Programming to Learn: Logic and Computation from a Programming Perspective

Matthew Farrugia-Roberts
The University of Melbourne
Melbourne, Victoria, Australia
matt.farrugia@unimelb.edu.au

Bryn Jeffries*
Grok Academy
Sydney, New South Wales, Australia
bryn.jeffries@grokacademy.org

Harald Søndergaard
The University of Melbourne
Melbourne, Victoria, Australia
harald@unimelb.edu.au

ABSTRACT

Programming problems are commonly used as a learning and assessment activity for learning to program. We believe that programming problems can be effective for broader learning goals. In our large-enrolment course, we have designed special programming problems relevant to logic, discrete mathematics, and the theory of computation, and we have used them for formative and summative assessment. In this report, we reflect on our experience. We aim to leverage our students' programming backgrounds by offering a code-based formalism for our mathematical syllabus. We find we can translate many traditional questions into programming problems of a special kind—calling for 'programs' as simple as a single expression, such as a formula or automaton represented in code. A web-based platform enables self-paced learning with rapid contextual corrective feedback, and helps us scale summative assessment to the size of our cohort. We identify several barriers arising with our approach and discuss how we have attempted to negate them. We highlight the potential of programming problems as a digital learning activity even beyond a logic and computation course.

CCS CONCEPTS

• **Social and professional topics** → **Computer science education**; • **Applied computing** → **Computer-assisted instruction**; **Interactive learning environments**; • **Software and its engineering** → *Functional languages.*

KEYWORDS

Automaton theory, logic, formal languages, discrete mathematics, computability, programming, code-in-the-browser, question types, interactive learning, auto-grading

ACM Reference Format:
Matthew Farrugia-Roberts, Bryn Jeffries, and Harald Søndergaard. 2022. Programming to Learn: Logic and Computation from a Programming Perspective. In *Proceedings of the 27th ACM Conference on Innovation and Technology in Computer Science Education Vol 1 (ITiCSE 2022), July 8–13, 2022, Dublin, Ireland.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3502718.3524814

*Also with The University of Sydney, School of Computer Science.

ITiCSE 2022, July 8–13, 2022, Dublin, Ireland

ACM ISBN 978-1-4503-9201-3/22/07…$15.00
https://doi.org/10.1145/3502718.3524814

1 INTRODUCTION

The recent pandemic caused a widespread shift in learning and assessment to digital modes of interaction. While this was challenging for many, we benefited from a head start in our large-enrolment course on introductory logic, discrete structures, automata, and computability. Over the years, we have increasingly adopted web-based *programming problems* as a medium for formative and summative assessment. In 2020 and 2021, this included final exams.

Programming competency is not an intended learning outcome of our course—we study mathematical topics in logic and computation. Still we have found it useful to re-purpose programming as aligned learning and assessment activity. In particular, we have developed problems that focus on a program's *subject* (e.g., a digital representation of a logical formula or an automaton) rather than the programming constructs involved. Often, these problems require 'programs' comprising a small static expression, rather than a function or algorithm proper. Such problems are more akin to traditional pen-and-paper exercises than usual programming problems.

One motivation has been to leverage students' existing programming skills, to give students a 'bridge' to notoriously challenging theoretical topics. Having *learned to program* in foundational courses, they are now *programming to learn* formal concepts in logic, discrete structures, automata, and computability. Second, we aim to harness the pedagogical and logistic affordances of digital learning environments. For students, these include asynchronous and self-paced learning with rapid feedback [2]. For us, the prospect of scalability of formative and summative assessment through automation has been of particular value in our large-enrolment course.

In this paper, we share our experience and reflect upon the exploration of programming problems as a learning and assessment activity in a non-programming course. We briefly review related instructional approaches (Section 2), before detailing our particular educational context (Section 3) and relevant details of our programming problem learning activities (Section 4). We reflect on our overall experience in Section 5, based on informal student feedback and our own subjective evaluation of the initiative.

We believe that logic and computation are important topics in a computer science curriculum, and students can benefit from tools that support mastering them. Programming problems could prove an effective learning activity for these topics. In Section 6, we comment on the potential of programming to learn as a general instructional approach for students with a background in computer science studying these and other topics.

2 RELATED WORK

We briefly review instructional approaches for programming students studying mathematical topics in logic and computation.

Pen-and-paper approach. The traditional approach for each of these topics, as embodied by standard textbooks [e.g., 18, 22, 36], emphasises pen-and-paper exercises for learning and assessment. Exercises vary in complexity from simple procedural and construction tasks to proving complex theorems [cf. 14]. We retain elements of this approach, but recast most traditional exercises, including certain constructive proof exercises, as programming problems.

Implementation approach. It is common to augment traditional exercises with implementation of key algorithms pertaining to logic and computation [e.g., 47]. This is not primarily intended to improve programming skills *per se*, but rather to deepen understanding of the underlying formal concepts. Our approach naturally includes this kind of programming problem, alongside less conventional problems (discussed below).

Programmatic representation approach. Several textbooks use a programmatic representation of the mathematical objects of their syllabus [9, 19, 28]. An executable version of the expressions and diagrams in traditional textbooks, this representation provides a complementary formalism, perhaps more readily assimilated by students with a programming background. Such textbooks originally inspired our initiative. We have taken the representations a step further by lifting them off the page and into a web-based programming environment, as a digital format for expressing answers to traditional-style exercises.

Programming to learn. We use programmatic representations in both implementation tasks and digitised traditional exercises. We know of a few similar initiatives. In use for over two decades [31, 42], the web-based AUTOTOOL uses Haskell programming problems for learning and assessment in logic [43], computation [31], and many other topics [43–46]. Stoughton [38] used Standard ML for the learning of formal language theory. FSM [25, 26] provides programmatic automata exercises using a Lisp variant. Our reflections may be relevant to the use of these similar tools.

Specialised environment approach. It is common to augment a traditional approach with specialised digital exercise interface. The interface may be textual, such as a domain-specific language for logic formula manipulation [20, 21] or logical modelling [37]. Or it may be graphical, such as in AUTOMATA TUTOR [5, 7] and OPENFLAP [24] for automata. Typically, instructors can configure instances of certain *exercise types* as automatically-graded learning activities for students. Some tools offer automatic generation of instances from a template, for individuating assessment or providing additional drill for students [5]. Creating fundamentally new exercise types requires extending the environment itself, including providing a suitable interface [5, 24]. In contrast, our digital exercises use a *single, uniform environment* supporting many exercise types, with a programming language as a *universal textual interface*.

Other approaches. Software for the teaching of logic has a long history [12]. Extension of the logical or mathematical syllabus to include programming concepts as a case study or application of more abstract principles has also been common [16, 40]. Many exploratory visual learning environments have also been developed for topics in the theory of computation [3]. JFLAP [32, 33], predecessor to OPENFLAP, is a leading example.

Stepping back from our syllabus, the formative elements of our initiative constitute an *automated feedback system* [8]. Such systems go back seemingly as long as instructors have had computers [e.g., 39]. The best modern systems match human tutors in efficacy [41]. Our rudimentary web-based system is best classified as offering on-request corrective feedback based on expert knowledge [8]. However, our programming approach is compatible in principle with more sophisticated features.

3 LEARNING CONTEXT

Our course is an introduction to logic and computation. Intended learning outcomes include the ability to: (1) explain propositional and predicate logic, mechanised reasoning, sets, functions, relations; (2) use these to reason about computational problems; (3) create, use and analyse finite automata, regular expressions, pushdown automata, context-free grammars, and Turing machines.

We have large enrolments (over 500 students in the 2021 offering). Most students take the course for an undergraduate major or coursework graduate program in computer science or software engineering. Students know one or more programming languages as a prerequisite, and most have previously used the same web-based programming platform. Haskell—the language we adopt for programming problems—is *not* a prerequisite. Some students take an elective in functional programming, usually concurrently. Either way, semester begins with a self-paced introduction to basic Haskell as necessary for use in assessment. Most of our students have some university-level mathematics experience, but the course is considered mathematically demanding by many of our students.

<table>
<tr><th>Week</th><th>Topic in lectures and tutorials</th><th colspan="2">Programming problems for learning and assessment</th></tr>
<tr><td>1</td><td>Introduction</td><td rowspan="2">Introduction to Haskell basics (self-paced, self-contained)</td><td rowspan="12">Worksheets: 6% (four fortnightly formative explorations of algorithms for propositional logic, regular languages, and formal grammars)</td></tr>
<tr><td>2
3
4
5</td><td rowspan="4">Logic (propositional and predicate logic, resolution algorithms)</td></tr>
<tr><td>6
7</td><td>Discrete maths (sets, functions, relations)</td><td rowspan="2">Assignmt. 1: 12% (mathematical and algorithmic logic challenges)</td></tr>
<tr><td>8
9
10
11</td><td rowspan="4">Computation (finite automata, reg. expressions, context-free grammars, Turing machines, etc.)</td><td rowspan="3">Assignmt. 2: 12% (mathematical and algorithmic challenges in discrete maths / comp. theory)</td></tr>
<tr><td>12
…</td></tr>
<tr><td>Exam period</td><td>Exam: 70% (3hrs)</td></tr>
</table>

Table 1: Example semester calendar.

Our learning activities have evolved over several years. We introduced Haskell programming problems as supporting exercises in 2016. We then expanded the role of programming problems in assessment and moved to the web-based platform. In 2020 and 2021, we conducted practically all assessment, including the final exams, via programming problems, as shown in Table 1. In total, students face about 30 exercises in assignments and 20–30 questions in a final exam.

Beyond assessment, our contact with students has comprised two one-hour lectures and a one-hour tutorial each week.

4 PROGRAMMING PROBLEMS

Our students complete programming problems on the online learning platform Grok Academy (Grok) [13], a tool originally designed for *learning to program*. Grok supports Haskell and has an appealing and engaging design, known to most of our students from previous coursework. In principle, any platform offering sufficient customisability of programming problems could be re-purposed for *programming to learn*.

A programming problem essentially has three components (the interface from the student perspective is shown in Figure 1):

(a) A *description*, explaining the task to the student. This can contain formatted text with embedded equations, images, videos, and interactive code listings.
(b) A *text editor*, where the student writes their answer program. We configure initial scaffolding and importable libraries.
(c) A suite of *tests*, programs to run against the student's answer on request, including customisable feedback messages. We configure the tests to give the student contextual feedback.

We set up dozens of programming problems (one per exercise, assignment question, or exam question) in a series of *modules* (one per assessment task). Within each module, problems are interleaved with *slides* with supporting and motivating information, creating a kind of interactive e-textbook.

Our programming problems can mostly be classified as *instance* or *implementation* problems. Figure 2 provides an example of each.

Instance problems directly replace traditional procedural or construction exercises in our assignments and exams. Such exercises already asked students to instantiate a mathematical object (a propositional formula or model, a set or other discrete structure, a string, regular expression, grammar, or one of various automata models). The conditions for correctness often constitute a decidable property of the object, verifiable by a simple program. We can usually offer helpful feedback for incorrect answers in an automated way.

Thus, we designed and introduced the students to a programmatic representation of the various mathematical objects in our syllabus. We then created programming problems asking students to define an appropriate object (otherwise using a traditional-style problem description). The student solves the exercise as they otherwise would, before (or while) entering their answer as an expression.

We configure tests to compile the expression and apply some wellformedness checks (e.g., an automaton's initial state should be declared in its state set). During formative assessment, we add tests to convey the verification result and/or contextual feedback.

Implementation problems have students write a topical function (rather than a mere expression). Important algorithms abound in our course (normal form conversions, mechanised reasoning, simulation, minimisation, and determinisation of finite-state automata, etc.) and construction algorithms are central to many important mathematical proofs that we study (conversions between computation models, or closure properties of language families, etc.).

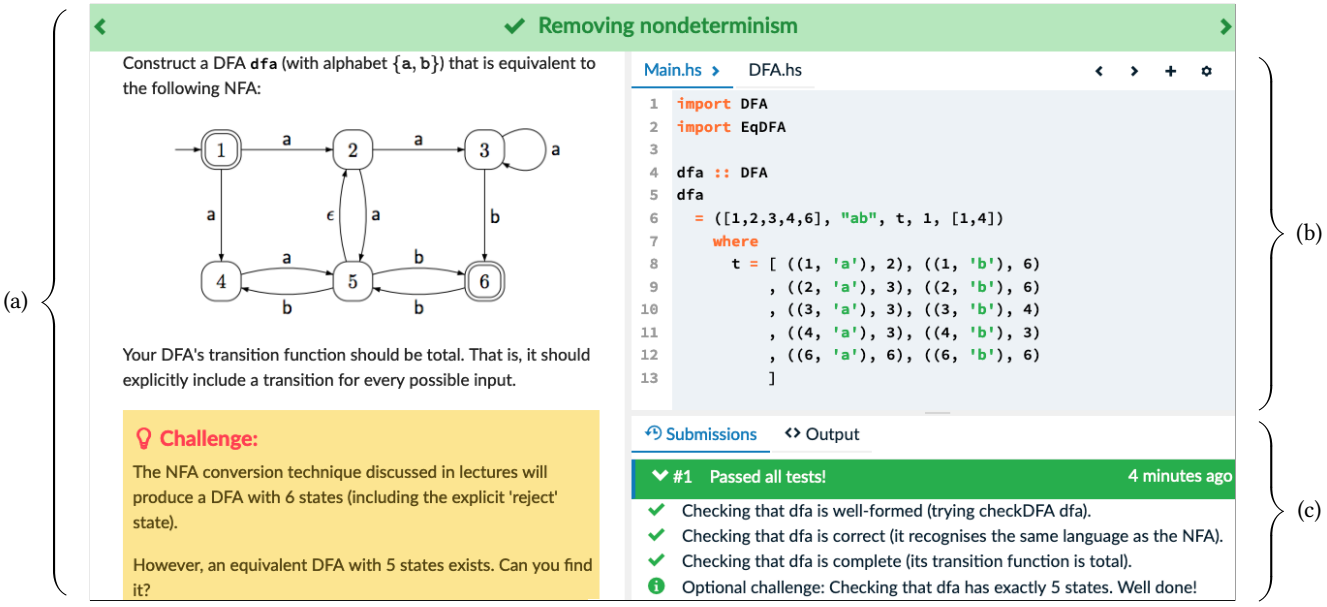

Figure 1: From the student perspective, a programming problem comprises (a) a description (shown: an embedded diagram); (b) a Haskell program editor; (c) feedback messages from a suite of tests. Each aspect (a, b, c) is configurable.

(a)

Problem description: A propositional formula is in *negation normal form* (NNF) if it only contains conjunctions, disjunctions, negations, and propositions, with all negations applied directly to propositions.

NNF is not a canonical form. Give two syntactically different but logically equivalent formulas in NNF.

Haskell representation of propositional formulas using a recursive data type (some connectives omitted for brevity):

```
data Formula
  = PROP Char
  | NOT  Formula
  | AND  Formula Formula
  | OR   Formula Formula
```

Haskell expression answer:

```
f1 = PROP 'A'
f2 = PROP 'A' `OR` PROP 'A'
```

Traditional non-program answer:
The negation normal form formulas 'A' and '$A \vee A$' are logically equivalent but syntactically distinct.

(b)

Algorithmic problem description:
Give an algorithm that takes a deterministic finite automaton (DFA) D as input and converts it into a DFA for the complement language $\Sigma^* \setminus L(D)$.

Proof-based problem description:
Show that the class of regular languages is closed under the complement operation, by constructing a DFA for the complement language $\Sigma^* \setminus L(D)$ of an arbitrary DFA D.

(Dual description shows two possible framings.)

Haskell representation for deterministic finite-state automata (`State` and `Symbol` definitions omitted for brevity):

```
type DFA
  = ( [State]  -- states
    , [Symbol] -- alphabet
    , [((State, Symbol), State)]
               -- transitions
    , State    -- start state
    , [State]  -- accept states
    )
```

Haskell function answer:

```
complementDFA :: DFA -> DFA
complementDFA (q, s, d, q0, f)
    = (q, s, d, q0, q \\ f)
```

Traditional non-program answer:
Given a DFA $D = (Q, \Sigma, \delta, q_0, F)$, form a new DFA

$$D' = (Q, \Sigma, \delta, q_0, Q \setminus F)\,.$$

We have $L(D') = \Sigma^* \setminus L(D)$ as required.

Figure 2: Problem (a), an *instance problem*, asks students to synthesise two propositional formulas as Haskell expressions. Problem (b), an *implementation problem*, asks students for a Haskell function representing a basic construction algorithm.

These algorithms and proofs can be represented as functions involving our programmatic representations of objects. We thus ask students to implement such functions as a way to study the algorithms. We can scale the complexity of the task by providing scaffolding (partially-implemented functions missing key components, or 'helper functions' abstracting away irrelevant ones).

5 REFLECTION

Through our experiment in learning with programming problems we have noted several logistic and pedagogical benefits. We have also identified and addressed some key challenges.

5.1 Problem development

As instructors, we have found programming problems an ergonomic and flexible medium through which to design a high-quality digital learning experience for students.

A rich digital exercise format. We have found it straightforward to convert most traditional exercises into programming problems. We have even had some success converting 'proof' exercises into implementation problems, though we still have students engage with some proofs through traditional activities.

We have come to see instance problems in particular as a compelling alternative to the standard digital exercise types provided by learning management systems (multiple-choice questions, 'fill in the blanks', matching questions, etc.). It is challenging to exercise certain cognitive skills with these formats [cf. 34]. As a salient example, consider aligning standard digital exercises with the intended learning outcome 'creating finite-state automata'. Instance problems, on the other hand, with unrestricted degrees of freedom for student answers, can readily assess a broad range of skills, including 'creating'. At the same time, they offer similar advantages to other digital formats.

Rapid development of new exercise types. A programming problem is a traditional-style problem description plus the minimal extra requirements of an answer representation and a program for automatic feedback and verification. Since many problems share the same answer representation and even feedback/verification primitives, we find it feasible to regularly develop several new kinds of problem for each assignment or exam. In contrast, adding a new exercise type to a graphical tool requires the same components *plus* an extension to the graphical interface [5, 24]. Since instructors always want more exercises for student practice and for assessments, these lightweight requirements can be considered a virtue.

A unified interface. While specialised tools abound for learning logic, discrete mathematics, and the theory of computation, fewer tools cater to the full breadth of our syllabus. Using Haskell as a *lingua franca*, we offer exercises on a breadth of topics in a unified learning interface. This range is crucial for our exams, with questions spanning the curriculum.

A programming platform is also a natural setting for providing implementation problems *alongside* instance problems in one platform. In courses using graphical tools for instance-style problems (albeit with graphical 'editors'), students must use a separate system if they are to *program with* the objects they construct.

5.2 Pedagogical support

Though a more thorough evaluation is needed, in principle the programming approach offers students several direct benefits.

Rapid (contextual) formative feedback. We use Grok's testing system to deliver rapid feedback. Wellformedness feedback helps students internalise the structure of mathematical objects. Through appropriate test programs we can provide other contextual feedback based on student answers, such as showing a misclassified string to help a student improve an automaton [cf. 1, 6, 31]. We expect rapid verification feedback builds student confidence.

Leverage from student backgrounds. Our students have always considered our course mathematically demanding. At the same time, they typically have a strong programming background. We expect that a programmatic representation offers students untrained in mathematical notation a means to assimilate the formal concepts in our syllabus. Moreover, implementation problems allow students to (literally) codify their understanding of methods and algorithms.

Making it hands-on. Programming offers a hands-on experience with mathematical objects [33], as students follow a process of "constructing, debugging and testing" in line with the rest of a computing curriculum. Implementation problems offer students a hands-on experience in another sense: they get to analyse and transform these mathematical objects with code. We believe such activities help our students develop an 'operational' understanding of the concepts as a prerequisite for a 'structural' understanding [35].

Student empowerment. Programming languages are powerful tools for shaping the world. With a programming approach, that power is always within reach, and students can use it to take more ownership of their learning experience. We see many students write custom test scripts to verify hand-derived answers. Some submit code that *computes instances systematically*, to check or replace a hard-coded, hand-derived answer. This could be seen as subversive, but we encourage it, as it demonstrates engagement and a high level of understanding. Also, in our fortnightly worksheets, students complete the implementation of software tools useful later in the course (e.g. inference and automata algorithms, expression parsers, and automaton visualisers). Having had a hand in building these tools, students can feel ownership when they are later used.

We do not suggest that programming problems are alone sufficient for learning. Even with most of our assessment conducted with programming problems, student interaction with peers and tutors is of critical importance. For all the rich contextual feedback provided by our most intricate feedback algorithms, we attain only 'corrective' feedback, just one of many forms of feedback important in a student's development [15]. For all the concepts we can codify through the expressive power of programs, code does not by itself motivate the study of our syllabus.

5.3 Barriers and their mitigation

The use of a programming language as an expressive medium also raises barriers to student engagement. We have identified four distinct barriers, and taken steps to mitigate them.

Language barrier. The most fundamental barrier our students face is learning enough Haskell to participate in assessment. Most of our students enter with no Haskell experience. So we provide a self-contained introduction to the language in Grok itself and we minimise feature use—students use a consistent and manageable subset of the language. To reduce cognitive load, we provide basic scaffolding, like type signatures, for all programming problems.

Syntax barrier. Low-level (syntax) details can present a barrier to expressing objects. For example, students often face compilation errors due to poor syntax or typos. Due to its advanced type system, Haskell's errors sometimes reflect important conceptual mistakes. Unfortunately, error messages can be cryptic to novices even if they would be meaningful to an expert [cf. 4, 23].

We address this issue with live exam support for errors. Nevertheless, many students still submit answers with errors, and several report losing exam time hunting errors. In response, we reviewed all non-compiling answers, to fix trivial errors or grade them manually. Going forward, we could advertise exam support more effectively, extend this support to the semester, and include error handling in the Haskell introduction.

Digitisation barrier. For some questions, students work manually before typing their answer as code we can process. Especially during exams, this two-stage process is a fundamental limitation compared to pen-and-paper. In particular, when a student fails to digitise an answer, we cannot assess any rough work that may have demonstrated merit.

We advise students to avoid the issue through careful time management, and we offer a timed practice exam to aid preparation. A handful of students (1%) still reported this as an issue after the 2021 final exam, so the issue cannot be neglected.

Encoding barrier. Ideally, students can express themselves as effortlessly with code as they might with natural language and mathematical notation. But in practice they will be hampered by tedious syntactic incantations, especially if a representation is poorly designed. We use Haskell partly because of its clean, maths-like syntax, making it a popular choice for our syllabus [9, 28, 31]. Within Haskell, we try carefully to design natural representations. We offer students *parsers* for notations such as regular expressions and logic formulas, as a way to limit syntactic overhead. But these parsers also need careful design to be effective. In particular, they should offer good feedback messages, lest students suffer even greater syntax barriers (e.g., hunting for a single unbalanced parenthesis). For some objects, a textual encoding is quite unnatural. Early student feedback expressed difficulty constructing automata without the popular graphical 'state diagram' [cf. 33]. We responded by integrating a visualiser (using Graphviz [11] to draw automata) into the programming environment.

Due to our use of an unfamiliar programming language, success becomes a question of whether the pedagogical benefits arising from a programming perspective are sufficient to offset the overhead of learning the language. So are they? It is credible, at least: In similar examples of instrumentally teaching a new language, Rahn and Waldmann report students overcoming Haskell barriers to use autotool [31], and several instructors successfully taught their students to use a proof assistant to meet teaching goals in logic and proof [17, 27, 29]. The possibility requires more rigorous study.

But we can also ask, need we take on such overhead? By using a language *already* familiar to our students, we might sacrifice syntactic elegance, but students' ability to see through low-level details might make up for it, for greater overall pedagogical benefits.

5.4 Programming for summative assessment

Our experience is a case study in scaling summative assessment using a web-based interface and automatic grading rubrics.

Immediate feedback. In exams, we provide some automatic feedback to students—compiler feedback and simple wellformedness checks (not complete verification). We primarily want to assess content, not form. By prompting students to fix trivial errors in their work, we can better assess answers at the conceptual level. As intended learning outcomes operate at this level, wellformedness feedback enhances assessment validity, compared to pen-and-paper exams, where students often submit answers with small typos or omissions clouding merit. 'Silly' mistakes, like forgetting to indicate an automaton's start state, are flagged at the Haskell level.

Automation pressure. Tight grading timelines provide incentive to automate assessment. But we must also counter this pressure by diligently evaluating automation decisions, and resorting to manual grading when we cannot otherwise match the skills of a human grader, especially when it comes to assigning partial credit. Our grading pipeline involves use of an automatic grading scheme if feasible, otherwise pre-processing student answers before manual grading. Pre-processing includes visualisation to help graders, and a process of 'clustering' answers so that only a limited number of representative cases need be graded manually. For example, if asking for a regular expression for some language, we can automatically verify correctness. However, it is unclear which 'formula' defines appropriate partial grades for incorrect answers. We resort to automatically detecting correct answers and then clustering and manually grading equivalence classes of incorrect expressions.

Timeline pressure could also steer our choice of assessment items away from those that resist automation or are ill-suited as programming problems. We must constantly refer to our intended learning outcomes to ensure assessment tasks remain aligned.

Scaling assessment. We have seen an overall decrease in time spent grading assignments and exams since adopting programming problems and partially automatic grading. Since tutors categorise their work when submitting pay claims, 'grading' being one category, we can quantify the change in workload quite accurately. Measured per student, exam preparation time has more than doubled, totalling roughly 10 minutes per student, but grading time has dropped from 19 to roughly 6 minutes per student. Notably, automating grading took significant *manual* work—to create an ad-hoc automatic grading scheme or pre-processing pipeline for each question. Automation is a manual process! The investment would not have paid off for a small class.

Through systematic grading schemes and the pre-processing technique of clustering equivalent answers to be graded at once, we increase grading consistency in that equivalent answers from different students are guaranteed to receive equal grades.

We stress that the automation of summative assessment should be undertaken with care. Digital problems can be manually graded, and sometimes should be. It would be an 'abuse of automation' [30] to insist otherwise. We can appeal to automation as a supplementary aide to our manual efforts [10], consistent with our duty to assess students accurately and fairly.

6 LOOKING AHEAD

We have begun to explore the possibilities afforded by programming problems as a medium for non-programming learning goals. There is much room for future work to build upon our current approach. While we have compared our system against existing tools, we see these tools as inspiration rather than competition. Advanced features such as using formal methods to generate richer feedback [14] and verify instances and implementations [5], or automatically generating problem instances from a template [5, 31], are compatible with a *programming to learn* approach.

We also envisage a synthesis between graphical and programmatic representations in a single e-textbook. Our students could certainly benefit from a more natural visual interface for constructing objects. Perhaps students using graphical tools could also benefit from exploring a programmatic representation of the objects they are constructing, and the opportunity to *program with* these objects.

We also believe our approach has potential beyond our syllabus. Our programming problems already span logic, discrete maths, and theory of computation. Waldmann's AUTOTOOL [42] addresses further topics including constraint problem solving [43], programming languages [44], data structures [45], and functional programming theory [46]. Programming languages provide a digital, executable formalism, able to encode any structured object, model, or process. Such things abound in the broader computer science curriculum, and in the neighbouring disciplines of mathematics, engineering, and sciences. Thus programming problems could offer a bridge for computer science students studying a wide range of topics.

So far, evaluation of our approach has been based on our subjective assessment and incidental student feedback. An imperative for future work on this approach is a more rigorous investigation of the student experience, both as it is perceived by students, and also as far as we can measure it, for example through empirical study of reactions to the barriers discussed in Section 5.3.

7 CONCLUSION

A student who has learnt to program has gained a powerful and flexible means of expression. We try to harness this power by communicating with students about complex formal objects and algorithms in the language of programs. This constitutes an instructional approach of *programming to learn*, exploited in our course on logic, discrete maths, and the theory of computation. The use of programming tasks as learning exercises for a non-programming course has pedagogic and logistical benefits over traditional pen-and-paper activities and other digital learning activities, and Haskell and Grok have provided a uniform platform for formative and summative assessment across our broad syllabus. We have also identified and begun to address certain limitations of learning and automated assessment through programming problems. We believe the *program to learn* approach is flexible enough to be used in other courses where students with a programming background study unfamiliar objects and algorithms. The approach would be worth a careful evaluation, particularly from the student perspective.

ACKNOWLEDGMENTS

We thank our teaching colleagues Anna Kalenkova, Bach Le, Billy Price, Rohan Hitchcock, and rest of the COMP30026 teaching team.

REFERENCES

[1] R. Alur, L. D'Antoni, S. Gulwani, D. Kini, and M. Viswanathan. Automated grading of DFA constructions. In *Proceedings of the 23rd International Joint Conference on Artificial Intelligence*, pages 1976–1982, 2013.

[2] D. E. Bostow, K. M. Kritch, and B. F. Tompkins. Computers and pedagogy: Replacing telling with interactive computer-programmed instruction. *Behavior Research Methods, Instruments, & Computers*, 27(2):297–300, 1995.

[3] P. Chakraborty, P. C. Saxena, and C. P. Katti. Fifty years of automata simulation: A review. *ACM Inroads*, 2(4):59–70, Dec. 2011.

[4] C. Clack and C. Myers. The dys-functional student. In P. H. Hartel and R. Plasmeijer, editors, *Functional Programming Languages in Education*, pages 289–309. Springer, 1995.

[5] L. D'Antoni, M. Helfrich, J. Kretinsky, E. Ramneantu, and M. Weininger. Automata Tutor v3. In S. K. Lahiri and C. Wang, editors, *Computer Aided Verification, Proceedings Part II*, volume 12225 of *Lecture Notes in Computer Science*, pages 3–14. Springer, 2020.

[6] L. D'Antoni, D. Kini, R. Alur, S. Gulwani, M. Viswanathan, and B. Hartmann. How can automatic feedback help students construct automata? *ACM Transactions on Computer-Human Interaction*, 22(2), Mar. 2015.

[7] L. D'Antoni, M. Weavery, A. Weinert, and R. Alur. Automata Tutor and what we learned from building an online teaching tool. *Bulletin of the EATCS*, 3(117), Oct. 2015.

[8] G. Deeva, D. Bogdanova, E. Serral, M. Snoeck, and J. De Weerdt. A review of automated feedback systems for learners: Classification framework, challenges and opportunities. *Computers & Education*, 162:1–43, 2021. Article 104094.

[9] K. Doets and J. van Eijck. *The Haskell Road to Logic, Maths and Programming*. King's College Publ., 2004.

[10] C. Douce, D. Livingstone, and J. Orwell. Automatic test-based assessment of programming: A review. *Journal on Educational Resources in Computing (JERIC)*, 5(3):Article 4, 2005.

[11] E. R. Gansner and S. C. North. An open graph visualization system and its applications to software engineering. *Software—Practice and Experience*, 30(11):1203–1233, 2000.

[12] D. Goldson, S. Reeves, and R. Bornat. A review of several programs for the teaching of logic. *The Computer Journal*, 36(4):373–386, 1993.

[13] Grok Academy webpage. https://grokacademy.org/. Last accessed Feb 2022.

[14] S. Gulwani. Example-based learning in computer-aided STEM education. *Communications of the ACM*, 57(8):70–80, Aug. 2014.

[15] J. Hattie and H. Timperley. The power of feedback. *Review of Educational Research*, 77(1):81–112, 2007.

[16] P. B. Henderson. Functional and declarative languages for learning discrete mathematics. In M. Hanus, editor, *Proceedings of the International Workshop on Functional and Declarative Programming in Education*, 2002. Available from https://www.informatik.uni-kiel.de/~mh/publications/reports/fdpe02/.

[17] M. Hendriks, C. Kaliszyk, F. Van Raamsdonk, and F. Wiedijk. Teaching logic using a state-of-the-art proof assistant. *Acta Didactica Napocensia*, 3(2):35–48, 2010.

[18] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Pearson, third edition, 2013.

[19] N. D. Jones. *Computability and Complexity from a Programming Perspective*. MIT Press, 1997.

[20] J. Lodder. *The Design and Use of Tools for Teaching Logic*. PhD thesis, Open Universiteit, Heerlen, Netherlands, 2020.

[21] J. Lodder, B. Heeren, and J. Jeuring. A domain reasoner for propositional logic. *Journal of Universal Computer Science*, 22(8):1097–1122, 2016.

[22] D. Makinson. *Sets, Logic and Maths for Computing*. Springer, third edition, 2020.

[23] G. Marceau, K. Fisler, and S. Krishnamurthi. Mind your language: On novices' interactions with error messages. In *Proceedings of the 10th SIGPLAN Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2011, pages 3–18. ACM Press, 2011.

[24] M. Mohammed, C. A. Shaffer, and S. H. Rodger. Teaching formal languages with visualizations and auto-graded exercises. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*, pages 569–575. ACM Press, 2021.

[25] M. T. Morazán. FSM webpage. https://morazanm.github.io/fsm/index.html. Retrieved Apr 2022.

[26] M. T. Morazán and R. Antunez. Functional automata: Formal languages for computer science students. In J. Caldwell, P. Hölzenspies, and P. Achten, editors, *Proceedings of the Third International Workshop on Trends in Functional Programming in Education (TFPIE 2014)*, number 170 in EPTCS, pages 19–32, 2014.

[27] T. Nipkow. Teaching semantics with a proof assistant: No more LSD trip proofs. In V. Kuncak and A. Rybalchenko, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 7148 of *Lecture Notes in Computer Science*, pages 24–38. Springer, 2012.

[28] J. O'Donnell, C. Hall, and R. Page. *Discrete Mathematics Using a Computer*. Springer, 2006.

[29] P.-M. Osera and S. Zdancewic. Teaching induction with functional programming and a proof assistant. In *SPLASH Educators Symposium (SPLASH-E)*, 2013.

[30] R. Parasuraman and V. Riley. Humans and automation: Use, misuse, disuse, abuse. *Human Factors*, 39(2):230–253, 1997.

[31] M. Rahn and J. Waldmann. The Leipzig `autotool` system for grading student homework. In M. Hanus, editor, *Proceedings of the International Workshop on Functional and Declarative Programming in Education*, 2002. Available from https://www.informatik.uni-kiel.de/~mh/publications/reports/fdpe02/.

[32] S. H. Rodger. JFLAP webpage. https://www.jflap.org/. Retrieved Jan 2022.

[33] S. H. Rodger, B. Bressler, T. Finley, and S. Reading. Turning automata theory into a hands-on course. In *Proceedings of 37th SIGCSE Technical Symposium on Computer Science Education*, pages 379–383. ACM Press, 2006.

[34] K. Sanders et al. The Canterbury QuestionBank: Building a repository of multiple-choice CS1 and CS2 questions. In *Working Group Reports from the Conference on Innovation and Technology in Computer Science Education*, pages 33–52. ACM, 2013.

[35] A. Sfard. On the dual nature of mathematical conceptions: Reflections on processes and objects as different sides of the same coin. *Educational Studies in Mathematics*, 22(1):1–36, 1991.

[36] M. Sipser. *Introduction to the Theory of Computation*. Cengage Learning, third edition, 2012.

[37] J. Slaney. Logic for fun: An online tool for logical modelling. *The IfColog Journal of Logics and their Applications (FLAP)*, 4(1):171–192, 2017.

[38] A. Stoughton. Experimenting with formal languages using Forlan. In *Proceedings of the 2008 International Workshop on Functional and Declarative Programming in Education*, pages 41–50, 2008.

[39] P. Suppes. Computer-assisted instruction at Stanford. Technical Report 174, Institute for Mathematical Studies in the Social Sciences, Stanford University, 1971.

[40] T. VanDrunen. Functional programming as a discrete mathematics topic. *ACM Inroads*, 8(2):51–58, 2017.

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

[42] J. Waldmann. Leipzig Autotool webpage. https://www.imn.htwk-leipzig.de/~waldmann/autotool/. Retrieved Jan 2022.

[43] J. Waldmann. Automated exercises for constraint programming. In *28th Workshop on (Constraint) Logic Programming (WLP 2014)*, pages 66–80, 2014.

[44] J. Waldmann. Automatisierte Bewertung und Erzeugung von Übungsaufgaben zu Prinzipien von Programmiersprachen. In *18. Kolloquium Programmiersprachen und Grundlagen der Programmierung (KPS 2015)*, 2015.

[45] J. Waldmann. Automatische Erzeugung und Bewertung von Aufgaben zu Algorithmen und Datenstrukturen. In *Proceedings of the Third Workshop "Automatische Bewertung von Programmieraufgaben" (ABP2017)*, volume 2015, 2017. http://ceur-ws.org/Vol-2015/#ABP2017_paper_02.

[46] J. Waldmann. How I teach functional programming. In *Workshop on Functional Logic Programming (WFLP 2017)*, 2017.

[47] A. B. Webber. *Formal Language: A Practical Introduction*. Franklin, Beedle & Associates, Inc., 2008.