarId2,'y')

HasValue(VarId2,1)

HasInitialValue(VarId2,1)

Since this only happens when the if condition is satisfied and therefore, the *LessThanOrEqualTo(val_x,10)* fact is true, the state of the program can now be written as below.

GreaterThan(val_x,10) → HasValue(VarId2,1)

This program does not contain an else part and therefore, nothing happens when the condition is not satisfied. Therefore, the final state of the program contains the following facts.

$HasName(VarId2,y)$

$\wedge (GreaterThan(val_x,10) \rightarrow HasValue(VarId2,1))$

When comparing this final state against the overall goal, it can be seen that the goal is not met since only part of the necessary implication conditions are present. Therefore, this program is identified as incorrect.

Exercise 2

Exercise : Write a PHP program to display 'A' if \$marks is greater than 80.

Otherwise, if \$marks is greater than 50, display 'B'. Display 'F' in all other instances. Note that when execution reaches the point where the code has to be completed, the variable \$marks already contains a value.

Initial State: $HasName(VarId1,'marks')$

$HasValue(VarId1,val_m)$

$HasInitialValue(VarId1,val_m)$

Suggested Goal: $(GreaterThan(val_m,80) \rightarrow OnPage('A',i))$

$\wedge (LessThanOrEqual(val_m,80) \rightarrow$

$(GreaterThan(val_m,50) \rightarrow OnPage('B',j))$

$\wedge (LessThanOrEqual(val_m,50) \rightarrow OnPage('F',k)))$

Goal: $(GreaterThan(val_m,80) \rightarrow OnPage('A',i))$

$\wedge (LessThanOrEqual(val_m,80) \wedge GreaterThan(val_m,50) \rightarrow OnPage('B',j))$

$(LessThanOrEqual(val_m,50) \rightarrow OnPage('F',k))$

Solution 2a

```
if($marks<=50)
{
    echo('F');
}
else if ($marks<=80)
{
    echo('B');
}
else
```

```

{
    echo('A');
}

```

AST :

```

(DOCUMENT (BODY (PHP (If (<= ($ marks) 50) (echo 'F') ((If (<= ($
marks) 80) (echo 'B') ((echo 'A'))))))))

```

Analysis:

The first part of the AST to be analysed is the first if structure. Let the *BooleanExpression* created at this point have id ExprId1. Let the ids of the *VariableExpr* and the *LiteralExpr* that make up the *BooleanExpression* be VarExprId1 and LitExprId1 respective. Let the id of the created literal be LitId1. Then, the following facts are created.

HasId(LessEqualExpr(VarExprId1,LitExprId1),ExprId1)

HasVariable(VarExprId1,VarId1)

HasLiteral(LitExprId1,LitId1)

HasLitValue(LitId1,50)

Finding the value of these expressions as explained in Section 4.4.1.1 results in the following facts being created.

ValueOf(VarExprId1,val_m)

ValueOf(LitExprId1,50)

When considering the case when the condition is satisfied, the following fact is created.

ValueOf(ExprId1,True)

This fact results in the following fact being created using the rules in *Figure 5.4*.

LessThanOrEqual(val_m,50)

When this condition is satisfied, an ‘echo’ statement is executed. This results in the *Display* action being used to create the following fact.

OnPage('F',1)

So the entire state for when the condition is satisfied can be written as below.

$$\text{LessThanOrEqual}(\text{val_m}, 50) \rightarrow \text{OnPage}('F', 1)$$

When the condition is not satisfied, i.e. in the else section, the following fact is created.

$$\text{ValueOf}(\text{ExprId1}, \text{False})$$

Again using the rules in *Figure 5.4*, the following fact is then created in the system for the case where the condition is not satisfied.

$$\text{GreaterThanOr}(\text{val_m}, 50)$$

At this point, another selection structure is encountered. This means that whatever facts are created after this are implied by the above fact. The condition for this second selection structure results in the following set of facts being created. Let the ids of the relevant *BooleanExpression*, *VarExpr* and *LitExpr* be ExprId2, VarExprId2 and LitExprId2 respectively. Let the id of the created *Literal* be LitId2.

$$\text{HasId}(\text{LessEqualExpr}(\text{VarExprId2}, \text{LitExprId2}), \text{ExprId2})$$
$$\text{HasVariable}(\text{VarExprId2}, \text{VarId2})$$
$$\text{HasLiteral}(\text{LitExprId2}, \text{LitId2})$$
$$\text{HasLitValue}(\text{LitId2}, 80)$$

Finding the value of these expressions as explained in Section 4.4.1.1 results in the following facts being created.

$$\text{ValueOf}(\text{VarExprId2}, \text{val_m})$$
$$\text{ValueOf}(\text{LitExprId2}, 80)$$

When this second condition is satisfied the ValueOf the expression is set to True and this results in a comparison fact being created using the rules in *Figure 5.4*. This means that the following facts are created.

$$\text{ValueOf}(\text{ExprId2}, \text{True})$$
$$\text{LessThanOrEqual}(\text{val_m}, 80)$$

When the second condition is satisfied, a *Display* action is again used to create the following fact.

$OnPage('B',2)$

So the result of the second condition being true can be written as below.

$LessThanOrEqual(val_m,80) \rightarrow OnPage('B',1)$

When the second condition is not satisfied, the *Display* action is used to create the following facts.

$ValueOf(ExprId2,False)$

$GreaterThan(val_m,80)$

For this situation, the *Display* action results in the following fact.

$OnPage('A',3)$

So the state when the second condition is not satisfied is as below.

$GreaterThan(val_m,80) \rightarrow OnPage('A',3)$

Using the above description, it can be seen that the entire state for the second condition is as below.

$(LessThanOrEqual(val_m,80) \rightarrow OnPage('B',2))$

$\wedge (GreaterThan(val_m,80) \rightarrow OnPage('A',3))$

But as described earlier, the second condition is only satisfied if the first one is not so this entire state is an implication of when the first condition is not satisfied. Therefore, the final state of this program is as below.

$(LessThanOrEqual(val_m,50) \rightarrow OnPage('A',1))$

$\wedge (GreaterThan(val_m,50) \rightarrow (LessThanOrEqual(val_m,80) \rightarrow OnPage('B',2)))$

$\wedge (GreaterThan(val_m,80) \rightarrow OnPage('F',3))$

This final state does not satisfied either the suggested goal or the overall goal above so the program is identified as incorrect even though it achieves the objective.

Solution 2b

```
if($marks>80)
{
    echo('A');
}
if($marks<=80 && $marks>50)
```

```

{
    echo('B');
}
if($marks<=50)
{
    echo('F');
}

```

Analysis:

The first part of the AST to be analysed is the first if structure. Let the *BooleanExpression* created at this point have id ExprId1. Let the ids of the *VariableExpr* and the *LiteralExpr* that make up the *BooleanExpression* be VarExprId1 and LitExprId1 respectively. Let the id of the created literal be LitId1. Then, the following facts are created.

HasId(GreaterExpr(VarExprId1,LitExprId1),ExprId1)
HasVariable(VarExprId1,VarId1)
HasLiteral(LitExprId1,LitId1)
HasLitValue(LitId1,80)

Finding the value of these expressions as explained in Section 4.4.1.1 results in the following facts being created.

ValueOf(VarExprId1,val_m)
ValueOf(LitExprId1,80)

When considering the case when the condition is satisfied, the following fact is created.

ValueOf(ExprId1,True)

This fact results in the following fact being created using the rules in *Figure 5.4*.

GreaterThan(val_m,80)

When this condition is satisfied, an ‘echo’ statement is executed. This results in the *Display* action being used to create the following fact.

OnPage('A',1)

So the entire state for when the condition is satisfied can be written as below.

GreaterThan(val_m,80) → OnPage('A',1)

The next section of the AST to be analysed is the second if condition. Here, the *BooleanExpression* is an *AndExpr*. Let the id of this be ExprId2. Let the id of the *LessEqualExpr* on the left hand side of this be ExprId3 and the id of the *GreaterExpr* on the right hand side be ExprId4. Then, the following facts are created.

HasId(AndExpr(ExprId3,ExprId4),ExprId2)

Let the ids of the *VariableExpr* on the left hand side of the *LessEqualExpr* be VarExprId3 and the id of the *LiteralExpr* on the right hand side be LitExprId3. Then, the following facts are created.

HasId(LessEqualExpr(VarExprId3,LitExprId3),ExprId3)

HasVariable(VarExprId3,VarId1)

HasLiteral(LitExprId3,LitId1)

Finding the value of these expressions as explained in Section 4.4.1.1 results in the following facts being created.

ValueOf(VarExprId3,val_m)

ValueOf(LitExprId3,80)

Similarly, let the ids of the *VariableExpr* on the left hand side of the *GreaterExpr* be VarExprId4 and the id of the *LiteralExpr* on the right hand side be LitExprId4. Let the id of the created *Literal* be LitId2. Then, the following facts are created.

HasId(GreaterExpr(VarExprId4,LitExprId4),ExprId4)

HasVariable(VarExprId4,VarId1)

HasLiteral(LitExprId4,LitId2)

HasLitValue(LitId2,50)

Finding the value of these expressions as explained in Section 4.4.1.1 results in the following facts being created.

ValueOf(VarExprId4,val_m)

ValueOf(LitExprId4,50)

When the second condition is satisfied, the following predicate is created.

ValueOf(ExprId2,True)

The rules in *Figure 5.12*, results in the following facts being created.

ValueOf(ExprId3,True)

ValueOf(ExprId3,True)

These facts result in the following fact being created using the rules in *Figure 5.4*.

LessThanOrEqual(val_m,80)

GreaterThan(val_m,50)

When this condition is satisfied, an ‘echo’ statement is executed. This results in the *Display* action being used to create the following fact.

OnPage('B',2)

So the entire state for when the condition is satisfied can be written as below.

LessThanOrEqual(val_m,80) \wedge GreaterThan(val_m,50) \rightarrow OnPage('B',2)

The final section of the AST to be analysed is the first if structure. Let the *BooleanExpression* created at this point have id ExprId5. Let the ids of the *VariableExpr* and the *LiteralExpr* that make up the *BooleanExpression* be VarExprId5 and LitExprId5 respective.

HasId(LessEqualExpr(VarExprId5,LitExprId5),ExprId5)

HasVariable(VarExprId5,VarId1)

HasLiteral(LitExprId5,LitId2)

Finding the value of these expressions as explained in Section 4.4.1.1 results in the following facts being created.

ValueOf(VarExprId5,val_m)

ValueOf(LitExprId5,50)

When considering the case when the condition is satisfied, the following fact is created.

ValueOf(ExprId5,True)

This fact results in the following fact being created using the rules in *Figure 5.4*.

LessThanOrEqualTo(val_m,50)

When this condition is satisfied, an ‘echo’ statement is executed. This results in the *Display* action being used to create the following fact.

OnPage('F',3)

So the entire state for when the condition is satisfied can be written as below.

LessThanOrEqualTo(val_m,50) → OnPage('F',3)

So the final state of the program is as below.

(GreaterThanOrEqualTo(val_m,80) → OnPage('A',1))

∧ (LessThanOrEqualTo(val_m,80) ∧ GreaterThanOrEqualTo(val_m,50) → OnPage('B',2))

∧ (LessThanOrEqualTo(val_m,50) → OnPage('F',3))

Therefore, the overall goal is satisfied when $i=1$, $j=2$ and $k=3$ so the program is identified as correct.

Exercise 3

Exercise : Write a PHP program to display ‘Excellent’ if the grade is ‘A’. Otherwise, if the grade is ‘B’ display ‘Good’. In all other instances display ‘Try Harder’. Note that when execution reaches the point where the code has to be completed, the variable \$grade already contains a value.

Initial State: *HasName(VarId1,'grade')*

HasValue(VarId1,val_g)

HasInitialValue(VarId1,val_g)

Goal: *(EqualTo(val_g,'A') → OnPage('Excellent',i))*
∧ (EqualTo(val_g,'B') → OnPage('Good',j))

$\wedge (\text{NotEqualTo}(\text{val_g}, 'A') \wedge \text{NotEqualTo}(\text{val_g}, 'B')) \rightarrow \text{OnPage}(\text{'Try Harder'}, k)$

Solution 3a

```
switch($grade)
{
    case 'A': echo('Excellent');
              break;
    case 'B': echo('Good');
              break;
    default: echo('Try Harder');
}

```

AST :

(PHP (switch (\$ grade) (case 'A' (echo 'Excellent') break) (case 'B' (echo 'Good') break) (default (echo 'Try Harder'))))

Analysis:

The first part of the AST to be analysed is the switch structure. When this structure is encountered, the switch variable \$grade is noted. When the first case statement is encountered a new AST is created for the conditional expression as below.

(== (\$ grade) 'A')

Now, this is processed as an if structure with this as the condition and the second node of the 'case' node as what to do when the condition is satisfied.

Let the *BooleanExpression* created at this point have id ExprId1. Let the ids of the *VariableExpr* and the *LiteralExpr* that make up the *BooleanExpression* be VarExprId1 and LitExprId1 respectively. Let the id of the created literal be LitId1. Then, the following facts are created.

HasId(EqualExpr(VarExprId1,LitExprId1),ExprId1)

HasVariable(VarExprId1,VarId1)

HasLiteral(LitExprId1,LitId1)

HasLitValue(LitId1,'A')

Finding the value of these expressions as explained in Section 4.4.1.1 results in the following facts being created.

ValueOf(VarExprId1, val_g)

ValueOf(LitExprId1, 'A')

Considering the section of the AST where the condition is true, the following fact is created.

ValueOf(ExprId1, True)

This fact translates to the following fact using the rules defined in *Figure 5.4*.

EqualTo(val_g, 'A')

When the second condition is satisfied, a *Display* action is again used to create the following fact.

OnPage('Excellent', 1)

So the result of the first condition being true can be written as below.

EqualTo(val_g, 'A') → OnPage('Excellent', 1)

Similarly the second case node results in the following new AST for the conditional expression.

(== (\$ grade) 'B')

Let the *BooleanExpression* created at this point have id *ExprId2*. Let the ids of the *VariableExpr* and the *LiteralExpr* that make up the *BooleanExpression* be *VarExprId2* and *LitExprId2* respective. Let the id of the created literal be *LitId2*. Then, the following facts are created.

HasId(EqualExpr(VarExprId2, LitExprId2), ExprId2)

HasVariable(VarExprId2, VarId2)

HasLiteral(LitExprId2, LitId2)

HasLitValue(LitId2, 'B')

Finding the value of these expressions as explained in Section 4.4.1.1 results in the following facts being created.

ValueOf(VarExprId1, val_g)

ValueOf(LitExprId1, 'B')

Considering the section of the AST where the condition is true, the following fact is created.

$ValueOf(ExprId2, True)$

This fact translates to the following fact using the rules defined in *Figure 5.4*.

$EqualTo(val_g, 'B')$

When the second condition is satisfied, a *Display* action is again used to create the following fact.

$OnPage('Good', 2)$

So the result of the second condition being true can be written as below.

$EqualTo(val_g, 'B') \rightarrow OnPage('Good', 2)$

The default statement results in all conditions for previous expressions being false so the following facts are created.

$ValueOf(ExprId1, False)$

$ValueOf(ExprId2, False)$

These facts translates to the following facts using the rules defined in *Figure 5.4*.

$NotEqualTo(val_g, 'A')$

$NotEqualTo(val_g, 'B')$

In the default case, a *Display* action is again used to create the following fact.

$OnPage('Try Harder', 3)$

So the result of the default section can be written as below.

$NotEqualTo(val_g, 'A') \wedge NotEqualTo(val_g, 'B') \rightarrow OnPage('Try Harder', 3)$

So the final state of the program is given below.

$(EqualTo(val_g, 'A') \rightarrow OnPage('Excellent', 1))$

$\wedge (EqualTo(val_g, 'B') \rightarrow OnPage('Good', 2))$

$\wedge (NotEqualTo(val_g, 'A') \wedge NotEqualTo(val_g, 'B') \rightarrow OnPage('Try Harder', 3))$

So the overall goal is satisfied when $i=1$, $j=2$ and $k=3$ and the program is identified as correct.

Appendix F

Examples for Analysis of Functions and Forms

Example 1

Consider the following PHP function

```
function display()
{
    echo($_POST['num']);
}
```

When analysing this function, let the id of the *Function* be *FuncId1*. Then, the *CurrentScope* is set as below.

CurrentScope(FuncId1)

When the `$_POST` array is encountered, an array with this name is created. Let the id of the *Array* be *ArrId1*. Since `$_POST` is a super-global array, the following facts are created.

HasArrayName(ArrId1, '\$_POST')

HasArrayScope(ArrId1, Null)

Global('\$_POST', FuncId1)

Now, the third rule in *Figure 6.9* is used to create the following fact.

HasArrayScope(ArrId1, FuncId1)

An *ArrayVariable* corresponding to the array element is also created at this point. Let the id of the *ArrayVariable* be *ArrId1*. Let the id of the corresponding *Key* be *KeyId1*. Let the *LiteralExpression* corresponding to the *Key* have an id of *LitExprId1* and the created *Literal* have an id of *LitId1*. Then, the following facts are created.

HasVariableId(HasElement(ArrId1, KeyId1), VarId1)

HasKeyExpression(KeyId1, LitExprId1)

HasLiteral(LitExprId1, LitId1)

HasLiteral(LitId1, 'num')

Next, the first rule in Figure 6.9 is used to find the scope of the ArrayVariable resulting in the following fact.

HasVariableScope(VarId1,FuncId1)

Now, the array element of the super-global array is in scope within the function and therefore, can be accessed within it.

Example 2

Exercise : Write a PHP function called `displayMotto` that displays the text ‘We are the best!’.

Goal : `FunctionOK(FUNCID1)`

Constraints : `HasFunctionName(FUNCID1,'displayMotto')`

Condition of Subplan(`FunctionOK(FUNCID1)`):

PRECOND :

POSTCOND: `OnPage('We are the best',i)`

Solution

```
function displayMotto()  
{  
    echo("We are the best!");  
}
```

Analysis:

The function definition is the first node of the AST to be processed and results in the following facts. Let the id of the created *Function* be `FuncId1`.

CurrentScope(FuncId1)

HasFunctionName(FuncId1,'findTotal')

Now, a check is made to see whether the preconditions of any of the sub-plans are satisfied. Since the conditions of the sub-plan has no precondition, it is automatically satisfied.

Next, the statements within the function are processed. The ‘echo’ node results in the *Display* action being activated, resulting in the following predicate.

OnPage('We are the Best!',1)

Since all nodes within the function definition have now been analysed, a check is carried out to see whether the post-conditions of the sub-plan are satisfied. It can be seen that the post-condition is satisfied when $i=1$. This results in the following fact being created.

FunctionOK(FuncId1)

This is the final state of the program. When comparing this against goal, it is satisfied when $FUNCID1=FuncId1$. When comparing this state against the constraints, these are also satisfied so the program is identified as correct.

Example 3

Exercise : Write a PHP function called `globAdd` that adds the value passed in as a parameter to the value of the global variable `$x` and returns the result. Note that when execution reaches the point where the code has to be completed, the variable `$x` already contains a value.

Initial State: *CurrentScope(Null)*
 HasName(VarId1,'x')
 HasValue(VarId1,val_x)
 HasInitialValue(VarId1,val_x)
 HasVariableScope(VarId1,Null)

Goal : *FunctionOK(FUNCID1)*

Constraints : *HasFunctionName(FUNCID1,'globAdd')*

Conditions of Subplan(*FunctionOK(FUNCID1)*):
 PRECOND : *HasParameter(FUNCID1,1,VARID1)*
 \wedge *HasValue(VARID1, VALUEa)*
 POSTCOND: *Add(VALUEa, val_x,VALUEc)*
 \wedge *HasReturnExpression(FUNCID1, RETEXPRID1)*
 \wedge *ValueOf(RETEXPRID1,VALUEc)*

Solution

```
function globAdd($num)
{
    global $x;
    $tot=$num+$x;
    return($tot);
}
```

Analysis:

The function definition is the first node of the AST to be processed and results in the following facts.

CurrentScope(FuncId1)

HasFunctionName(FuncId1,'globAdd')

HasParameter(FuncId1,1,ParamVarId1)

HasName(ParamVarId1,'num')

Since the *ParameterVariables* are only in scope within the function, a new fact is created to indicate this.

HasVariableScope(ParamVarId1,FuncId1)

Assigning values to the *ParameterVariables* as described in Section 6.2.3.2 results in the following fact.

HasValue(ParamVarId1,'num')

Now a check is made to see whether the preconditions of a sub-plan are satisfied. In this case, it is satisfied when $FUNCID1=FuncId1$, $VARID1=ParamVarId1$ and $VALUEa='num'$.

Next the AST nodes corresponding to the statements within the function definition are analysed. The 'global' definition results in the following fact.

Global('x',FuncId1)

Using the process described in Section 6.2.2.2, the following fact is created.

HasVariableScope(VarId1,FuncId1)

The next node corresponds to an assign statement with an *AddExpr* on the right hand side. Let the id of the *AddExpr* be *ExprId1* and the values of the *VarExprs* on either side of this expression be *VarExprId1* and *VarExprId2* respectively. Then, the following facts are created.

HasId(AddExpr(VarExprId1,VarExprId2),ExprId1)

HasVariable(VarExprId1,ParamVarId1)

HasVariable(VarExprId2,VarId1)

The *ValueOf* each of these sub-expressions is then found using the rules in *Figure 6.11*.

ValueOf(VarExprId1,'num')

ValueOf(VarExprId2,val_x)

The *ValueOf* the *AddExpr* is next found using the rules in *Figure 4.8*. Let the sum of 'num' and val_x be tot so *Add('num',val_x,tot)*.

ValueOf(ExprId1,tot)

The value of this is assigned to a new variable, \$tot and the following facts are created as given in the *Assign* action in *Figure 6.12*. Let the id of the newly created *Variable* be *VarId1*.

HasName(VarId2,'tot')

HasValue(VarId2,tot)

HasInitialValue(VarId2,tot)

HasVariableScope(VarId2,FuncId1)

Next, the AST node corresponding to the return expression is analysed. Here, the return expression is actually a *VarExpr* returning the \$tot variable. This is used together with the rules to find the *ValueOf* the expression to create the following facts.

HasReturnExpression(FuncId1,RetExprId1)

HasVariable(RetExprId1,VarId2)

ValueOf(RetExprId1,tot)

Now, a check is made to see if the post-condition of the sub-plan is satisfied. It can be seen that this is satisfied when $RETEXPRID1=RetExprId1$ and $VALUEc=tot$. Therefore, the following fact is created.

FunctionOK(FuncId1)

This is the final state of the system. It can be seen now that the overall goal is satisfied when $FUNCID1=FuncId1$ so the program is identified as correct.

Appendix G Examples for Analysis of Loops

Example 1

Goal : $\forall j [(1 \leq j \leq 5) \rightarrow \{ \text{OnPage}(\text{"Hello"}, Y) \}]$

Constraints : For(FORID1)
 \wedge LoopBodyOK(FORID1)

Conditions of Subplan(FunctionOK(FORID1)):

PRECOND :

POSTCOND: OnPage("Hello",x)

Solution

```
$i=1;
while($i<=5)
{
    echo("Hello");
    $x++;
}
```

Analysis:

The first assignment statement results in a new variable named VarId1 being created and assigned a value of 1, resulting in the following facts.

HasName(VarId1, 'i')

HasInitialValue(VarId1, 1)

HasValue(VarId1, 1)

Now, a *while* loop is encountered. It is first checked to see whether it has a condition with a *BooleanExpression* that is valid for a *for* loop. In this case, it is a *LessEqualExpr* with a *VariableExpr* on the left hand side and a *LiteralExpr* on the right hand side so it corresponds to the expression in a *for* loop. Also, the *VariableExpr* in the condition refers to the variable \$i, which already has a value, as it should in a *for* loop.

Next, the statements within the loop are analysed to see whether the variable \$i is updated within the loop so that it updates during every instance. Since no

selection expressions are found within the loop, all the statements within it are executed at all times. There is a statement $\$i++$, which updates the variable within the loop. Therefore, this while loop is identified as similar to a for loop with a loop variable of $\$i$, resulting in the following facts being created. Let the id of the *While* loop be *WhileId1*, the id of the Variable $\$i$ be *VarId1* and the id of the *LessEqualExpr* be *ExprId1*. Let the ids of the *VariableExpr* and the *LiteralExpr* on either side of the *LessEqualExpr* be *VarExprId1* and *LitExprId1* respectively. Let the id of the corresponding *Literal* be *LitId1*.

HasName(VarId1,'i')

HasValue(VarId1,1)

HasInitialValue(VarId1,1)

HasLoopVariable(WhileId1,VarId1)

HasForStartValue(WhileId1,1)

HasId(LessEqualExpr(VarExprId1,LitExprId1),ExprId1)

HasVariable(VarExprId1,VarId1)

HasLiteral(LitExprId1,LitId1)

HasLitValue(LitId1,5)

HasLoopCondition(WhileId1,ExprId1)

Using the rules in *Figure 4.8*, the *ValueOf* the *LiteralExpr* is found, resulting in the following fact.

ValueOf(LitExprId1,5)

Using the rules in *Figure 7.5*, the following fact is created.

HasForEndValue(WhileId1,5)

Based on the analysis of the first iteration of the *while* loop to determine if it corresponds to a *for* loop, the following fact is obtained.

HasValue(VarId1,2)

Since this is the value of the loop variable at the end of the first iteration, the following fact is created.

HasForFirstLoopValue(WhileId1,2)

Next, the rule in *Figure 7.6* is activated, resulting in the following fact.

HasForIncrement(WhileId1,1)

Now, the loop itself needs to be analysed. The effects of the overall loop can be written as below.

repeat(WhileActionEffects,WhileId1)

Now, the conditions of the sub-plan needs to be analysed. Let the value of \$i at the beginning of each iteration be *val_i*. Then, the following facts are created.

HasValue(VarId1,val_i)

HasIterationValue(ForId1,VarId1,val_i)

Since the conditions of the sub-plan have no pre-conditions, they are automatically satisfied. Now, the statements within the loop need to be analysed. The first statement is an echo statement which results in a *Display* action. The following fact is created as a result of this action.

OnPage("Hello",1)

Next, the variable \$i is incremented from its current value. The relevant *AssignAdd* action results in the following fact.

HasValue(VarId1,val_j) where *Add(val_i,1,val_j)*

This is the state of the program at the end of execution of the rule. When comparing against the conditions of the sub-plan, it can be seen that it is satisfied when $x=1$. Therefore, the following fact is created.

LoopBodyOK(WhileId1)

Now, the rules in *Figure 7.10* are activated to create the following facts.

RepeatLoop(WhileId1,1,5,1)

RepeaAll(WhileId1,1,5)

$\forall val_i [(1 \leq val_i \leq 5) \rightarrow OnPage("Hello",count)]$

The resultant state is the final state of the system. When comparing this against the overall goal, it can be seen that they are satisfied when

FORID1=WhileId1, j=val_i and Y=count. Therefore, this program is identified as correct.

Exercise 2

Exercise : Write a program segment to store the result of the multiplication of two variables \$a and \$b into a new variable. Use the definition of multiplication as a result of repeated addition to use a for loop to perform the calculation. Note that when execution reaches the point where the code needs to be completed, the variables \$a and \$b already contain a value.

Initial State: *HasName(VarId1, 'a')*
 HasValue(VarId1, val_a)
 HasInitialValue(VarId1, val_a)
 HasName(VarId2, 'b')
 HasValue(VarId2, val_b)
 HasInitialValue(VarId2, val_b)

Goal:
 Multiply(VALUE_a, VALUE_b, VALUE_m)
 \wedge HasValue(VARID_m, VALUE_m)

Constraints:
 ForLoop(FORID1)
 \wedge LoopBodyOK(FORID1)

Conditions of Subplan1(LoopBodyOK(FORID1),
 PRECOND : HasValue(VARID_m, VALUE_ms)
 \wedge Add(VALUE_ms, VALUE_a, VALUE_me)
 POSTCOND: HasValue(VARID_m, VALUE_me))

Conditions of Subplan2(LoopBodyOK(FORID1),
 PRECOND : HasValue(VARID_m, VALUE_ms)
 \wedge Add(VALUE_ms, VALUE_b, VALUE_me)
 POSTCOND: HasValue(VARID_m, VALUE_me))

Solution 2a

```
$multiply=0;
for($i=1;$i<=$b;$i++)
{
    $multiply+=$a;
}
```

Analysis:

The first assignment statement results in a new variable named `VarId3` being created and assigned a value of 0, resulting in the following facts.

HasName(VarId3,'multiply')

HasInitialValue(VarId3,0)

HasValue(VarId3,0)

The following facts are created as a result of the for loop as described in Section 7.2.1. Let the id of the variable `$i` be `VarId4`. Let the id of the *LessEqualExpr* be `ExprId1`. Also, let the *VariableExprs* on either side of this expression have ids `VarExprId1` and `VarExprId2` respectively.

HasName(VarId4,'i')

HasValue(VarId4,1)

HasInitialValue(VarId4,1)

HasLoopVariable(ForId1,VarId4)

HasForStartValue(ForId1,1)

HasId(LessEqualExpr(VarExprId1,VarExprId2),ExprId1)

HasVariable(VarExprId1,VarId4)

HasVariable(VarExprId2,VarId2)

HasLoopCondition(ForId1,ExprId1)

Using the rules in *Figure 4.8*, the *ValueOf* the `VarExprId2` is found, resulting in the following fact.

ValueOf(VarExprId2,val_b)

Using the rule in *Figure 7.5*, the end value of the loop is found as below.

HasForEndValue(ForId1, val_b)

Next, it is necessary to find the value of the counter variable at the end of the first iteration. The post-increment operator results in an *AssignAdd* action which creates the following fact.

HasValue(VarId4, 2)

Since this is the value of the loop variable at the end of the first iteration, the following fact is created.

HasForFirstLoopValue(ForId1, 2)

Next, the rule in *Figure 7.6* is activated, resulting in the following fact.

HasForIncrement(ForId1, 1)

Now, the actual loop has to be analysed. The repetition of the loop can be written as below.

repeat(ForActionEffects, ForId1)

Only two variables, \$i and \$multiply change their value during the loop so it is only necessary to consider initial values for these two variables for each iteration of the loop. Let the initial values be val_i and val_m respectively. Then, the following facts are created.

HasValue(VarId4, val_i)

HasIterationValue(ForId1, VarId4, val_i)

HasValue(VarId3, val_m)

HasIterationValue(ForId1, VarId3, val_m)

It can be seen that at this point, the pre-conditions of both sub-plans are satisfied. Next, the actions performed by the loop have to be analysed. Here, it is an assignment statement resulting in a *AddAssign* action being activated, resulting in the following fact.

HasValue(VarId3, val_new) where *Add(val_m, val_a, val_new)*

It can be seen that the post-condition of the first sub-plan is now satisfied when VALUE_me=val_new, so the following fact is created.

LoopBodyOK(ForId1)

Next, the rules in *Figure 7.7* are executed to consolidate the actions performed by the loop, resulting in the following facts.

RepeatLoop(ForId1,1,val_b,1)

RepeatAll(ForId1,1,val_b)

In this case, the *ActionEffects* is the result of the assignment which is the *HasValue(VarId3,val_new)* fact so the consolidated effect is as below.

$\forall val_i [(1 \leq val_i \leq val_b) \rightarrow HasValue(VarId3, val_new)]$

Next, the rule in *Figure 7.17* is activated, resulting in the following fact.

HasValue(VarId3, val_mul) where *Multiply(val_a, val_b, val_mul)*

When comparing this final state against the overall goal, it can be seen that it is satisfied when *VALUE_m=val_mul*, *VARID_m=VarId3* and *FORID1=ForId1*. Therefore, the program segment is identified as correct.

Solution 2b

```
$multiply=0;
for($i=1;$i<=$a;$i++)
{
    $multiply+=$b;
}
```

Analysis:

The first assignment statement results in a new variable named *VarId3* being created and assigned a value of 0, resulting in the following facts.

HasName(VarId3,'multiply')

HasInitialValue(VarId3,0)

HasValue(VarId3,0)

The following facts are created as a result of the for loop as described in Section 7.2.1. Let the id of the variable *\$i* be *VarId4*. Let the id of the *LessEqualExpr* be *ExprId1*. Also, let the *VariableExprs* on either side of this expression have ids *VarExprId1* and *VarExprId2* respectively.

HasName(VarId4, 'i')

HasValue(VarId4, 1)

HasInitialValue(VarId4, 1)

HasLoopVariable(ForId1, VarId4)

HasForStartValue(ForId1, 1)

HasId(LessEqualExpr(VarExprId1, VarExprId2), ExprId1)

HasVariable(VarExprId1, VarId4)

HasVariable(VarExprId2, VarId2)

HasLoopCondition(ForId1, ExprId1)

Using the rules in *Figure 4.8*, the *ValueOf* the *VarExprId2* is found, resulting in the following fact.

ValueOf(VarExprId2, val_a)

Using the rule in *Figure 7.5*, the end value of the loop is found as below.

HasForEndValue(ForId1, val_a)

Next, it is necessary to find the value of the counter variable at the end of the first iteration. The post-increment operator results in an *AssignAdd* action which creates the following fact.

HasValue(VarId4, 2)

Since this is the value of the loop variable at the end of the first iteration, the following fact is created.

HasForFirstLoopValue(ForId1, 2)

Next, the rule in *Figure 7.6* is activated, resulting in the following fact.

HasForIncrement(ForId1, 1)

Now, the actual loop has to be analysed. The repetition of the loop can be written as below.

repeat(ForActionEffects, ForId1)

Only two variables, *\$i* and *\$multiply* change their value during the loop so it is only necessary to consider initial values for these two variables for each iteration of

the loop. Let the initial values be val_i and val_m respectively. Then, the following facts are created.

HasValue(VarId4, val_i)

HasIterationValue(ForId1, VarId4, val_i)

HasValue(VarId3, val_m)

HasIterationValue(ForId1, VarId3, val_m)

It can be seen that at this point, the pre-conditions of both sub-plans are satisfied. Next, the actions performed by the loop have to be analysed. Here, it is an assignment statement resulting in a *AddAssign* action being activated, resulting in the following fact.

HasValue(VarId3, val_new) where *Add(val_m, val_b, val_new)*

It can be seen that the post-condition of the second sub-plan is now satisfied when $VALUE_me=val_new$, so the following fact is created.

LoopBodyOK(ForId1)

Next, the rules in *Figure 7.7* are executed to consolidate the actions performed by the loop, resulting in the following facts.

RepeatLoop(ForId1, 1, val_a, 1)

RepeatAll(ForId1, 1, val_a)

In this case, the *ActionEffects* is the result of the assignment which is the *HasValue(VarId3, val_new)* fact so the consolidated effect is as below.

$\forall val_i [(1 \leq val_i \leq val_a) \rightarrow HasValue(VarId3, val_new)]$

Next, the rule in *Figure 7.17* is activated, resulting in the following fact.

HasValue(VarId3, val_mul) where *Multiply(val_a, val_b, val_mul)*

When comparing this final state against the overall goal, it can be seen that it is satisfied when $VALUE_m=val_mul$, $VARID_m=VarId3$ and $FORID1=ForId1$. Therefore, the program segment is identified as correct.

Appendix H

Implementation Details

1. At certain times, it becomes necessary to manipulate the AST created by the grammar files (Section 4.6.2, Section 5.6). However, the position returned by the grammar file is used when highlighting syntax error nodes (Section 8.3.1.2). In order to maintain accurate position information, this information from the original node is copied on to any newly created nodes.
2. In order to analyse the program HTML attributes need to be converted into AST form. However, the HTML grammar file treats attribute nodes as simple text. The conversion to AST form is done during the AST walking process.
3. As mentioned in the description, the PHP grammar file used during program analysis is one that has been downloaded from the web (Section 4.5.2). This grammar file does not check to see whether a '\$' sign is present before variable names although it accepts variable names with a '\$' sign. Therefore, no syntax error is identified if no '\$' sign precedes a variable name. This problem is handled by manually checking for the '\$' sign in all places where it is expected and generating a syntax error.
4. Function calls can be used anywhere where expressions are expected. However, the syntax is only correct if the function is either a pre-defined function or it has been defined in the same program. This cannot be checked during parsing using the grammar files. This is also checked during the AST walking process and a syntax error is generated if an unacceptable function name is used.
5. Two types of array keys, keystings and indexes, have been modelled in the system (Section 6.1). However, there is no change in the program analysis, whatever the type of key. Therefore, although this distinction has been modelled in theory, it has been ignored during the actual system building for ease of implementation.
6. It is possible to infinitely convert from one expression type to another when converting between equivalent Boolean expressions as described in Section 5.3. Therefore, this conversion is also implemented using CLIPS functions which are executed at the time of goal checking.

7. As described in Section 6.2.4, predefined functions are handled by storing a definition. This definition contains a link to a CLIPS function that is executed when the predefined function is called. This function creates the predicates that result from executing the predefined function.
8. When handling function calls to user defined functions, the relevant facts are formed by creating the post-conditions of the selected sub-plan (Section 6.2.3.2). However, in reality, this is handled by calling a separate CLIPS function.

Appendix I Pre and Post Test

- Which of the following delimiter syntax is PHP's default delimiter syntax
 - `<?php ?>`
 - `<% %>`
 - `<? ?>`
 - `<script language="php"> </script>`
- The left association operator % is used in PHP for
 - percentage
 - bitwise or
 - division
 - modulus
- To produce the output "I love the summer time", which of the following statement can be used?
 - `<? php print ("<p> I love the summer time</p>");?>`
 - `<? php $season="summer time"; print"<p> I love the $season</p>"; ?>`
 - `<?php $message="<p> I love the summer time </p>"; echo $message; ?>`
 - All of above
- What will be displayed?

```
$var = 'a';  
$VAR = 'b';  
  
echo "$var$VAR";
```

 - aa
 - bb
 - ab
 - error
- A value that has no defined value is expressed in PHP with the following keyword:
 - undef
 - null
 - None
 - There is no such concept in PHP
- All variables in PHP start with which symbol?
 - !
 - \$
 - &
 - %
- Which of the following ways will add 1 to the variable \$count?
 - `$count++;`
 - `incr $count;`
 - `count++;`
 - `$count +=1`
- Which of the following is NOT a valid PHP comparison operator?
 - `!=`
 - `>=`
 - `<=>`
 - `<>`

9. What will be displayed?

```
if ('2' == '02') {  
    echo 'true';  
} else {  
    echo 'false';  
}
```

- a. true
- b. false

10. When the statement `$alive= 5;` is executed, and then `$alive` is tested as a boolean condition, e.g. `if($alive)`, then

- a. `$alive` is false
- b. `$alive` is true
- c. `$alive` is overflow
- d. the statement is not valid

11. Which of the following method sends input to a script where the input is displayed in the URL of the resultant page?

- a. Get
- b. Post
- c. Both
- d. None

12. How do we access the value of 'd' later?

```
$a = array(  
    'a',  
    3 => 'b',  
    1 => 'c',  
    'd');
```

- a. `a[0]`
- b. `a[4]`
- c. `a[3]`
- d. `a[2]`

13. What will be displayed by the code below?

```
<?php  
  
    FUNCTION TEST()  
    {  
        ECHO 'HELLO' . ' WORLD!\n';  
    }  
  
    test();  
?>
```

- a. HELLO WORLD!
- b. Nothing
- c. it's a compiler error
- d. hello world!

14. How do you get information from a form that is submitted using the "post" method?

- a. `$_POST[];`
- b. `Request.Form;`

- c. Request.QueryString;
- d. \$_GET[];

15. What value is displayed for "a" below?

```
<?php
    $a = 2;

    function Test($a)
    {
        echo "a = $a";
    }
    $a--;
    Test($a);
?>
```

- a. 1
- b. 2
- c. 3
- d. No value

16. Consider the following php webpage. Assume that this webpage is loaded into a browser and the user enters the text 'Hello' into the textbox and clicks the submit button. What will then be displayed on the web page?

```
<?php
if(isset($_POST['submit']))
{
    echo($_POST['mytext']);
}
else
{
?>
<form action="" method=post>
<input type=text name=mytext>
<input type=submit name=submit>
</form>
<?php
}
?>
```

- a. The text 'Hello' followed by a form containing a textbox and a submit button.
- b. A form containing a textbox and a submit button with the text 'Hello' inside the text box.
- c. Only the text 'Hello'.
- d. An empty form containing a textbox and a submit button

17. Which of the PHP code segments is equivalent to the code segment given below?

```
<?php
switch($a)
{
case 1:$b=$b+10;
    break;
case 2:$b=$b+5;
    break;
```

```
default:$b=$b+15;
}
?>
```

a. <?php
if(\$a=1)
 \$b+=10;
if(\$a=2)
 \$b+=5;
else
 \$b+=15;
?>

b. <?php
if(\$a=1)
 \$b=\$b+10;
if(\$a=2)
 \$b=\$b+5;
else
 \$b=\$b+15;
?>

c. <?php
if(\$a==1)
 \$b+=10;
else if(\$a==2)
 \$b+=5;
else
 \$b+=15;
?>

d. <?php
if(\$a==1)
 \$b=\$b+10;
else if(\$a==2)
 \$b=\$b+5;
else if(\$a==3)
 \$b=\$b+15;
?>

18. Consider the following PHP code segment. Which of the PHP code segments below will display the elements of the array in the given order?

```
$a[1]=' PHP' ;  
$a[2]=' Java' ;  
$a[3]=' C' ;
```

a. foreach(\$a as \$value)
{
 echo(\$value);
}

- b.

```
foreach($a as $value)
{
    echo($a);
}
```
- c.

```
foreach($a as $key=>$value)
{
    echo($key);
}
```
- d. None of the above

19. Which of the following statements is *incorrect* regarding PHP for loops?

- a. A for loop can always be converted to an equivalent while loop.
- b. For loops can be nested within each other.
- c. The 'for' keyword is followed by three expressions within a pair of brackets.
- d. The condition in a for loop (the second expression within the bracket) can never be blank.

Appendix J
Questionnaire

PHP Intelligent Tutoring System

Feedback Form

This feedback form is used to obtain feedback about the PHP Intelligent Tutoring System. Your answers will not be recorded against your username. They will be used solely for the purpose of improving the system for future users. Your support in submitting this feedback is highly appreciated.

Please rate your prior use of the following.

1. Programming in C (not C#)

- Never used it
- Very basic knowledge
- Good knowledge
- Very good knowledge
- Expert

2. Web development using HTML

- Never used it
- Very basic knowledge
- Good knowledge
- Very good knowledge
- Expert

3. PHP

- Never used it
- Very basic knowledge
- Good knowledge
- Very good knowledge
- Expert

4. Database Management Systems

- Never used it
- Very basic knowledge
- Good knowledge

- Very good knowledge
- Expert

5. MySQL

- Never used it
- Very basic knowledge
- Good knowledge
- Very good knowledge
- Expert

Please rate the following aspects of the system.

6. Overall impression of the system

- Excellent
- Good
- Neutral
- Poor
- Very poor

7. Ease of use

- Excellent
- Good
- Neutral
- Poor
- Very poor

8. Look and feel

- Excellent
- Good
- Neutral
- Poor
- Very poor

9. Programming exercises

- Excellent
- Good
- Neutral

- Poor
- Very poor

10. Feedback messages

- Excellent
- Good
- Neutral
- Poor
- Very poor

11. Success in gaining student knowledge and understanding

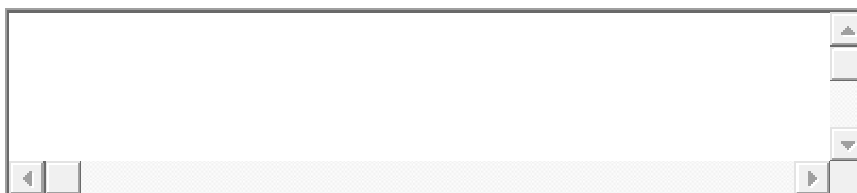
- Excellent
- Good
- Neutral
- Poor
- Very poor

12. Speed of response of the system

- Excellent
- Good
- Neutral
- Poor
- Very poor

Please give short descriptive answers to the following questions

13. How much time (in total across the semester) did you spend learning web development using the Intelligent Tutoring System?



14. Do you feel that your knowledge of dynamic web development using PHP improved as a result of using the system?

An empty rectangular text input field with a light gray background and a thin border. It features a vertical scrollbar on the right side and a horizontal scrollbar at the bottom.

15. Would you like to use a similar system again to gain better knowledge of the subject matter?

An empty rectangular text input field with a light gray background and a thin border. It features a vertical scrollbar on the right side and a horizontal scrollbar at the bottom.

16. Would you recommend the system be used by other students?

An empty rectangular text input field with a light gray background and a thin border. It features a vertical scrollbar on the right side and a horizontal scrollbar at the bottom.

17. Did you feel that the feedback provided by the system was helpful in understanding why your program was incorrect?

An empty rectangular text input field with a light gray background and a thin border. It features a vertical scrollbar on the right side and a horizontal scrollbar at the bottom.

18. Were you happy with the system's suggestions for the next programming exercise or did you often feel that you should try something else because the system's suggestion was inappropriate?

An empty rectangular text input field with a light gray background and a thin border. It features a vertical scrollbar on the right side and a horizontal scrollbar at the bottom.

19. Did you at any time feel that the system analysed your program incorrectly (i.e. it

accepted a solution you knew was wrong or rejected a solution you knew was correct)? If so, please provide more details.

An empty rectangular text input box with a light gray background and a thin black border. It features a vertical scrollbar on the right side and a horizontal scrollbar at the bottom, both with standard arrow and track icons.

20. What aspect of the user interface did you find most appealing?

An empty rectangular text input box with a light gray background and a thin black border. It features a vertical scrollbar on the right side and a horizontal scrollbar at the bottom, both with standard arrow and track icons.

21. What aspect of the user interface did you find least appealing?

An empty rectangular text input box with a light gray background and a thin black border. It features a vertical scrollbar on the right side and a horizontal scrollbar at the bottom, both with standard arrow and track icons.

22. What extra features would you most like to see added to the user interface?

An empty rectangular text input box with a light gray background and a thin black border. It features a vertical scrollbar on the right side and a horizontal scrollbar at the bottom, both with standard arrow and track icons.

23. Any other comments

An empty rectangular text input box with a light gray background and a thin black border. It features a vertical scrollbar on the right side and a horizontal scrollbar at the bottom, both with standard arrow and track icons.

Appendix K

Focus Group Questions

1. What aspects of the user interface should be changed to make the system more user-friendly?
2. How would you compare this system with any other online learning system that you have used?
3. Do you think the learning resources supplied with this system are used effectively to teach the subject of dynamic web development? Suggest areas of improvement.
4. Do you think that the exercises suggested by the system are useful in improving your knowledge?
5. What other improvements can you suggest to make the educational process more productive?

Appendix L Complete ORM Diagram

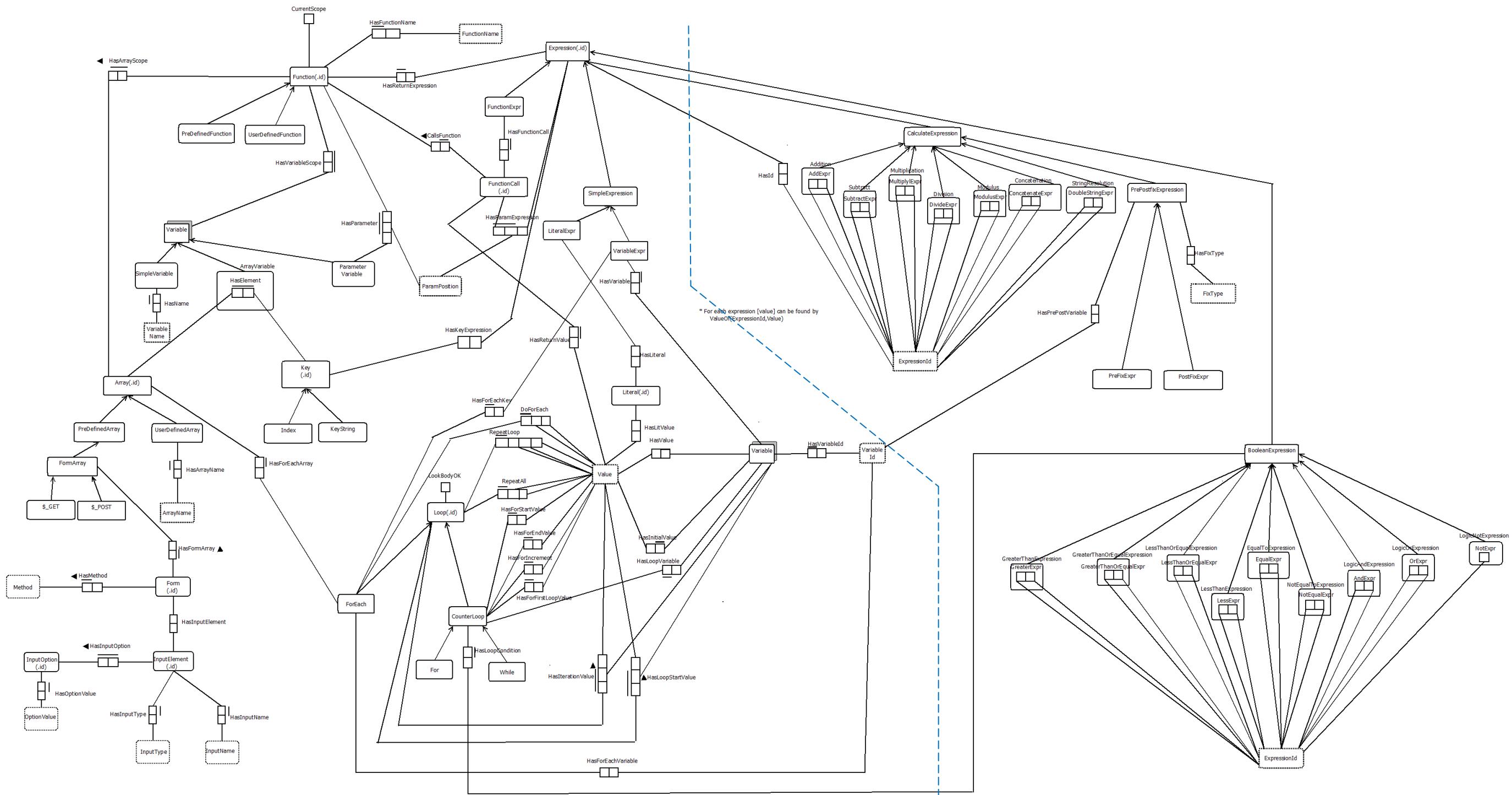


Figure L1. Complete ORM diagram.

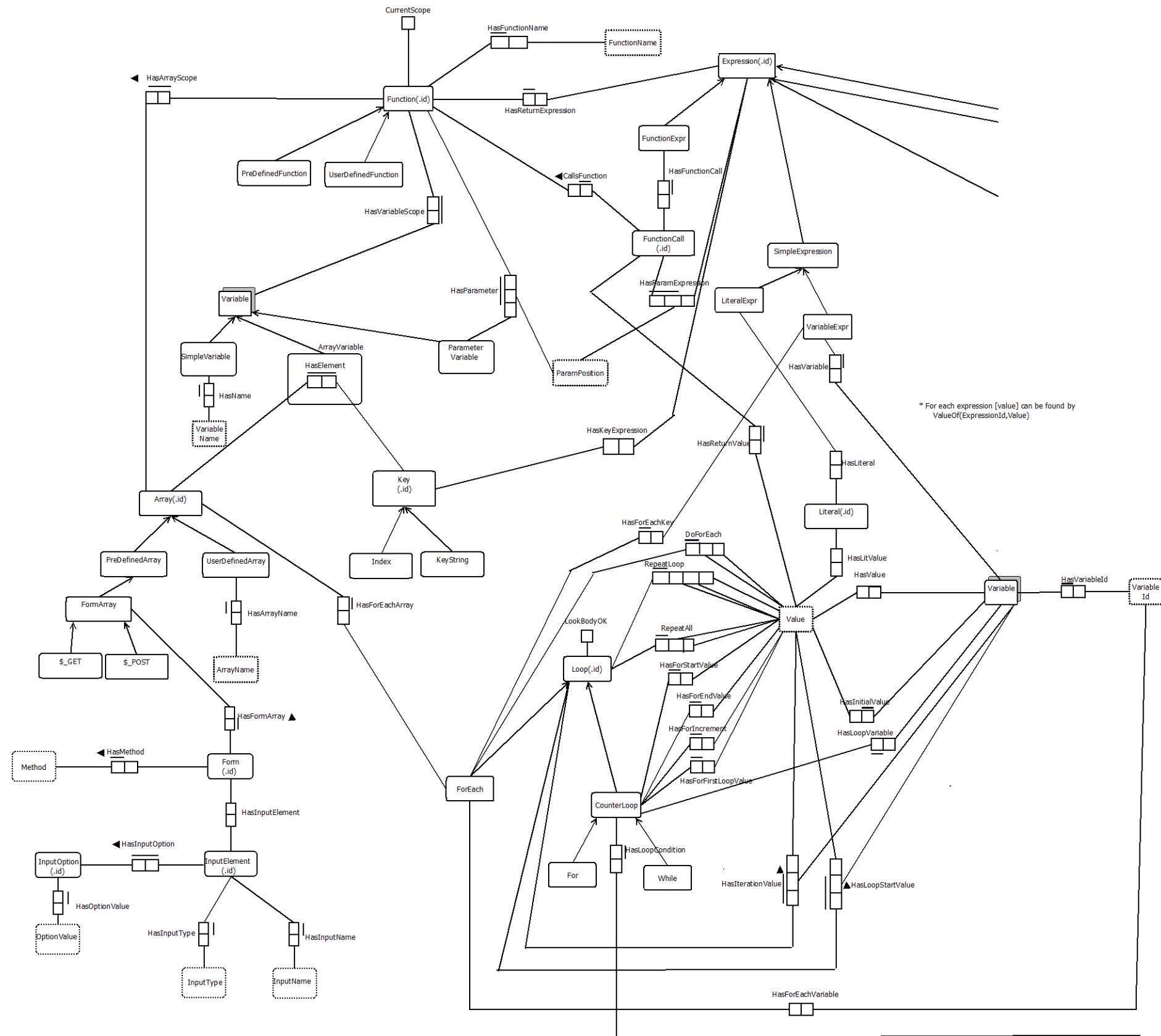


Figure L2. Complete ORM diagram – left half

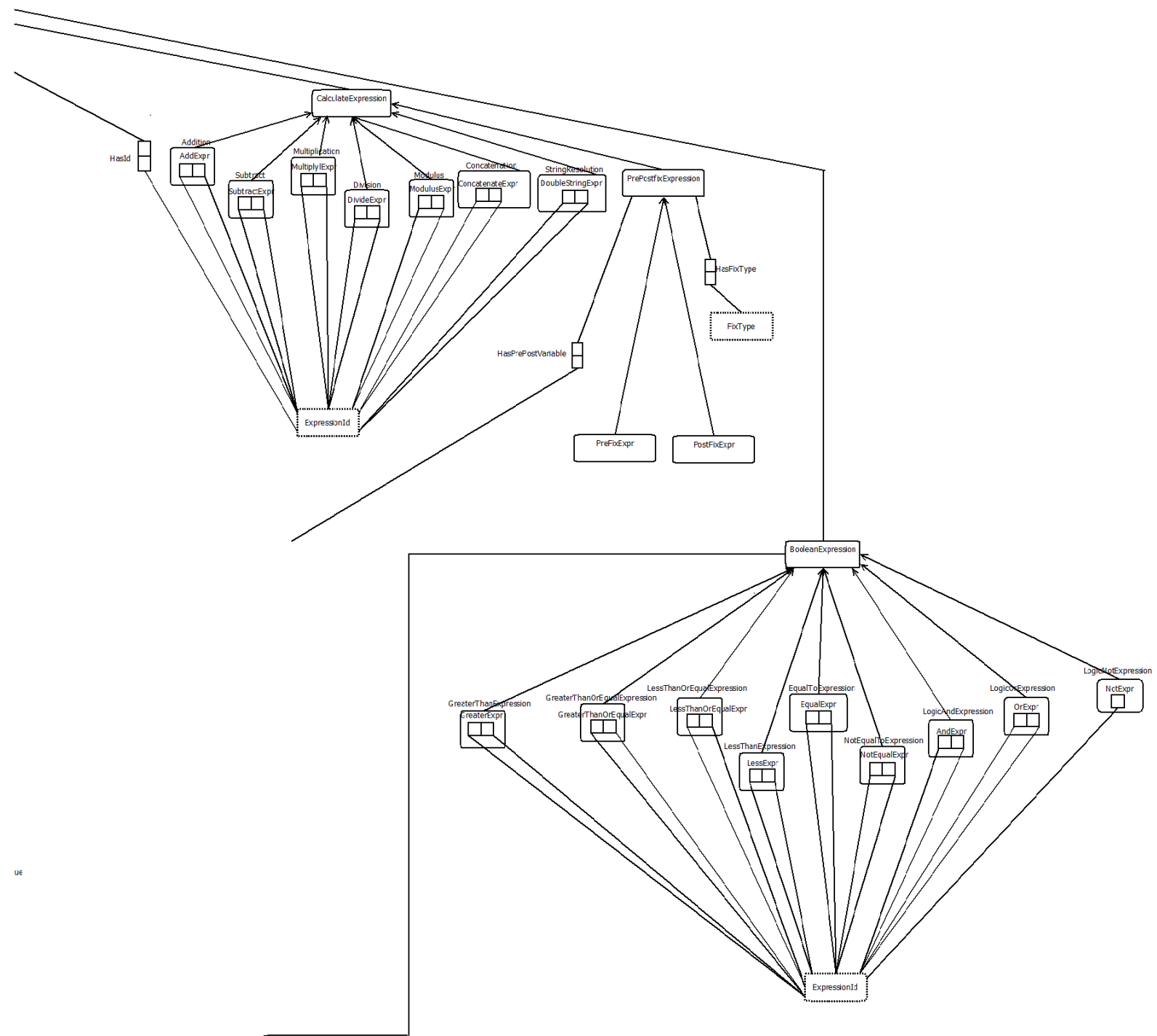


Figure L3. Complete ORM diagram – right half