Hint generation in programming tutors

A dissertation presented

by

Timotej Lazar

to

The Faculty of Computer and Information Science

in partial fulfilment of the requirements for the degree of

Doctor of Philosophy

in the subject of

Computer and Information Science

Ljubljana, 2018

The author has dedicated this work to the public domain by waiving all rights to the work worldwide under copyright law, including all related and neighboring rights, to the extent allowed by law. For details, see https://creativecommons.org/publicdomain/zero/1.0/legalcode.

APPROVAL

I hereby declare that this submission is my own work and that, to the best of my knowledge and belief, it contains no material previously published or written by another person nor material which to a substantial extent has been accepted for the award of any other degree or diploma of the university or other institute of higher learning, except where due acknowledgement has been made in the text.

— Timotej Lazar —
March 2018

The submission has been approved by

acad. dr. Ivan Bratko
Professor of Computer and Information Science
supervisor

dr. Igor Kononenko
Professor of Computer and Information Science
examiner

dr. Marko Robnik-Šikonja
Professor of Computer and Information Science
examiner

dr. Gerhard Friedrich
Professor of Computer and Information Science
external examiner
Alpen-Adria-Universität Klagenfurt

PREVIOUS PUBLICATION

I hereby declare that the research reported herein was previously published/submitted for publication in peer reviewed journals or publicly presented at the following occasions:

[1] T. Lazar, and I. Bratko. Data-driven program synthesis for hint generation in programming tutors. In S. Trausan-Matu, K.E. Boyer, M. Crosby, and K. Panourgia, editors, *Proc. of ITS 2014*, volume 8474 of *Lecture Notes in Computer Science*, pages 306-311, Switzerland, 2014. Springer.
doi: 10.1007/978-3-319-07221-0_38

[2] T. Lazar, A. Sadikov, and I. Bratko. Rewrite rules for debugging student programs in programming tutors. In *IEEE Transactions on Learning Technologies*. [Accepted for publication in August 2017.]
doi: 10.1109/TLT.2017.2743701

[3] T. Lazar, M. Možina, and I. Bratko. Automatic extraction of AST patterns for debugging student programs. In E. André, R. Baker, X. Hu, M. Rodrigo, B. du Boulay, editors, *Proc. of AIED 2017*, volume 10331 of *Lecture Notes in Computer Science*, pages 162-174, Switzerland, 2017. Springer.
doi: 10.1007/978-3-319-61425-0_14

I certify that I have obtained a written permission from the copyright owner(s) to include the above published material(s) in my thesis. I certify that the above material describes work completed during my registration as graduate student at the University of Ljubljana.

Univerza *v Ljubljani*
Fakulteta *za računalništvo in informatiko*
Timotej Lazar
Generiranje namigov v sistemih za poučevanje programiranja

POVZETEK

Programiranje je uporabna in zmeraj pomembnejša veščina. V zadnjem desetletju so se pojavili mnogi spletni tečaji programiranja, za katere je izkazalo interes mnogo uporabnikov. Na takih tečajih je običajno preveč udeležencev, da bi učitelj delal z vsakim posameznikom. Prav neposredne povratne informacije pa lahko zelo olajšajo učenje.

Področje inteligentnih sistemov za poučevanje oziroma tutorjev se ukvarja s problemom samodejnega podajanja povratnih informacij. Ti sistemi so tradicionalno temeljili na domenskem modelu, ki ga učitelj definira vnaprej. Izdelava takega modela je težavna naloga, sploh v kompleksnih domenah, kot je programiranje.

Potencialna rešitev tega problema je uporaba podatkovno vodenih modelov, ki jih tutor samodejno zgradi tako, da opazuje, kako so učenci reševali naloge v preteklosti. Ko nov učenec naleti na podobno težavo, ga lahko sistem z namigi usmeri na pravo pot. Pri poučevanju programiranja je tak pristop precej zahteven, saj akcij pri pisanju programa ni lahko interpretirati.

Disertacija predstavlja dva nova pristopa k podatkovno vodenemu modeliranju programerskih domen. Prvi pristop modelira pisanje programa z zaporedjem popravkov kode in se uči prepisovalnih pravil za spreminjanje programov. S temi pravili lahko tutor samodejno odpravi napake v novih nepravilnih programih. Drugi pristop uporablja sintaktične vzorce v abstraktnih sintaktičnih drevesih, na podlagi katerih se uči pravil za ločevanje med pravilnimi in nepravilnimi programi. Oba modela lahko uporabimo za samodejno odkrivanje tipičnih napak v programih in generiranje namigov.

Razvili smo spletno aplikacijo za učenje programiranja, v kateri smo preizkusili oba pristopa. Rezultati kažejo, da lahko na podlagi obeh modelov generiramo namige, ki učencem pomagajo pri reševanju programerskih nalog.

Ključne besede: inteligentni sistemi za poučevanje, programiranje, odkrivanje napak

University *of Ljubljana*
Faculty *of Computer and Information Science*
Timotej Lazar
Hint generation in programming tutors

ABSTRACT

Programming is increasingly recognized as a useful and important skill. Online programming courses that have appeared in the past decade have proven extremely popular with a wide audience. Learning in such courses is however not as effective as working directly with a teacher, who can provide students with immediate relevant feedback.

The field of intelligent tutoring systems seeks to provide such feedback automatically. Traditionally, tutors have depended on a domain model defined by the teacher in advance. Creating such a model is a difficult task that requires a lot of knowledge-engineering effort, especially in complex domains such as programming.

A potential solution to this problem is to use data-driven methods. The idea is to build the domain model by observing how students have solved an exercise in the past. New students can then be given feedback that directs them along successful solution paths. Implementing this approach is particularly challenging for programming domains, since the only directly observable student actions are not easily interpretable.

We present two novel approaches to creating a domain model for programming exercises in a data-driven fashion. The first approach models programming as a sequence of textual rewrites, and learns rewrite rules for transforming programs. With these rules new student-submitted programs can be automatically debugged. The second approach uses structural patterns in programs' abstract syntax trees to learn rules for classifying submissions as correct or incorrect. These rules can be used to find erroneous parts of an incorrect program. Both models support automatic hint generation.

We have implemented an online application for learning programming and used it to evaluate both approaches. Results indicate that hints generated using either approach have a positive effect on student performance.

Key words: intelligent tutoring systems, programming, error discovery

ACKNOWLEDGEMENTS

I thank my adviser, prof. dr. Ivan Bratko, for the many insightful comments that helped clarify my thoughts.

I am grateful to Aleš, Marko and Saša for help with CodeQ; Martin for help with code patterns; Vida for answers about life; Jure and Tadej for questions about the universe; and Aljaž, Domen, Erik, Matevž et al. for laughs about everything.

Fala алексею, cupavi, luncoji pa piratki, ka že tak dugo z menof odite skozi život.

Fala staršon: brez vajine ljubezni pa podpore ne bi nej začo nej končo te disertacije.

— Timotej Lazar, Ljubljana, March 2018.

CONTENTS

Povzetek *i*

Abstract *iii*

Acknowledgements *v*

1 Introduction *1*
- 1.1 Motivation 4
- 1.2 Scientific contributions 6
- 1.3 Thesis overview 6

2 Background *9*
- 2.1 Intelligent tutoring systems 10
  - 2.1.1 Model-tracing 13
  - 2.1.2 Constraint-based models 17
  - 2.1.3 Ad hoc models 18
- 2.2 Data-driven tutoring 19

3 Code rewrites *25*
- 3.1 Dataset 27
- 3.2 Rewrites 31
  - 3.2.1 Normalization 32
- 3.3 Rewrite rules 34
- 3.4 Learning rewrite rules 36
  - 3.4.1 Extracting rewrites 38
  - 3.4.2 Selecting fragments 41

3.4.3 Rewrite probabilities 42
3.5 Debugging 43
3.5.1 Evaluation 44
3.6 Generating hints 49
3.6.1 Automatic feedback 49
3.6.2 Manual feedback 51
3.7 Future directions 56

4 Code patterns *59*
4.1 AST patterns 61
4.1.1 Examples 62
4.2 Extracting patterns 66
4.3 Learning rules 67
4.4 Generating hints 69
4.4.1 Automatic feedback 70
4.4.2 Manual feedback 71
4.5 Evaluation 73
4.6 Python 78
4.7 Future directions 81

5 CodeQ *83*
5.1 Feedback 86
5.2 Evaluation 89
5.2.1 Data set 90
5.2.2 First study: rewrites 92
5.2.3 Second study: patterns 95
5.2.4 Discussion 97
5.2.5 User survey 99

6 Conclusion *101*

A Razširjeni povzetek *105*

B Prolog grammar *115*

Bibliography *121*

I

Introduction

> General-purpose computers are astounding. They're so astounding that our society still struggles to come to grips with them, what they're for, how to accommodate them, and how to cope with them.
>
> – Cory Doctorow

The computer is a general-purpose tool that can be programmed to perform arbitrary tasks: it is a tool for creating tools. Most people do not interact with computers this way, and only use tools – programs – that have been created by others. Their computing needs would be served just as well by an advanced typewriter, telephone or videocassette player. In fact, a significant portion of personal computing is now done with phones and tablets: computers that have been locked down to restrict programming.

While computers have been at the center of many technological and societal advances, a great deal of their potential remains latent. For instance, instead of taking advantage of the programmable machine to empower children exploring and learning about the world – a big idea from half a century ago [1] – most schools still use the computer only as a slightly more convenient book or TV.

Computers are in a similar position today as the printing press in the 15th century. While the press made the written word accessible to a much larger population, the full effects of that invention did not appear until centuries later. Once texts could be easily copied without errors, the arguments in those texts became more precise and elaborate. This spurred the creation of formal systems for writing (and thinking) about science and mathematics, ultimately supporting new ways of conceiving the world [2].

Like the printing press, the programmable computer is poised to support new modes of thinking, by allowing us to easily explore and formalize dynamic processes. It takes time, however, for a society to understand any invention, and despite early optimism, "the real computer revolution hasn't happened yet" [2]. Douglas Adams nicely summarized the evolution of our understanding of computers so far:

> First we thought the PC was a calculator. Then we found out how to turn numbers into letters with ASCII – and we thought it was a typewriter. Then we discovered graphics, and we thought it was a television. With the World Wide Web, we've realized it's a brochure.

Lawmakers talking about banning encryption show us that, as a society, we are still far from understanding just what the computer is. Just as the written word cannot

have a significant effect on how people think until the majority can read, the effects computers can have – and our understanding of those effects – will be limited until the majority can use them (in the sense of creating tools, ie. programming).

It is not surprising then that programming is often considered the "new literacy" [3, 4]. What is usually meant is that knowing how to program is an increasingly important skill that should be accessible to everyone. However, just as literacy is not only about translating between letters and sounds, the essence of programming is not in writing code but rather the ability to express one's mental model within the confines of a well-defined formal system.

Programming – especially discovering and fixing errors in incorrect programs – requires a large degree of introspection to uncover the hidden assumptions underlying our understanding of the world. While debugging we often discover that our mental model of the program is incorrect or insufficiently detailed. After correcting our understanding we can usually fix the program. Programming thus provides many opportunities for practicing general cognitive skills [5].

Many initiatives exist to introduce more people to programming, such as the Hour of Code[1] and the EU Code Week[2]. Similarly, many massive open online courses teach programming and other areas of computer science. Online courses typically provide video lectures and a problem-solving environment that allows students to practice their skills and assess their knowledge. Large numbers of participants in these courses indicate the widespread interest in programming among people from all backgrounds.

Since students in these courses can number in hundreds of thousands, providing individual feedback presents a large burden for instructors. The task is especially daunting when teaching programming, where the variability of student submissions is practically unbounded. Generating feedback in an automated manner, at least for the common issues, would significantly reduce the teachers' workload, freeing them to deal with more complicated cases.

The idea dates back to the earliest systems for computer-aided instruction, where feedback was limited to simple *correct/incorrect* responses. To check whether a solution is correct we can simply compare it to the expected answer or, for programming problems, run the submitted program on a set of inputs and check the program's output.

While useful, telling the student that their program is incorrect falls far short of

[1] https://hourofcode.com/

[2] http://codeweek.eu/

the feedback an experienced human teacher can provide. Intelligent tutoring systems (ITSs), first introduced in the 1980s, improve on this by analyzing errors in a submission and generating tailored hints to guide the student towards a solution.

An ITS must include a domain model that enables it to "understand" what the student is doing and suggest a sensible course of action. We distinguish two kinds of domain models for ITSs. *Dynamic* or *process-oriented* models describe the problem-solving *process*; in other words, the sequence of actions a student must perform to get from the initial state (the empty program) to a solution (a correct program). *Static* or *solution-oriented* models, on the other hand, only describe the properties of correct and incorrect states (programs the student submits for testing), and disregard actions the student performed to reach the current state.

Authoring a domain model for programming involves a substantial knowledge-engineering effort. For example, the dynamic domain model used by the Lisp tutor required three person-years of development to support 40 hours of educational content [6]. Static models – for example those used in constraint-based tutors [7] – tend to be somewhat easier to create. While several tools exist to support ITS authoring, tens or even hundreds of development hours are still needed to manually produce one hour of content [8, 9].

Massive online courses are a great use case for intelligent tutors. Furthermore, they also present a new opportunity for automatically creating the domain model [10]. The idea behind *data-driven modeling* is simple: observe how thousands of actual students solve a problem, then use that knowledge to provide feedback based on what successful students did in the past. While the quality of automatic feedback is unlikely to match the output of a hand-crafted model, a data-driven approach can adapt to new exercises without requiring additional work.

1.1 Motivation

We have investigated data-driven domain models for generating hints in programming tutors. It turns out that creating a domain model for programming is particularly challenging, for two reasons outlined below. We have developed both a dynamic and a static data-driven model for programming, each implementing a novel approach to dealing with these challenges.

The greatest obstacle to creating a dynamic model of programming is the fact that writing a program typically proceeds through unstructured text editing. No general,

well-defined (in programming terms) actions exist to describe how students transform the empty program into different solutions. The only directly observable "programming actions" are modifications of the program's text, which are difficult to interpret in terms of programming concepts.

Interestingly, the Lisp tutor – as the first major ITS – used a dynamic rule-based model for writing programs. The rules approximated students' cognitive processes, allowing the tutor to understand the student's actions and also generate new programs on its own. Creating the ruleset was a highly involved process, however, and no later attempts were made to manually construct a model of programming with similar generative power. Furthermore, the Lisp tutor placed several constraints on how students typed their programs, in order to be able to follow their progress.

Our first, dynamic model learns typical *code rewrites* that represent basic programming actions in terms of text-editing operations. We debug incorrect programs by searching for a sequence of code rewrites, and generate feedback based on such sequences. While rewrites do not necessarily represent meaningful "programming steps", they can be learned automatically.

The other main challenge when modeling programming domains is the large variability of possible solutions. Most programming problems can be solved in several ways, and the number of distinct *incorrect* programs submitted by students is practically unbounded. Even the simplest problems, which can be solved in a few lines, tend to accumulate thousands of distinct submissions [11, 12]. Any domain model for a programming tutor will need some way of accounting for these variations.

The second domain model we developed is static, dealing only with individual submissions and not how they evolved. It employs *code patterns* in the programs' abstract syntax trees to describe only those parts of a program that indicate a particular bug or solution strategy. By considering only the relevant parts we can locate the same mistake in different programs, even if no student has submitted the same program before.

One of our primary goals when creating both domain models was to keep them independent of a particular programming language. Our models require very little language-specific knowledge beyond a parser for constructing tree-based representations of programs. We have developed these models using student data from solving Prolog exercises, as it was most readily available. The methods presented in this dissertation are therefore explained using examples from that language, but should be relatively easy to port to other languages given appropriate problem-solving data as

described in Section 3.1.

1.2 Scientific contributions

We present two new methods for creating a data-driven domain model for use in programming tutors. We explain how each method can directly support hint generation, and how they can assist authoring tutors. Finally we describe CodeQ, our web application for learning programming. This dissertation presents the following contributions to science.

- *Programming model based on code rewrites.* We formalize the process of writing a program as a sequence of problem-specific transformations or *rewrites*. We present an algorithm for automatically extracting rewrites from observed student solutions, and give examples of rules discovered from student data. Rewrites represent typical transformations of program code that can be used to generate new versions from a given program, even when that program has not been observed before. We model debugging as a search for suitable sequences of rewrites, and explain how feedback can be constructed from such sequences. We evaluate the rewrite-based debugger on past student submissions and in the classroom using the online programming environment CodeQ.
- *Programming model based on code patterns.* We use abstract-syntax-tree patterns to encode dependencies between variables and literals in a program, and induce a rule-based model to predict program correctness. For each problem, induced rules for correct programs can be interpreted as different possible solution strategies, while rules for incorrect programs encode typical mistakes. We show how both kinds of rules may be used to discover and highlight errors in students' programs. We evaluate hint generation on past student submissions and in the classroom.

1.3 Thesis overview

The following chapter presents related work in the field of intelligent tutoring systems, focusing on existing domain models for programming tutors. Chapters 3 and 4 describe and evaluate the two models we have developed to support hint generation for programming exercises. Chapter 5 presents CodeQ, an online programming tutor we

implemented to collect data and evaluate the effectiveness of feedback generated using these models. The final chapter compares the strengths and weaknesses of each model.

2

Background

This chapter presents existing research on programming tutors. Our work concerns the knowledge component (usually called the *domain model*, or sometimes the *expert module*), which allows an intelligent tutor to discover and correct students' mistakes. How and when to present feedback to students is of course also important, but can for the most part be considered independently of the domain model. We therefore limit this overview to how different programming tutors encode domain knowledge.

The following section gives an overview of main concepts in intelligent tutoring systems (ITSs). Next we describe the major paradigms used for knowledge representation and how they are used in programming tutors. Finally we explain the challenges for data-driven programming tutors and how existing implementations address them.

2.1 Intelligent tutoring systems

Technology has been used in education even before the microcomputer [13, 14]. The early attempts were physical devices, such as Skinner's teaching machines [15], using mechanical components to implement student-machine interaction. Software-based solutions quickly supplanted mechanical devices as digital computers became more affordable in the 1960s. One of the most prominent educational software frameworks was PLATO with many advanced (at the time) features such as graphics, support for collaboration between users, and an authoring environment for teachers [16].

The main functions of all teaching systems – whether implemented in hardware or software – are: presenting information, allowing the student to interact with the system, and providing feedback to the student [13]. Most systems for computer-assisted instruction (CAI) are problem-oriented: each unit of information is accompanied by a set of exercises for the students to test and improve their understanding. The main advantage of these systems over textbooks and other static learning materials is the ability to provide feedback for the student's responses.

Before describing ITSs we briefly mention *Microworlds*, which represent an alternative CAI paradigm. They provide a simple open-ended world for the student to explore, and have been particularly effective for teaching programming [17, 18]. Notable examples include Logo [19], Alice [20] and Scratch [21]. Fig. 2.1 shows the user interface for Snap![1], a visual programming language and environment based on Scratch but extended with advanced features such as classes and continuations.

[1] Available at https://snap.berkeley.edu/.

Unlike other CAI systems, microworlds typically do not provide clearly defined goals or tasks for the student. With no information about what the student is trying to achieve, feedback is usually limited to messages about syntactic errors. The methods presented in this dissertation assume a problem-oriented learning environment with clearly defined tasks and solutions, so we focus on such systems in this chapter.

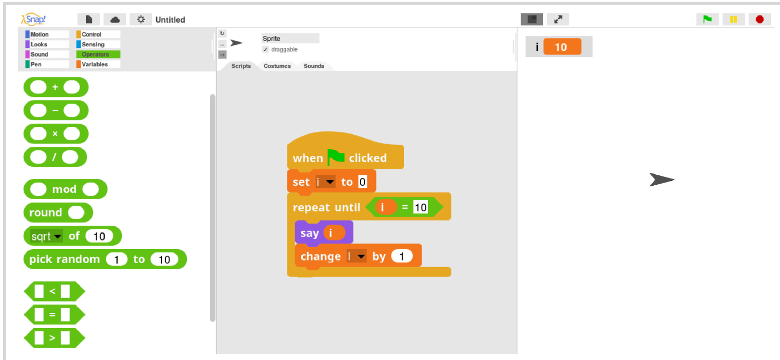

Figure 2.1

The Snap! visual programming language and environment. The left column shows available building blocks, which are used to compose a program in the middle. The right column shows the sprite controlled by the program, and current variable values.

Early CAI systems were styled after traditional textbooks, encoding and presenting educational content as opaque blocks or *frames* of information [22]. Questions were usually limited to simple numeric or multiple-choice answers. While the teacher could program a system to respond in a certain way when a student submits an incorrect answer, the system itself was unable to adapt to individual students or support their problem-solving efforts with immediate feedback [23].

Like most CAI systems, ITSs are usually problem-oriented [24]. The tutor presents a list of problems in some subject domain and helps the student solve them. Some tutors focus on teaching specific skills, e.g. operations on fractions [25] or linked lists [26], while others cover broad subjects such as high-school-level physics [27] or geometry [28]. While many tutors include learning materials to support self-study, they have often been designed to augment existing classes by replacing traditional exercises [26–29].

ITSs improve the traditional CAI systems in two important ways [14]. First, an ITS models the state of each student's knowledge, allowing it to adapt the order and presentation of educational content to the needs of individual students. For example, if a student keeps repeating the same mistake, an ITS could suggest to reread the relevant

sections or provide additional exercises. Conversely, a student doing well might be encouraged to skip ahead to the more difficult problems.

Second, an ITS provides a domain-specific problem-solving interface, so the student can do all their work on the computer instead of just inputting the answer. The user interface is typically specialized for the subject domain, reducing the cognitive load for the student and enabling the tutor to “observe” the student’s actions. This allows the ITS to provide *immediate* feedback when a student makes an incorrect step and explain the error in the terms of the subject domain. Fig. 2.2 shows the user interface of the Andes physics tutor[2].

Figure 2.2

The problem-solving screen of the Andes physics tutor. The main part is the free-body diagram, where the student draws objects and the forces acting on them. The right-hand side shows defined quantities and equations relating them. In place of pen and canvas the interface provides specialized tools for drawing vectors and objects. This allows Andes to follow and understand the student’s progress in terms of high-level operations such as decomposing a force vector acting on a body on an inclined plane.

Operation of ITSs can be described as consisting of two loops [30]. The *outer loop* is executed once per problem to select the next problem for the student to solve. The two components of an ITS that enable it to suggest appropriate problems are the *student* and *tutoring models* [31]. The student model keeps track of the concepts the student has mastered, while the tutoring model encodes the pedagogical policy used to decide

[2] Image from http://andestutor.org/its2008-demo/.

which problems to present to the student in order to yield the greatest learning gains. This policy can be fixed or adapt to each student. Machine learning has also been used to optimize these models based on student data [32–34].

The other defining feature of an ITS, and the one we focus on in this dissertation, is the *inner loop*, providing feedback as the student works on a problem. *Model-tracing tutors* execute this loop for every problem-solving step, analyzing the student's progress towards a solution and alerting them to any mistakes. Other tutors do not consider individual steps and instead only provide feedback when the student submits a solution. In the following subsections we look at the main tutoring paradigms and their implications for programming tutors.

Several intelligent tutors have been successfully deployed. In the US, many high schools have incorporated cognitive and other tutors into their curricula, particularly for mathematics and physics [27, 28]. Small- and large-scale studies have confirmed that tutoring systems increase student performance in these subjects [29, 35–38].

Few programming tutors have seen such widespread use. Notable examples are the ACT programming tutor for Lisp, Pascal and Prolog [39], and the SQL tutor [7], both of which have been used to enhance university-level courses. Other tutors focus on teaching specific concepts or skills, such as the iList tutor for teaching linked lists [26]. On the other hand, there are a plethora of commercial online learning environments, such as Coursera and Codecademy[3], that share some attributes with the classic programming tutors: individual learner modeling, immediate feedback and bug libraries. Courses provided by these platforms can attract hundreds of thousands of students, but suffer from high attrition rates [40]. In Slovenia, Projekt Tomo is used to teach Python in several high school and university-level courses [41].

2.1.1 Model-tracing

Model-tracing tutors represent one of the earliest tutoring paradigms. They employ a detailed cognitive model for solving problems in the target domain, allowing them to find step-by-step solution paths [42]. In terms from the previous chapter, model-tracing tutors use a dynamic domain model as they are concerned with the correctness of individual problem-solving steps a student takes. They are called *model-tracing tutors* because they compare the trace of student actions with the correct sequence of steps

[3]Available at `https://coursera.com` and `https://codecademy.com`.

predicted by the domain model, and base their feedback on the differences between the two [14].

Cognitive tutors, based on the ACT cognitive theory [43], represent the most prominent and well-researched instance of the model-tracing approach. Here we describe the Lisp tutor [6], as the first cognitive tutor and one of the earliest modern ITSs overall. The core ACT principle is to distinguish between declarative and procedural knowledge. Students assimilate *chunks* of declarative knowledge from lectures and books; a typical chunk for Lisp programming is:

> The function `car` takes a list and returns the first element.
> For example, `(car '(a b c d))` returns `a`.

Procedural knowledge, on the other hand, supports goal-oriented problem solving. It is encoded as a set of production (*if-then*) rules. For example, the rule

> *If* the goal is to code an expression that returns the first element of a list,
> *then* code the operator `car` and set a goal to code the list as its argument.

gives the procedural counterpart to the declarative chunk above. Productions cover both planning (how to decompose a problem into subproblems) and operative (which action achieves a goal) aspects of solving a programming problem.

The distinction between the two kinds of knowledge informs the pedagogical strategy for cognitive tutors [44]. The student first acquires declarative knowledge through explanations and worked examples. Production rules are then learned through solving problems, by applying general strategies like analogy and planning to declarative knowledge. The learned productions strengthen and become more refined through practice.

Fig. 2.3 shows the main user-interface elements of the Lisp cognitive tutor: problem statement, a feedback window, and a structured code editor [39]. The student has partially written the function `last-item` for extracting the last element from a list, and has selected `<EXPR1>` as the next fragment to refine. The tutor offers a menu of common Lisp functions, where the student can choose a replacement for the selected fragment.

Equipped with a formal definition of the problem and a catalog of productions, the tutor can use a planning algorithm to generate all possible variants of this function. If the student attempts to replace `<EXPR1>` with a fragment that does not appear in

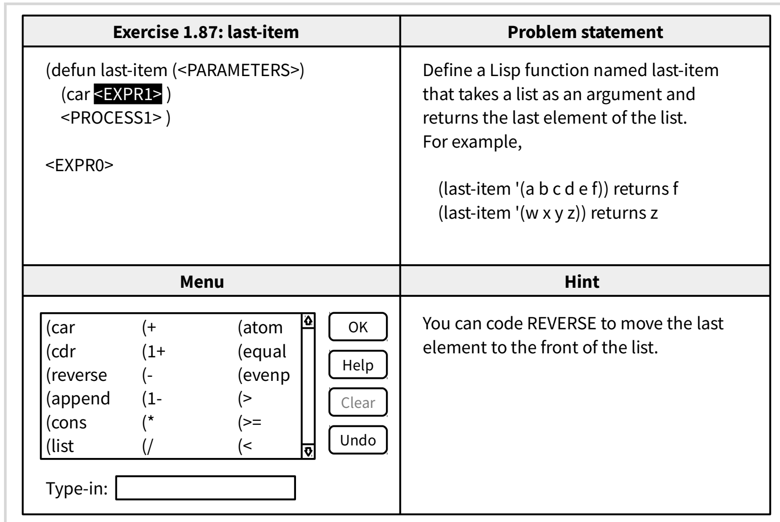

Figure 2.3

Schematic representation of the Lisp tutor's main components. The right-hand side shows the problem definition and feedback, while the left-hand side contains the structured code editor with a menu for inserting code fragments. In the editor (upper-left) window, the student has selected a missing fragment `<EXPR1>` to complete.

any generated solution, the tutor responds with a hint – in this example, a chunk of declarative knowledge guiding the student to use the `reverse` function.

While model-tracing tutors can be very effective, they have two significant drawbacks. First, the underlying domain model is complex and difficult to create. The domain model for the Lisp tutor contained over 1,200 production rules to support about 30 hours of educational content, and required three person-years to construct [6]. This is not a major issue since an ITS, once created, may be used by any number of students. Additionally, authoring tools exist that alleviate some of the effort associated with creating domain models [8, 45].

The other limitation is that a model-tracing tutor must be able to understand what the student is doing in terms of actions used in production rules. Problem-solving steps therefore usually correspond to user-interface events [30]. For example, in the Deep Thought tutor for deductive logic, each step is an application of an inference rule to one or more premises [46]. The user interface contains buttons for different rules, so that the tutor can follow the student's chain of inference and compare it to solutions generated by the domain model. Similarly, the Andes physics tutor offers a specialized interface for drawing free-body diagrams and solving equations [27].

To put it another way: model-tracing tutors represent the problem-solving process as a sequence of general and meaningful steps, and limit student interaction to these steps. When writing code, however, the only directly observable actions are inserting and deleting arbitrary text fragments. It would be impossible to model programming in a meaningful way using only actions of the form "insert the letter e at position 42".

For this reason, the Lisp tutor uses a structured code editor, which ensures that the program is always syntactically correct. More importantly, structured editing is the essential feature enabling the tutor to understand what the student is doing in terms of high-level actions like "coding function parameters" or "replacing a value with a function call". These are the same actions that are used in production rules, allowing the tutor to compare the student's progress to model-based solutions.

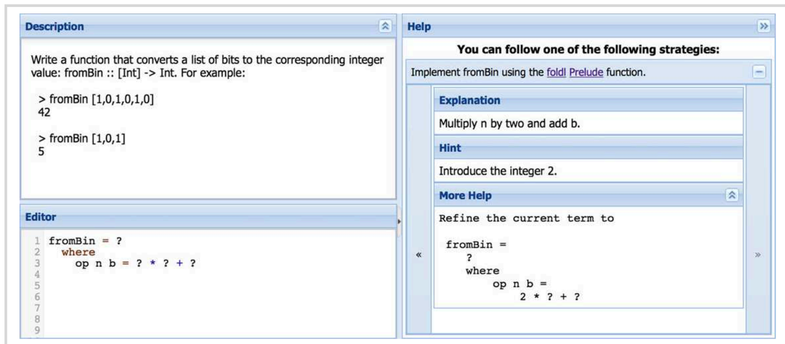

Figure 2.4

Ask-Elle: a Haskell tutor. The student can code freely in the lower-left window, and use question marks to indicate "holes" (unfinished fragments) in the program. Programming is modeled as a sequence of refinements, where a hole is replaced with a value, a function call, or some other construct.

A structured editor frees the student from having to worry about syntax [6]. On the other hand, it forces them to program in a somewhat unnatural, top-down fashion. We conclude that the model-tracing approach is not very suitable for domains where meaningful problem-solving steps are difficult to observe from student interactions. While the Lisp tutor has been extended to Prolog and Pascal [47], no other cognitive tutors for programming have been developed.

A much more recent model-tracing tutor for Haskell[4] uses per-problem solution strategies – models that can be instantiated in different ways to account for potential variations in student programs [48]. Fig. 2.4 shows the tutor's user interface. While the tutor allows unstructured code editing, the students are still required to write programs

[4]Available at http://ideas.cs.uu.nl/AskElle/.

top-down and indicate unfinished parts of the program with “holes”. Model-tracing thus appears to be most amenable to functional programming languages.

2.1.2 *Constraint-based models*

Since the process of writing a solution to a programming problem is difficult to formalize in terms of meaningful and observable actions, many programming tutors ignore the process altogether. Instead they use a static domain model to analyze only individual submissions, i.e. versions of the program the student submits to the tutor as a potential solution. Just like ACT tutors are an instance of a general model-tracing approach, *constraint-based modeling* [49] represents a specific implementation of this approach that has received the most attention; we look at it in this section.

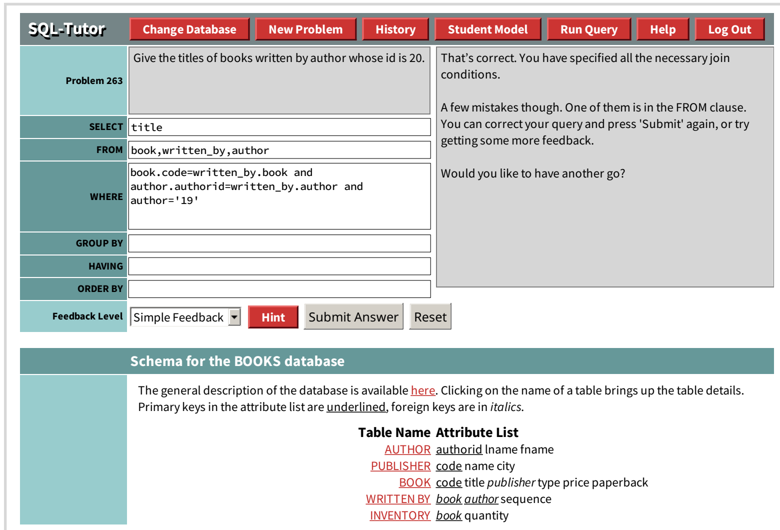

Figure 2.5

SQL tutor’s problem-solving interface. The student must write an SQL query given a database schema. The structure of the query is given, and only the individual fields must be completed. The tutor provides no feedback until an answer is submitted, or a hint is explicitly requested.

Fig. 2.5 shows an online version of the SQL tutor [50], one of the earliest constraint-based tutors. The task is to write an SQL query to answer the given question, based on the provided database schema. While the query structure is fixed and the student only needs to fill out individual clauses, this is only done to reduce the cognitive load; the tutor would function just as well with a single text field for the whole query.

The student is free to write parts of the query in any order. Their progress is analyzed

only when the student clicks the "Submit Answer" button. At that point, the tutor checks whether the submitted solution satisfies all relevant *constraints*. Like procedural knowledge in cognitive tutors, constraints are typically encoded using *if-then* rules, for instance:

> *If* the `FROM` clause contains the `JOIN` keyword
> *then* it must also contain the `ON` keyword.

The first part of a rule (relevance condition) determines for which solutions a constraint applies, while the second part (satisfaction condition) tells us what properties must hold for those solutions. The above rule expresses a syntactic constraint for SQL queries. Rules can also describe semantic constraints, such as how the results of a query should be ordered [51]. Each constraint has an associated explanation in natural language, offered as a hint when a solution violates that constraint.

Constraint-based domain models are descriptive: each constraint encodes a certain property that must hold for all correct solutions. Unlike cognitive models, constraints typically cannot be used to directly generate new solutions. On the other hand, they are easier to define, especially for complex domains. Several other programming tutors have been constructed independently using constraint-based modeling [52–54]. The INCOM tutor for Prolog extends the constraint-based model by weighting different types of constraints (e.g. syntactic or semantic errors) according to severity [53].

2.1.3 *Ad hoc models*

Other programming tutors use ad hoc domain models, not based on a certain tutoring paradigm. Like constraint-based tutors, such models are practically always solution-oriented: the tutor only provides feedback when a program is submitted, and not while the student is writing it. These tutors usually model domain knowledge in one of two ways: either with a set reference programs for each problem that represent the different ways of solving it [55–60], or with a library of common programming techniques and mistakes [61–63].

PROUST was an early Pascal tutor [64], which used *programming plans* to statically analyze student submissions. Plans relate program fragments that perform a certain function. For example, the "counter variable" plan covers statements in the program that initialize and increment a loop counter, while the "running-total loop" plan covers statements that read and add values to a running total in a loop. By matching a program

to a database of correct and buggy plans, PROUST was able to discover what the student's intended to do, and point out mistakes. The database of plans was manually designed for a single problem and later extended to several others [56].

A similar plan-based approach was used in ELM-PE [59], another tutor for the Lisp language, and the C-Tutor [58], which extracted plans automatically from a reference solution. Knowledge base of Hong's Prolog tutor [57] was implemented as a hierarchy of plans (called programming techniques [65]). For example, the "if-then-else" technique covers programs that use cut (!) to limit search to exactly one branch:

```
<cond> :- <test>, !, <case 1>.
<cond> :- <case 2>.
```

Another example is the "recursion with accumulator" technique, describing programs that use an accumulator argument to recursively construct a data structure.

Singh et al. define a bug library in terms of *correction rules* using their Error Model Language [63]. These rules allow them to synthesize new programs from an incorrect submission in order to find a sequence of corrections that will fix it. Our rewrite-based model, presented in Chapter 3, employs the same approach. Correction rules must however be defined manually, while our rewrites are learned automatically.

The methods above analyze submitted programs statically to discover known patterns in the code. An alternative is to run the program and observe its behavior. With this approach, the tutor executes each student programs to record the trace of its runtime behavior, and compares this trace to the correct solution. The differences can be used to match programs using the same algorithm, and to pinpoint errors in the code [66–68]. Note that only individual submissions are analyzed, while evolution of the program from one version to the next is ignored.

Since tutors described in this section do not track how each program is written, they are easier to develop than model-tracing tutors. On the other hand, considerable work is still required to construct a useful set of programming plans or reference solutions for each problem. Data-driven tutors attempt to reduce this effort, by constructing or updating the domain model automatically from observed student solutions.

2.2 *Data-driven tutoring*

With increasing use of technology in education – massive online courses being a prominent example – ever larger amounts of educational data are becoming available. This

presents new opportunities for improving the student and pedagogical models in ITSs, for example by finding the optimal ordering for a set of problems [10]. Data-driven tutors can also utilize student data to learn or update the domain model, which is the focus of this dissertation. The basic idea is this: use past observed student behavior to "learn" how to solve individual problems. This reduces the expert's workload when building a tutor.

For reasons discussed in this section, building a data-driven model to support completely automated feedback is quite difficult for programming domains. Several existing approaches focus instead on the easier problem of helping a human teacher provide feedback for a large number of student submissions [69–72]. Other tutors can generate feedback automatically from past student solutions [12, 61, 62, 73, 74]. Both methods described in Chapters 3 and 4 of this dissertation are geared towards autonomous hint generation. However, they can also be used to help teachers provide feedback in programming tutors.

One of the main challenges for data-driven programming tutors is that programming is unique in the number of different solutions the students come up with. For example, students submitted over 40,000 distinct programs implementing gradient descent in an online course on machine learning; the standard solution for this exercise consists of seven lines of code [11]. In an introductory programming course using a Scratch-like visual programming language, Piech et al. similarly found over 10,000 distinct submissions for a very simple problem with the following correct solution [12]:

```
move forward
turn left
move forward
turn right
move forward
```

A small number of common programs usually account for approximately half of all submissions. The remaining programs are submitted by much fewer students, forming a long tail of rarely occurring submissions [12, 75].

One way to address this problem is to normalize student programs before analyzing them [61, 72–74, 76]. Normalization steps typically include rewriting expressions into a canonical form (e.g. "`b>a`" into "`a<b`"), renaming variables according to some consistent scheme, inlining functions, and so on [75]. Such transformations are language-specific, and often not completely general – for example, in certain languages reorder-

ing the expression "b>a" might change its result in some situations. This is however not a major concern in the tutoring context.

Using such canonicalization techniques, Rivers et al. were able to reduce the space of syntactically correct Python programs by 60% [73]. Using equivalence classes of *code phrases*, the CodeWebs tool was able to transform 25,000 different programs (out of 40,000) for the machine-learning problem into just 200 canonical versions [69]. The reduced number of programs allows "force-multiplying" teacher-provided feedback to cover submissions from many students with little or no extra work.

One early data-driven programming tutor is the MEDD system, which uses multistrategy conceptual clustering to discover common errors in Prolog programs [61]. The system first extracts discrepancies between the student's program and the closest reference solution. Then, it uses an incremental clustering algorithm with different similarity measures to discover misconceptions in terms of discrepancies that occur in many solutions. Discovered misconceptions are used to detect errors and generate feedback for student programs. Before comparing programs, MEDD transforms programs using various Prolog-specific transformation rules. The system was extended to support programs written in Java [62].

Several programming tutors adopt the Hint Factory approach, first developed to automatically generate hints in the Deep Thought logic tutor [77]. The tutor provides an environment for the students to practice propositional calculus by deriving conclusions from premises using standard rules of inference. The problem domain is represented as a state space of partial and complete solutions, where each state represents a set of derived premises, and each action represents an application of an inference rule to one or more premises. Hint Factory derives a problem-solving policy for each problem as a Markov decision process from observed student solution traces (i.e., sequences of inferences used to derive a conclusion) [78]. This policy allows the tutor to generate next-step hints, based on the student's current state.

The idea behind the Hint Factory is to automatically build a problem-solving model that supports generating new solutions, similar to handcrafted cognitive domain models. The state-space approach is however not easily applied to programming because transitions and states are not easy to formalize in a way that would lend itself to conceptual analysis.

The first problem is the lack of meaningful actions. While deriving a logic proof can be usefully described as a sequence of well-defined steps, no such steps exist in

free-form programming. Programming tutors employing the Hint Factory approach thus typically model the problem-solving process in terms of sequences of programs the students have submitted for testing [12, 73, 74, 79, 80].

These tutors combine many such sequences into a *solution space*, where nodes represent correct and incorrect submissions, and edges connect successive submissions. In other words, an edge $s_1 \longrightarrow s_2$ means (only) that one or more students have submitted the program s_1, followed by s_2 [12]. The Hint Factory approach can thus be used to determine a problem-solving policy. Unlike the logic tutor, however, transitions in the solution space do not correspond to meaningful student actions, which means the tutor cannot use transitions on programs that have not been observed before. In our first programming model based on code rewrites (Chapter 3) we attempt to learn meaningful transitions that can be used on previously unseen programs.

A related problem with using a solution-space representation for programming domains is that individual states (submitted programs) are not easily inspectable. Unlike the logic tutor, where each problem-solving state is represented simply as a set of currently derived logic formulas, programs are difficult to decompose into meaningful independent elements, especially without relying on language- and problem-specific knowledge [81]. Data-driven programming tutors usually use syntactic or run-time features to distinguish between programs.

The CodeWebs tool learns semantically equivalent *code phrases* in MATLAB programs collected from an online course on machine learning [69]. Code phrases are subtrees of the program's abstract syntax tree (AST) that occur in many solutions. To determine whether two code phrases are equivalent, CodeWebs tests whether the program's behavior remains the same (determined using a battery of test cases) after replacing one phrase with another, for all programs containing these phrases. This way it builds a database of equivalence classes of code-phrase that can be used to canonicalize equivalent programs into the same normal form.

Zimmerman et al. use approximate subtree matching based on pq-grams [82] to recommend program elements (derived from relevant AST subtrees) for programs in a full IDE [76].

Instead of modeling the evolution of the whole program, Price et al. extend the Hint Factory algorithm to use subtrees of the program's AST in the solution space [74]. This way they can model modifications to individual subtrees. If an incorrect program is not found among existing observed solutions, each part of the program can be considered

independently to generate hints.

Jin et al. used *linkage graphs* as program features more amenable to the solution-space approach [79]. A linkage graph encodes the dependencies between statements in a program: a statement b depends on another statement a if b references at least one variable created or modified by a. Linkage graphs represent programs at a high level and are robust against small code variations.

Another tool to help manage many student submissions is OverCode [70], which uses simple dynamic analysis to cluster solutions. Two programs are considered equivalent when their variables take on the same sequences of values during execution. OverCode presents these clusters with a specialized interface that allows the teacher to define custom rewrite rules specifying additional normalizations and further reducing the number of distinct program clusters.

A more involved approach using dynamic analysis [72] executes programs on different inputs, recording the resulting Hoare triples (precondition, program, postcondition) [83] for every subtree in the program's AST. These triples are embedded into an Euclidean space where each (sub)program is viewed as a linear mapping between pre- and postconditions. Using recursive neural networks they learned how to propagate teacher feedback for a small sample of submitted programs to many other relevant solutions. Their approach is limited to programs without variables and requires a large number (tens of thousands) of student submissions.

Outside the tutoring setting, many methods exist for assessing similarity between programs and predicting faults, in terms of code features such as the number of functions or classes and cyclomatic complexity [84, 85]. Another option is using generic text- or graph-based similarity measures [86, 87]. White et al. used deep learning to find code fragments for detecting duplicated code [88]. While successful, most of these methods pick the low hanging fruit – easy-to-detect errors in large software projects. In the tutoring setting we conversely have very little code and a comparatively large space of possible errors to handle.

3

Code rewrites

We first present a dynamic model of programming we created to describe the process of solving a programming problem. We model this process as a sequence of *code rewrites* describing specific transformations of the program code. One of the few existing examples of this approach is the cognitive Lisp tutor [6], where the various possible programming actions are manually encoded as production rules. We learn code rewrites automatically by observing how students program.

Any definition of the problem-solving steps in programming must have certain properties to be useful for a tutor. First, steps must be *observable* from the students' interaction with the tutor. Second, each step should be *meaningful* in terms of the learning domain; in other words, it should allow us to reason about what the student is doing. Finally, programming steps should be *general* and not specific to individual programs.

As explained in the previous chapter, there are no directly observable meaningful steps in a free-form programming task. Many programming tutors and environments avoid this problem by using a structured or visual editor (e.g. Scratch) or requiring program code to be entered in specific order (e.g. the Lisp tutor). One of our main goals was to support hint generation in a programming interface that approximates the "real world" as closely as possible, which means using an ordinary text editor for writing programs.

Using such an editor means that only insertions and deletions of individual characters can be observed directly. While these atomic editing actions are generic (they can be applied to any program to generate a new version), they have no semantic content: the action "insert the letter `e`" does not involve, and cannot be used to reason about, any programming concepts. Such actions hence cannot be used for building a programming model.

We address this problem by grouping related single-character edits into *code rewrites* representing the basic problem-solving steps in programming. Rewrites can be viewed as macro-operators for modifying code fragments. We automatically extract rewrites from student solutions and generalize them into *rewrite rules* that encode information about situations where a particular rewrite is likely to be useful.

After we have obtained a catalog of rewrite rules for some problem, we can model debugging as search: starting from the incorrect program, find a suitable sequence of rewrites that transforms it into a correct version. While single-character editing actions could be used in the same way, the search would be infeasible due to a large branching

factor and lack of meaningful heuristics.

The following sections describe rewrites, how they are extracted from student solutions, and how we generalize them into rewrite rules. We also explain the debugging procedure, and how rewrite rules can support both automatic hints and manual feedback authoring.

3.1 Dataset

We first describe the format of data used for developing and learning our programming models. The data covers student interactions that can be observed in most programming tutors. Collecting it does not require a specialized interface such as a structured code editor; we used our CodeQ tutor (described in Chapter 5). The same data was used for both the rewrite-based programming model presented here, and the pattern-based static model presented in the next chapter.

We store collected data as a list of *traces*, where each trace describes one *solution attempt* – that is, a particular student working on one problem. A trace is simply the sequence of actions the student performed while solving the problem which allows us to reconstruct the entire problem-solving process. We recorded the following types of actions:

- insert/delete: These are the main actions that actually modify the program. Typing actions generally add or remove a single character. Other possibilities are cutting and pasting, which modify larger chunks of texts. Since these chunks are always contiguous, these larger actions can be considered as sequences of single-character insertions or removals.
- test: All tutoring systems allow the student to check whether their current solution is correct. In programming tutors, this is typically done by checking the program's output on a predefined set of test cases. We record an action each time the student submits a program for testing.
- query: Like many other programming tutors, CodeQ provides an interactive interpreter for the target language, allowing students to run their programs on arbitrary inputs. We record all queries the student ran while working on a problem.

- `feedback`: For evaluation purposes, we also record all feedback the student receives, including test results and generated hints.
- `open`/`close`: In CodeQ, the student may stop working on a problem at any point and resume the attempt later. We record the time when the student started or stopped working on a problem.

A timestamp is included with each action. We do not record mouse input such as clicks or movements (besides well-defined actions such as pressing the `Test` button). While such actions are not directly related to the programming task, they could help analyze students' emotional states like boredom or frustration. Such analysis is however out of scope of our work.

Table 3.1 shows a cleaned up and abridged trace for one student's solution of the Prolog problem `dup/2`. In this problem, students must write the predicate `dup(L,L2)`, which duplicates each element in the list `L` to produce the new list `L2`. This exercise often appears among the introductory problems for Prolog lists. Row 13 of the table shows the canonical solution to this problem:

```
dup([],[]).
dup([H|T],[H,H|DupT]):-
    dup(T,DupT).
```

This is a typical recursive program with two clauses. The base case (first line) states that duplicating elements in the empty list `[]` again yields the empty list. The second, recursive clause tells us how to duplicate elements in a nonempty list `[H|T]`, composed from the first (head) element `H` and the remaining (tail) elements `T`: duplicate the first element and recursively process the remainder of the list. To be classified as correct, a program must return the correct answer to the following queries:

```
1. ?- dup([],X).                       % X = []
2. ?- dup([y],X).                      % X = [y,y]
3. ?- dup([k,f,f,g,a],X).              % X = [k,k,f,f,f,f,g,g,a,a]
4. ?- dup(X,[k,k,f,f,f,f,g,g,a,a]).    % X = [k,f,f,g,a]
5. ?- dup(X,[4,4,3,3,8,8]).            % X = [4,3,8]
```

Actual traces contain separate actions for every inserted and deleted character. This allows us to reconstruct every version of the program, beginning with an empty string

Table 3.1

Abridged solution trace for a solution to the problem dup/2. Groups of insert and delete actions that modify the program from one version into another are shown as a single modify actions, with the changed parts of the program shown in blue.

<table>
<tr><th>#</th><th>Time</th><th>Action</th><th>Comment</th><th>Program</th></tr>
<tr><td>1.</td><td>0 s</td><td>open</td><td>Start with an empty program.</td><td></td></tr>
<tr><td>2.</td><td>68 s</td><td>modify</td><td>Insert the (incorrect) base case.</td><td><code>dup([X],[X|X]).</code></td></tr>
<tr><td>3.</td><td>112 s</td><td>modify</td><td>Start the recursive rule.</td><td><code>dup([X],[X|X]).</code>
<code>dup(L,L2):-</code>
<code>   dup(</code></td></tr>
<tr><td>4.</td><td>135 s</td><td>modify</td><td>Refine the first argument.</td><td><code>dup([X],[X|X]).</code>
<code>dup([H|T],L2):-</code>
<code>   dup(</code></td></tr>
<tr><td>5.</td><td>179 s</td><td>modify</td><td>Complete the recursive rule.</td><td><code>dup([X],[X|X]).</code>
<code>dup([H|T],L2):-</code>
<code>   dup(T,L2).</code></td></tr>
<tr><td>6.</td><td>181 s</td><td>test</td><td>No tests passed.</td><td></td></tr>
<tr><td>7.</td><td>232 s</td><td>modify</td><td>Fix the base case.</td><td><code>dup([X],[X,X]).</code>
<code>dup([H|T],L2):-</code>
<code>   dup(T,L2).</code></td></tr>
<tr><td>8.</td><td>296 s</td><td>modify</td><td>Partially fix the recursive rule.</td><td><code>dup([X],[X,X]).</code>
<code>dup([H|T],[H|L2]):-</code>
<code>   dup(T,L2).</code></td></tr>
<tr><td>9.</td><td>299 s</td><td>test</td><td>One test passed: 2.</td><td></td></tr>
<tr><td>10.</td><td>313 s</td><td>query</td><td><code>?- dup([1,2,3],X).</code></td><td></td></tr>
<tr><td>11.</td><td>352 s</td><td>modify</td><td>Fix the recursive rule.</td><td><code>dup([X],[X,X]).</code>
<code>dup([H|T],[H,H|L2]):-</code>
<code>   dup(T,L2).</code></td></tr>
<tr><td>12.</td><td>354 s</td><td>test</td><td>Four tests passed: 2,3,4,5.</td><td></td></tr>
<tr><td>13.</td><td>402 s</td><td>modify</td><td>Reduce the base case.</td><td><code>dup([],[]).</code>
<code>dup([H|T],[H,H|L2]):-</code>
<code>   dup(T,L2).</code></td></tr>
<tr><td>14.</td><td>405 s</td><td>test</td><td>All five tests passed.</td><td></td></tr>
<tr><td>15.</td><td>424 s</td><td>close</td><td>Close the editor.</td><td></td></tr>
</table>

and ending with the solution. Since most traces contain hundreds of actions, Table 3.1 only shows some of the significant versions for brevity. A sequence of insert and delete actions that transforms the program from one version to the next is shown as single modify action. For each modify action, the parts of the program changed from the previous version are shown in blue.

As always, the student begins with the empty program (row 1). After the initial burst of insertions and deletions, the program's basic structure is complete (row 4). This is very common for the kinds of problems considered here: students write the initial version of the whole program quickly, then spend most of the time locating and removing bugs. Both clauses in row 4 initially contain errors that will be fixed later.

Tracking the programming process at the character level allows us to make some interesting observations. For instance, the student initially (row 3) writes the head of the recursive rule with the generic, non-refined arguments `dup(L,L2)`. Only when they reach the point where `L`'s tail must be passed to the recursive call (open parenthesis in the last line in row 3) do they realize that the tail is not accessible. They return to the second line to refine `L` into `[H|T]` (row 4), and then complete the rule (incorrectly) in row 5. This behavior is very common with beginners, until they learn that this is a general pattern when solving list problems recursively.

Next, the student submits the program for testing (row 6). Due to the erroneous base case (in the term `[X|X]` the variable `X` represents both head and tail of the list, which is almost never correct or intended), the program passes no test cases. The student fixes this error in the next version (row 7), and adds the missing head to the output list in row 8. However, since this only prepends one copy of each element to the output list instead of duplicating them, the program still passes only one test case (where the input list contains only one element and is thus covered by the base case).

At this point the student submits a query to the Prolog interpreter (row 10):

```
?- dup([1,2,3],X).
  X=[1,2,3,3].
```

Only the last element is duplicated by the base-case rule. The student realizes that the recursive rule does not actually duplicate elements, and fixes the bug in row 11.

The next test confirms that the program is now almost correct. The student realizes that it does not work for the empty list, because both rules require a list with at least one element as the first argument. After simplifying the base case in row 13, the program works as expected.

3.2 Rewrites

A rewrite $a \longrightarrow b$ transforms a program by replacing the code fragment a (left-hand or "before" part) with the new version b (right-hand or "after" part). Rewrites consolidate a set of related insertions and deletions into meaningful code transformations. Since they are defined in terms of character strings, they are independent of the programming language. For example, the following rewrite in C

```
for (i=1; i<=n; i++)  ⟶  for (i=0; i<n; i++)
```

groups several character-level deletions and insertions[1] that together change the loop counter `i` to use zero-based indexing. This rewrite fixes a mistake that a student learning C arrays might make. Of course, the rewrite is not necessarily appropriate for every program. Whether it should be used or not depends on the loop body and the programmer's intent.

Unmodified fragments on the left-hand side of a rewrite serve as local context that limits the rewrite's applicability. For example, the small Prolog rewrite

```
dup([H|T],[H,H,DupT])  ⟶  dup([H|T],[H,H|DupT])
```

fixes the incorrect list construction in the second argument. The context here is the structure `dup/2` with two list arguments. The student's mistake was using the `,` operator, used for enumerating items in a list, instead of the `|` operator, which joins the head and tail of a list (similar to the `cons` operator in Lisp).

A smaller context yields more generic rules that can be applied to more programs. The transformation above could, for instance, also be represented by the minimal rewrite `,` $\longrightarrow$ `|`. This rewrite could be applied to any comma in the program, and would in most cases result in a broken program.

At the other extreme, the same transformation can be represented by a rewrite that includes the whole program on the left-hand side, for instance:

```
dup([],[]).                       dup([],[]).
dup([H|T],[H,H,DupT]) :-  ⟶  dup([H|T],[H,H|DupT]) :-
    dup(T,DupT).                      dup(T,DupT).
```

Applying this rewrite to any program matching the left-hand side will result in the correct implementation of the `dup/2` predicate, unless the program contains additional

[1]Red parts on the left-hand side indicate deletions; green parts on the right-hand side indicate insertions.

incorrect `dup/2` clauses (even then, the modified version will be closer to the solution, since only the extraneous clauses must be removed). On the other hand, this rewrite is not generic at all since it is only applicable to one particular program.

The amount of context used in rewrites is thus a trade-off between allowing rewrites to generalize to more programs, and ensuring that each rewrite is appropriate wherever it can be applied. We choose the appropriate context based on the program's structure; this is explained in Section 3.3 on rewrite rules.

3.2.1 Normalization

Rewrites described above are simple text-replacement operators. While this makes them independent of the programming language and easy to extract from student traces, it also means that even small variations in program code can make a rewrite inapplicable. For example, the above rewrite

```
dup([H|T],[H,H,DupT]) ⟶ dup([H|T],[H,H|DupT])
```

cannot be applied to the program

```
dup([], []).
dup([H|T], [H, H, DupT]) :-
    dup(T, DupT).
```

due to the extra whitespace, or the program

```
dup([],[]).
dup([X|Y],[X,X|NewY]) :-
    dup(Y,NewY).
```

due to a different choice of variable names. In other words, the rewrites

- `dup([H|T],[H,H,DupT])` ⟶ `dup([H|T],[H,H|DupT])`,
- `dup([H|T], [H, H, DupT])` ⟶ `dup([H|T], [H, H|DupT])`, and
- `dup([X|Y],[X,X,NewY])` ⟶ `dup([X|Y],[X,X|NewY])`

are all distinct and apply to different programs.

To account for these superficial differences between rewrites, we perform two normalization steps. First, instead of storing the left- and right-hand sides of a rewrite as character strings, we pass them through a lexer and store the corresponding token

sequences. For example, the first rewrite listed above would be represented with the following "before" and "after" token sequences:

```
atom(dup), lparen, lbracket, var(H),          atom(dup), lparen, lbracket, var(H),
pipe, var(T), rbracket, comma,          ⟶     pipe, var(T), rbracket, comma,
lbracket, var(H), comma, var(H),              lbracket, var(H), comma, var(H),
comma, var(DupT), rbracket, rparen            pipe, var(DupT), rbracket, rparen
```

Presenting rewrites with token sequences is rather unwieldy. Since there is a one-to-one correspondence between strings and token sequences, we will keep using the same notation as above to present rewrites. Unless indicated otherwise, all rewrites in this chapter should be interpreted as (pairs of) token sequences.

Comparing token sequences instead of character strings allows us to reliably ignore differences in whitespace. Converting a string to a token-based representation is straightforward and requires only a lexer for the target programming language. Lexing is a context-independent operation that can be performed on individual program fragments without considering other parts of the program.

To deal with the second problem, i.e. different variable-naming schemes, we rename all variables into canonical names. All occurrences of the first variable (on both sides of the rewrite) are renamed to `A`, the second variable to `B` and so on. If the right-hand side of a rewrite introduces new variables, their normalized names are selected so that they do not clash with existing variables on the left-hand side. For example, the rewrite

```
dup(L,L2) ⟶ dup([H|T],L2)
```

becomes

```
dup(A,B) ⟶ dup([C|D],B)
```

Renaming variables in this manner works in any language, as long as rewrites use a token-based representation that tells us which parts of the code correspond to variables (for Prolog we use the token `var(H)`, as in the example above). The same normalization step can also be done for other identifiers in the program, such as function or class names. Here we only rename variables; the vast majority of the introductory programs we deal with involve a single predicate or function with a prescribed name, which is the same in all submissions.

Unlike other, more extensive canonicalization techniques [89, 90], both normalization steps are simple and require only limited language-dependent knowledge (the lexer). Using these steps the three distinct rewrites listed above normalize into the same form:

```
dup([A|B],[A,A,C]) ⟶ dup([A|B],[A,A|C])
```

To improve clarity, we will often use meaningful variable names when giving examples of rewrites, such as `[Head|Tail]` instead of `[A|B]`. However, all examples of rewrites in this chapter should be considered normalized as described here.

3.3 Rewrite rules

By grouping related editing actions, rewrites encode program transformations at a higher level than individual insertions and deletions. The left-hand side of a rewrite also provides context that establishes some limits on its applicability. This context is very superficial, however, and only takes into account the program tokens in the immediate vicinity of the modified fragment. Furthermore, given several rewrites with the same left-hand side we also have no way of knowing which rewrite is most likely to result in a working program.

To address these issues, we annotate rewrites with additional information to form *rewrite rules*. Rewrite rules add structural information to rewrites, describing where in a program each rewrite may be used. They also prioritize rewrites based on how often they were used in student traces.

To understand the first problem – why the limited context provided by the left-hand side of a rewrite can be insufficient – consider the rewrite

```
dup(L,L2) ⟶ dup([Head|Tail],L2)
```

that refines the first argument from a generic variable into a list, consisting of at least one element `Head` and a list of remaining elements `Tail`. While this rewrite brings the program

```
dup(L,L2):-
    dup().
```

closer to a solution (by allowing `Tail` to be passed to the recursive call), it would be incorrect to apply it to the goal in the body of the clause

```
dup([H|T],[H|DupT]):-
    dup(T,DupT).
```

where the actual error is that `H` is not duplicated in the head of the clause.

Rewrite rules use additional structural information to disambiguate these situations. This information is recorded as the path from the root of the program's abstract syntax tree (AST) to the node containing the left-hand side of the rewrite. For example, the following rewrite rule specializes the above rewrite to only apply when the left-hand side matches the head of a clause:

text ▷ clause ▷ head ▷ compound : `dup(L,L2)` ⟶ `dup([Head|Tail],L2)`

For example, we can only apply this rewrite rule to the third line (the head of the second clause, with the AST path text ▷ clause ▷ head ▷ compound) in the following program:

```
dup([],L2):-
    L2 = [].
dup(L,L2):-
    dup(L,DupL),
    L2 = [H,H|DupL].
```

On the other hand, we cannot apply this rule to the fourth line (first goal in the second clause) even though the normalized left-hand side matches that line, because the AST path to that line (text ▷ clause ▷ body ▷ and ▷ compound) does not match the rule. Appendix B gives the Prolog grammar used in this chapter.

Fig. 3.1 shows the (simplified) AST for this program. The figure also shows how the rewrite is applied at the given path: replace the content of the dotted node *a* with the new version *b*. Remaining examples of Prolog rewrite rules in this chapter omit the two initial path elements (text ▷ clause), since they are the same in all programs.

To address the second issue – how to prioritize the application of different rewrites – rewrite rules also assign probabilities to rewrites. These probabilities indicate how often a rewrite was used given its path and left-hand side. For example, each of the rewrites

- head ▷ compound : `dup(L,L2)` ⟶ `dup([],[])`,
- head ▷ compound : `dup(L,L2)` ⟶ `dup([X],[X,X])`, and
- head ▷ compound : `dup(L,L2)` ⟶ `dup([X],[X|X])`

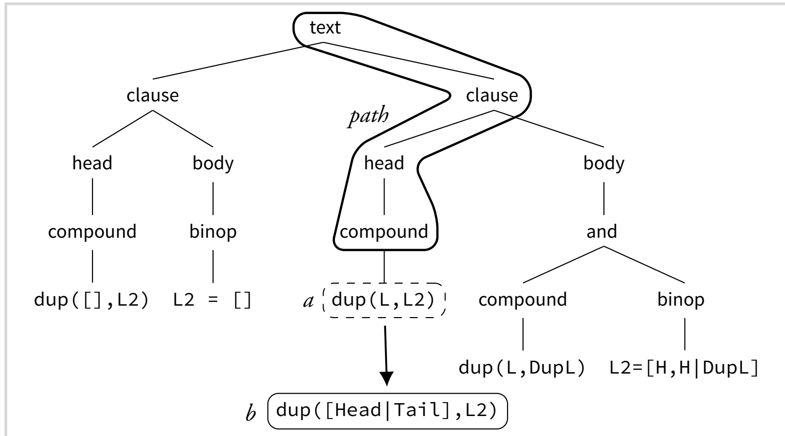

Figure 3.1

Applying a rewrite rule *path* : *a* ⟶ *b* to the student's submission. The original fragment *a* at *path* from the AST root is replaced with the modified version *b*.

refines the generic `dup/2` structure into a different base-case clause. Only the first rule, however, will ultimately result in a working program. It was used most often by students and is thus assigned the highest probability. These probabilities allow us to try the most promising rules first when debugging incorrect programs; section 3.4.3 describes how we assign priorities to rewrite rules.

3.4 *Learning rewrite rules*

This section describes the learning phase of our approach: how rewrites are extracted from student solutions and generalized into rewrite rules. We illustrate the learning algorithm using the example trace shown in Fig. 3.2. This trace describes one student's process of solving the problem `rev/2`, where the task is to write the predicate that reverses a list.

In this example, the student implemented the most common naive recursive solution [57], which reverses the tail `T` of the original list to obtain the new list `RT`, and appends the head (first) element `H` at the back of `RT`. Just as for the `dup/2` problem, this solution contains separate rules for reversing the empty and non-empty lists.

The left-most (thick) line in Fig. 3.2 represents the sequence of actions in the student's trace, beginning and ending respectively with `open` and `close` actions. The four test actions, where the student submitted the program for testing, are shown as points

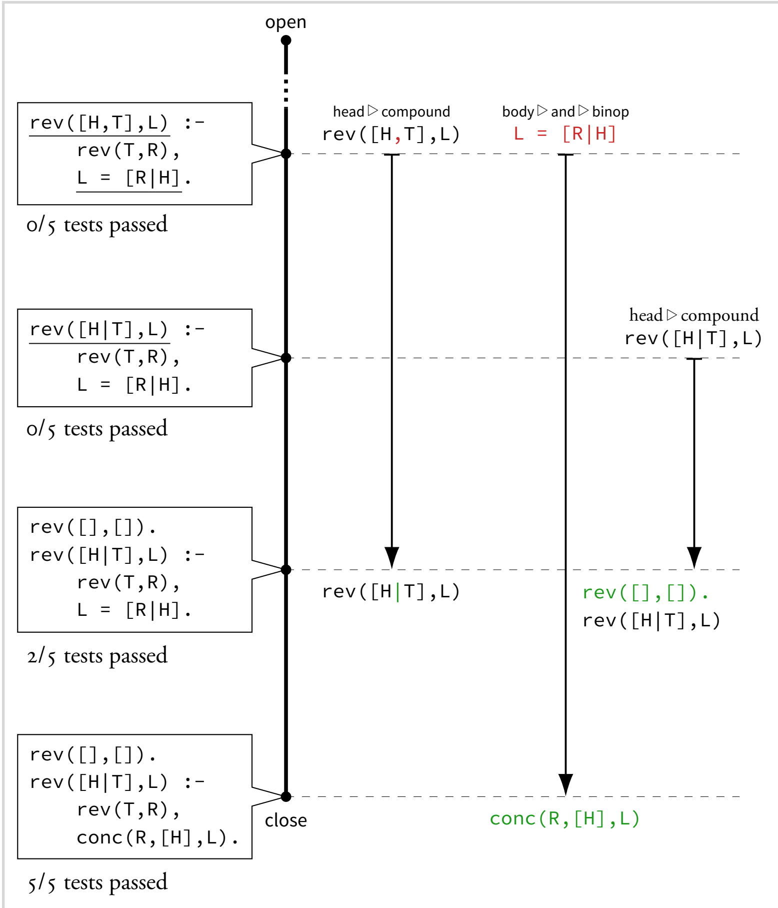

Figure 3.2

Sequence of submissions (in boxes) for an attempt at solving `rev(A,B)`. Tracking changes to the underlined fragments yields the three shown rewrite rules.

on this line, with the corresponding code in boxes to the left. Individual insertion and deletion actions are not shown. The three arrows to the right of the trace line indicate three of the rewrite rules extracted from this trace.

We extract rewrite rules by following modifications to the program code, and tracking how certain code fragments change between significant versions. Here, a *fragment* means any contiguous sequence of tokens in a program. Only the program versions submitted with test actions are considered significant in our implementation. Other possibilities are discussed at the end of this section. Furthermore, we also track only the "interesting" fragments from each incorrect submission. The next subsection describes which fragments are selected for tracking.

3.4.1 Extracting rewrites

The algorithm for extracting rewrite rules from a trace can be conceptualized similarly to sweep line algorithms in computational geometry: we maintain a set of tracked fragments and follow the sequence of actions in the trace, performing certain operations for each action. Specifically, we keep a set F of tracked fragments to follow the evolution of interesting fragments between submissions, and a result set R of extracted rewrites. For every action in the trace we update these sets depending on action type, as follows:

- test: For each "interesting" fragment present in this submission we add a new item (a, path, t, start, end) to F: a is the fragment (token sequence) at path from AST root, spanning character indexes from start to end, and t is the number of tests passed by this submission. We track the fragment's evolution by updating indexes start and end as characters are inserted and deleted.

  For each tracked fragment already in F we check if the current submission passed more tests than the stored value t. If so, we add a new rewrite $\mathsf{path} : \mathsf{a} \longrightarrow \mathsf{b}$ to R, where a is the stored original fragment with AST path, and b is the modified fragment in the current submission (delimited by the updated indexes start and end – see the next item).

  As mentioned, we only add rewrites for submissions that pass more tests than the submission with the original fragment. We found that using this simple common-sense heuristic allows us to find fixes for more programs, mainly by

reducing the branching factor in the debugging algorithm described in Section 3.5.

- insert/delete: For every tracked fragment (a, path, t, start, end) in F we update the indexes start and end that delimit this fragment. If a new character is inserted in front of the fragment, we increment both start and end; if a character is deleted within the fragment, we decrement end; and so on. This allows us to track how a part of the code evolved locally, even when there are changes to other parts of the program.

  The only nontrivial case is when a new character is inserted at the final index end of a fragment: how can we tell whether the new character should be considered as an addition to the original fragment or not? For example, if the initial incorrect fragment was

  ```
  rev([H|T],[T|H])
  ```

  and the student modified it by first deleting everything after the comma:

  ```
  rev([H|T],
  ```

  we should include the subsequent insertions that change the fragment into

  ```
  rev([H|T],R)
  ```

  On the other hand, always extending a fragment when a character is inserted at the end would result in rewrite rules such as

  ```
                          rev([H|T],L):-
  rev([H|T],[T|H])  ⟶        rev(T,RT),
                              conc(RT,[H],L).
  ```

  where a single rewrite inserts the whole clause, which is not useful as a problem-solving step for modeling the programming process. We deal with such situations by only extending the fragment (by incrementing the index end) if the token immediately following the modified fragment is the same as in the original version.

Fig. 3.2 shows three of the tracked fragments (underlined) from the first two submissions. Several other fragments are tracked but omitted in the figure for clarity. The line for each fragment extends from the submission where it is added to F to the submission where the corresponding rewrite is added to R.

For the first rewrite, we start tracking the fragment f=“`rev([H,T],L)`” in the initial program by adding the tuple F_1 = (f, head ▷ compound, t=0, start=0, end=11) to F. As the student modifies the program, we update the indexes start and end accordingly. The fragment is corrected by the second submission, but we do not add the corresponding rewrite because the program still passes no test cases. In the second submission, start and end still have their initial values, since no characters have been inserted before the fragment and its length has not been modified.

Between the second and third submissions, the student inserted the base-case rule (containing 11 characters plus a newline) at the beginning of the program. The start and end indexes are therefore incremented to 12 and 23. Since the third submission passed more tests than the stored value t, we add to R the rewrite

```
head▷compound: rev([H,T],L) ⟶ rev([H|T],L)
```

where the path and left-hand side are the original values stored in F_1, and the right-hand side is the fragment in the current submission delimited by the updated values start and end. This rewrite fixes a common mistake of using the wrong operator to construct a list.

The second (middle) rewrite in Fig. 3.2 is extracted in the same way, except that this rewrite is only added to the result set R in the final submission, because the original fragment “`L = [R|H]`” remains unchanged during the first three submissions.

By directly tracking all insertions and deletions, we are able to follow modifications to each part of the program independently, allowing for overlapping fragments that modify the same part of a program. One such example is the third (rightmost) rewrite

```
head▷compound: rev([H|T],L) ⟶ rev([],[]). rev([H|T],L)
```

for which tracking starts with the fragment “`rev([H|T],L)`” – an already modified version of the fragment tracked by the first rewrite.

The third rewrite represents the step of adding a base case to the program. This and other rewrites that only insert new text could also use an empty left-hand side; however, having some context is still useful and allows us to determine where in the program it

makes sense to perform such insertions. In this case, inserted text is anchored to the head of the following clause.

As mentioned above, we could add rewrites for every submission, regardless of how many tests it passes, or even for program versions between submissions. We found, however, that doing so yields many useless rewrites that are more likely to break a program further than fix it. By only considering improved submissions we increase the probability that discovered rules will be useful for debugging. That said, the number of passed test cases is only a rough measure of correctness. Finding appropriate "checkpoints" at which to consider rewrites is an interesting topic that would benefit from further research.

3.4.2 Selecting fragments

When encountering an incorrect submission, one option would be to simply add every fragment to the set F of tracked fragments. However, a program with n tokens contains $\binom{n+1}{2}$ nonempty fragments, and tracking every possible fragment would result in a huge catalog of rewrite rules, with many "nonsensical" rewrites like

```
,A,B),B=  ⟶  ,[A],B),B=
```

While such rewrites can be used for debugging, a large rule catalog means a large branching factor, slowing down the search for a correct program. More importantly, our goal is to find meaningful transformations that can give us some insight into the programming process. For instance, the left-hand side of the rewrite

```
conc(A,B,C)  ⟶  conc(A,[B],C)
```

provides a more meaningful context (a complete compound term in Prolog), and describes the same modification much better. Instead of tracking all fragments from each submission, we therefore select only such "interesting" fragments.

In our first attempt we used only individual lines of code as fragments [91]. This is easy to implement, but not very robust. Beginners often do not conform to the suggested coding style, writing multiple goals or even entire clauses in the same line, or breaking lines in unusual places. Line-based fragment selection also makes it difficult to limit rewrites to specific places in the program's structure.

For these reasons we added the constraint that each selected fragment should represent a complete syntactic unit. In other words, we consider only fragments that correspond to subtrees of certain non-terminal nodes in the program's AST. In Prolog

programs we select the fragments representing the head of a clause and the goals in its body. Specifically, we track fragments defined by subtrees rooted at symbols `clause`, `head`, `body`, `and`, `or`, `compound`, `binop` (binary operator) and `unop` (unary operator). Besides the parser, this is the only additional language-specific information required by our method.

Using the AST to select fragments means that we can only consider syntactically correct submissions. While this places an additional limit on the versions of the program we can consider when extracting edits, we have found that this is not a significant limitation in practice. Syntax errors are not very common after the introductory exercises, and can be resolved using error messages from the interpreter (perhaps augmented with additional explanations more suitable for beginners).

On the other hand, having an AST allows us to add structural information to rewrite rules. This is the *path* component of a rule, which allows the debugging algorithm to only apply each rewrite rule in the same context in which it was learned.

3.4.3 Rewrite probabilities

Once we have extracted rewrite rules from all traces for a problem, we associate a probability with each rule to guide the debugging algorithm described in the next section. This probability describes how likely a rule $path : a \longrightarrow b$ is used in a program that contains the fragment a at *path* from the root of program's AST.

Specifically, we calculate the conditional probability of using a rule $path : a \longrightarrow b$ when the program contains the fragment a (the AST path must also match, but we omit it here for clarity) as:

$$p(a \to b|a) = \frac{\text{\# of traces containing } a \to b}{\sum_x \text{\# of traces containing } a \to x}.$$

We wish to avoid assigning very high or very low probabilities to rewrite rules. If the probability of a rewrite is too low, it will rarely or never be attempted during debugging; on the other-hand, very high-probability rewrites can prevent less common alternatives from being explored. For this reason we compress the range of probabilities using the logistic function with steepness $k = 3$ and the average probability $\bar{p} = \text{avg}(p)$ as the midpoint. We calculate the final value p' for each probability p as:

$$p' = \frac{1}{1 + e^{-k(p-\bar{p})}}.$$

This moves the extreme values of p closer to the average, while leaving other values mostly unchanged. The function and parameters were chosen ad hoc; they performed well in our evaluations, but better options might exist.

For each problem we thus obtain a catalog of rewrite rules and associated probabilities. The next section explains how these rules can be used for debugging student programs.

3.5 Debugging

We formalize the task of debugging an incorrect program P_0 as a search for a sequence of rewrites. Algorithm 1 outlines the procedure. We keep a priority queue of generated programs, initially containing only the original incorrect program P_0. In every iteration we test the highest-priority program P in the queue; if it is correct, we return it along with the corresponding sequence of rewrites transforming P_0 into P. Otherwise, we use applicable rewrite rules to generate new programs from P and add them to the queue.

Algorithm 1

Debugging with rewrite rules.

Input: incorrect program P_0, catalog of rewrite rules R
Output: correct program with associated rewrite sequence

```
let Q be the empty priority queue
let S_0 be the empty rewrite sequence
add (P_0, S_0) with priority 1.0 to priority queue Q
while Q not empty do
    pop (P, S) with highest priority c from Q
    if P is correct then
        return (P, S)
    for all r ∈ R do
        if rule r is applicable to P then
            apply r to P to get new program P′
            append r to S to get new rewrite sequence S′
            add (P′, S′) to Q with priority c ∗ p(r)  /* defined in Section 3.4.3 */
```

Essentially, our method performs a best-first search guided by rewrite-rule probabilities. Specifically, we define the "probability" of a sequence of rewrites $r_1 r_2 \ldots r_n$ as the product of the probabilities of individual rewrites:

$$p(r_1 r_2 \ldots r_n) = \prod_{i=1}^{n} p(r_i).$$

We use these probabilities as priorities of candidate programs in the queue in order to first visit programs generated using likelier sequences of rewrites. This heuristic is based on the assumption that rewrites that were used in more traces are more likely to reflect successful problem-solving strategies. By using the product of probabilities, we also implicitly prefer shorter rewrite sequences.

In general there is no guarantee that the process described by Algorithm 1 will finish in a certain amount of time – depending on the catalog of rules it might keep generating new candidate programs indefinitely, without finding a correct solution. To properly call it an algorithm would require some terminating condition, for example by disallowing p from falling under a certain value. This could be problematic in an interactive application, where we instead terminate the search after some time if no solution has been found.

3.5.1 Evaluation

We have tested the rewrite-rule-based debugger on six groups of introductory Prolog problems (50 in total) that cover all basic features of the language. For each problem we randomly divided the set of student problem-solving traces into training and testing sets in the ratio $70 : 30$, and extracted rewrite rules from the traces in the training set.

We then evaluated the debugger using those rewrites on the incorrect submissions from the testing set. We only attempted to debug the incorrect programs before the first correct submission from each trace. For example, in a trace containing the sequence of submissions $S_1 \ldots S_c \ldots S_n$, where S_c is the first correct submission, we only attempted to debug submissions $S_1 \ldots S_{c-1}$. This is because after finding one solution, students often try out other approaches and sometimes even add unrelated code; we are primarily interested in helping them achieve that first working version. We also excluded submissions containing syntax errors, which can be handled by the interpreter. Finally, we only considered each distinct submission once per trace, even if a student submitted the same program multiple times.

For this and subsequent experiments in this dissertation we used an ordinary desktop computer with a 3 GHz Intel Core 2 Duo processor and 4 GB of memory. In this evaluation we stop debugging each program if no solution has been found after 10 seconds; a longer timeout would make it impractical for real-time use in a programming tutor.

Tables 3.2 and 3.3 summarize the results. The first row for each problem group gives the total statistics, followed by a row for every problem in that group. In each row, the "Traces" column shows the number of traces (one per student) in the testing set. The two columns under the "Submissions" heading show the number of considered incorrect programs in those traces, and how many of those programs our debugger was able to fix. The final two columns show the average time and the number of tests (i.e., how many candidate programs were tested) required before finding a solution.

Overall we are able to fix between one and two thirds of submitted incorrect programs. Results vary between problems, with success rate generally dropping with increasing program complexity. The main reason for this is the fact that we need to find the complete sequence of rewrites required to fix an incorrect program; in complex programs multiple rewrites may be needed to remove all errors, with exponentially growing search space.

The average time to solution shows that, for the programs we are able to fix, we can usually do so very quickly. This can also be seen from the "Tests" column, showing the number of programs (generated by applying different sequences of rewrites) that had to be tested before finding a solution. The last column similarly shows that most submissions are fixed by a short sequence of one or two rewrites. These results indicate that rewrite rules work well for debugging programs with few bugs, but are not very efficient for programs that are far from the correct solution.

The lowest success rate was for the `powerset/2` problem, where a fix was found for only one out of 15 incorrect submissions. This problem can be solved easily using the `findall` meta-predicate. However, most students submitting incorrect programs first attempt to solve the problem with a recursive predicate and fail. These attempts are very different from the final solution, so no good rewrite rules can be learned.

The debugger will usually fail to complete a recursive program containing only the base case, such as the following submission for the `rev/2` problem:

```
rev(A,A) :- A = [].
```

This is partly because we limit the amount of new code a single rewrite can add –

Table 3.2

Evaluating the rewrite-based debugger in Algorithm 1 on incorrect student submissions for Prolog exercises in the *Family relations*, *Lists I* and *Lists II* groups. See Table 3.3 on the next page for a description of the data.

		Submissions				
	Traces	Incorrect	Fixed	Time [s]	Tests	Rewrites
Family relations	1119	1081	749	1.05	5.2	1.2
ancestor/2	126	145	100	0.92	5.3	1.2
aunt/2	130	113	83	1.24	5.4	1.2
brother/2	134	65	47	0.36	2.6	1.3
cousin/2	119	232	139	1.72	7.7	1.1
descendant/2	124	94	66	0.85	5.0	1.2
father/2	75	18	10	0.19	1.5	1.2
grandparent/2	136	65	38	1.33	11.9	1.6
mother/2	150	64	31	0.44	3.7	1.4
sister/2	125	285	235	0.91	3.6	1.1
Lists I	836	1714	824	1.73	8.3	1.4
conc/3	117	348	148	1.54	6.6	1.3
del/3	118	257	160	1.91	9.9	1.4
divide/3	96	255	102	1.39	4.6	1.2
dup/2	102	313	155	1.61	8.0	1.5
insert/3	126	185	111	1.78	10.6	1.6
last_elem/2	51	37	9	2.02	10.3	1.3
memb/2	129	100	65	1.67	8.6	1.8
permute/2	97	219	74	2.36	9.9	1.2
Lists II	978	1751	799	1.63	8.2	1.4
{even,odd}len/1	98	169	75	2.27	12.1	1.3
len/2	101	96	61	0.97	6.4	1.4
max/2	103	94	31	2.09	11.4	1.5
min/2	90	236	59	1.84	7.1	1.3
palindrome/1	96	184	129	2.17	10.4	1.3
rev/2	104	348	127	1.21	5.0	1.3
shiftleft/2	105	165	102	0.60	4.0	1.2
shiftright/2	100	127	33	1.51	7.7	1.3
sublist/2	78	243	139	2.36	11.9	1.6
sum/2	103	89	43	0.63	3.5	1.3

Table 3.3

Evaluating the rewrite-based debugger in Algorithm 1 on incorrect student submissions for Prolog exercises in the *Sorting*, *Sets* and *Trees* groups. The *Traces* column shows the number of traces in the testing set for each problem. The two columns under the *Submissions* heading show the total number of incorrect submissions in those traces, and the number of those programs we were able to fix. The final three columns show the average time, number of generated programs and rewrites necessary to debug fixed programs. For each problem group the total (for the first three columns) or average (for the last three columns) is given.

		Submissions				
	Traces	Incorrect	Fixed	Time [s]	Tests	Rewrites
Sorting	510	1228	456	1.60	6.9	1.3
is_sorted/1	96	257	174	1.54	8.6	1.3
isort/2	87	213	104	1.08	4.8	1.4
pivoting/4	79	194	27	1.77	5.0	1.3
quick_sort/2	79	221	65	1.99	5.1	1.4
sins/3	86	292	60	2.57	8.5	1.4
slowest_sort/2	83	51	26	0.74	7.2	1.3
Sets	663	1015	315	1.72	8.2	1.2
count/3	87	263	91	1.99	9.9	1.2
diff/3	85	101	37	1.89	7.1	1.1
intersect/3	83	171	63	1.57	5.9	1.3
is_subset/2	86	37	9	2.26	6.8	1.1
is_superset/2	85	139	55	1.04	5.0	1.1
powerset/2	75	15	1	4.42	15.0	3.0
subset/2	77	143	38	1.94	15.7	1.4
union/3	85	146	21	1.75	5.0	1.1
Trees	649	1582	566	1.54	6.9	1.4
deletebt/3	62	195	18	2.88	8.0	1.3
depthbt/2	81	114	43	1.71	8.6	1.4
insertbt/3	61	73	28	1.32	6.5	1.4
maxt/2	51	111	11	1.95	6.4	1.2
memberbt/2	91	131	96	1.45	7.0	1.5
membert/2	62	251	27	2.21	8.1	1.4
mirrorbt/2	73	284	141	1.63	7.6	1.3
numberbt/2	86	204	110	1.50	7.4	1.5
tolistbt/2	82	219	92	1.00	4.2	1.3

otherwise, a rewrite could simply insert the whole correct rule in one step, which would not be useful for generating hints. Additionally, each step must result in a syntactically correct program so that the next rewrite can be applied. Rewrite rules are thus not suitable for generating programs from scratch.

Another type of problematic submissions are programs like the following implementation of the list-concatenation predicate conc/3:

```
conc(L1,L2,L):-
    L1 = [X|L3],
    L4 = [X|St],
    St = L4,
    L1 = L3,
    L = L2
    ;
    St = [X|L6],
    St = L6,
    L7 = [X|L],
    L = L7.
```

With great effort, a teacher might be able to divine the student's intent in such cases, but the program is so far removed from the solution (and any other incorrect submission) that there is little hope for automated methods. Students that encounter such difficulties are especially prone to tinkering, and often submit many small variations of the same incorrect code.

Another common submission is the empty program (or only the base case of a predicate), which is obviously incorrect – perhaps students wish to see how many test cases there are. In both cases – not enough code or too much incorrect code – debugging often fails, especially for more complex problems.

It is interesting to note that a significant percentage (about a half overall) of incorrect submissions are actually close to a solution and can be fixed by a single rewrite. This agrees with our observation that, unlike many other tutoring domains, solving programming problems typically proceeds through two distinct stages. First, a student writes the initial version of the program, which is more or less complete but may contain errors. This stage is relatively short and serves only to "load" the student's initial concept of the solution into the editor.

The main problem-solving activity takes place during the second stage, in which the student locates and removes errors in the initial program. The rewrite-based approach

can work well in these cases, as long as the program is not too far removed from a solution. Experimental data shows that this is indeed often the case. Our debugger can thus still be useful in many cases. We discuss the possible methods for providing feedback based on rewrite rules in the next section.

3.6 Generating hints

We have explained how a catalog of rewrite rules can be learned in a (mostly) language-independent manner, and how these rules can be used to debug incorrect programs by searching for appropriate sequences of rewrites. To use discovered rewrites in a programming tutor, we must turn them into an appropriate message for the student. This section outlines two possible approaches: automatically highlighting erroneous fragments in the student's code, and using common rewrite sequences to aid the authoring process for teacher-provided feedback.

The hints described below are likely not optimal in terms of improving the learning process. Our main intent here is to demonstrate that rewrite rules are a feasible basis for generating data-driven feedback in a programming tutor.

3.6.1 Automatic feedback

Automatic feedback could be provided by simply showing the rewrites required to fix an incorrect program. Showing the solution is called a *bottom-out* hint, which should only be used as the last resort for students that are unable to solve the problem otherwise. When such hints are available, students often "game the system" by repeatedly requesting help from the tutor until a bottom-out hint is shown [92].

We instead wish to help students practice their debugging skills by pointing them in the right direction without revealing the solution. We do this by highlighting incorrect or missing fragments in the program, based on the rewrite sequence found by the automatic debugger. This should allow the student to focus their analysis on the critical parts of the program.

Consider for example the following incorrect implementation of the list-reversal predicate `rev/2`:

```
rev([Head|Tail],Reversed):-
    rev(Tail,RevTail),
    conc(RevTail,Head,Reversed).
```

The automatic debugger fixes it with two rewrites:

1. `rev([A|B],C)` ⟶ `rev([],[]). rev([A|B],C)`
2. `conc(A,B,C)` ⟶ `conc(A,[B],C)`

Given a list of rewrites we highlight the modified fragments in the original program. Instead of simply highlighting the entire left-hand side of each rewrite, we extract only the differences between the two versions using Python's implementation of Ratcliff and Metzener's diff algorithm [93]. We distinguish three cases and mark them with different colors: inserting (green), removing (red), and modifying (yellow) fragments. The above program is annotated as follows:

```
rev([Head|Tail],Reversed):-
    rev(Tail,RevTail),
    conc(RevTail, Head ,Reversed).
```

For rewrites that only add new code, green highlights ( ) indicate the positions where new fragments (in this case a new clause) should be inserted. If a rewrite inserts two fragments near each other – like the second rewrite above that places a bracket on either side of the variable `Head` – we use a single "modify" highlight.

Another example highlights the incorrect base case (where the list with one element `X` should be replaced with the empty list) and the incorrect operator (`=` instead of `is`) in the `sum/2` predicate:

```
sum([ X ], X ).
sum([H|T],Sum) :-
    sum(T,S),
    Sum = S + H.
```

We have implemented rewrite-based automatic hints in CodeQ and evaluated them in the classroom. Section 5.2.2 presents the results of that evaluation. Here we note two possible improvements to how hints are presented. First, red highlights are too informative – it would be better to use the yellow ("modify") highlight instead for those cases. Second, we highlight all modified fragments at once, since we generally cannot know whether two rewrites are related to the same error (so basing highlights on a single rewrite might be misleading). The pattern-based error model discussed in the next chapter avoids both issues.

3.6.2 *Manual feedback*

This section presents another application of rewrite rules: assisting the instructor when manually authoring feedback, by enumerating typical errors and selecting sets of relevant incorrect programs. The idea is to take a set of incorrect programs fixed by Algorithm 1 and group them according to the sequence of rewrites that was required to fix them.

For example, to define the `permute(List,Permuted)` predicate, which generates permutations of a list through backtracking, one should recursively permute the tail of a list and then successively insert the head element at every possible position. Many students do this incorrectly, using `[H|TP]` to only prepend the head element `H` to the permuted tail `TP`. The following rewrite rule fixes this mistake:

```
and▷compound : permute(T,TP), P = [H|TP] ⟶
                permute(T,TP), insert(H,TP,P)
```

This correction represents one of the most common error classes for this problem. Along with the rewrite we can present the instructor with several examples of incorrect submissions, with the erroneous fragment highlighted:

```
permute([],[]).
permute([H|T],P):-
    permute(T,TP),
    P = [H|TP].
```

```
permute([],[]).
permute(L,P):-
    [H|T] = L,
    permute(T,TP),
    P = [H|TP].
```

```
permute(L,L).
permute(L,P):-
    L = [H|T],
    permute(T,TP),
    P = [H|TP].
```

Even though the debugging algorithm has no concept of what the actual error is, we can use it to produce examples of incorrect programs that contain this error, and rewrites to fix it. From this, the instructor can easily see the misconception behind the mistake and provide an appropriate explanation, for instance: "Reinserting the element at the beginning in each recursive step will leave the list unchanged" or "Try inserting the element at other locations in the list".

To test this approach in practice, we manually predicted common error classes for two introductory sets of problems – *Family relations* and *Lists* – based on our experience teaching Prolog. For each problem we then analyzed the ten largest groups of incorrect programs obtained as described above. We first describe manually predicted and automatically discovered errors for two selected problems, and then present overall results.

Example: sister/2

One of the introductory problems in the Family relations set is the `sister(A,B)` predicate, defining the relation "`A` is a sister of `B`" and typically written as

```
sister(A,B):-      % A is B's sister when:
  parent(P,A),     %   A and B share a common parent P,
  parent(P,B),
  female(A),       %   A is female, and
  A \= B.          %   A and B are not the same person.
```

For this program, an experienced Prolog instructor predicted five error classes, all of which have later been observed in many student submissions:

1. "`A` and `B` must share a parent"
   A call to `parent/2` is missing or has wrong arguments.
2. "`A` must be female"
   A call to `female/1` is missing or has wrong arguments.
3. "`B` may be of any gender"
   There is an incorrect call to `female(B)`.
4. "`A` and `B` must be different"
   The comparison `A \= B` is missing.
5. "the `\=` operator used too early"
   `A` and `B` must be instantiated before they can be compared.

The question is: can these error classes be induced from the rewrites and example programs returned by our method? As it turns out, they can. For example, the third error is represented by the following rewrite:

```
and▷compound : female(A), female(B) ⟶ female(A)
```

Other rewrites (with corresponding examples of incorrect programs) may sometimes indicate the same error. In this case, one such rewrite is

```
and▷compound : parent(P,A), parent(P,B), female(B) ⟶
               parent(P,A), parent(P,B), female(A)
```

This rewrite corresponds to the second and third items in the list. In fact we have discovered, with the help of our method, a new and much more typical mistake of incorrectly interpreting predicate argument order: `A` is a sister of `B` and not vice versa. Showing such examples to the instructor would be beneficial when enumerating the error classes for each problem. The next rewrite gives another example of incorrect argument order:

```
and▷compound : parent(A,P), parent(B,P) ⟶
               parent(P,A), parent(P,B)
```

Some errors require several rewrites to fix. The last item in the above list of five common errors is represented by a sequence of two rewrites, removing the offending goal and then inserting it at a later point in the program:

```
1. and▷compound : A \= B, parent(P,A), parent(P,B) ⟶
                   parent(P,A), parent(P,B)
2. and▷compound : parent(P,A), parent(P,B) ⟶
                   parent(P,A), parent(P,B), A \= B.
```

Example: sum/2

As another example take the `sum(List,Sum)` predicate from the *Lists* problem set, which is most commonly written as

```
sum([],0).
sum([H|T],S) :-
    sum(T,ST),
    S is ST+H.
```

and for which the instructor manually predicted the following four common errors:

1. "base case with non-empty list"
2. "incorrect base case"
   We anticipated the incorrect base case `sum([],_)`, but it was only observed in two submissions. Examples returned by our method revealed the much more common error `sum([],[])`.
3. "using the operator `=` instead of `is`"

4. "the `is` operator used too early"
   Similar error as in the `sister(A,B)` example; it was detected in the same way.

Each of these errors can easily be induced from the rewrites and example programs returned by our method. For instance, the appropriate rewrite for the third error is:

```
and▷compound : S = ST+H ⟶ S is ST+H
```

We have additionally discovered two significant new error classes, both concerning incorrect arithmetic operations. The first is an incorrect attempt to "update" the value of a variable in Prolog, which likely stems from students' prior familiarity with imperative programming languages:

```
and▷compound : sum(T,S), S is S+H ⟶ sum(T,ST), S is ST+H
```

The other error not predicted by the instructor results from poor understanding of the `is` operator semantics:

```
and▷compound : ST is S+H ⟶ S is ST+H
```

Evaluation

In both cases above we have been able to automatically discover all manually predicted error classes, simply by grouping incorrect programs according to rewrites needed to fix them. We did the same for 19 other problems from the first three problem groups. Table 3.4 shows the results.

The first two columns show the number of manually predicted error classes and the number of those error classes actually observed in student submissions. The last two columns show the number of errors found by analyzing automatically selected groups of programs, divided into errors matching one of the manually predicted classes, and errors not predicted by the instructor (i.e., newly discovered from the results of our method).

In total we were able to discover over 70% of manually predicted errors. For many problems our method also discovered one or more error types that were not predicted manually; altogether 27 new errors, that is a 37% increase over the number of manually defined errors. Equally important, about a quarter of predicted errors have occurred only rarely or never, meaning that a significant part of the authoring effort could have been avoided.

Table 3.4

Number of errors predicted manually, or induced from automatically selected groups of incorrect programs. The first column gives the number of error classes predicted by the instructor, and the second column gives the number of those errors actually observed in student submissions. The third column gives the number of predicted error classes that were also found using the method described in Section 3.6.2, and the last columns gives the number of newly discovered error classes (not predicted by the instructor).

	Predicted		*Found*	
	Total	Seen	Predicted	New
Family relations	42	35	28	6
ancestor/2	6	5	3	0
aunt/2	8	6	4	2
brother/2	5	5	5	1
cousin/2	5	5	4	0
descendant/2	6	4	2	0
grandparent/2	7	5	5	2
sister/2	5	5	5	1
Lists I	25	14	8	6
conc/3	7	1	1	1
del/3	5	4	2	0
divide/3	5	3	2	1
dup/2	5	3	2	1
permute/2	3	3	1	3
Lists II	33	24	16	15
len/2	4	2	2	2
max/2	5	4	1	2
min/2	5	5	1	2
palindrome/1	3	2	2	1
rev/2	6	4	4	1
shiftleft/2	3	3	3	0
shiftright/2	2	1	0	3
sublist/2	1	1	1	2
sum/2	4	2	2	2
Total	100	73	52	27

Certain predicted error classes turned out to be somewhat non-specific, such as "`X` must have a parent" and "`X` need not be a parent" in the *Family relations* group problem. Such errors are typically caused by incorrect argument order in the `parent(X,Y)` goal, and the instructor-provided hint was not very helpful for debugging. Using examples produced by our method we would instead have defined an "incorrect argument order" error, which better captures the meaning of such mistakes.

A text-based approach has some inherent limitations, but it is conceptually simple and does not require any language-specific knowledge beyond a parser. While classifying errors is only the first step when developing a domain model for a tutoring system, it can be just as time-consuming as devising a way of detecting errors and writing feedback. A method that automatically returns common error classes can thus save significant teacher effort.

3.7 Future directions

While we have shown that debugging with rewrites works, there is a lot of room for improvement. When ordering applicable rewrite rules during the search, we could take into account other features of the program besides the probability of a rewrite given its left-hand side. These features might include other fragments in the program, and structural features such as the number of clauses and variables in the program. We could then use reinforcement learning to find a "bug-fixing policy" based on these features. This would greatly expedite the search by first considering actions (rewrites) with the highest expected return.

Another approach would be to learn rewrites with a deep neural network, which would learn to combine individual characters, tokens, expressions and so on in higher layers. If successful, this method would have a number of benefits. Since each program can be considered separately, there would be no need to track modifications across successive submissions. The model would allow us to discover both syntactic and semantic errors and would not depend on a parser. Recurrent neural networks using the long short-term memory architecture have been successful in natural language processing, so there is a reason to believe they are also applicable to the programming domain.

One important difference between natural and programming languages is robustness: while using one incorrect word or character will usually not render a sentence incomprehensible, it will almost surely render a program incorrect. Recent experiments with recurrent neural networks have shown, however, that learning successful language

models is plausible for both natural [94] and programming [95, 96] languages.

These approaches require significant amounts of learning data – more than we have collected with CodeQ so far. One option would be to use freely available code from online repositories. It is unlikely that a model build from such disparate programs would allow us to discover conceptual errors related to specific programming exercises. It could however serve to find common bugs in a given programming language, like the tools for static code analysis.

Another potential solution is to generate learning examples automatically. This could be done in a language-dependent manner with predefined semantics-preserving program transformations [89], or by applying known "bad" rewrites to introduce errors in other programs. Generative adversarial networks could also be used to generate useful new learning examples to improve classification accuracy [97].

4

Code patterns

The previous chapter described rewrite rules we have used to model the programming process. Rewrite rules are conceptually simple and have been used successfully for both automatic hint generation and to support the authoring process in a programming tutor. We have implemented automatic feedback based on rewrite rules in our programming tutor CodeQ, and show in Section 5.2.2 that such hints can have a significant effect in the classroom.

While developing that model we encountered certain problems that motivated us to look at other options for data-driven programming feedback. First, the debugger does not scale very well: in order to generate feedback, it needs to find a complete sequence of rewrites to fix an incorrect program. We cannot predict in advance how long this will take, or whether a solution will be found. Since testing generated programs is a processor-intensive task, this presents a significant scaling issue for CodeQ, which uses a central server to process hints.

Furthermore, rewrite-based debugging is an "all or nothing" affair: no feedback can be provided unless we find the complete sequence of rewrites to fix a program. While we could simply offer the student a list of some of the applicable rewrites as "coding suggestions", there is no guarantee that any individual rewrite – no matter how commonly it was used in the past – would be useful, and might very well lead the student down a wrong path. What is missing is a simple way of finding the incorrect parts in a program, without necessarily knowing the exact steps required to fix it.

The most important realization, however, is that simulating the problem-solving process might not be necessary (or indeed even make much sense) in programming domains. Observing human tutors in the classroom we see that, even though they have little or no information on how a specific incorrect program evolved, they are usually able to quickly pinpoint the error – in code and in students' understanding. Unlike other domains such as deriving logic proofs or solving physics problems, where the steps taken are as important as the answer, a solution to a programming problem already encodes all the necessary "steps" for solving it.

For these reasons we have developed an alternative, *static* or *solution-oriented* model for describing typical bugs and solution strategies for programming problems. We define *patterns* in abstract syntax trees (ASTs) and use them as features to learn classification rules for distinguishing between correct and incorrect programs. Induced rules are easily comprehensible and can be interpreted as common bugs and solution strategies. As with rewrite rules we show how they can be used both directly for generating

feedback, or for assisting the authoring process when building a programming tutor.

Before describing our model, let us briefly mention tools for static code analysis, used to detect common errors in a particular programming language [98–100]. They contain extensive knowledge of the target language, but can only discover generic (not problem-specific) bugs, such as dereferencing a null pointer in C. Without a formal problem specification, bugs in the program's logic cannot be found. Conversely, our goal here is to automatically discover problem-specific mistakes without relying on language-specific knowledge or a formal specification of each problem.

Tools for static code analysis focus on discovering errors in programs. *Code smells*, first defined by Beck and Fowler [101], indicate instead program structures that are correct but should be refactored. Examples include long methods or classes, duplicated code, and overly-general abstract classes. Several tools have been developed to automatically discover code smells, typically using a set of detection rules based on various software metrics [102]. These rules are defined by hand and are language-specific. While we implemented code patterns primarily to discover errors, it should be possible also to use them as attributes for specifying code smells.

4.1 AST patterns

The main challenge when building a data-driven model is finding appropriate invariant features in programs that could support machine learning. As noted in the introductory chapter, the programming domain presents a particular challenge due to high variability of student solutions. What is needed is some way of capturing only those parts of the program that are relevant to the mistake we wish to describe, while ignoring unimportant code variations.

For this purpose we define *AST patterns* that describe relations between different parts of the program's AST. In this section we explain AST patterns on several examples, and then explain how patterns are extracted from student programs in the next section. Finally we show how, like rewrite rules, patterns can be used to produce automatic feedback or assist the authoring process.

AST patterns are inspired by Tregex [103] and trx [104], two languages extending regular expressions to tree structures. While Tregex is primarily used in the field of natural language processing to query text corpora for sentences with a given structure, we have used it to describe interesting substructures in a program's AST. Initially we

used the original Tregex syntax[1] to specify patterns, but have later replaced it with the much simpler version.

An AST is an ordered rooted tree: the order of children of each node is fixed. AST patterns describe relations between nodes in such trees. Just as an ordinary (string) regular expression is again a string, an AST pattern is again an ordered rooted tree. In this chapter we use the S-expression notation to denote trees. For example, $(a\ b\ (c\ d))$ denotes a tree with the root a and two child nodes b and c (in that order), where the node c has one child d.

The patterns we use here encode (only) the following two relations between nodes in an AST: "node a is an ancestor of b", and "a precedes b in a depth-first tree walk". Each edge $a \rightarrow b$ in the pattern means that any matching tree must contain a path from a to b. Each pair of sibling nodes a and b (in that order) in a pattern means that a must precede b in a depth-first walk through any matching tree. With these two relations we can encode AST structures we are interested in – described in Sec. 4.2 – with sufficient precision.

When interpreted as a pattern, the tree $(a\ b\ c)$ thus means that the nodes b and c are *descended from* a, and that b *precedes* c in a depth-first tree walk. Formally, an AST matches the pattern (`name` $p_1\ \ldots\ p_k$) if the AST 1) contains a node n labeled `name` and 2) the subtree rooted at n contains, in depth-first order, distinct nodes n_1 to n_k matching subpatterns p_1 to p_k. The next section shows several examples of AST patterns.

4.1.1 Examples

Regardless of the language, most programming is about manipulating data. Almost every line of any program will involve – access or modify – at least one variable or literal. The kinds of patterns selected for this study reflect that observation. We describe our patterns on two programs. First, consider the Prolog program implementing the relation `sister(A,B)`:

```
sister(A,B):-     % A is B's sister when:
  parent(P,A),    %   A and B share a common parent P,
  parent(P,B),
  female(A),      %   A is female, and
  A \= B.         %   A and B are not the same person.
```

[1]Described in the TGrep2 manual, available at https://tedlab.mit.edu/~dr/Tgrep2/tgrep2.pdf.

Figure 4.1 shows this program's AST with two patterns overlaid. The pattern drawn with blue dotted arrows encodes the fact that the first argument to the `sister` predicate also appears in the call to `female`. In other words, this pattern states that `A` must be female to be a sister. We write it as the S-expression

```
(clause
 (head (compound (functor 'sister') (args var)))
 (compound (functor 'female') (args var)))
```

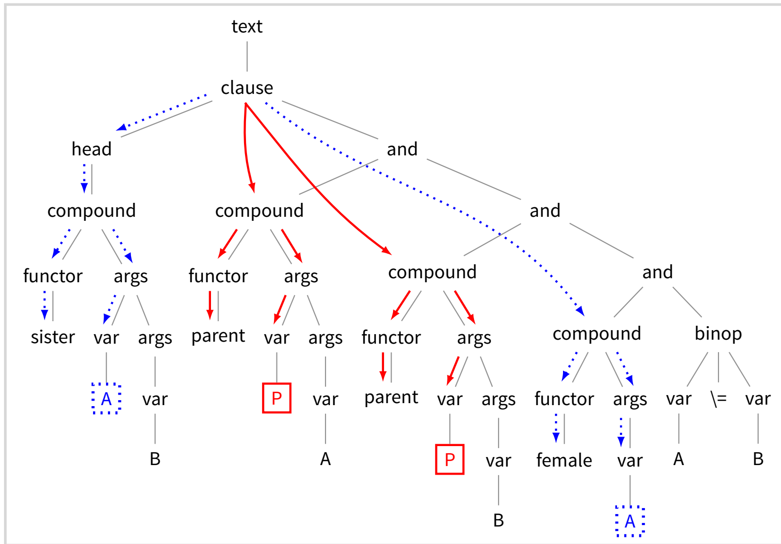

Figure 4.1 The AST for the `sister` program, showing two patterns and the leaf nodes inducing them. The solid arrows equate the first arguments in the two calls to `parent`. The dotted arrows encode the necessary condition that `A` must be female to be a sister.

We only consider Prolog patterns with the same basic structure – describing paths from a clause node to one or two leaf nodes containing variables or values. All patterns in Figs. 4.1 and 4.2 are induced from such node pairs. We regard these patterns as the smallest units of meaning in Prolog programs: each pattern encodes a syntactic relation between two objects in the program (i.e., a path from one variable or value to another).

Other kinds of patterns might also be useful; for example, patterns relating all instances of a variable or a function symbol in a program. However, generating hints from such patterns as described in Section 4.4 would be difficult, since we would not be able to tell which parts of the erroneous pattern are the most relevant. Patterns that

relate only two objects in a program are almost always easy to interpret in terms of conceptual errors for the given exercise, and can provide better support for both manual and automatic analysis.

The patterns we use here therefore contain at most two `var` nodes, and we require they both refer to the same variable; relating two nodes with different variables would not tell us much about the program. This allows us to omit actual variable names from patterns, so that the same pattern can cover programs using different variable-naming schemes.

When extracting patterns we include some local context with each leaf node, for example the predicate name (e.g. `parent` or `sister`) in `compound` nodes; without this context patterns could not distinguish between e.g. `parent(X,…)` and `sister(X,…)`.

We handle certain syntactic variations by omitting some nodes from patterns. For example, by not including `and` nodes, the above pattern can match a clause regardless of the presence (and order) of other goals in its body (in other words, the pattern matches any arrangement of `and` nodes in the AST). Order *is* important for those nodes that are included in the pattern; this is explained below.

The second pattern in Fig. 4.1, drawn with solid red arrows, encodes the fact that the two calls to `parent` share the first argument. In domain-specific terms, `A` and `B` must have the same parent `P`:

```
(clause
 (compound (functor ‘parent’) (args var))
 (compound (functor ‘parent’) (args var)))
```

This pattern matches only the last of the following programs. The first program is missing one call to `parent`, while the second has different variables in the positions encoded by the pattern.

```
% no match
sister(A,B):-
  female(A),
  parent(P,A),
  A \= B.
```

```
% no match
sister(A,B):-
  female(A),
  parent(P1,A),
  parent(P2,B),
  A \= B.
```

```
% match
sister(A,B):-
  parent(P1,A),
  female(A),
  parent(P1,B),
  A \= B.
```

A single relation between any two objects in a program is generally insufficient to reason about the program's behavior. In the tutoring context, however, there are pat-

terns that strongly indicate the presence of certain bugs. Take for instance the following incorrect program to sum a list:

```
sum([],0).           % the empty list sums to zero
sum([H|T],Sum):-     % to sum the list [H|T],
  sum(T,Sum),        %  sum the tail T and
  Sum is Sum + H.    %  add first element H (bug: reused variable)
```

This error is fairly common with Prolog novices: the variable `Sum` is used to represent both the sum of the whole list in the second line, and the sum of only the tail elements in the third line. The last line then fails since Prolog cannot unify `Sum` with a (generally) different value of `Sum + H`.

This mistake can be described with several different patterns. Fig. 4.2 shows three patterns overlaying the program's AST. Solid and dashed arrows indicate two of the possible patterns capturing the variable-reuse bug. The first of these patterns states that the `Sum` returned by the predicate should not be the same as the `Sum` from the recursive call:

```
(clause
 (head (compound (functor 'sum') (args (args var))))
 (compound (functor 'sum') (args (args var))))
```

Another possible pattern for the same bug is drawn with dashed orange arrows. It indicates the likely error in the arithmetic expression "`Sum is Sum + H`":

```
(clause (binop var 'is' (binop var '+')))
```

Finally, the leftmost pattern in Fig. 4.2, drawn with dotted blue arrows, describes the correct relation between the two constants in the base-case rule:

```
(clause (head (compound (functor 'sum') (args '[]' (args '0')))))
```

We use such patterns to relate pairs of literals (or a variable and a literal) occurring in the same goal. The main reason for including these patterns in our feature set is to handle recursive programs for list-processing tasks, which often include a base-case rule with no variables – like the above example.

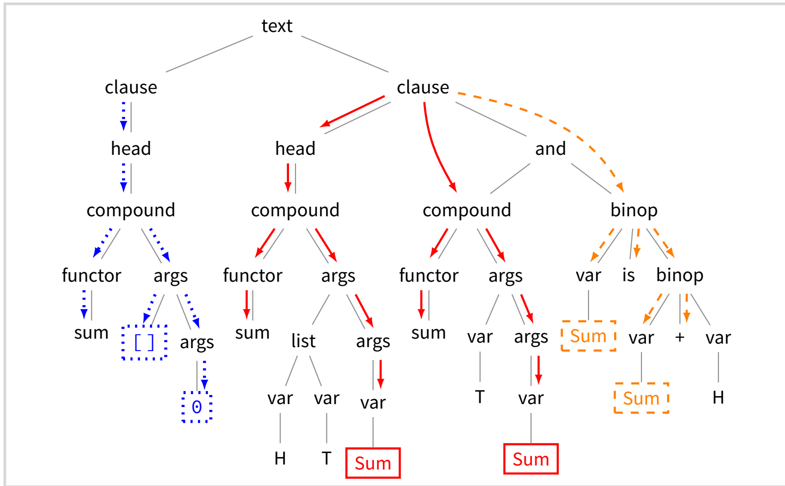

Figure 4.2

The AST for the buggy `sum` program. Dotted arrows relate the correct values in the base case. Solid and dashed arrows denote two patterns describing incorrect reuse of the `Sum` variable in the recursive case.

4.2 Extracting patterns

We construct each pattern by connecting some pair of leaf nodes in a program's AST. Here we always select a pair of nodes from the same clause: either two nodes referring to the same variable (like the examples in Fig. 4.1), or a value (such as the empty list `[]` or the number `0`) and another variable or value in the same `compound` or `binop` (like the blue dotted pattern in Fig. 4.2). For example, in the clause[2]

```
foo(A,B):-
  bar(A',[]),
  baz(A",C),
  B' is C' + 1.
```

we would select the following node pairs: {`A`, `A'`}, {`A`, `A"`}, {`A'`, `A"`}, {`B`, `B'`}, {`C`, `C'`}, {`A'`, `[]`}, {`B'`, `1`} and {`C'`, `1`}.

For each selected pair of leaf nodes (a, b) we build a pattern by walking the AST in depth-first order, and recording nodes that lie on the paths to a and b. We omit `and` nodes, as explained in the previous section. We also include certain nodes that do not

[2] The second and third occurrences of each variable (`A`, `B` and `C`) are marked with ' and " for disambiguation.

lie on a path to any selected leaf. Specifically, we include the functor or operator name for all compound, binop and unop nodes containing a or b.

Patterns constructed in this way form the set of features for rule learning. To weed out very unusual patterns and keep this set at a reasonable size, we only use patterns that have occurred in at least five submitted programs.

4.3 Learning rules

We represent students' submissions in the feature space of AST patterns described above. Each pattern corresponds to one binary feature, with the value true when the pattern is present and false when it is absent. We classify each program as correct if it passes a predefined set of test cases, and incorrect otherwise. We use these labels for machine learning.

Since we can establish program correctness using appropriate test cases, our goal for learning rules is not actually classifying new submissions. Instead, we wish to discover patterns associated with correct and incorrect programs. This approach to machine learning has been called *descriptive induction* – automatic discovery of patterns that describe regularities in data [105]. We use rule learning for this task, because rule conditions are easy to translate into hints.

Before explaining the algorithm, let us discuss the reasons why a program can be incorrect. Our experience indicates that bugs in student programs can often be described either by some incorrect or *buggy* relation between objects which needs to be corrected, or some missing relation that should be added before the program will pass the test cases. We now explain how both types of errors can be identified with rules.

To discover buggy patterns, the algorithm first learns *negative rules* – those that classify programs as incorrect. We use a variant of the CN2 algorithm [106] implemented within the Orange data-mining toolbox [107]. Since the primary use of rules is to generate hints, we wish to ensure that induced rules are correct so as to avoid presenting misleading hints. To this end we impose several additional constraints on the rule learner:

- classification accuracy of each learned rule must exceed a given threshold (we used 90%, as a 10% error seems acceptable for our application);
- each conjunct in a condition must be significant according to the likelihood-ratio test (set the significance threshold to $p = 0.05$);

- conjuncts can only specify the presence of a pattern (in other words, we only allow feature-value pairs with the value true).

The first two constraints ensure we only get high-quality rules that contain only significant patterns. The third constraint is less obvious. It ensures that rules do not mention the absence of a pattern as a reason for the program to be incorrect. This is important when generating hints from negative rules: we wish to be able to point to a specific incorrect pattern in a program, which would not always be possible if negative rules specified that some pattern must be *absent* from the program.

For example, say a program P is covered by the rule "$\neg A \wedge \neg B \Rightarrow \text{incorrect}$". In other words, P is incorrect because it is missing at least one of the patterns A and B. Adding either pattern might fix P, but we cannot point to any erroneous part of the program. Instead of a negative rule specifying missing patterns, we handle this case using two positive rules "$A \Rightarrow \text{correct}$" and "$B \Rightarrow \text{correct}$". Alternatively, if both patterns must be present for the program to be correct, we would instead only have the single rule "$A \wedge B \Rightarrow \text{correct}$". Either way, the conditions in such positive rules will likely contain one or more additional patterns besides A and B.

For the second type of error – missing relations in a program – we induce *positive rules* for the class of correct programs. Positive rules specify the necessary conditions for a program to be correct. To support hint generation, the combination of all conditions in a positive rule should also be sufficient (with some degree of certainty) to determine correctness. To this end, we use the same constraints on rules and conditions as above.

Learning accurate positive rules turns out to be difficult: there are many programs that are incorrect despite having all necessary patterns, because they also include some incorrect patterns. A possible way to solve this problem is to ignore programs covered by some negative rule when learning positive rules. This way all known buggy patterns are removed from the data, and will not be included in positive rules. However, removing incorrect patterns also removes the need for specifying relevant patterns in positive rules: if all incorrect programs were ignored, the single (useless) rule "$\text{true} \Rightarrow \text{correct}$" would suffice. We achieved good results by learning positive rules from the complete data set and estimating their accuracy only on programs not covered by negative rules.

While our main interest is discovering important patterns, induced rules can also be used to classify new programs, for instance when evaluate rule quality. Classification proceeds in three steps:

1. if a negative rule covers the program, classify it as incorrect;
2. else if a positive rule covers the program, classify it as correct;
3. otherwise, if no rule covers the program, classify it as incorrect – correct programs are very likely to be covered by at least one positive rule.

As with any programming language, functionally equivalent Prolog clauses can often be written in different ways. For example, the clause

```
sum([],0).
```

can also be written as

```
sum(List,Sum):-
    List = [],
    Sum = 0.
```

Given enough data, our approach will cover such variations by inducing additional patterns and rules. Another option would be to use rules in conjunction with program canonicalization, by transforming each submission into a semantically equivalent normalized form before extracting patterns [73].

This is another advantage AST patterns have over rewrite rules: since they only use individual submissions, they can be easily combined with other approaches, such as program normalization. Learning rewrites, on the other hand, requires a full trace of modified characters for each solution, from which changes to individual fragments are extracted. Since canonicalizing a program changes its fragments, it would be difficult to combine learning rewrites with this approach.

4.4 Generating hints

Once we have induced classification rules for a given problem, we can use them to provide hints based on buggy or missing patterns. As in the previous chapter we describe two options: generating hints directly from matching rules for a submission, and using rules to assist the authoring process.

4.4.1 Automatic feedback

To generate a hint for an incorrect program, each rule is considered in turn. We consider two types of automatic feedback: *buggy* and *intent* hints based on negative and positive rules (i.e. for incorrect and correct programs).

First, all negative rules are checked to find any known incorrect patterns in the program. To find the most likely incorrect patterns, negative rules are considered in the order of decreasing quality (i.e., we consider the negative rule that covers the fewest correct programs first). If all patterns in the rule "$p_1 \wedge \cdots \wedge p_k \Rightarrow$ incorrect" match (i.e., the program contains the patterns $p_1, \ldots, p_k$), we highlight the relevant leaf nodes. In our evaluation (described in the following section) we found that most negative rules are based on the presence of a single pattern.

For the incorrect list-sum program from the previous section this method produces the following highlight

```
sum([],0).
sum([H|T], Sum ):-
  sum(T, Sum ),
  Sum is Sum + H.
```

based on the rule "$p \Rightarrow$ incorrect", where p corresponds to the solid red pattern from Fig. 4.2:

```
(clause
 (head (compound (functor 'sum') (args (args var))))
 (compound (functor 'sum') (args (args var)))).
```

Along with the highlight we provide a generic message pointing out possible causes and solutions for the bug:

> The variable `Sum` is used incorrectly. Are all goals that reference it correct? Check whether you are using the right predicate or operator, and that the highlighted arguments make sense.
>
> Also, ensure that all occurrences of `Sum` denote the same value – within a Prolog rule, each variable can only refer to a single value (such as a name or a number).

If the program is not covered by any negative rule, we try to determine the student's intent using positive rules. Recall that positive rules group patterns that together indicate a high likelihood that the program is correct. Each such rule thus defines a particular "solution strategy" in terms of AST patterns. We reason that alerting the student to a missing pattern could help them complete the program, without revealing the whole solution.

To generate an intent hint we consider all *partially matching* positive rules "$p_1 \wedge \cdots \wedge p_k \Rightarrow \text{correct}$", where the student's program matches some (but not all) patterns p_i. For each such rule we store the number of matching patterns, and the set of missing patterns. We are interested in those rules that have the most matching patterns, since those rules are most likely to correctly capture the students intent. We then return the most common missing pattern among the rules with the most matching patterns.

For example, if we find the following missing pattern for an incorrect program implementing the `sister` predicate:

```
(clause
 (head (compound (functor 'sister') (args var)))
 (binop var '\=')),
```

we could display a message to the student saying "a comparison between `A` and some other value is missing", or "your program is missing a goal with the form `A \= ?`".

This method can find more than one missing pattern for a given partial program. In such cases we can return the most commonly occurring pattern as the main hint, and other candidate patterns as alternative hints. We use main and alternative intent hints to establish the upper and lower bounds when evaluating automatic hints in Section 4.5.

4.4.2 *Manual feedback*

Only generic feedback messages – like the examples in the previous section – can be provided automatically. In order to explain errors in terms of the specific problem being solved, a human teacher must add appropriate messages to the tutor. Pattern-based rules are transparent and understandable, and can thus aid the authoring process.

In many cases, negative rules map directly to misconceptions in the target programming language. When writing feedback for a tutor, a teacher can simply annotate each rule with an explanatory message. For example, consider one of the top negative rules

for the `sum` problem (for each rule in this section we also give the quality and the number of correct and incorrect programs it covered in the experiment described in the next section):

Rule 1 (quality = 0.931, # incorrect = 29, # correct = 0):
(clause (binop var "is" (binop var "+"))) ⇒ *incorrect*

Like most negative rules, this rule is based on a single code pattern – in this case describing the erroneous expression "`Sum is Sum + ?`" (see Fig. 4.2 for a graphical representation on a concrete AST). A teacher could add the following explanatory text:

> The variable `Sum` appears on both sides of the `is` operator. In Prolog, you cannot "update" the value of a variable – each variable can only represent a single value.

The same feedback could be used for any problem with this rule (i.e., where the `is` operator has been used incorrectly). A more specific message could explain the error in terms of values that are actually used in the `sum` problem:

> It appears you are using the same variable `Sum` to represent a) the sum of the whole list and b) the sum of its tail. In Prolog, you need a different variable for each value – try introducing a new variable to denote the recursively calculated length of the tail of the list.

For most problems we induced between 10 and 30 negative rules, covering the majority of student errors. Annotating these rules is much easier than writing feedback from scratch – as explained in the previous chapter, enumerating all the possible student errors is a complex task even for experienced instructors. Additionally, looking at the number of submissions covered by each rule allows us to prioritize writing feedback for the most common mistakes first.

Positive rules, on the other hand, can be used to discover the most important or difficult parts of each program. For example, consider the following rule for the `sum` problem, relating three important patterns:

Rule 2 (quality = 0.913, # incorrect = 4, # correct = 78):
(clause (head (compound (functor "sum") (args (args "0")))))
(clause (head (compound (functor "sum") (args (args var)))) (binop var "is")) ⇒ *correct*
(clause (compound (functor "sum") (args (args var))) (binop "is" (binop "+" var)))

A student's program is likely to be correct when it includes all these patterns. The first pattern describes the base case of the empty list with sum zero. Note that this pattern, or any other pattern in the rule, says nothing about the first argument (the empty list) in the base case. We can reason that once a student figures out that the base case should handle the empty list with zero sum – and not, for example, a list with one element – they have no problems coding the corresponding rule; be it as the fact "`sum([],0)`" or as the Prolog rule

```
sum(L,0) :-
    L = [].
```

The two remaining patterns ensure the `is` operator is applied correctly. Since the rule contains no other patterns for the recursive clause, we can again conclude that this is the most challenging part of the `sum` problem: once a student has coded the fragment "`Sum is SumT + H`" correctly, the rest of the program is also very likely to be correct.

The two main "knowledge components" in the `sum` problem appear to be the empty-list base case, and the `is` operator in the recursive clause. Positive rules can thus help us analyze the most important or difficult concepts for each problem. This can potentially help an instructor plan and improve a course.

4.5 Evaluation

We evaluated automatic hints based on AST patterns on 50 programming assignments, using the data set described in Section 3.1. As when evaluating rewrites in the previous chapter, we divided the set of student traces for each problem into training and testing sets in the ratio 70 : 30. We extracted patterns and induced rules from submissions in the training set, then tested those rules on the incorrect submissions from the testing set. We evaluated both rule classification accuracy and the generated hints by retrospectively analyzing the proportion of cases in which students removed or added a suggested pattern.

Tables 4.1 and 4.2 show our results. The second, third, and fourth columns provide classification accuracies of the rule-based, majority, and random-forest classifiers on testing data. The majority classifier and the random forests method, which had the best overall performance, serve as references for bad and good classification accuracy on particular data sets.

Table 4.1

Evaluating hints based on AST patterns on historic student data for Prolog exercises in the *Family relations*, *Lists I* and *Lists II* groups. See Table 4.2 on the next page for a description of the data.

	CA			Buggy		Intent			
	Rules	RF	Maj.	All	Imp.	All	Imp.	Alt.	No hint
Family rel.	0.940	0.978	0.621	540	539	613	351	32	306
ancestor/2	0.934	0.980	0.525	60	60	95	76	1	17
aunt/2	0.882	0.953	0.480	72	72	71	31	0	20
brother/2	0.846	0.952	0.654	27	27	47	20	2	15
cousin/2	0.871	0.931	0.650	102	101	104	40	19	61
descendant/2	0.981	0.989	0.567	72	72	24	13	6	29
father/2	0.990	1.000	0.707	8	8	11	11	0	15
grandparent/2	0.969	1.000	0.725	33	33	18	18	0	15
mother/2	1.000	1.000	0.573	20	20	30	30	0	73
sister/2	0.988	0.994	0.711	146	146	213	112	4	61
Lists I	0.908	0.957	0.634	985	965	457	334	20	318
conc/3	0.907	0.965	0.653	166	157	37	31	1	36
del/3	0.968	0.946	0.574	155	155	28	12	6	62
divide/3	0.923	0.942	0.724	161	155	126	100	0	36
dup/2	0.940	0.963	0.677	155	155	111	89	8	28
insert/3	0.932	0.969	0.589	120	118	52	32	1	44
last_elem/2	0.806	0.935	0.620	9	9	15	3	0	21
memb/2	0.880	0.967	0.516	59	58	33	25	1	48
permute/2	0.910	0.966	0.723	160	158	55	42	3	43
Lists II	0.877	0.924	0.615	1034	1001	493	289	42	543
{even,odd}len/1	0.748	0.900	0.618	19	19	85	45	2	104
len/2	0.940	0.988	0.496	88	88	27	24	0	23
max/2	0.778	0.809	0.467	32	32	53	13	11	45
min/2	0.828	0.867	0.740	150	150	115	64	22	49
palindrome/1	0.849	0.923	0.663	118	118	16	7	4	133
rev/2	0.944	0.967	0.744	242	241	72	58	0	33
shiftleft/2	0.927	0.949	0.633	132	121	14	7	0	63
shiftright/2	0.881	0.892	0.537	80	75	16	7	0	44
sublist/2	0.893	0.953	0.744	111	98	62	33	3	33
sum/2	0.981	0.990	0.510	62	59	33	31	0	16

Table 4.2

Evaluating hints based on AST patterns on historic student data for Prolog exercises in the *Sorting*, *Sets* and *Trees* groups. The first column group gives classification accuracies for rules, random forests, and the majority classifier. The next two groups show the number of all/implemented buggy and intent hints; for intent hints we also give the number of implemented alternative hints. The last column shows the number of submissions where a hint could not be generated.

	CA			Buggy		Intent			
	Rules	RF	Maj.	All	Imp.	All	Imp.	Alt.	No hint
Sorting	0.889	0.933	0.672	660	643	337	229	32	224
is_sorted/1	0.950	0.968	0.780	160	155	103	92	3	82
isort/2	0.935	0.976	0.628	106	106	24	19	3	22
pivoting/4	0.826	0.895	0.695	93	90	89	49	17	39
quick_sort/2	0.899	0.951	0.696	161	154	43	34	4	16
sins/3	0.862	0.918	0.691	111	110	67	26	5	23
slowest_sort/2	0.860	0.890	0.543	29	28	11	9	0	42
Sets	0.809	0.884	0.644	572	558	597	316	62	268
count/3	0.856	0.829	0.779	121	121	153	74	2	41
diff/3	0.734	0.837	0.547	21	21	64	43	3	25
intersect/3	0.653	0.807	0.627	32	32	120	57	12	28
is_subset/2	0.699	0.925	0.726	2	2	0	0	0	42
is_superset/2	0.874	0.910	0.466	72	71	31	30	0	41
powerset/2	0.947	0.989	0.582	87	84	5	5	0	15
subset/2	0.900	0.925	0.670	72	72	74	41	19	28
union/3	0.811	0.849	0.752	165	155	150	66	26	48
Trees	0.885	0.908	0.698	862	821	392	195	42	444
deletebt/3	0.865	0.829	0.777	101	100	70	24	11	32
depthbt/2	0.861	0.916	0.578	66	63	62	29	4	26
insertbt/3	0.951	0.969	0.698	62	59	0	0	0	18
maxt/2	0.838	0.873	0.775	44	38	26	5	0	46
memberbt/2	0.943	0.954	0.603	71	71	52	38	3	43
membert/2	0.898	0.923	0.760	85	73	77	11	23	54
mirrorbt/2	0.755	0.798	0.755	134	126	0	0	0	164
numberbt/2	0.895	0.941	0.624	113	107	51	38	1	30
tolistbt/2	0.958	0.969	0.716	186	184	54	50	0	31

For example, our rules correctly classified 99% of testing instances for the `sister` problem – almost the same as random forest, whereas the accuracy of the majority classifier was 71%. In most cases, rules perform slightly worse than random forests, mostly due to the constraints described in Section 4.3, which ensure more general rules at the cost of goodness of fit.

For the problems `intersect` and `is_subset`, rules perform significantly worse than random forests. This is likely due to the cut operator (`!`) used in many solutions to those problems. The AST patterns we used here do not capture this goal, because it contains no variables or values. Induced rules therefore cannot use it to distinguish between correct and incorrect submissions.

Results in the remaining columns were obtained by evaluating generated hints on existing student traces. For each incorrect program we generated a hint as described in Section 4.4.1, and then checked whether the suggestion was implemented in a subsequent correct submission. For buggy hints the offending pattern should be removed, while patterns suggested by intent hints should be added.

The columns under the *Buggy* heading contain evaluation of hints generated from negative rules (i.e. rules that predict a program is incorrect). For each generated buggy hint we checked whether it was implemented by the student (by removing the corresponding pattern) in the final submission. The *All* column shows the number of all generated buggy hints, while the *Imp.* column shows the number of implemented hints. The results indicate that buggy hints are very relevant, as over 97% (4527 out of 4653) were implemented in the final solution.

When no buggy hint is found for an incorrect program, the algorithm attempts to generate intent hints by looking for positive rules that most closely match the patterns in the student's code. While we evaluate buggy hints found by looking only at the top matching negative rule, the situation is somewhat more complex for intent hints. For many incomplete submissions there are several different but equally good ways to complete it, so we often find several possible intent hints. We call the hint based on the highest-quality positive rule the *main* hint (the one we would have shown to the student), and the others *alternative* hints.

Success rates for intent hints are given under the *Intent* heading. The *All* column shows the number of submissions for which an intent hint was generated, and the *Imp.* column shows the number of programs where the student has subsequently implemented the main intent hint (derived from the highest-quality rule). This gives the

lower bound on the effectiveness of our method.

Consider now the case where the main intent hint was not implemented in the final submission. This could be either because the hint was incorrect, or because the student decided to follow some other solution strategy. However, if we had actually shown the main intent hint, the student might have opted for that strategy. The *Alt.* column shows the number of programs where an alternative intent hint was implemented in the final solution. Combining the *Imp.* and *Alt.* columns thus gives the upper bound on the effectiveness of our method.

Notice that the percentage of implemented intent hints is significantly lower when compared to buggy hints: for the `ancestor` problem, 77 out of 95 (81%) of suggested intent hints were implemented, whereas only 24 out of 53 suggested hints were implemented for the `max` problem. On average, 59% of main intent hints and an additional 8% of alternative intent hints were implemented.

To sum up, buggy hints are good and reliable, since they are almost always implemented, even when testing on past data – the students' decisions were not actually influenced by these hints. The percentage of implemented intent hints is lower, which is still not a bad result, given that it is often difficult to determine the programmer's intent from incorrect submissions. Overall we were able to generate hints for approximately 78% of incorrect submissions.

High classification accuracies in many problems imply that it is possible to correctly determine the correctness of a program by simply checking for the presence of a small number of patterns. Our hypothesis is that there exist some crucial patterns for each exercise that students have difficulties with. When they figure out these patterns, implementing the rest of the program is usually straightforward.

Since this evaluation was done on past data and students did not actually see the hints, this is only a crude measure of how effective generated hints are. It is encouraging, however, that we are able to provide apparently useful hints for a large majority of incorrect submissions. Furthermore, unlike the debugger from the previous chapter, hints can be found for a program without having to generate and run new programs, reducing the load on the tutoring server. Since hints are generated simply by searching for matching rules, the whole process could also easily be implemented as a standalone program: given a set of induced rules in the S-expression notation, the only additional language-specific requirement is a parser.

4.6 Python

This section evaluates pattern-based classification of Python programs. We have used the same kinds of patterns as described in Section 4.1; that is, patterns connecting two instances of a variable, or a literal and a variable. As before, we extracted patterns from student programs and used them as attributes for machine learning.

Since CodeQ was not used as extensively for teaching Python, we have not been able to collect as much data. We tested classification accuracy for rules and random forests using 10-fold cross validation on problems with at least 200 collected submissions. Python programs evaluated here are more complex and variable than Prolog programs in previous sections. Rules described in Section 4.3 proved ineffective; we therefore lifted the restrictions on conditions and quality of induced rules in this evaluation. The following results thus only indicate the feasibility of AST patterns for predicting program correctness, and further research is needed to generate hints based on these predictions.

Table 4.3 shows classification accuracy for rules, random forests and the majority classifier. For most problems we are able to achieve a classification accuracy between 70% and 90%. Rule performance is on average noticeably worse than for Prolog programs, for reasons discussed in the remainder of this section. By showing that the unmodified method achieves a relatively high classification accuracy for most problems, we confirm that the pattern-based approach is also useful for imperative programming languages. Significant room for improvement remains, however.

A plausible reason for the relatively worse classification accuracy on Python programs could be insufficient data; we observed significantly lower performance for problems with fewer than a hundred submissions. Furthermore, most Prolog programs were submitted by students taking the same course, which decreased variations in collected programs. To test this we tried inducing rules from increasingly large subsets of submissions. Figure 4.3 shows the results for five problems.

While using more programs does increase classification accuracy, results show that the effect plateaus after a certain point. We are therefore unlikely to discover additional knowledge by inducing rules from larger data sets while using the same kinds of patterns. The remainder of this section discusses other possible reasons for the lower accuracy, and potential ways to improve it.

When analyzing Prolog programs, we discovered several kinds of patterns that im-

Table 4.3

Evaluating the quality of classification rules based on AST patterns on historic student data for Python exercises. The second and third columns give the number of correct and all submitted programs. The final three columns give the classification accuracy for rules and random forests (induced using AST patterns as attributes) and the majority classifier.

	Submissions		Classification accuracy		
	Correct	Total	Rules	RF	Majority
greatest negative	73	200	0.785	0.745	0.635
body mass index	72	233	0.708	0.687	0.691
molar mass	131	233	0.777	0.691	0.562
is palindrome	142	241	0.805	0.822	0.589
contains string	147	247	0.725	0.761	0.595
checking account	122	248	0.706	0.702	0.508
star tree	82	249	0.835	0.835	0.671
sum to n	137	259	0.861	0.826	0.529
contains number	121	267	0.914	0.861	0.547
temperatures	73	273	0.758	0.784	0.733
sum and average	81	276	0.793	0.775	0.707
area of a triangle	151	303	0.884	0.838	0.498
contains 42	138	312	0.747	0.728	0.558
hello world	189	313	0.652	0.658	0.604
competition	232	333	0.808	0.802	0.697
fast fingers 2	147	347	0.723	0.677	0.576
even odd	142	367	0.899	0.883	0.613
what is your name	132	409	0.954	0.941	0.677
pythagorean theorem	136	425	0.918	0.896	0.680
fast fingers	210	502	0.779	0.773	0.582
top shop	134	508	0.829	0.772	0.736
buy five	299	514	0.807	0.792	0.582
average	187	518	0.782	0.736	0.639
pythagorean theorem	353	758	0.681	0.654	0.534
speed of sound	165	817	0.931	0.874	0.798

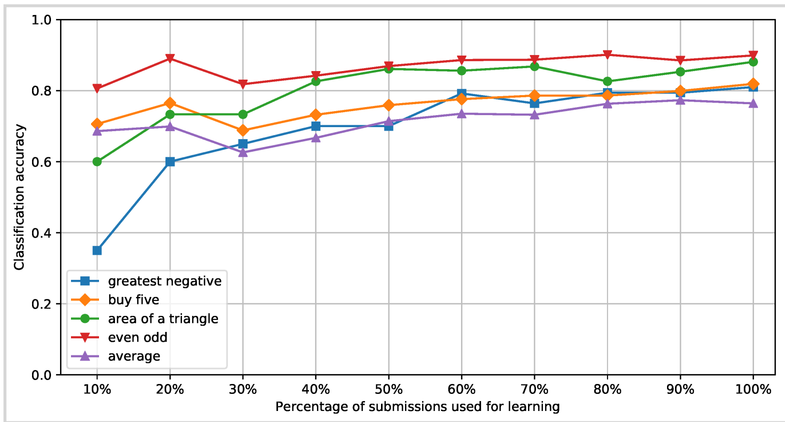

Figure 4.3

Classification accuracy for rules induced from subsets of submissions for selected problems. The x-axis shows the percentage of all submissions that were used for learning, and the y-axis shows the classification accuracy of induced rules.

proved the performance of classification rules. With Python we have used the same kinds of patterns as for Prolog, with no further tweaks. Analyzing misclassified programs for the most problematic problems should yield insight into the kinds of patterns that would be useful for Python. One possibility are patterns describing the control structure of the program (e.g., a for loop containing an if statement). We will perform this analysis in future research.

The problem of semantically equivalent program variations is more pronounced in Python. For this evaluation we performed no normalizations beyond renaming variables. It would be straightforward to apply program canonicalization [73] to submissions before extracting patterns, which should further improve prediction accuracy.

Finally, declarative programming languages do seem to be more amenable to pattern-based analysis. Just like AST patterns, Prolog programs define static relations between variables, and the run-time behavior follows relatively simple rules. On the other hand, Python and other imperative languages provide a rich set of control structures, resulting in complex behavior. Any kind of static analysis without the knowledge of Python semantics might thus be inherently limited. Still, AST patterns can be used in conjunction with other attributes.

4.7 Future directions

We have shown that code patterns perform as well, and in many cases better, than rewrites. Since pattern-based rules are simpler than rewrites, and easier to learn and use, we believe they provide a better and more stable foundation for further research.

In this work we focused on patterns relating two instances of a variable. These are the minimal meaningful attributes, and our approach was to use classification rules to group them. An obvious avenue for future work is exploring different types of patterns, for example relating multiple or even all instances of a variable, or patterns specifying (a part of) the internal structure of the AST. An interesting approach would be using a genetic algorithm to modify and combine patterns into new variants.

While researching patterns, we first defined the kind of patterns to use, extracted them automatically and induced classification rules. We then looked for problematic cases where existing patterns could not cover some important aspect of the program, and added new kinds of patterns (e.g. patterns covering singletons). Argument-based machine learning [108] can be used to facilitate such an iterative approach: the computer learns a model and finds most important misclassifiied examples, and the expert provides arguments or new attributes (patterns) to help the computer correct those mistakes. This process is repeated until the model is good enough.

Patterns denoted with S-expression are not easy to read. A tool to visualize patterns and corresponding matching programs in a clear and concise way would be extremely useful when analyzing student submissions. A useful feature of such a tool would be to allow the teacher to define new patterns by selecting nodes in one or more sample programs, and filter submissions based on newly defined patterns.

5

CodeQ

As described in preceding chapters, both rewrites and code patterns can be used to generate feedback. The most pertinent evaluation of such feedback is of course analyzing the effect it has on student problem-solving performance. To this end we have developed an online learning environment which allows us to both collect necessary learning data and evaluate the effectiveness of different kinds of feedback in the classroom. This chapter describes the main features of the application, and compares the effects of manually and automatically generated hints.

CodeQ[1] is a free[2] web application for learning programming. It provides an integrated online environment for writing and running programs in Prolog and Python, and can be extended to support other languages. CodeQ currently supports two courses based on classes taught at the Faculty of Computer and Information Science, University of Ljubljana:

- Programming 1, a first-year introduction to programming using Python, and
- Principles of Programming Languages, an elective course taught with Prolog.

In both classes, students solve sets of programming exercises that have been selected and tuned over several years. We implemented these exercises in CodeQ.

While using an ordinary programming environment allows students to practice "real-world" programming, it has several downsides. Before CodeQ, students solved Prolog problems using a simple text editor together with the standard SWI-Prolog interpreter[3]. Often a student would forget to reload a modified file, and spend a lot of time looking for a bug that did not exist anymore. For Python problems students use PyCharm[4], which is a complex integrated development environment (IDE). It is not trivial to set up – students often have trouble configuring paths, file encoding and other settings – and its incessant tips about coding style can be distracting or difficult to understand.

Using an online environment specialized for learning alleviates most of these issues. It avoids the overhead associated with solving programming exercises, such as installing the interpreter and managing files. Keeping the CodeQ feature set to a minimum reduces the cognitive load of a full-fledged IDE interface. Students log in, select

[1] `https://codeq.si/`

[2] Source code is available under AGPL3+ at `https://codeq.si/code`.

[3] `http://swi-prolog.org/`

[4] `https://jetbrains.com/pycharm/`

a problem to solve and can immediately start coding. They can log out at any time and resume their attempt later, or on another computer.

After logging in and selecting a course, CodeQ presents the student with a list of problem groups. Each group typically contains problems for one lab session. Fig. 5.1 shows the list of problems in the first group (*Family relations*) for the Prolog course. Most groups feature introductory text explaining the concepts required for solving that week's problems. Dots next to problem names indicate status: an empty dot means the problem has not yet been attempted, an orange dot means the student has started working on the problem but has not yet completed it, and green dots indicate successfully solved problems.

Figure 5.1 Overview of problems in the Prolog course (in Slovene). Links in group description describe basic Prolog syntax and the given database of family relations, used to test student solutions. The problem list shows the status for each problem, with green, orange and empty dots indicating solved, attempted and not attempted problems.

We have observed that, even though no deliberate gamification features [109] have been included in CodeQ, the status indicators tend to have a positive effect on student motivation and involvement. Compared to previous years, when they wrote programs using a normal text editor, students appear to solve more problems and remain in class longer in order to "complete the set" for each week – despite the fact that there was no explicit award for doing so.

Fig. 5.2 shows the main screen for the list-reversal problem from the *Lists II* group in the Prolog course. Problem description is displayed in the upper-left corner and

includes one or more examples of correct program behavior. Dynamically provided feedback from the tutor, including test results and hints, is shown below the line. The right-hand side contains a code editor and an interactive prompt for submitting queries to Prolog. The current program is loaded into the Prolog engine automatically with every query. Students can request feedback from the tutor using the two buttons above the editor.

rev/2

`rev(L1,L2)`: the list `L2` is obtained from `L1` by reversing the order of the elements.

```
?- rev([1,2,3], X).
  X = [3,2,1].
?- rev([], X).
  X = [].
```

Your code passed 0 / 5 tests.

All three arguments of predicate `conc/3` are *lists*. Are you sure you used it properly?

This is one of the most rewarding exercises. Classic recursion! Try to reduce the problem into a smaller one. That, of course, means reducing it to a shorter list.

Plan Test

```
1 % reverse a non-empty list
2 rev([Head|Tail], Reversed) :-
3     % reverse tail
4     rev(Tail, RevTail),
5     % append head to reversed tail
6     conc(RevTail, Head, Reversed).
```

line 4, column 24

```
?- rev([1,2,3], X).
false.
?-
```

Figure 5.2

CodeQ problem-solving screen. The left side gives a description of the problem with examples of correct behavior. Feedback in response to *Plan* and *Test* buttons is output below. On the right there is a code editor and an interpreter for running queries.

The Python interface is very similar. It contains an additional "Run" button that executes the student's program. Unlike Prolog, where each query is independent, state (variable assignments) persists between successive runs and queries in the Python interpreter.

5.1 Feedback

CodeQ provides several kinds of feedback: automatic testing against predefined test cases, problem-specific plans and code-specific hints. Feedback is only provided when requested by the student, by clicking on the appropriate button.

The "Plan" button provides additional advice when the student is unsure about how to approach a problem. These messages are written by the instructor and are included in problem definitions. In Fig. 5.2 a plan has been requested and shown under the line in the bottom-left corner. No limits are imposed on using the "Plan" button, but

students are encouraged to use it only as a last resort when stuck on a problem.

Correctness of a solution may be checked at any time using the "Test" button. As in most programming tutors this is done by checking program outputs on a predefined set of inputs. CodeQ responds with the number of test cases the program answered correctly, in some cases also including one of the failing tests. For incorrect programs the response may contain additional feedback, hidden behind a "Hint" button to allow students to decide whether to ask the tutor for help or try finding the error on their own.

In order to provide code-specific hints, CodeQ analyzes an incorrect program in several stages. If a mistake is found in any stage, a corresponding hint is returned and no further processing is done. The checks are described below in the order in which they are performed.

1. *Syntax check.* The program is run through the interpreter to ensure it is syntactically correct. Any syntax errors are reported. We have noted that students, especially when beginning to learn a new language, often have difficulty understanding errors and warnings from the interpreter. These messages could potentially be clarified by instructor-provided annotations; however, providing too detailed or verbose feedback might not be beneficial [110].
2. *Problem-specific hints.* An optional problem-specific hint function is invoked. This function is written by the instructor as part of problem definition. Typically it runs the program on selected inputs to detect the presence of common errors. The hint function for each new problem is written in an ad hoc manner, but could be standardized in the future using test-output vectors [11] to detect specific errors based on which tests fail.

   An example of a manually defined hint for the `sister(X,Y)` problem is:

   > "If `X` is `Y`'s sister, they should have a common parent."

   This hint is triggered if the student's program returns any solution to the query

   ```
   ?- sister(X,Y), \+(parent(P,X), parent(P,Y)).
   ```
3. *Automatic hints.* Next, CodeQ uses one of the methods described in Chapters 3 and 4 to generate feedback for the given program. The first option is attempting

```
1 % reverse a non-empty list
2 rev([Head|Tail], Reversed) :-
3     % reverse tail
4     rev(Tail, RevTail),
5     % append head to reversed tail
6     conc(RevTail, Head, Reversed).
```

Check the highlighted part.

Figure 5.3

Incorrect program with automatic highlights. The first (green) highlight indicates the position where another clause must be inserted, while the second (yellow) highlight points out an incorrect argument to the conc predicate.

to fix the program using the rewrite-based debugger. If successful, the debugger returns a list of rewrites required. Areas of the program modified by these rewrites are highlighted, directing the student's debugging efforts. Fig. 5.3 shows an incorrect solution for the `rev(List,Reversed)` predicate, with automatically generated highlights.

The rewriting debugger may fail to produce a result within the acceptable time frame. The second option is checking the program against learned pattern-based rules, which is much faster and uses a predictable amount of time. To decrease the load on the server it could also easily be done on the client-side, since no programs need to be generated or tested. Unlike rewrites, patterns do not give us the exact steps needed to fix the program; however, they can be turned into hints in a similar manner.

4. *Language-specific hints.* If no feedback was provided by any of the above steps, the program is run through a language-specific hint function. This function attempts to detect certain common mistakes, such as writing a variable with a lowercase initial in Prolog. It can also point out stylistic mistakes, for instance rules of the form

   ```
   <head> :- fail.
   ```

   (such rules may be omitted without changing the meaning of the program).

5.2 Evaluation

We have already demonstrated in Sections 3.5.1 and 4.5 that both rewrite-based debugging and AST patterns can be used to generate hints for many incorrect programs. Here we present two studies performed to determine whether feedback – coded manually or generated automatically – is actually helpful to students. To this end, we evaluated CodeQ in the usual classroom setting. On the one hand, performing the experiment during regular lab sessions limited our ability to control for hidden variables. On the other hand, evaluating a tutoring system in a real-world situation should provide the most pertinent results, indicating whether hints can help at all or not.

We performed the experiments during the first three regular Principles of Programming Languages lab sessions in the spring semesters of 2016 and 2017. The purpose of these lab sessions is to familiarize students with Prolog programming. At the beginning of each session the instructor explained new concepts (Prolog basics with recursion, lists and arithmetic) and showed a solution to a sample problem on the whiteboard; the same explanation was also available in written form for reference. Students then solved exercises for the remainder of the session.

The three lab sessions in the studies covered nine problems from the *Family relations* group and 18 problems from the *Lists* and *Lists II* groups. Four of those problems were either new (with no data available from previous years to build a model for automatic hints) or solved by the teacher as examples; we exclude these problems from the analysis below.

For each study, students were randomly assigned to three groups: no hints, automatic hints only and manual hints only. In the 2016 study, the *automatic* group received hints based on the rewrite-based debugger (described in Section 3.6.1), while the 2017 *automatic* group received hints based on AST patterns (described in Section 4.4.1). Both *manual* groups received the same teacher-provided hints. All students received test results and hints related to syntax errors, and had the option of using the *Plan* button. Students solved problems in the CodeQ programming environment. Those who did not wish to participate in the study could use SWI-Prolog or create an anonymous account; there were only a few such students.

To see whether hints help with problem-solving, we measured the time and number of distinct incorrect submissions before a correct program was submitted. Existing research shows that test achievement is strongly related to the number of problems

students successfully solved [111], with content and presentation of feedback playing a secondary role. Decreasing solving time with automatic hints should therefore be beneficial to students.

5.2.1 Data set

We have collected solution traces during previous three iterations of the same course in spring semesters from 2013 to 2015, using an online application similar to a simplified version of CodeQ. These data were used to learn rewrite rules for the 2016 study, and code-pattern rules for the 2017 study. On average, our method discovered 42 rewrite rules and 26 code-pattern rules per problem.

Altogether 355 students have attended the course over the years 2013 to 2015. Of these, 254 students have solved ten or more problems using the online environment. Overall we have collected traces for 4,649 successful solutions to problems considered in this study, for an average of 202 solutions per problem. The distribution of attempts across problems looks very similar each year, with the number of students attempting each problem dropping by a half by the end of the third session, and then slowly decreasing for the remainder of the course.

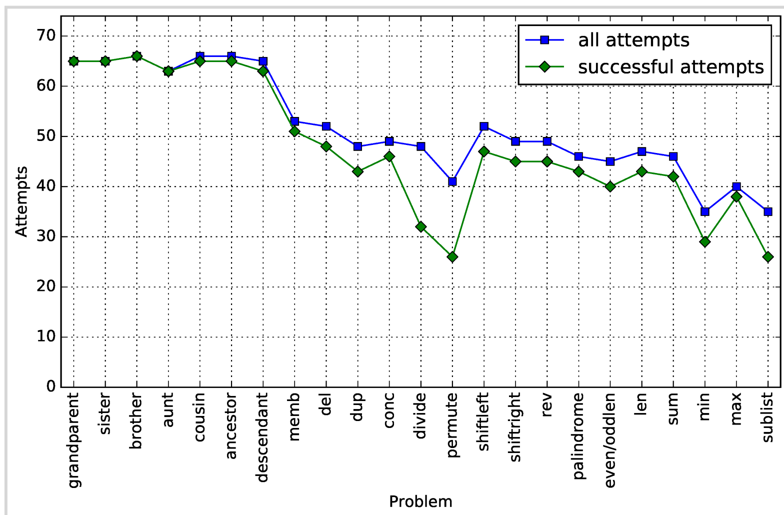

Figure 5.4

Number of students attempting and solving each problem in the 2016 study.

Fig. 5.4 graphs the number of students attempting and solving each exercise during the first study. Problem difficulty increases overall and within each problem group; both trends can be clearly seen in the graph. The drop in the number of attempts is especially noticeable after the first session; this is probably due to students who come in only once to check out the class. We observed no difference in attrition rates between the three experimental groups in either study.

Several authors have noted the high variability of student submissions – with a long tail of unique programs – characteristic of the programming domain [12, 69, 75]. Our experiments confirm this: even after normalizing submitted programs by removing superfluous white space and renaming variables, 2,978 distinct submissions were observed across all attempts (including unsuccessful attempts without a final correct submission) in the first study. Only 228 of those programs have been submitted by more than one student. Fig. 5.5 shows how many programs were submitted by one or more students. Note that the y-axis uses a logarithmic scale. The program at the right edge is the most common correct submission.

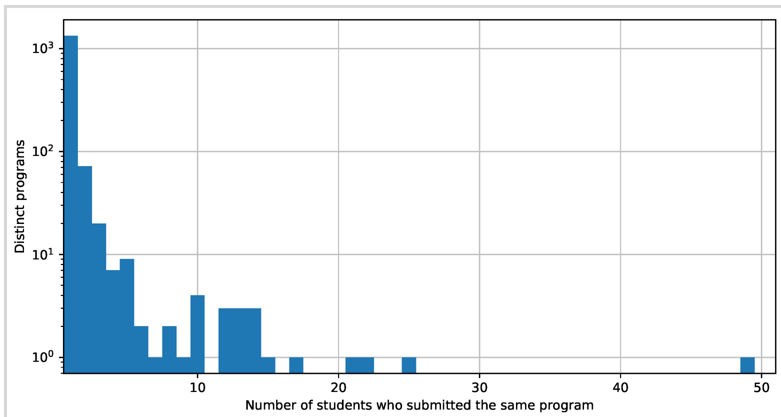

Figure 5.5

Histogram showing how many programs were submitted by 1, 2, 3,... students. The first bin shows that over a thousand distinct programs were submitted only once (i.e., by only one student). The second bin shows that fewer than a hundred distinct programs were submitted twice (i.e., by exactly two students). The exponential drop-off is clear, even though the y-axis uses a logarithmic scale.

For a concrete example, consider the canonical solution for the problem `palindrome`, which uses the list-reversal predicate `rev` and fits in a single line:

```
palindrome(L) :- rev(L,L).
```

Almost 50 students attempted this problem in 2016, and over 40 eventually submitted a working program. In total we received – after normalizing white space and variable

names – 166 distinct submissions. Only five of those programs were submitted by more than one student, and the remaining 161 programs were unique (i.e. appeared in a single attempt). These results are particularly striking in light of the fact that the problem `rev` directly precedes `palindrome` in their problem group.

Each year there were over a hundred students enrolled in the course. We excluded students who have taken the course before (without passing the final exam), exchange students, and those who enrolled after the class had started, leaving 76 participants in the first study and 93 participants in the second study. To ensure the experimental groups were balanced, we controlled for the average grade received on exams in the student's first year (for all classes, and programming classes only). Table 5.1 shows details about the groups for both studies.

Table 5.1

Experimental groups in the first and second study, evaluating hints based on rewrites and AST patterns respectively. N gives the number of participants, and the following columns give the mean grade and standard deviation for all exams taken previously by the students, and for programming exams only.

Group	N	Average grade (<6 = fail, 10 = best)			
		All exams		Programming exams	
2016: rewrites					
No hints	25	7.94	±0.79	7.90	±0.98
Automatic	26	7.92	±0.80	7.90	±1.19
Manual	25	7.92	±0.84	8.08	±1.10
2017: patterns					
No hints	31	8.19	±1.08	7.89	±0.83
Automatic	31	8.19	±1.19	7.91	±0.89
Manual	31	8.16	±1.10	7.92	±0.87

5.2.2 *First study: rewrites*

In 2016 we ran the study using the rewrite-based debugger to provide hints to the *automatic* group. The study was done during regular lab sessions with a teacher available for help. Students were however encouraged to solve problems on their own for the duration of the study and consult hints when necessary. There were 1,216 attempts in total, with 1,133 attempts containing a correct submission.

Table 5.2 breaks down successful attempts by problem. The second column (Time)

shows the average solving time for each problem, defined as the sum of time deltas between successive actions. We only consider actions before the first correct program is submitted, as students sometimes experiment with the code after it has passed all tests. Time deltas are capped at five minutes – if a student is idle longer than that, we consider them to have gone off-task. We exclude solutions where the student spent over ten times longer than the overall average to reach the solution for the given problem; such attempts are very rare and unusual, and the great majority of solutions are found much sooner. The third column (Subs) shows the average number of incorrect programs submitted before a submission passed all tests.

The remaining columns show, for the two hint groups, the average number of non-syntax-related hints offered (by displaying the *Hint* button after an incorrect program is submitted) during one attempt, and the percentage of those hints that were actually viewed (the student pressed the *Hint* button). Due to an unfortunate oversight, the data on viewed hints are not available for the first week of the 2016 study. Exercises in this group serve as an introduction to Prolog and are not very difficult, evidenced by the low average number of incorrect submissions.

In general, many more hints were offered to the manual group. While the automatic debugger would often fail to produce a rewrite sequence in the allotted time, there were "catch-all" manual hints defined for most problems. These hints would always trigger when more specific feedback was not available, presenting generic instructions such as "check that the recursive rule is correctly implemented". Both groups viewed about a half of the offered hints.

Table 5.3 shows the average and standard deviation for problem-solving time, number of incorrect submissions, and the number of plans requested for one attempt. Since problems vary greatly in difficulty, we normalized all three values to the average across all attempts for a given problem. When comparing the experimental groups, we only consider those attempts where at least one plan or non-syntax-related hint was shown (on request) to the student: in attempts where the first submission was correct, or no hints were available or requested, there was no chance for feedback to have an effect.

Since we select for attempts that required feedback to reach a solution, all solving times are above the overall average. Students in the *no hints* group, receiving only plans and no code-specific feedback, needed $T_0 = 1.45$ times as long as the average to solve a problem, while students receiving either automatic or manual hints only needed $T_A = T_M = 1.20$ as long as the average. Availability of hints also reduced the number

Table 5.2

Statistics for generated hints using rewrite rules. Second and third columns give the average time and incorrect submissions before a solution was found. Fourth and fifth columns give the average number of generated hints offered during one attempt, and the percentage of all offered hints that were actually viewed by students. The final two columns give the same values for manually defined hints.

Problem	Time	Subs.	Automatic hints		Manual hints	
	(s)	(#)	Offered	Viewed	Offered	Viewed
Family rel.						
grandparent/2	104	2.0	0.05	n/a	0.00	n/a
sister/2	378	4.2	1.38	n/a	1.00	n/a
brother/2	86	2.5	0.33	n/a	0.10	n/a
aunt/2	190	2.5	0.10	n/a	0.37	n/a
cousin/2	348	3.0	0.22	n/a	0.67	n/a
ancestor/2	245	3.0	0.29	n/a	0.29	n/a
descendant/2	165	2.3	0.00	n/a	0.00	n/a
Lists I						
memb/2	397	3.4	0.47	71%	1.31	24%
del/3	678	6.2	1.18	40%	2.64	49%
dup/2	752	6.9	0.75	11%	5.64	55%
conc/3	682	5.5	2.08	36%	0.79	36%
divide/3	665	5.1	1.70	47%	3.08	43%
permute/2	606	3.2	0.75	44%	2.12	53%
Lists II						
shiftleft/2	508	4.0	1.00	56%	1.36	42%
shiftright/2	462	4.3	0.40	25%	1.13	53%
rev/2	532	3.7	1.38	59%	1.88	23%
palindrome/1	370	5.5	1.93	52%	2.75	57%
{even,odd}len/1	341	3.4	1.63	71%	1.47	44%
len/2	233	3.2	0.88	53%	1.50	33%
sum/2	144	2.1	0.78	57%	0.59	30%
min/2	596	4.6	1.31	29%	2.00	62%
max/2	169	2.8	0.83	27%	1.53	22%
sublist/2	656	6.2	0.78	71%	2.27	40%

Table 5.3

Relative time and number of submissions until solution in the first study. For each experimental group, the first pair of columns shows the mean and standard deviation for the time until solution (T). The second pair of columns shows the mean and standard deviation for the number of incorrect submissions (S). The final two columns show the mean and standard deviation of plans requested (P) during one attempt. All values are normalized to the average for each problem. Statistically significant results are marked with * ($p < 0.05$).

Group	Solving time (T)		Submissions (S)		Plans (P)	
No hints	1.45	±1.15	1.47	±1.35	2.58	±2.35
Automatic	1.20	±0.66	1.12	±0.89	2.07	±2.16
Manual	1.20	±0.78*	0.99	±0.75*	2.06	±2.43*

of submissions required before reaching a solution. Students in the *no hints*, *automatic* and *manual* groups submitted $S_0 = 1.47$, $S_A = 1.12$ and $S_M = 0.99$ as many distinct incorrect programs as the average. When hints were available, users tended to request fewer plans, but the difference was not statistically significant.

Solving times and submission counts are not distributed normally, so we used the Kruskal–Wallis H-test to determine the significance of our results. We found a significant ($p < 0.05$) difference between the *no hints* and *manual* groups. While there was also a decrease in solving times and number of submissions between the *no hints* and *automatic* groups, it fell short of the significance threshold.

5.2.3 Second study: patterns

The design of the 2017 study was the same as the previous year, with the only difference in the kind of hints provided to the automatic group: instead of highlighting code fragments using the rewrite-based debugger, we pointed out buggy and missing code patterns based on negative and positive rules, as described in Section 4.4.1.

As in the previous study, we only considered first-time students, splitting them into three groups while controlling for grades in the previous year. There were 1,315 problem-solving attempts in total, with 1,223 attempts with a correct submission. Like in the previous section, Table 5.4 gives per-problem statistics for successful attempts. Data presented in each column is the same as in Table 5.2.

Unlike the previous year with hints based on rewrite rules, the *automatic* group was offered – except for a few exercises – about as many hints as the *manual* group. This confirms that pattern-based rules can provide hints in many more cases. The percentage of viewed hints was again about a half for both groups.

Table 5.4

Statistics for generated hints using code patterns. Second and third columns give the average time and incorrect submissions before a solution was found. Fourth and fifth columns give the average number of generated hints offered during one attempt, and the percentage of all offered hints that were actually viewed by students. The final two columns give the same values for manually defined hints.

Problem	Time (s)	Subs. (#)	Automatic hints Offered	Automatic hints Viewed	Manual hints Offered	Manual hints Viewed
Family rel.						
grandparent/2	105	2.6	0.79	68%	0.04	100%
sister/2	343	5.1	3.24	56%	2.47	34%
brother/2	95	2.4	0.71	41%	1.08	46%
aunt/2	188	2.2	0.38	33%	1.10	48%
cousin/2	329	3.7	3.60	38%	1.61	66%
ancestor/2	235	3.0	2.00	31%	1.29	37%
descendant/2	155	2.4	0.90	39%	0.64	64%
Lists I						
memb/2	307	2.8	1.17	52%	1.27	43%
del/3	493	4.1	2.07	38%	2.40	62%
dup/2	748	6.5	4.33	38%	5.09	63%
conc/3	555	4.7	2.00	53%	1.35	30%
divide/3	526	3.7	1.25	67%	3.06	67%
permute/2	492	4.7	0.38	100%	2.54	67%
Lists II						
shiftleft/2	478	3.7	2.08	48%	1.44	61%
shiftright/2	389	3.5	1.18	62%	0.50	88%
rev/2	617	7.2	3.43	46%	5.25	39%
palindrome/1	379	7.1	1.73	63%	1.25	70%
{even,odd}len/1	334	3.3	0.07	100%	1.79	62%
len/2	229	3.5	1.53	30%	1.50	67%
sum/2	138	1.6	0.75	42%	0.42	38%
min/2	567	5.5	4.75	44%	3.36	81%
max/2	117	2.5	0.71	40%	0.94	73%
sublist/2	541	6.6	7.00	20%	2.11	74%

As in the previous study, we compared problem-solving time and the number of submissions in the three groups. We again considered only attempts where some form of help was requested by the student: either one or more plans, or at least one non-syntax-related hint. Table 5.5 shows the average and standard deviation for problem-solving time, number of incorrect submissions, and the number of plans requested.

Table 5.5

Relative time and number of submissions until solution. For each experimental group, the first pair of columns shows the mean and standard deviation for the time until solution (T). The second pair of columns shows the mean and standard deviation for the number of incorrect submissions (S). The final two columns show the mean and standard deviation of plans requested (P) during one attempt. All values are normalized to the average for each problem. Statistically significant differences from the *no hints* group are marked with * ($p < 0.05$) or ** ($p < 0.01$).

Group	Solving time (T)		Submissions (S)		Plans (P)	
No hints	1.65	±1.08	1.31	±1.27	5.41	±3.94
Automatic	1.41	±1.05*	1.22	±1.04	0.99	±1.91**
Manual	1.22	±1.00**	1.15	±1.05	1.85	±2.95**

This year, students receiving no hints needed $T_0 = 1.65$ times as long as the average to solve a problem, while students receiving automatic and manual hints needed $T_A = 1.41$ and $T_M = 1.20$ as long, respectively. Availability of hints did not have a significant impact on the number of submissions needed to reach a solution, with students in the *no hints*, *automatic* and *manual* groups submitting $S_0 = 1.31$, $S_A = 1.22$ and $S_M = 1.15$ as many distinct incorrect programs as the average. The number of requested plans was however much lower in the *automatic* and *manual* groups.

5.2.4 Discussion

Results from both studies indicate that automatic and manual hints decrease the time needed to solve Prolog problems. Solving time was consistently lower for the *manual* group, which needed between one third and one half as much additional time as the *no hints* group, compared to the overall per-problem average. Teacher-programmed hints provide carefully crafted feedback based on years of teaching experience, and thus give us a useful baseline – it is difficult to imagine a purely data-driven and language-independent method to provide better explanations.

Still, both studies show that automatic hints do help. While the difference is not as pronounced as with manual hints, this shows that automatic feedback can provide at least some of the benefit at a fraction of teacher effort. Since the focus of our work was

on the underlying models, the hints presented in these experiments were rather basic, highlighting required modifications in the first study and erroneous variables or literals in the second. Pointing out the location of errors appears to contribute significantly to the student's debugging process.

This finding agrees with our in-class experience, where we often observed students having difficulty locating bugs in a misbehaving program, especially when learning a new programming language. Furthermore, highlighting incorrect fragments also indicates which parts of the program are already correct, providing a degree of assurance that the student is on the right path.

In the past we observed that some students resort to tinkering when faced with a buggy program – making small modifications to the program in hope of stumbling onto a solution. Automatic testing and hints might have either positive or negative effects in such cases: they can serve as a starting point to motivate a more systematic approach to debugging, or they can encourage random tinkering by limiting the range and number of variations a student has to try.

Our results seem to suggest that students receiving hints submit fewer incorrect programs, so the first option seems more likely. In any case, undesired student behavior such as tinkering or requesting many hints may be discouraged by appropriate prompts from the tutor [92]. For example, a tutor could advise students to use the interpreter to test the program themselves, suggesting relevant inputs to try.

A significant percentage of students never – or very rarely – request any feedback from the tutor beyond testing the program for correctness. These students prefer to work out the problems on their own and might consider such help "cheating". They sometimes still resort to hints if they are unable to debug a program for a long time. In our experience, such students are often more likely to accept (or request) the teacher's help, indicating the social aspect of learning.

While the results of both experiments mostly agree, there are some differences. This can likely be attributed to the fact that the studies were done during ordinary lab sessions, without strict controls. Students worked on the problems on their own, but a teacher did help when there were major difficulties. One of the teachers was replaced for the 2017 class, which might also have affected the results.

The main goal of this investigation was to establish that feedback, either manual or automatic, can play a useful role in a programming tutor. We have shown that both manual and automatic hints positively affect students' problem-solving. However,

further experiments are required in order to determine the extent of these effects and to better understand how different kinds of hints influence learning. To measure learning gains directly, students' skills should be tested before and after a tutoring session in a controlled environment, with similar problems used both for tutoring and testing.

5.2.5 User survey

In 2016 we conducted a survey to see how CodeQ was received by the students. The survey consisted of four scaled questions, and three optional open-ended questions asking for comments about the system. Table 5.6 shows mean responses to the scaled questions for each experimental group. In the last two questions, "feedback" refers to all messages from the tutor: automatic or manual hints, test results, syntax errors and planning messages. Note that for the last question the "best" answer is 1.

Table 5.6

Mean responses to the post-experiment survey (1 = no, 5 = yes) for each experimental group.

Question	Type of hints		
	None	Automatic	Manual
1. Did you find CodeQ easy to use?	4.69	4.70	5.00
2. Did CodeQ help you learn Prolog?	4.69	4.70	4.88
3. Did you find the feedback useful?	4.08	3.90	4.43
4. Was the feedback ever unclear?	2.46	3.20	2.43

There are some variations in student responses across the three groups, though they do not approach statistical significance. Nearly all students answered 4 or 5 to the first two questions. Responses to questions 3 and 4 show that students find verbal feedback (as could be expected) more useful and easier to understand than simple highlights. This is expected, as manually written hints explain the problem and point to a solution, whereas automatic hints only highlight the problematic areas. The student is left the non-trivial task of understanding the error.

The open-ended questions asked about which aspects of CodeQ the students found most useful, and what could be improved. Positive comments mainly related to ease of use afforded by an integrated online application – no installation is required, programs are automatically loaded into Prolog interpreter, and per-problem test cases allowing students to easily determine whether a solution is correct.

Suggested improvements mainly concerned usability problems in the current version of the application. Most commonly raised issues were: cumbersome access to solutions to completed problems, no indication which test case(s) have failed, and the limited functionality of the Prolog engine compared to a locally installed interpreter.

6

Conclusion

We have developed and evaluated two programming models to support hint generation in programming tutors. With both models we were able to automatically discover many common mistakes. Automatically generated hints helped students find and eliminate bugs more quickly. Furthermore, rules learned using either approach are interpretable and can help a domain expert provide manual feedback for common errors.

In order to evaluate and compare different kinds of hints, we developed an online programming environment CodeQ. We used it to teach Prolog and Python in existing courses at the Faculty of Computer and Information Science, and in several programming workshops and tutorials. We confirmed that both manually written and automatically generated hints positively affect students' problem-solving rate.

We conclude the dissertation with a few observations made during our research. We started our research with the rewrite-based model in order to approximate the approach used by model-tracing tutors, where the problem-solving process is represented as a series of well-defined steps. We have shown how such steps can be learned for programming exercises as rewrite rules. While successful, the evaluation of these rules on student programs led us to believe that programming is inherently different from other tutoring domains.

When solving a math problem, for instance, a student will typically progress through one or more states with a partial solution. These intermediate states are not incorrect, merely incomplete. In programming, however, the first program submitted by a student will usually be complete in the sense that the student expects it to solve the problem. However, the first submission will often contain one or more errors. The tutor's job is then to discover these errors and suggest fixes.

This was our first major insight: for programming, a "bug library" approach makes much more sense than trying to account for all the different ways a student can type in a program. Our rewrite rules essentially serve as such a library, with each rule – or sometimes a pair of rules – corresponding to a common issue in some subset of student programs. Indeed, as can be seen from the results in Section 3.5.1, very few discovered fixes apply more than one or two rewrites to correct a program.

The main issue with rewrite rules is the long and unpredictable time required to debug a program. While the applicability of each rule is limited to some extent, the debugger must still try many possibilities. Ordering rules by frequency helps, but the problem remains. The real challenge for programming tutors is thus determining whether or not a specific bug is present. With our second model we therefore focused on discovering

program patterns that would indicate the presence of errors.

While it is generally impossible to reason about a program's behavior purely based on its syntactic structure, we have found that an extension of regular expressions to trees – based on Tregex [103] – works very well for our domains. Using only simple patterns relating pairs of variables or values we were able to predict program correctness with high accuracy. Despite an extensive literature survey we found almost no use of regular expressions on trees outside the natural-language-processing community. Such expressions can succinctly describe tree features while being more general than subtrees or other commonly used attributes, and we believe that this approach should be useful for other kinds of tree-structured data.

For our classroom evaluations we introduced CodeQ to lab sessions in existing courses. We found that hints are helpful to students, but it is important to also note the effect of the learning environment itself. Even when no hints were available, students were more eager to solve all problems each week than when using an ordinary editor and interpreter. This is likely due to the ease of use afforded by the environment. Another possible explanation is that participating in a research project motivated students to care more about the class, and thus spend more effort to do well.

This last observation brings us back to ideas about the evolving role of computers from the introductory section. Intelligent tutoring systems can certainly improve the learning experience: on the one hand by providing some feedback to the student when a teacher is not available, and, on the other hand, by freeing teachers from having to explain simple errors over and over, allowing them to focus instead on more difficult cases. We should be careful, however, not to diminish the teacher's role to simply providing problem-solving feedback. Learning is a social process, and teachers should first and foremost motivate and contextualize knowledge – both firmly out of reach for current AI methods.

Razširjeni povzetek

A

Uvod

Računalniki so prisotni na vseh področjih človekovega delovanja. Ključna lastnost, ki mu to omogoča, je, da je računalnik *splošnonamenski* stroj – torej stroj, ki ga lahko sprogramiramo za poljubno opravilo. Večina ljudi danes uporablja vrsto programov za različne namene, le malo pa jih računalnik uporablja za izdelavo novih orodij s programiranjem. Zato obstaja velik razkorak med računalnikovim potencialom in vlogami, ki jih trenutno opravlja. Preden lahko ta potencial dosežemo, pa ga mora razumeti dovolj ljudi [2]. Dosedanji razvoj našega razumevanje računalnika lepo povzame Douglas Adams:

> Najprej smo mislili, da je računalnik kalkulator. Potem smo ugotovili, kako spremeniti številke v črke z ASCII – in smo mislili, da je pisalni stroj. Potem smo odkrili grafiko in mislili, da je televizija. S svetovnim spletom smo končno ugotovili, da je reklamna brošura.

Politiki, ki želijo prepovedi šifriranje, kažejo, da smo kot družba še zmeraj daleč od razumevanja tega, kaj računalnik v resnici je. Dokler večina ljudi ni znala brati, pisana beseda ni mogla imeti pomembnega vpliva na mišljenje. Prav tako bodo učinki računalnikov – in naše razumevanje teh učinkov – omejeni, dokler jih večina ne bo znala uporabljati (v smislu ustvarjanja orodij, tj. programiranja).

Zato ni presenetljivo, da programiranje mnogi smatrajo za „novo pismenost" [3, 4]. S tem običajno mislijo, da je znanje programiranja zmeraj pomembnejše in bi moralo biti dostopno vsem. A tako kot pri pismenosti ne gre samo za prevajanje med črkami in glasovi, bistvo programiranja ni v pisanju kode, temveč v zmožnosti, da izrazimo svoj miselni model v okvirih dobro definiranega formalnega sistema.

Programiranje – še posebej odkrivanje in popravljanje napak v nepravilnih programih – zahteva visoko stopnjo introspekcije, da odkrijemo skrite predpostavke v našem razumevanju sveta. Med razhroščevanjem pogosto ugotovimo, da je naš miselni model programa napačen ali premalo podroben. Ko popravimo model, lahko običajno popravimo tudi program. Programiranje tako ponuja ogromno priložnosti za razvoj splošnih kognitivnih spretnosti [5].

Tudi zato postaja pouk programiranja vse bolj razširjen. Nekatere države so ga že uvedle v programe osnovnih in srednjih šol, poleg tega pa tako na državni kot meddržavni ravni obstajajo številne iniciative za popularizacijo programiranja. V zadnjih

desetih letih so zelo priljubljeni tudi spletni tečaji (angl. *massive open online course* oziroma *MOOC*) programiranja. Na te tečaje je lahko vpisanih več deset tisoč učencev, zato učitelj ne more vsakemu posamezniku dati individualnih komentarjev in nasvetov; ravno ti pa so zelo koristne pri učenju.

S problemom samodejnega podajanja povratnih informacij se ukvarja področje inteligentnih sistemov za poučevanje (angl. *intelligent tutoring system*, v nadaljevanju *ITS* oz. tutor). Pričujoča disertacija raziskuje problem samodejnega generiranja namigov (angl. *hint*) pri poučevanju programiranja. Najtežja naloga pri razvoju takega sistema je gradnja domenskega modela, s katerim sistem v učenčevi rešitvi odkrije napake in na podlagi teh napak učenci svetuje, kako nadaljevati. Za razvoj ustreznega modela je lahko potrebnih tudi več sto ur dela za vsako učno uro materiala [9].

Poleg motivacije za razvoj metod za podajanje namigov nam spletni tečaji omogočajo tudi samodejno učenje domenskega modela. Z njimi lahko namreč zberemo veliko količino podatkov o reševanju programerskih nalog, na podlagi katerih lahko računalnik sam odkrije tipične pristope in napake. Samodejni namigi se sicer po kakovosti običajno ne morejo kosati z ročno izdelanimi modeli, vendar se tak sistem lahko nauči podajati namige za nove naloge brez dodatnega dela.

Prispevki k znanosti

V okviru doktorske naloge smo razvili dva pristopa za samodejno učenje pogostih napak pri programiranju. Poleg generiranja povratnih informacij pokažemo tudi, da sta oba modela lahko v pomoč učitelju pri pisanju in izboljšavi razlag. Opišemo tudi spletno okolje za poučevanje programiranja CodeQ, v katerem smo razvite metode preizkusili.

- *Model programiranja na podlagi prepisovalnih pravil.* Proces reševanja programerskih nalog formaliziramo z zaporedjem transformacij programske kode. Predstavimo algoritem, ki iz obstoječih rešitev posamezne naloge izlušči prepisovalna pravila, ki opisujejo te transformacije, in podamo primere pravil iz zbranih podatkov. Prepisovalna pravila lahko uporabimo za generiranje novih različic programov. Razhroščevanje modeliramo kot iskanje primernega zaporedja transformacij in razložimo, kako lahko iz najdenega zaporedja podamo povratne informacije učencu. Pristop ovrednotimo na obstoječih programih v prologu in v razredu z uporabo spletnega okolja za programiranje CodeQ.

- *Model programiranja na podlagi sintaktičnih vzorcev.* Relacije med spremenljivkami in vrednostmi predstavimo z vzorci v abstraktnih sintaktičnih drevesih. Vzorce dobimo iz obstoječih rešitev in jih uporabimo kot atribute pri učenju klasifikacijskih pravil za napovedovanje pravilnosti programov. Pravila za pravilne programe interpretiramo kot različne možne rešitve posamezne naloge, medtem ko lahko iz pravil za nepravilne programe vidimo pogoste napake. Pokažemo, kako obe vrsti pravil uporabimo za samodejno podajanje namigov in analizo pogostih napak. Namige ovrednotimo na obstoječih programih v prologu in pythonu ter v razredu.

Znanstvena izhodišča

Že prvi računalniki so se uporabljali tudi za izobraževanje. Korenine inteligentnih sistemov za poučevanje lahko najdemo v sistemih kot je PLATO [16]. Ti zgodnji sistemi so služili predvsem kot interaktivna zbirka nalog, kjer je vsak učenec lahko napredoval s svojim tempom. Povratne informacije so bile največkrat omejene na preverjanje rešitev: sistem je učencu lahko povedal le, da njegova rešitev ni pravilna, ne pa zakaj.

Še pred razvojem ITS, ki gradijo predvsem na izboljšanih povratnih informacijah, so se pojavili t.i. mikrosvetovi (angl. *microworld*). Ti učencu nudijo poenostavljeno okolje z agenti, ki jih lahko programira. Najstarejši in najbolj znan primer je Logo [19], danes pa sta precej razširjena tudi Scratch [21] (glej sliko 2.1 na strani 11) in Alice [20]. Za razliko od ITS mikrosvetovi ponavadi nimajo vnaprej določenih nalog, zato so povratne informacije omejene na opozorila o sintaktičnih napakah.

ITS razširijo delovanje prejšnjih sistemov v dveh glavnih smereh. Prvič, znajo se prilagoditi nivoju znanja posameznega učenca, in drugič, analizirati znajo tudi vmesne rešitve in učencu svetovati, kako naprej. Delovanje ITS lahko opišemo skozi *zunanjo* in *notranjo zanko* [30]. Naloga zunanje zanke je izbrati naslednjo nalogo, ki bo učencu najbolj koristila, notranja zanka pa sledi učenčevemu reševanju posamezne naloge in mu nudi sprotno povratno informacijo. V našem delu smo se osredotočili na implementacijo notranje zanke v sistemih za poučevanje programiranja.

Prvo večjo strujo v razvoju ITS predstavljajo kognitivni tutorji (angl. *cognitive* oziroma *model-tracing tutors*). Delujejo na kognitivni theory ACT, ki razlikuje med deklarativnim in proceduralnim znanjem [43]. Kognitivni tutor vsebuje nabor pravil, s pomočjo katerih zna sam reševati naloge, kar mu omogoča sledenje učenčevim korakom in zaznavanje napak. Izdelava kognitivnega modela je zahtevna naloga, predvsem

v kompleksnih domenah kot je programiranje. Izdelanih je bilo le nekaj kognitivnih tutorjev za programiranje; večinoma za funkcijske jezike, v katerih se da razvoj programa opisati z zaporedjem izostritev [47, 48].

Drug in pogosteje uporabljan pristop je model z omejitvami (angl. *constraint-based modeling*) [49]. Tak model namesto pravil za reševanje vsebuje množico omejitev, ki morajo veljati za vse pravilne rešitve. Če učenčeva rešitev ne ustreza kakšni omejitvi, mu sistem na podlagi te omejitve poda nasvet. S takim modelom sicer ne moremo iskati novih rešitev nalog, ga je pa precej lažje razviti in posodabljati. Model z omejitvami uporablja SQL-tutor [50] (glej sliko 2.5 na strani 17), zaradi enostavnega razvoja pa tudi številni programerski tutorji [52, 53, 53, 54].

Ostali programerski tutorji temeljijo na referenčnih rešitvah [55–60] ali katalogih pogostih napak [61–63]. Pri prvem pristopu učitelj za vsak problem definira nabor tipičnih rešitev, tutor pa išče razlike med njimi in učenčevo napačno rešitvijo. Za drugi pristop učitelj definira katalog pogostih napak za posamezne programe.

Pri vseh opisanih pristopih potrebuje tutor za vsako nalogo poleg opisa in pričakovane rešitve tudi vnaprej definiran model napak. Ne glede na pristop zahteva razvoj tega modela precej truda. Pri vse večjih količinah podatkov, ki jih lahko zbiramo preko spletnih tečajev, se poraja ideja, da bi se računalnik domenskega modela naučil samodejno iz preteklih uporabniških rešitev z uporabo podatkovno vodenih metod [10].

Številni podatkovno vodeni pristopi opišejo reševanje naloge z zaporedjem različic programa, ki jih je učenec poslal v testiranje [12, 73, 74, 79, 80]. Glavna težava, s katero se soočajo ti pristopi, je, da spremembe med dvema zaporednima različicama ponavadi ne ustrezajo dobro definiranim akcijam. Druga težava je raznolikost rešitev v programerskih domenah, saj se da isti program zapisati na mnogo načinov. Ta problem lahko omilimo s poenotenjem različnih programov v najbolj tipično obliko [61, 72–74, 76].

Prepisovalna pravila

Naš prvi domenski model definira *prepisovalna pravila* (angl. *rewrite rules*), s katerimi lahko tutor iz obstoječega programa generira nove različice. Za učenje modela smo uporabili podatke, ki smo jih zbrali v naši spletni aplikaciji CodeQ. Ti podatki nam omogočajo podroben uvid v razvoj posameznih rešitev, saj zajemajo vse učenčeve akcije: vpisane in izbrisane znake programske kode ter informacije o testih in poizvedbah, ki jih je učenec pognal med reševanjem. Zaporedje akcij enega učenca pri reševanju enega

problema imenujemo *sled*.

Prepisovalna pravila delujejo na nivoju besedila oziroma programske kode. Pravilo $pot : a \longrightarrow b$ izbriše kos kode, ki se ujema z a in se pojavi na *poti* od korena sintaktičnega drevesa programa, in namesto njega vstavi novo različico b. Vsako prepisovalno pravilo definira neko zaključeno spremembo programske kode (glej sliko 3.1 na strani 36). Z uporabo teh pravil lahko iz poljubnega programa ustvarimo nove različice, tudi, če ta program vidimo prvič.

Pravil se učimo tako, da v posameznih sledeh združimo zaporedne akcije (tj. vstavljene in izbrisane znake) v določenem kosu programa (glej sliko 3.2 na strani 37). Pri tem si zapomnimo tudi pot do tega kosa od korena abstraktnega sintaktičnega drevesa originalnega programa. Ker želimo najti pravila, ki bi nepravilen program s čim večjo verjetnostjo spremenila v pravilnega, obravnavamo le zaporedja akcij, ki so privedla do programa, ki je bliže rešitvi. Da lahko pravila primerjamo, imena spremenljivk v dobljenih pravilih popravimo na standardne vrednosti.

Na ta način iz obstoječih sledi izluščimo nabor prepisovalnih pravil, ki jih nato uporabljamo za popravljanje programov po naslednjem postopku. V nepravilnem programu P uporabimo vsa primerna pravila $pot : a \longrightarrow b$ – torej tista, za katera najdemo v programu P kos a na mestu pot – tako, da kos a zamenjamo z b. S tem dobimo več novih programov. Če kateri izmed njih predstavlja pravilno rešitev, smo končali, sicer pa tako dobljene programe popravljamo naprej po istem postopku, pri čemer najprej obdelamo programe, ki smo jih dobili po pogosteje uporabljanih pravilih.

Ko najdemo pravilen program, lahko iz dobljenega zaporedja pravil poiščemo kose originalnega programa, ki jih je potrebno popraviti. Namig za učenca samodejno tvorimo tako, da te kose označimo (glej sliko 5.2 na strani 86). Ta postopek je preprost in neodvisen od programskega jezika.

Poleg samodejnih namigov nam prepisovalna pravila omogočajo tudi analizo tipičnih napak pri posameznih nalogah. Učitelju prikažemo pravila (ali kombinacij pravil), ki se pojavijo v največ sledeh, poleg vsakega pravila pa še seznam nepravilnih programov, za katere je bilo to pravilo uporabno. Pokazali smo, da se da iz teh informacij enostavno razbrati pogoste napake. Učitelj mora nato dodati le še komentar, ki naj se prikaže učencu, ko sistem v njegovem programu odkrije katero od teh napak.

S prepisovalnimi pravili smo želeli ustvariti model, ki bi deloval podobno kot kognitivni tutorji: s pomočjo tipičnih „programerskih akcij“ (prepisovalnih pravil) bi znal sam generirati programe. To nam je do neke mere uspelo, vendar smo pri tem spoznali,

da učenci pri reševanju programerskih nalog postopajo drugače kot recimo pri nalogah iz fizike. Pri slednjih se da proces reševanja lepo opisati z zaporedjem korakov od problema do rešitve, medtem ko pri programiranju učenec običajno napiše cel program v enem zamahu, nato pa v njem popravlja posamezne napake. Naš drugi model, ki temelji na sintaktičnih vzorcih, je zato namenjen predvsem odkrivanju napak v programih.

Sintaktični vzorci

Največja slabost prepisovalnih pravil je, da popravljanje programov lahko traja precej časa, saj moramo vsako novo različico preizkusiti. Pri preizkušanju moramo program pognati, kar oteži izvedbo. Po drugi strani pa nam najdeno zaporedje popravkov zagotavlja, da smo odkrili vse napake v originalnem programu. V drugem delu raziskovalne naloge smo si zastavili bolj ambiciozen cilj: ali lahko samo na podlagi strukture programa z dovolj veliko gotovostjo ugotovimo, če je pravilen, in v nasprotnem primeru ugotovimo, kateri del programa predstavlja napako?

Glavna ovira pri tem je neverjetna raznolikost – še posebej nepravilnih – programov, ki jih oddajo učenci tudi za najenostavnejše naloge. Na nekem spletnem tečaju so recimo za preprosto nalogo (rešitev je prikazana na strani 20) prejeli čez deset tisoč različnih programov. Poiskati želimo invariante, ki označujejo prisotnost določenih napak v čim bolj različnih programih.

V ta namen smo uporabili *sintaktične vzorce* (angl. *AST patterns*), s katerimi posplošimo regularne izraze na drevesne strukture. Ideja izhaja iz programa Tregex [103], kjer tak pristop uporabljajo za iskanje stavkov z določeno strukturo v besedilnih korpusih. Sintaktični vzorec opiše strukturo abstraktnega sintaktičnega drevesa programa, pri čemer upošteva le določene dele te strukture.

S sintaktičnimi vzorci lahko predstavimo dva tipa relacij. Vzorec $(n\ p_1\ \dots\ p_k)$ pomeni, da 1) drevo vsebuje vozlišče n in 2) poddrevo s korenom v n vsebuje različna vozlišča n_1 do n_k, ki se ujemajo z vzorci p_1 do p_k. Pri tem morajo vozlišča n_1 do n_k nastopati v tem vrstnem redu pri obhodu drevesa v globino.

V našem delu smo uporabili le vzorce, ki povezujejo dve pojavitvi spremenljivke ali vrednosti v programu (za primer glej sliko 4.1 na strani 63). Ti vzorci opisujejo interakcije med pari podatkovnih objektov v programu; take pare obravnavamo kot najmanjše zaključene pomenske enote. Na ta način smo ustvarili nabor atributov, na podlagi katerih lahko izvajamo strojno učenje na programih. Za vsako nalogo smo se iz množice pravilnih in nepravilnih programov, ki so jih napisali učenci, naučili

klasifikacijskih pravil, ki na podlagi vzorcev povejo, ali je program pravilen (pozitivna pravila) ali ne (negativna pravila).

Negativno pravilo oblike „$p_1 \wedge \ldots \wedge p_n \Rightarrow$ incorrect“ pomeni, da kombinacija vzorcev p_1 do p_n predstavlja napako v programu, ki jo je potrebno odpraviti. Namig za učenca enostavno pripravimo tako, da označimo ustrezne kose programa (glej primer na strani 70). Pri tem za razliko od prepisovalnih pravil ne potrebujemo točnega postopka, kako program popraviti.

Pozitivno pravilo oblike „$p_1 \wedge \ldots \wedge p_n \Rightarrow$ correct“ pomeni, da kombinacija vzorcev p_1 do p_n nakazuje veliko verjetnost, da je program že pravilen. Če za učenčev nepravilen program nismo našli nobenega negativnega pravila, poiščemo najbližje pozitivno pravilo – torej tisto, za katero program vsebuje največ vzorcev v pogoju pravila. Kot namig nato učencu prikažemo vzorce, ki v programu še manjkajo.

Z uporabo pravil smo uspeli pravilno odkriti napake v treh četrtinah nepravilnih programov. Zanimivo je, da prevladujejo pravila oblike „$p_1 \Rightarrow$ incorrect“, torej pravila, ki program na podlagi enega samega vzorca označijo za napačnega. Na podlagi tega sklepamo, da so izbrani vzorci ustrezni za opis posameznih konceptov oziroma napak v programih. Tako kot prepisovalna pravila lahko sintaktični vzorci služijo tudi kot podlaga za analizo tipičnih napak. Učitelj lahko poda razlage za nekaj najpogostejših pravil in s tem pokrije dobršen odstotek vprašanj, ki jih imajo učenci pri reševanju.

CodeQ

Razvili smo spletno aplikacijo za učenje programiranja CodeQ[1]. Aplikacija omogoča samostojno reševanje programerskih nalog v jezikih prolog in python, uporabljamo pa jo tudi pri nekaterih predmetih na Fakulteti za računalništvo in informatiko ter raznih tečajih.

Po prijavi v aplikacijo učenec izbere nalogo (glej sliko 5.1 na strani 85) in začne z reševanjem. Prednost spletnega okolja je, da ne zahteva namestitve, vse rešitve pa se hranijo na strežniku, tako da so dostopne od koderkoli. Vmesnik za reševanje nalog ima tri glavne komponente: opis naloge, urejevalnik besedila in tolmač za izbran programski jezik (glej sliko 5.2 na strani 86).

Učenec lahko s pomočjo tolmača poganja poljubne poizvedbe, pri čemer se v okolje samodejno naloži trenutna različica programa. Za vsako nalogo je napisanih še ne-

[1]Dostopna na `https://codeq.si`. Koda je dostopna pod licenco AGPL3+ na `https://codeq.si/code`.

kaj dodatnih napotkov, ki jih lahko učenec zahteva, če pri reševanju naleti na težave. CodeQ nudi tudi možnost preverjanja rešitev; pri tem požene učenčev program na različnih testnih primerih in primerja izhod programa s pričakovanimi vrednostmi. Če program ni pravilen, poskusi z zgoraj opisanimi metodami v njem samodejno odkriti napake in jih predstaviti učencu v obliki namiga.

CodeQ smo uporabili za evalvacijo učinkovitosti namigov, ustvarjenih na podlagi prepisovalnih pravil in vzorcev, in namigov, ki jih je napisal učitelj. Pri tem smo opazovali čas, ki so ga učenci porabili za reševanje posameznih nalog, in število nepravilnih programov, ki so jih pri tem oddali. Čas reševanja – oziroma z njim povezano število rešenih nalog – je namreč močno povezan z učnimi dosežki [111].

Ugotovili smo, da je skupina brez namigov po pričakovanjih potrebovala v povprečju največ časa za reševanje, skupina, ki je prejemala učiteljeve namige, pa najmanj. Namigi, ki smo jih dobili samodejno s pomočjo zgoraj opisanih modelov, so bili nekje vmes. Pri tem so se namigi na podlagi strukturnih vzorcev izkazali za boljše kot namigi na podlagi prepisovalnih pravil. To ni presenetljivo, saj drugi pristop deluje na veliko večjem naboru programov.

Zaključek

Razvili smo dva modela programiranja, uporabna za podajanje namigov v inteligentnih sistemih za poučevanje programiranja. Z obema modeloma smo uspešno odkrili številne pogoste napake pri programiranju. Uporaba metod v razredu kaže, da so samodejno generirani namigi pomagali učencem hitreje odkriti napake. Poleg tega so odkrita pravila razumljiva in lahko služijo učitelju pri analizi tipičnih napak.

Za preizkus metod smo razvili spletno aplikacijo CodeQ za učenje programiranja, ki smo jo uporabili za poučevanje prologa in pythona. Pokazali smo, da tako ročno izdelani kot samodejni namigi učencem pomagajo hitreje reševati naloge.

Na koncu izpostavimo nekaj ugotovitev, do katerih smo prišli med raziskovanjem. Najpomembnejša je ta, da v sistemih za poučevanje programiranja domenski model, ki opisuje postopek reševanja naloge – kot npr. v kognitivnih tutorjih – ni najbolj primeren. Programiranje običajno ne moremo opisati z zaporedjem smiselnih, dobro definiranih korakov. Učenci namreč tipično napišejo cel program v enem zamahu, nato pa iščejo in odpravljajo posamezne napake. Domenski model, ki opisuje posamezne napake, je zato precej primernejši kot model, ki bi opisoval razvoj celotnega programa (kot v primeru kognitivnih tutorjev).

Zanimivo je, da nam je na podlagi dokaj omejenega nabora sintaktičnih vzorcev uspelo z veliko gotovostjo napovedati pravilnost programa, ne da bi ga pognali. To kaže na možnost širše uporabe regularnih izrazov za drevesa, ki pa je izven področja procesiranja naravnega jezika praktično nismo našli.

Pokazali smo, da namigi pomagajo učencem hitreje reševati naloge. Pomembno je omeniti še učinek samega učnega okolja CodeQ. Glede na naše izkušnje iz prejšnjih let so učenci bili bolj motivirani za reševanje, tudi, ko namigi niso bili na voljo. Vsaj deloma je to zagotovo zaradi enostavnejšega pisanja in poganjanja programov, pa tudi zaradi nabiranja „točk“ – čeprav ni bilo nobene eksplicitne nagrade, so študentje vztrajali dlje časa, da bi rešili vse naloge za posamezni teden. Tudi sodelovanje v raziskovalnem projektu je študente morda dodatno motiviralo.

Računalnik nam torej lahko pomaga pri učenju programiranja: po eni strani lahko učencu poda povratno informacijo, kadar učitelj ni na voljo, po drugi strani pa lahko učitelja deloma razbremeni in mu pomaga bolje razumeti težave, ki jih imajo učenci. Pri tem razmišljanju pa moramo paziti, da učiteljeve vloge ne zreduciramo na podajanje povratnih informacij pri reševanju problemov. Učenje je socialni proces, ki ga mora učitelj predvsem motivirati, iskano znanje po postaviti v ustrezen kontekst – naloge, ki so daleč izven dosega obstoječih metod umetne inteligence.

Prolog grammar

B

This chapter gives the grammar used to parse Prolog programs when extracting rewrites, implemented in PLY[1]. Our parser only supports a subset of Prolog programs; in particular, dynamic operators are not supported.

Token definitions

```
operators = {
    r':-': 'FROM',
    r'-->': 'FROMDCG',
    r'->': 'IMPLIES',
    r'\+': 'NOT',
    r'=': 'EQU',
    r'\=': 'NEQU',
    r'==': 'EQ',
    r'\==': 'NEQ',
    r'=..': 'UNIV',
    r'is': 'IS',
    r'=:=': 'EQA',
    r'=\=': 'NEQA',
    r'<': 'LT',
    r'=<': 'LE',
    r'>': 'GT',
    r'>=': 'GE',
    r'@<': 'LTL',
    r'@=<': 'LEL',
    r'@>': 'GTL',
    r'@>=': 'GEL',
    r'#=': 'EQFD',
    r'#\=': 'NEQFD',
    r'#<': 'LTFD',
    r'#=<': 'LEFD',
    r'#>': 'GTFD',
    r'#>=': 'GEFD',
    r'in': 'IN',
    r'ins': 'INS',
    r'..': 'THROUGH',
    r'+': 'PLUS',
    r'-': 'MINUS',
    r'*': 'STAR',
    r'/': 'DIV',
```

[1] Available at http://www.dabeaz.com/ply/ply.html.

```
    r'//': 'IDIV',
    r'mod': 'MOD',
    r'**': 'POW',
    r'^': 'POW',
    r'.': 'PERIOD',
    r',': 'COMMA',
    r';': 'SEMI'
}
tokens = sorted(list(operators.values())) + [
    'UINTEGER', 'UREAL',
    'NAME', 'VARIABLE', 'STRING',
    'LBRACKET', 'RBRACKET', 'LPAREN', 'RPAREN', 'PIPE',
    'LBRACE', 'RBRACE', 'INVALID'
]

# punctuation
t_LBRACKET = r'\['
t_RBRACKET = r'\]'
t_LPAREN = r'\('
t_RPAREN = r'\)'
t_PIPE = r'\|'
t_LBRACE = r'{'
t_RBRACE = r'}'

# literals
t_UINTEGER = r'[0-9]+'
t_UREAL    = r'[0-9]+\.[0-9]+([eE][-+]?[0-9]+)?|inf|nan'
t_VARIABLE = r'(_|[A-Z])[a-zA-Z0-9_]*'
t_STRING   = r'"(""|\\.|[^\"])*"'

# strongest-binding operators first
precedence = (
    ('nonassoc', 'FROM', 'FROMDCG'),
    ('right', 'PIPE'),
    ('right', 'IMPLIES'),
    ('right', 'NOT'),
    ('nonassoc', 'EQU', 'NEQU', 'EQ', 'NEQ', 'UNIV', 'IS',
                 'EQA', 'NEQA', 'LT', 'LE', 'GT', 'GE', 'LTL',
                 'LEL', 'GTL', 'GEL', 'IN', 'INS', 'THROUGH',
                 'EQFD', 'NEQFD', 'LTFD', 'LEFD', 'GTFD',
                 'GEFD'),
    ('left', 'PLUS', 'MINUS'),
```

```
    ('left', 'STAR', 'DIV', 'IDIV', 'MOD'),
    ('nonassoc', 'POW'),
    ('right', 'UMINUS', 'UPLUS'),
    ('nonassoc', 'UINTEGER', 'UREAL'),
    ('nonassoc', 'NAME', 'VARIABLE', 'STRING'),
    ('nonassoc', 'PERIOD'),
    ('nonassoc', 'LBRACKET', 'RBRACKET', 'LPAREN', 'RPAREN',
                 'COMMA', 'SEMI', 'LBRACE', 'RBRACE')
)
```

Parser rules

```
text : text clause
clause : head PERIOD
clause : head FROM or PERIOD
       | head FROMDCG or PERIOD
head : term
or : if
or : or SEMI if
if : and
if : and IMPLIES if
and : term
and : and COMMA term
term : functor LPAREN RPAREN
term : functor LPAREN args RPAREN
term : LPAREN or RPAREN
term : term PLUS term
     | term MINUS term
     | term STAR term
     | term POW term
     | term DIV term
     | term IDIV term
     | term MOD term
     | term EQU term
     | term NEQU term
     | term EQ term
     | term NEQ term
     | term UNIV term
     | term IS term
     | term EQA term
     | term NEQA term
     | term LT term
     | term LE term
```

```
      | term GT term
      | term GE term
      | term LTL term
      | term LEL term
      | term GTL term
      | term GEL term
      | term PIPE term
      | term THROUGH term
      | term IN term
      | term INS term
      | term EQFD term
      | term NEQFD term
      | term LTFD term
      | term LEFD term
      | term GTFD term
      | term GEFD term
term : NOT term
      | MINUS term %prec UMINUS
      | PLUS term %prec UPLUS
term : list
term : STRING
      | NAME
      | UINTEGER
      | UREAL
      | VARIABLE
term : LBRACE clpr RBRACE
args : term
args : args COMMA term
list : LBRACKET RBRACKET
list : LBRACKET args RBRACKET
list : LBRACKET args PIPE term RBRACKET
functor : NAME
clpr : clpr_constr
clpr : clpr_constr COMMA clpr
      | clpr_constr SEMI clpr
clpr_constr : term
```

BIBLIOGRAPHY

[1] Seymour Papert. What's the big idea? Toward a pedagogy of idea power. *IBM Systems Journal*, 39(3.4): 720–729, 2000.

[2] Alan Kay. The real computer revolution hasn't happened yet. *Viewpoints Research Institute*, 15, 2007.

[3] Annette Vee. Understanding computer programming as a literacy. *Literacy in Composition Studies*, 1(2): 42–64, 2013.

[4] Douglas Belshaw et al. *What is 'digital literacy'?* PhD thesis, Durham University, 2012.

[5] Yuen-Kuang Cliff Liao and George W Bright. Effects of computer programming on cognitive outcomes: A meta-analysis. *Journal of Educational Computing Research*, 7(3):251–268, 1991.

[6] John R Anderson and Edward Skwarecki. The automated tutoring of introductory computer programming. *Communications of the ACM*, 29(9): 842–849, 1986.

[7] Antonija Mitrovic. Fifteen years of constraint-based tutors: what we have achieved and where we are going. *User Modeling and User-Adapted Interaction*, 22 (1-2):39–72, 2012.

[8] Tom Murray. An overview of intelligent tutoring system authoring tools: Updated analysis of the state of the art. In Tom Murray, Stephen Blessing, and Shaaron Ainsworth, editors, *Authoring tools for advanced technology learning environments*, chapter 17, pages 491–544. Springer Netherlands, 2003.

[9] Jeremiah T Folsom-Kovarik, Sae Schatz, and Denise Nicholson. Plan ahead: Pricing ITS learner models. In *Proc. 19th Behavior Representation in Modeling & Simulation Conference*, pages 47–54, 2010.

[10] Kenneth R Koedinger, Emma Brunskill, Ryan S J d Baker, Elizabeth A McLaughlin, and John Stamper. New potentials for data-driven intelligent tutoring system development and optimization. *AI Magazine*, 34(3):27–41, 2013.

[11] Jonathan Huang, Chris Piech, Andy Nguyen, and Leonidas Guibas. Syntactic and functional variability of a million code submissions in a machine learning MOOC. In *Proc. Workshops 16th Int'l Conf. Artificial Intelligence in Education (AIED 13)*, pages 25–32, 2013.

[12] Chris Piech, Mehran Sahami, Jonathan Huang, and Leonidas Guibas. Autonomously generating hints by inferring problem solving policies. In *Proc. 2nd ACM Conference on Learning @ Scale (L@S 2015)*, pages 195–204, 2015.

[13] Ludy T Benjamin. A history of teaching machines. *American psychologist*, 43(9):703, 1988.

[14] Valerie J Shute and Joseph Psotka. Intelligent tutoring systems: Past, present, and future. In D Jonassen, editor, *Handbook of research for educational communications and technology*, chapter 19, pages 570–600. Macmillan, New York, 1996.

[15] B F Skinner. Teaching machines. *Science*, 128(3330): 969–977, 1958. doi: 10.1126/science.128.3330.969.

[16] D Bitzer, P Braunfeld, and W Lichtenberger. PLATO: An automatic teaching device. *IRE Transactions on Education*, 4(4):157–161, Dec 1961. ISSN 0893-7141. doi: 10.1109/TE.1961.4322215.

[17] Lloyd P. Rieber. Computer-based microworlds: A bridge between constructivism and direct instruction. *Educational Technology Research and Development*, 40 (1):93–106, 1992. ISSN 10421629, 15566501.

[18] Y Papadopoulos and S Tegos. Using microworlds to introduce programming to novices. In *16th Panhellenic Conference on Informatics*, pages 180–185, 2012. doi: 10.1109/PCi.2012.18.

[19] Seymour Papert. Microworlds: transforming education. In *Artificial intelligence and education*, volume 1, pages 79–94, 1987.

[20] Stephen Cooper, Wanda Dann, and Randy Pausch. Alice: a 3-d tool for introductory programming concepts. In *Journal of Computing Sciences in Colleges*, volume 15, pages 107–116. Consortium for Computing Sciences in Colleges, 2000.

[21] John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. The scratch programming language and environment. *ACM Transactions on Computing Education (TOCE)*, 10(4): 16, 2010.

[22] E Wenger. *Artificial Intelligence and Tutoring Systems*. Morgan Kaufmann, 1987. ISBN 9781483221113.

[23] Jaime R Carbonell. AI in CAI: An artificial-intelligence approach to computer-assisted instruction. *IEEE transactions on man-machine systems*, 11(4): 190–202, 1970.

[24] Beverly Park Woolf. *Building intelligent interactive tutors: Student-centered strategies for revolutionizing e-learning*. Morgan Kaufmann, 2010.

[25] Martina A Rau, Vincent Aleven, and Nikol Rummel. Intelligent tutoring systems with multiple representations and self-explanation prompts support learning of fractions. In *Artificial Intelligence in Education: Building Learning Systems that Care: From Knowledge Representation to Affective Modelling, Proceedings of the 14th International Conference on Artificial Intelligence in Education, AIED 2009, July 6-10, 2009, Brighton, UK*, pages 441–448, 2009. doi: 10.3233/978-1-60750-028-5-441.

[26] Davide Fossati, Barbara Di Eugenio, Stellan Ohlsson, Christopher Brown, and Lin Chen. Data driven automatic feedback generation in the iList intelligent tutoring system. *Technology, Instruction, Cognition and Learning*, 10(1):5–26, 2015.

[27] Kurt VanLehn, Collin Lynch, Kay Schulze, Joel A Shapiro, Robert Shelby, Linwood Taylor, Don Treacy, Anders Weinstein, and Mary Wintersgill. The Andes physics tutoring system: Lessons learned. *International Journal of Artificial Intelligence in Education*, 15(3): 147–204, 2005.

[28] Kenneth R Koedinger, John R Anderson, William H Hadley, and Mary A Mark. Intelligent tutoring goes to school in the big city. *International Journal of Artificial Intelligence in Education*, 8(1):30–43, 1997.

[29] Carole R Beal, Ivon Arroyo, Paul R Cohen, and Beverly P Woolf. Evaluation of AnimalWatch: An intelligent tutoring system for arithmetic and fractions. *Journal of Interactive Online Learning*, 9(1):64–77, 2010.

[30] Kurt VanLehn. The behavior of tutoring systems. *International Journal of Artificial Intelligence in Education*, 16(3):227–265, 2006.

[31] John Self. The defining characteristics of intelligent tutoring systems research: ITSs care, precisely. *International Journal of Artificial Intelligence in Education*, 10:350–364, 1998.

[32] Min Chi, Kurt VanLehn, Diane Litman, and Pamela Jordan. Empirically evaluating the application of reinforcement learning to the induction of effective and adaptive pedagogical strategies. *User Modeling and User-Adapted Interaction*, 21(1-2):137–180, 2011.

[33] Michael Mayo and Antonija Mitrovic. Optimising ITS behaviour with bayesian networks and decision theory. *International Journal of Artificial Intelligence in Education*, 12:124–153, 2001.

[34] Noboru Matsuda, William W Cohen, Jonathan Sewall, Gustavo Lacerda, and Kenneth R Koedinger. Predicting students' performance with simstudent: Learning cognitive skills from observation. *Frontiers in Artificial Intelligence and Applications*, 158:467, 2007.

[35] Kenneth R Koedinger and Albert Corbett. *Cognitive Tutors: Technology Bringing Learning Science to the Classroom*, chapter 5. Cambridge University Press, 2006.

[36] Kurt VanLehn. The relative effectiveness of human tutoring, intelligent tutoring systems, and other tutoring systems. *Educational Psychologist*, 46(4):197–221, 2011.

[37] Saiying Steenbergen-Hu and Harris Cooper. A meta-analysis of the effectiveness of intelligent tutoring systems on college students' academic learning. *Journal of Educational Psychology*, 105(4):970–987, 2013.

[38] James A. Kulik and J. D. Fletcher. Effectiveness of intelligent tutoring systems. *Review of Educational Research*, 86(1):42–78, 2016. doi: 10.3102/0034654315581420.

[39] Albert Corbett. Cognitive mastery learning in the ACT programming tutor. Technical report, Human-Computer Interaction Institute, Carnegie Mellon University, 2000.

[40] Katy Jordan. Massive open online course completion rates revisited: Assessment, length and attrition. *The International Review of Research in Open and Distributed Learning*, 16(3), 2015.

[41] Matija Lokar and Matija Pretnar. A low overhead automated service for teaching programming. In *Proceedings of the 15th Koli Calling Conference on Computing Education Research*, pages 132–136. ACM, 2015.

[42] Alan Lesgold. Context-specific requirements for models of expertise. In *Cognitive Science in Medicine*, pages 373–400. MIT Press, 1989.

[43] John R Anderson. ACT: A simple theory of complex cognition. *American Psychologist*, 51(4):355, 1996.

[44] John R Anderson, Albert T Corbett, Kenneth R Koedinger, and Ray Pelletier. Cognitive tutors: Lessons learned. *The journal of the learning sciences*, 4 (2):167–207, 1995.

[45] Vincent Aleven, Bruce M McLaren, Jonathan Sewall, and Kenneth R Koedinger. The cognitive tutor authoring tools (ctat): preliminary evaluation of efficiency gains. In *International Conference on Intelligent Tutoring Systems*, pages 61–70. Springer, 2006.

[46] John Stamper, Michael Eagle, Tiffany Barnes, and Marvin Croy. Experimental evaluation of automatic hint generation for a logic tutor. *International Journal of Artificial Intelligence in Education*, 22(1-2):3–17, 2013.

[47] Albert T. Corbett and Akshat Bhatnagar. Student modeling in the ACT programming tutor: Adjusting a procedural learning model with declarative knowledge. In Anthony Jameson, Cécile Paris, and Carlo Tasso, editors, *User Modeling: Proceedings of the Sixth International Conference, UM97*, pages 243–254. Springer Wien New York, Vienna, New York, 1997.

[48] Alex Gerdes, Bastiaan Heeren, Johan Jeuring, and L Thomas van Binsbergen. Ask-elle: an adaptable programming tutor for haskell giving automated feedback. *International Journal of Artificial Intelligence in Education*, pages 1–36, 2016.

[49] Stellan Ohlsson. Constraint-based student modeling. In *Student modelling: the key to individualized knowledge-based instruction*, pages 167–189. Springer, 1994.

[50] Antonija Mitrovic, Kenneth R Koedinger, and Brent Martin. A comparative analysis of cognitive tutoring and constraint-based modeling. In *Proceedings of the 9th international conference on User modeling*, pages 313–322. Springer-Verlag, 2003.

[51] Antonija Mitrovic and Stellan Ohlsson. An intelligent SQL tutor on the web. *International Journal of Artificial Intelligence in Education*, 10:238–256, 1999.

[52] Jay Holland, Antonija Mitrovic, and Brent Martin. J-LATTE: a constraint-based tutor for Java. In *Proc. 17th Int'l Conf. Computers in Education (ICCE 2009)*, pages 142–146, 2009.

[53] Nguyen-Thinh Le and Wolfgang Menzel. Using weighted constraints to diagnose errors in logic programming – the case of an ill-defined domain. *International Journal of Artificial Intelligence in Education*, 19(4):381–400, 2009.

[54] Jaime Gálvez, Eduardo Guzmán, and Ricardo Conejo. A blended e-learning experience in a course of object oriented programming fundamentals. *Knowledge-Based Systems*, 22(4):279–286, 2009.

[55] Timothy S Gegg-Harrison. Exploiting program schemata in an automated program debugger. *Journal of Interactive Learning Research*, 5(2):255, 1994.

[56] W Lewis Johnson. Understanding and debugging novice programs. *Artificial Intelligence*, 42(1):51–97, 1990.

[57] Jun Hong. Guided programming and automated error analysis in an intelligent Prolog tutor. *International Journal of Human-Computer Studies*, 61(4): 505–534, 2004.

[58] JS Song, SH Hahn, KY Tak, and JH Kim. An intelligent tutoring system for introductory c language course. *Computers & Education*, 28(2):93–102, 1997.

[59] Gerhard Weber and Antje Mollenberg. ELM-PE: A knowledge-based programming environment for learning lisp. In *Proceedings of ED-MEDIA 94*, pages 557–562, 1994.

[60] Cristoph Peylo, Tobias Thelen, Claus Rollinger, and Helmar Gust. A web-based intelligent educational system for PROLOG. In *Proceedings of the International Workshop on Adaptive and Intelligent Web-Based Education Systems*, Montreal, QC, Canada, 2000.

[61] Raymund C Sison, Masayuki Numao, and Masamichi Shimura. Multistrategy discovery and detection of novice programmer errors. *Machine Learning*, 38 (1-2):157–180, 2000.

[62] Merlin Suarez and Raymund Sison. Automatic construction of a bug library for object-oriented novice java programmer errors. In *Proceedings of the 9th International Conference on Intelligent Tutoring Systems*, page 184. Springer Science & Business Media, 2008.

[63] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. Automated feedback generation for introductory programming assignments. *ACM SIGPLAN Notices*, 48(6):15–26, 2013.

[64] W Lewis Johnson and Elliot Soloway. PROUST: Knowledge-based program understanding. *IEEE Transactions on Software Engineering*, (3):267–275, 1985.

[65] Paul Brna, Alan Bundy, Tony Dodd, Marc Eisenstadt, Chee Kit Looi, Helen Pain, Dave Robertson, Barbara Smith, and Maarten van Someren. Prolog programming techniques. *Instructional science*, 20 (2-3):111–133, 1991.

[66] Michael Striewe and Michael Goedicke. Using run time traces in automated programming tutoring. In *Proceedings of the 16th annual joint conference on Innovation and technology in computer science education*, pages 303–307. ACM, 2011.

[67] Sumit Gulwani, Ivan Radiček, and Florian Zuleger. Feedback generation for performance problems in introductory programming assignments. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 41–51. ACM, 2014.

[68] Dohyeong Kim, Yonghwi Kwon, Peng Liu, I Luk Kim, David Mitchel Perry, Xiangyu Zhang, and Gustavo Rodriguez-Rivera. Apex: automatic programming assignment error explanation. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 311–327. ACM, 2016.

[69] Andy Nguyen, Christopher Piech, Jonathan Huang, and Leonidas Guibas. Codewebs: scalable homework search for massive open online programming courses. In *Proc. 23rd Int'l World Wide Web Conf. (WWW 14)*, pages 491–502, 2014.

[70] Elena L Glassman, Jeremy Scott, Rishabh Singh, Philip J Guo, and Robert C Miller. OverCode: Visualizing variation in student solutions to programming problems at scale. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 22(2):7, 2015.

[71] Philip J Guo. Codeopticon: Real-time, one-to-many human tutoring for computer programming. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*, pages 599–608. ACM, 2015.

[72] Chris Piech, Jonathan Huang, Andy Nguyen, Mike Phulsuksombati, Mehran Sahami, and Leonidas Guibas. Learning program embeddings to propagate feedback on student code. In *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)*, pages 1093–1102, 2015.

[73] Kelly Rivers and Kenneth R Koedinger. Data-driven hint generation in vast solution spaces: a self-improving Python programming tutor. *International Journal of Artificial Intelligence in Education*, pages 1–28, 2015. doi: 10.1007/s40593-015-0070-z.

[74] Thomas W. Price, Yihuan Dong, and Tiffany Barnes. Generating data-driven hints for open-ended programming. In *Proceedings of the 9th International Conference on Educational Data Mining, EDM 2016, Raleigh, North Carolina, USA, June 29 - July 2, 2016*, pages 191–198, 2016.

[75] Kelly Rivers and Kenneth R Koedinger. Automatic generation of programming feedback: A data-driven approach. In *Proc. Workshops 16th Int'l Conf. Artificial Intelligence in Education (AIED 13)*, pages 50–59, 2013.

[76] Kurtis Zimmerman and Chandan R Rupakheti. An automated framework for recommending program elements to novices. In *30th IEEE/ACM International Conference on Automated Software Engineering*, pages 283–288. IEEE, 2015.

[77] Marvin J Croy. Graphic interface design and deductive proof construction. *Journal of Computers in Mathematics and Science Teaching*, 18(4):371–386, 1999.

[78] John Stamper, Tiffany Barnes, Lorrie Lehmann, and Marvin Croy. The hint factory: Automatic generation of contextualized help for existing computer aided instruction. In *Proceedings of the 9th International Conference on Intelligent Tutoring Systems Young Researchers Track*, pages 71–78, 2008.

[79] Wei Jin, Tiffany Barnes, John Stamper, Michael John Eagle, Matthew W Johnson, and Lorrie Lehmann. Program representation for automatic hint generation for a data-driven novice programming tutor. In *Proc. 11th Int'l Conf. Intelligent Tutoring Systems (ITS 12)*, pages 304–309, 2012.

[80] Barry W. Peddycord III, Andrew Hicks, and Tiffany Barnes. Generating hints for programming problems using intermediate output. In *Proceedings of the 7th International Conference on Educational Data Mining, EDM 2014, London, UK, July 4-7, 2014*, pages 92–98, 2014.

[81] Ted J Biggerstaff, Bharat G Mitbander, and Dallas Webster. The concept assignment problem in program understanding. In *Proceedings of the 15th international conference on Software Engineering*, pages 482–498. IEEE Computer Society Press, 1993.

[82] Nikolaus Augsten, Michael Böhlen, and Johann Gamper. Approximate matching of hierarchical data using pq-grams. In *Proceedings of the 31st international conference on Very large data bases*, pages 301–312. VLDB Endowment, 2005.

[83] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[84] Chanchal K Roy, James R Cordy, and Rainer Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of computer programming*, 74(7):470–495, 2009.

[85] Thomas J McCabe. A complexity measure. *IEEE Transactions on software Engineering*, (4):308–320, 1976.

[86] James Walden, Jeff Stuckman, and Riccardo Scandariato. Predicting vulnerable components: Software metrics vs text mining. In *Software Reliability Engineering (ISSRE), 2014 IEEE 25th International Symposium on*, pages 23–33. IEEE, 2014.

[87] Anh Viet Phan, Phuong Ngoc Chau, Minh Le Nguyen, and Lam Thu Bui. Automatically classifying source code using tree-based approaches. *Data & Knowledge Engineering*, 2017.

[88] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. Deep learning code fragments for code clone detection. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 87–98. ACM, 2016.

[89] Kelly Rivers and Kenneth R Koedinger. A canonicalizing model for building programming tutors. In *Proc. 11th Int'l Conf. Intelligent Tutoring Systems (ITS 12)*, pages 591–593, 2012.

[90] Songwen Xu and Yam San Chee. Transformation-based diagnosis of student programs for programming tutoring systems. *IEEE Transactions on Software Engineering*, 29(4):360–384, 2003.

[91] Timotej Lazar and Ivan Bratko. Data-driven program synthesis for hint generation in programming tutors. In *Proc. 12th Int'l Conf. Intelligent Tutoring Systems (ITS 14)*, pages 306–311, 2014.

[92] Ryan Baker, Kenneth R Koedinger, Albert T Corbett, Angela Z Wagner, Shelley Evenson, Ido Roll, Meghan Naim, Jay Raspat, and Joseph E Beck. Adapting to when students game an intelligent tutoring system. In *Proc. 8th Int'l Conf. Intelligent Tutoring Systems (ITS 06)*, pages 392–401, 2006.

[93] John W Ratcliff and David E Metzener. Pattern matching: The gestalt approach. *Dr. Dobb's Journal*, 13(7):46, 1988.

[94] Ilya Sutskever, James Martens, and Geoffrey E Hinton. Generating text with recurrent neural networks. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pages 1017–1024, 2011.

[95] Eddie Antonio Santos, Joshua Charles Campbell, Dhvani Patel, Abram Hindle, José Nelson Amaral, Eddie Antonio Santos, Joshua Charles Campbell, Dhvani Patel, Abram Hindle, José Nelson Amaral, et al. Syntax and sensibility: Using language models to detect and correct syntax errors. In *25th IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER 2018)*, volume 29, pages 1–5, 2016.

[96] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. Deepfix: Fixing common c language errors by deep learning, 2017.

[97] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014.

[98] Pär Emanuelsson and Ulf Nilsson. A comparative study of industrial static analysis tools. *Electronic notes in theoretical computer science*, 217:5–21, 2008.

[99] Vijay D'silva, Daniel Kroening, and Georg Weissenbacher. A survey of automated techniques for formal software verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(7):1165–1178, 2008.

[100] Anjana Gosain and Ganga Sharma. Static analysis: A survey of techniques and tools. In *Intelligent Computing and Applications*, pages 581–591. Springer, 2015.

[101] Kent Beck and Martin Fowler. *Bad smells in code*, chapter 3, pages 75–88. Addison-Wesley Professional, 1999.

[102] Francesca Arcelli Fontana, Pietro Braione, and Marco Zanoni. Automatic detection of bad smells in code: An experimental assessment. *Journal of Object Technology*, 11(2), 2012.

[103] Roger Levy and Galen Andrew. Tregex and tsurgeon: tools for querying and manipulating tree data structures. In *5th International Conference on Language Resources and Evaluation (LREC 2006)*, 2006.

[104] Ilya Bagrak and Olin Shivers. trx: Regular-tree expressions, now in scheme. In *Proceedings of the Fifth Workshop on Scheme and Functional Programming*, pages 21–32, 2004.

[105] Petra Kralj Novak, Nada Lavrač, and Geoffrey I Webb. Supervised descriptive rule discovery: A unifying survey of contrast set, emerging pattern and subgroup mining. *Journal of Machine Learning Research*, 10(Feb):377–403, 2009.

[106] Peter Clark and Robin Boswell. Rule induction with CN2: Some recent improvements. In *Proceedings of the Fifth European Conference on Machine Learning*, pages 151–163, 1991.

[107] Janez Demšar, Tomaž Curk, Aleš Erjavec, Črt Gorup, Tomaž Hočevar, Mitar Milutinovič, Martin Možina, Matija Polajnar, Marko Toplak, Anže Starič, Miha Štajdohar, Lan Umek, Lan Žagar, Jure Žbontar, Marinka Žitnik, and Blaž Zupan. Orange: Data mining toolbox in python. *Journal of Machine Learning Research*, 14:2349–2353, 2013.

[108] Martin Možina, Jure Žabkar, and Ivan Bratko. Argument based machine learning. *Artificial Intelligence*, 171(10-15):922–937, 2007.

[109] Karl M Kapp. *The gamification of learning and instruction: game-based methods and strategies for training and education*. John Wiley & Sons, 2012.

[110] Marie-Hélène Nienaltowski, Michela Pedroni, and Bertrand Meyer. Compiler error messages: What can help novices? In *ACM SIGCSE Bulletin*, volume 40, pages 168–172. ACM, 2008.

[111] Albert T Corbett and John R Anderson. Locus of feedback control in computer-based tutoring: Impact on learning rate, achievement and attitudes. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 245–252. ACM, 2001.