

jsdare: a new approach to learning programming



Jan Paul Posma
St Hugh's College
University of Oxford

A thesis submitted for the degree of

Master of Science

August 2012

Abstract

We present an online framework for programming education, called *jsdare*. This framework can be used to create programming courses in, especially for secondary education. In this framework we implement a number of existing interface ideas in a coherent way, including a modern version of the LOGO turtle and Karel the Robot; debugging by quickly stepping forward and backward; a zero-response time compiler using a subset of Javascript; previewing, highlighting and manipulation of statements and expressions; a basic visualisation of variables in different scopes; an integrated reference manual; programming exercises called 'dares' in which the objective is to make a short program, similarly to Code Golf; support for events and HTML canvas in order to make games, with support for pausing time, rewinding time, debugging a specific event, editing the program while paused, and abstracting functions over time. Interface and implementation details are rooted in thorough literature research. The programming course has been evaluated in small-scale field trial, the results of which are mostly positive.

Notes

The work presented in this thesis is the sole work of the author. All text and figures in this thesis may be reproduced, as long as the author is properly given credit, as this work is released under the Creative Commons Attribution 2.0 UK: England & Wales License, except for the pictures that are under the copyright of the respective owners.¹ The source code has not been included since the author retains full copyright, and because around 16,000 lines of code would be too much to print. The report for the Requirements exam has not been included in this thesis as it would add many pages while the most important bits are reiterated in this thesis, but it is available to the examiners upon request. The software can be viewed at <http://preview.jsdare.com> using a modern web browser (Google Chrome, Mozilla Firefox, or Apple Safari). Please use this opportunity to experience the programming course yourself. This thesis is available for download at <http://thesis.jsdare.com>.

¹The details of this license can be found at:
http://creativecommons.org/licenses/by/2.0/uk/deed.en_GB

Contents

Contents	5
List of Figures	7
List of Tables	8
1 Introduction	11
2 Requirements	15
2.1 Requirements report	16
2.2 Related research	17
2.2.1 Pedagogy	17
2.2.2 Language choice	18
2.2.3 Automated assessment and games	19
2.2.4 Microworlds	20
2.2.5 Visualisation	21
2.3 Final requirements	25
3 Interface	27
3.1 Basic editor features	29
3.2 Language design	29
3.3 Error messages	31
3.4 Auto-completion	32
3.5 Stepping	33
3.6 Highlighting and manipulation	34
3.7 Events	36
3.8 Information tab	38
3.9 Output tabs	40
3.10 Dares	42
4 Implementation	45
4.1 Language	45
4.1.1 Parser implementation	45
4.1.2 Syntax tree	46
4.1.3 Compiling to safe Javascript	47
4.1.4 Context object	48
4.1.5 Runner	49
4.1.6 Extra language features	51
4.2 Editor	52
4.2.1 Surface	52
4.2.2 Manipulation	53
4.2.3 Toolbar	55
4.2.4 Editor object	55
4.3 Output tabs	56

4.3.1	Console	56
4.3.2	Robot	57
4.3.3	Canvas	60
4.3.4	Info	61
4.3.5	Input	62
4.3.6	Math	62
4.4	Dares	62
5	Evaluation	65
5.1	User evaluation experiment	65
5.1.1	Method	65
5.1.2	Results	66
5.1.3	Conclusions	68
5.2	Reflection	68
5.2.1	Program performance	69
5.2.2	Social features	69
5.2.3	Types of exercises	69
5.2.4	Interface features	70
5.2.5	Experimental research	70
6	Conclusions	73
	Acknowledgements	75
	Bibliography	77
A	Language	83
B	User evaluation experiment data	87

List of Figures

2.1	GeomLab	18
2.2	LOGO and Karel the Robot	20
2.3	Scratch overview	21
2.4	Scratch variable monitors	21
2.5	Inventing on Principle	22
2.6	Omniscient Debugger	24
3.1	Homepage	28
3.2	First dare	29
3.3	Interface overview	30
3.4	Error	31
3.5	Previewing when using auto-completion	32
3.6	Old statement insertion method	32
3.7	Stepping	33
3.9	Manipulation of colour strings	34
3.10	Manipulation of numbers	34
3.8	Highlighting	35
3.11	Events bar	36
3.12	Play/pause button overlay	36
3.13	Restart button flashing	37
3.14	Abstracting over time	38
3.15	Info tab	39
3.17	Robot moving like LOGO turtle	40
3.16	Output tabs	41
3.18	Canvas dares	42
3.19	Robot dares	43
3.20	Old dares points	43
4.1	Locations and offsets	46
4.2	Original and compiled code	47
4.3	Call ID of circles	49
4.4	Flowchart for updating syntax tree	51
4.5	Syntax tree	52
4.6	Balloon positions	53
4.7	Number manipulation graph	54
4.8	Errors in event slider	55
4.9	Robot path outlines	57
4.10	Saving and restoring canvas performance	59
4.11	Circular buffer	60
4.12	Canvas mirror	61

List of Tables

- 4.1 Synchronisation calls 50
- 5.1 Completion time of dares 67
- 5.2 Questionnaire quantitative questions 67
- A.1 Lexer rules 83
- B.1 Questionnaire scores 87
- B.2 Questionnaire answers (part 1) 87
- B.3 Questionnaire answers (part 2) 88
- B.4 Completion time of dares (part 1) 88
- B.5 Completion time of dares (part 2) 88

'The trouble with computers is you play with them. They are so wonderful!'
— Richard Feynman

Chapter 1

Introduction

It is hard to deny the importance of programming education nowadays. With our ever increasing reliance on code, not just in our computers but in our everyday household items as well, there is a growing need in our society to understand what is happening *under the hood*. Recently programming and computer science education has received widespread attention. In Britain, for example, the Secretary of State for Education Michael Gove recently announced an education reform in ICT, which until now consisted mainly of teaching the use of office programs and internet browsing [36]. He stressed the importance of rigorous computer science education, both for general knowledge and for practical use in many professions. This announcement came shortly after the publication of the *Next Gen.* report commissioned by the government, which gives as a first recommendation to teach computer science at schools; and a speech by Eric Schmidt of Google, criticising the current ICT curriculum [57, 77]. Another example is Michael Bloomberg, mayor of New York City, who pledged to learn programming in 2012 using the *Codecademy* online course [14, 2], sparking the debate on whether or not everyone should be taught programming in schools [10, 38, 78].

In this thesis we present a website for learning programming, aimed at secondary school education. This is essentially a programming course, but not one in the conventional sense, where students first learn some theory, and then do some exercises. We use a more *explorative* approach, in which students first discover how programming works, to get an intuition and motivation. The focus of this thesis is on the implementation of a number of user interface concepts that help students explore their programs. We have also implemented a couple of exercises, and evaluated them in a small-scale field study. These exercises are just examples of how our website can be used for teaching programming. Our website is in fact a *framework* in which different kinds of exercises can be built. In this thesis we will refer to our software as a '*programming course*', since we have not only thought about the user interface, but also about the pedagogy of programming, even though the exercises themselves are not the main focus of this thesis. We call our programming course *jsdare*, for *Javascript 'dares'* (exercises or challenges).¹

We started investigating the requirements for such a programming course in January 2012, for the Requirements exam, which is part of the MSc program at the University of Oxford. In our report we asked the question [74]: '*What are the requirements for an interface used for learning programming?*' We interviewed four teachers in order to answer this question, and we used these interviews as the basis of this project. In chapter 2 we look at this report in more detail. We also discuss an extensive list of requirements for this project, based on the report and a thorough study of the literature.

In chapter 3 we look at the interface concepts we have used in our programming course. We mostly build upon tried and tested concepts, such as the *LOGO turtle* [69]. Nowadays students have higher expectations of graphics and interactivity, so our contribution is the adaptation of these existing ideas to a modern context. An additional challenge is the implementation inside the web browser, which is much more restrictive than when using a general purpose compiler.

We do not only use tried and tested ideas, but also new ones. One particularly notable source of ideas is the recent presentation by Victor, *Inventing on Principle* [87]. In this presentation a lot of novel concepts are shown

¹The reason for this name is simple: the domain name `www.jsdare.com` was still available. Please go to `http://preview.jsdare.com` to view the latest version.

as prototypes, and we felt that many of them could be used effectively in our course. The reason for this is that most of these ideas are difficult to implement when dealing with relatively large computer programs, but are very applicable to smaller programming exercises, such as in our course. Therefore we dedicated a large part of this thesis to the implementation of these ideas. We present a complete and efficient implementation in the web browser.

We also present an interesting way of building programming exercises by simply restricting the number of lines of code that can be used, a variation on *Code Golf* [1]. As far as we know, this is the first time this method is used for teaching the basics of programming, and not just as a recreational activity for experts. Note that in general we use the word ‘*exercises*’, a subset of which are those with a length restriction, which we call ‘*dares*’. In this thesis we only look at dares, but it is possible to also include other kinds of exercises in the future.

In order to make these interface concepts possible, we have built a custom programming language which is a subset of Javascript, which we call *js--*. This way we have greater control over the execution, and it allows us to produce better error messages. In chapter 4 we look at the implementation details of this language and the interface concepts in detail. Besides discussing the different modules of the software, we discuss optimisation considerations, and how we solved some of the challenges of implementing the interfaces in a web browser.

To evaluate the programming course, we did a small-scale user evaluation experiment, in which 7 children tested the software. While this was only a short test with a few children, and without a proper comparison, it did give us some insight in how they used and perceived the course. This may well guide future development of the course. In Chapter 5 we present the results of this evaluation, and we also reflect on which requirements we have actually implemented.

In chapter 6 we discuss some ideas for future work, such as making further optimisations for performance, testing other interaction concepts, and implementing other types of exercises. We have the intention to keep on working on this software, and to build it into a platform in which any teacher can create different kinds of exercises, which can then be used by any student.

We finish with some concluding remarks in Chapter 7.

'He began playing with the blocks for incrementing variables, then reached out and shook the researcher's hand, saying "Thank you, thank you, thank you." The researcher wondered how many eighth-grade algebra teachers get thanked by their students for teaching them about variables?'
— Scratch: Programming for All

Chapter 2

Requirements

When designing a new programming course from the ground up, a lot of obvious questions arise: ‘Which programming language?’ ‘Which editor or IDE do we use?’ ‘What kind of examples?’ ‘How do we mark exercises?’ When examining each question, a lot of deeper considerations arise, which have wide implications. For example, when deciding which programming language to use, one has to decide which paradigm to teach. When teaching functional programming, the kind of examples that can be used becomes restricted, since most functional programming languages are mostly used for mathematical instead of graphical purposes. Then, when having chosen a paradigm, you can select a language from a list of existing and popular languages, keeping in mind the kind of examples and editors you might want to use. But since we are designing something new, from the ground up, these obvious questions with their obvious answers do not necessarily apply. They are part of the traditional framework of how we currently do programming.

There are a large number of tools and techniques out there designed to improve both education of programming, and programming itself. These works all have different goals, different scopes. Some are aimed at increasing motivation, others are more focused on mathematical skills. Some are focussed on computer science freshmen, while others focus on young children. In order to decide the goals and scope of our project, we use a requirements report we created earlier. For this report we have interviewed a number of teachers about how they teach, and what they value most in a computer science course. We take these interviews as a starting point, but also look at the existing literature more thoroughly.

In the current literature, there are a lot of different kinds of tools and techniques, such as (educational) programming languages and debugging methods. A comprehensive overview of the literature is given by Pears et. al. in 2007 [71]. They use the following classification of educational programming literature:

- Curricula
- Pedagogy
- Language choice
- Tools for teaching
 - Visualisation tools
 - Automated assessment tools
 - Programming environments
 - * Programming support tools
 - * Microworlds
 - * Other tools (e.g. plagiarism detection)

In this classification, *every category* is relevant to our project. When creating a completely new programming course from the ground up, we take a holistic approach, so we have to consider all the different possibilities, all the ideas that could be incorporated in the course. In this chapter, we try to formulate a number of requirements

that we used for designing this course, which we motivate using both the requirements report, and other research. For each topic we look at previous research in order to formulate accurate requirements.

2.1 Requirements report

In January 2012 we started thinking about the design of the programming course, for the Requirements exam at the University of Oxford. For this exam we had to write a report about a requirements elicitation process, and we decided to investigate the requirements of a programming course, with the intention of continuing with this for the thesis. Our report tries to answer the question [74]: ‘*What are the requirements for an interface used for learning programming?*’ In order to do this, we used a number of methods, such as a literature review and a think-aloud experiment. We did not execute these methods in-depth, however, so we will not discuss them here. Much more valuable are the four semi-structured interviews we conducted for this report [25]. We interviewed four teachers, who all had many years of experience with teaching programming classes. The first two were faculty members at a university, the third taught at a local youth centre, and the fourth one taught as a student assistant at a university. After these interviews we analysed what they value most when teaching programming, and how they implement these values and beliefs in their classes. We compiled a list of requirements from the recommendations they gave.

In this thesis we use these values and requirements as a starting point for creating a new programming course. It is therefore important to recognise that the *scope* of our programming course is heavily influenced by the scope of the classes taught by these participants. While our programming course is aimed at secondary school education, three of the participants teach at an academic level. However, they all teach introductory classes, to students who usually have no experience with programming at all. Also, the participant who teaches at a youth centre has more views and recommendations on teaching children. Therefore we would argue that the scope of our programming course would be the older half of secondary school education, for example high schools, with an academic focus.

We first look at the values the different participants uphold. First and foremost, they all seem to value teaching *applicable, real-world skills*. They all said that they teach programming languages that are widely used in practice, and that this is one of the most important aspects when choosing a language. This might be a bit surprising, as one might expect that general programming concepts would be more important to academics. While programming concepts are mentioned as well by most participants, they all choose their languages and tools based on actual applications in the domain they teach in. The second participant mentioned he would like to teach Pascal, but in fact teaches C because it is more widely used, and has similar semantics. The three academics all value, unsurprisingly, formal methods, such as program correctness. They do disagree somewhat about which methods are in fact effective.

To make programming engaging, and to increase motivation, participants mentioned the use of graphical elements. The first participant used Visual Basic for some classes, which has an attractive visual interface. Another one admitted that it is hard to motivate students when they all expect flashy animations, but does not address this in his classes. The third participant mentioned that he always asks children at the youth centre what *they* would like to make, and the answer is almost always *games*. Some participants mentioned that making software that students can show to their peers increases motivation as well.

We base our programming course on the two most agreed on values: *real-world applicability*, and *motivation*. Based on these values we select a number of most important requirements as discussed in the report. First of all, every participant mentioned a simple, real-world language, with a simple editor. The language should not have many features, as this might confuse novice students. One participant remarked that it is *not fair* to students to have them write things they cannot yet understand, such as the mandatory class with a ‘main’ method, when teaching imperative programming in Java. Because of the real-world applicability, we cannot use an educational language, which is rarely used outside an educational context. Since the third participant explicitly mentioned that children like to make games, we add this as a main requirement as well.

In order to get a better understanding of what a program does, some participants mention that it is important to have a short cycle between editing the program and seeing the result. An interactive console is a great example of an environment that gives immediate results, instead of having to wait until a program is compiled. Visualising programming concepts such as a call stack may also be beneficial.

Finally, some participants mention an automatic program verification system used by their university. They recommend such a system both for better understanding and for motivation. First, it gives error messages specific to the problem, not just compile- or runtime errors. Students can also benefit from this at home, without a teacher present. Second, it motivates students by making a game out of it; at every submission of the program it is exciting to see whether or not it works. The participants warn us, however, about using such an automatic system. It is possible, for example, to keep patching up a program based on the failed test cases, thus creating a program with a lot of special cases, where a general algorithm would have sufficed. Also, if the student gets stuck and is working at home, he cannot work on the problem further. Therefore, such an automated system cannot completely replace a human teacher, both for helping students, and grading programs.

2.2 Related research

In order to motivate the requirements and get a better understanding of what is involved in a programming course, we look at a number of topics. For each topic we consider previous research that is inside of the scope that we defined in the previous section: a course for high school children, valuing primarily real-world applicability and motivation.

2.2.1 Pedagogy

A paper by Robins et. al. gives a broad introduction to the research that has been done into programming pedagogy, and is particularly focused on novice programmers [76]. One of the interesting topics in this paper is the notion of different mental models when learning programming, such as the difference between *knowledge* and *strategies*.

Programming knowledge is defined as an understanding of what the different statements in a programming language do. There is always a gap between the *physical machine* and the *notational machine* that one learns to program. In this context the concept of a *notational machine model* is relevant, which is an idealised explanation about what happens inside the computer when using a certain programming language [76]. For example, the notational machine model for an imperative language is completely different from that of a functional language. The goal is to tell a convincing story that hides enough details of what actually happens inside the physical machine, but is useful enough to gain knowledge about the language.

Programming strategies are ways to build a program, and can be seen as small building blocks or patterns which are very common in programs [76]. An example is the strategy to repeat something a certain number of times using a for-loop. Both programming knowledge and strategies can be *fragile*; learned but missing, not used, or misused. Programming statements can be understood (knowledge), but not applied because of a lack of programming strategies. For example, someone may know what a for-loop is, how to increment a variable, and how to compare a variable with a constant, but still struggle with repeating a statement the correct number of times, because of lack of experience with this strategy. Strategies can be misused by a lack of understanding of the programming language or notational machine, even though in most cases the small building blocks or patterns do work. For example, one might know the for-loop strategy for counting upwards, but might fail at adapting it for counting downwards.

The question then arises of whether it is useful to teach programming knowledge and strategies explicitly. Robins et. al. cite Mayer, who gives some evidence that students who are given a notational machine model do better on some kinds of problems [62]. Perkins and Martin suggest that teaching specific problem solving techniques, such as tracing a program step by step, may help students in acquiring more robust knowledge [72]. It is unclear whether or not explicitly teaching strategies is helpful. Recent research by Kranch suggests that it is not, or that students should at least have some experience before presenting the strategies explicitly [52]. Another approach is to start with some problems, and only introduce knowledge and strategies when needed, *problem-driven* learning. By doing this, students begin with thinking about a solution for some problem, instead of being confronted by a lot of theory. Deek et. al. show that this method can have a significantly positive impact on competence, perception, and motivation [7].

In our requirements we include the problem-driven approach. We also want to provide users with explicit knowledge about the language in the form of an integrated reference manual. When designing exercises, strategies

should be taught by sometimes giving examples, while at other times giving users a clean slate, so that they can apply the learned techniques.

Another relevant finding mentioned by Robins et. al. is the notion of three types of people: *stoppers*, *movers* and *tinkerers* [76]. Stoppers give up easily when running into problems, are reluctant to find solutions themselves, and ask many questions. Movers try to find solutions when running into problems before giving up. Tinkerers are extreme movers, and try *too much* themselves, thus lacking understanding in what they are in fact doing. A perfect interface would help stoppers to move on, and slow down tinkerers, although this may be hard to accomplish. It is hard to make this into a requirement, but we would suggest an interface that invites users to experiment, but also forces users to stop and think from time to time.

When searching for information about programming, students prefer to seek this online, or by asking a colleague, as shown by Pierce and Aparício [73]. This further strengthens our requirement of having a built in reference manual. We can also consider the situation of students learning at home, in which case it is more difficult to ask fellow students for help. A thorough study by Bernard et. al. shows that in distance learning courses in general, student-student and student-teacher interactions improved both achievement and attitudes [13]. This suggests a requirement of social networking features in the course, potentially combined with teacher assessment tools (Section 2.2.3).

2.2.2 Language choice

We have established that the course should have real-world applicability, so this value should be reflected in the choice of programming language. Two paradigms that are widely used in industry are imperative/object-oriented programming, and functional programming. An example of a functional language for education is GeomLab, also developed at the University of Oxford, which uses image composition and manipulation to create artwork in the style of M.C. Escher (Figure 2.1) [79, 28]. Some functional languages that are widely used include Haskell, Erlang, and Lisp.

The problem with these, in our case, is that they are not often used for making games, another one of our requirements. For this the imperative or object-oriented paradigm is more suitable, used by languages such as C++, Java, Javascript, Python, Visual Basic, and Ruby. Since we are designing an online course, a logical choice of language would be Javascript, since it is the most popular online language, and is universally supported by web browsers [18, 89]. Recently it has become possible to make simple games easily using the `<canvas>` element [45]. Besides that, it is a modern and powerful language, and can now also be used on the server-side, and for native applications. Furthermore, the syntax is similar to other widely used languages, such as Java and C(++).

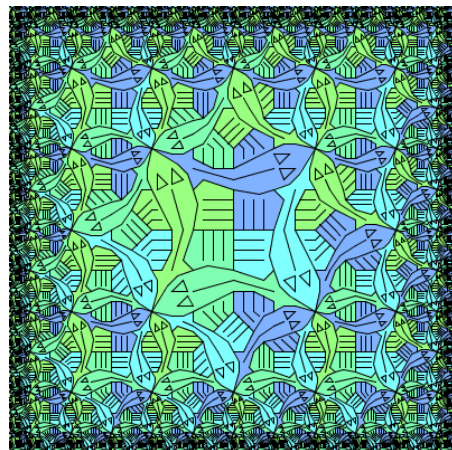


Figure 2.1: GeomLab can be used to create artwork in the style of M.C. Escher. (Copyright Michael Spivey)

One problem with Javascript is that it is object-oriented, while the requirements report recommends an *objects later* approach, which is cautiously supported by Robins et. al. [76]. On the other hand, most real-world programming languages are object-oriented instead of imperative, and recent research by Ehlert et. al. shows that the choice between objects first and later does not matter that much [26]. To further complicate things, the prototypical inheritance model of Javascript is relatively hard to understand, has a difficult syntax, and is uncommon in other languages [20]. Also, it is almost impossible to program in Javascript without using objects, since all the browser APIs use objects.

An approach recommended by Brusilovsky et. al. is to use a subset of a language, the *sub-language approach* [15]. The idea is to take a real-world language, but to remove a number of features that complicate the language. This seems like a good compromise, as we can keep the syntax to *access* objects, but to remove the syntax to *create* them. Other difficult syntax can be removed in the same way. As such, we add as a requirement that we should use a subset of Javascript.

2.2.3 Automated assessment and games

In the requirements report we discussed the possibility of using an automated assessment tool, as used by some of the teachers we interviewed. They gave a number of advantages: being able to verify exercises at home, making exercises into a game, and error messages specific to the problem stated in the exercise. They also mentioned a downside, which is that an automated assessment system cannot completely replace a teacher. Code that passes unit tests can still be bad code. And when a student works at home and gets stuck, he still has to wait for a class in order to be assisted by a teacher.

Some literature reviews compare different assessment systems, but unfortunately there is not much data on how effective these systems are [24, 47]. Still, there is one high-quality paper by Voit et. al. in which a number of automated assessment techniques are tested over a 5-year period [90]. They argue that automated testing using unit tests leads to better achievement, increased motivation, and reduced cheating. However, they also report increased stress among students, if only sometimes the online submissions were marked. They recommend doing many evaluations, and marking them all, but discarding 30%-50% of the lowest marks, as to reduce stress while maintaining motivation. They also recommend manually checking final exams, and they report a high correlation between the automatically and manually assessed exercises. Finally, an advantage is that manual assessment becomes easier in conjunction with automated testing, as the teacher already knows whether or not a program passed the test suite. These findings are consistent with earlier research [16, 27]. It seems therefore that a good requirement is to include automated assessment for exercises, which should be used very often. Furthermore, it is a good idea to have an integrated system for teachers to monitor the progress of the students, select some exercises to be included in the final mark, and manually assess some submissions.

Besides using test suites, Ihantola et. al. cite a number of other methods, such as simple output comparison using one or more model answers, static model checkers, and comparing program graphs with model answers [47]. There are also a number of methods to prevent ‘mindless trial and error’ (cf. ‘tinkerers’ from Robins et. al. [76]), which was also noted in the interviews of the requirements report. For example, we can limit the number of resubmissions, or the amount of feedback given. Another possibility is to introduce a compulsory time penalty after each submission, to force users to think again. An interesting idea mentioned by Ihantola et. al. is to have manual assessment done by having students check each others’ work, or by writing their own test cases [47]. We can include a requirement that some measure should be taken to prevent too much trial and error behaviour when submitting.

One method we think can be used to avoid too much trial and error behaviour, is forcing users to make short programs. This idea is based on the anecdotal evidence by the interviewees that users include a lot of special cases in their programs, instead of rethinking the general solution. This leads to longer programs, so perhaps imposing a limit on the length of a program can force users to stop and think. This is also known as *Code Golf*, a kind of competition in which the goal is to make programs as short as possible [1]. This is mostly done among more advanced programmers as a recreational activity, and we have not seen it in any educational setting. Moreover, there is ample research on Code Golf in general. Therefore, we feel it is worth trying this in our programming course. In this thesis we call these kinds of exercises *dares*.

One of the requirements we set based on the requirements report is that students have to be able to make games. The main reason for this is motivation, as children are highly motivated by the idea of making their own games. There is a lot of anecdotal evidence to support this notion [80, 54, 49, 23, 68, 39], although comparative data is hard to find. A study by Feldgen et. al. shows lower drop out rates out of computer science courses [29]. The reason that we bring up games in the context of automated assessment, is that it is hard to do automated assessment for games. It is still possible to do unit testing on the program, but integration or acceptance testing for an entire game, even simple ones, is difficult.

Of course, it is not impossible to test games. A blog post by Llopis details a system in which the state of a game is recorded for many simulated plays of that game, and this is then compared against new versions [58]. This is not an educational assessment system, but such a system could work in the same way, testing for different simulated plays against a model implementation. Obviously this only works with deterministic games, which is not necessarily an issue. The real problem is that this requires users to copy the model implementation exactly, and even a small rounding error can propagate after some seconds to a huge error compared to the original, while the game is still perfectly playable. Another, perhaps even bigger problem is that it takes away all opportunity for creativity when designing a game, which is one of the things that makes it motivating. We could not think of an

appropriate assessment system, except for manual assessment by teachers or fellow students.

When we are talking about games, there are other possibilities besides designing a game, such as writing programs that can play games, such as chess. One example is the *CodeCup*, an annual tournament in which programs compete in two-player games [85]. Another example is *Robocode*, in which programs control a robot that fights other robots in an arena [42]. Long shows that both novice and experienced users enjoy playing Robocode, and that their main motivation to do it is intrinsic [59]. An advantage of using two- or multi-player games in automated assessment, is that it is easy to compare how well the programs perform relative to each other. The disadvantage of this is that players who do worse than other players will get a lower ranking, while their code quality might be good in terms of clarity, maintainability, etc. Still, it is an option worth considering.

Finally, there is *gamification*, which has recently gained in popularity. It usually means making a game out of something by adding awards and badges. In the recent article by Deterding it is suggested that gamification can add to motivation, but only if there is already enough motivation present in the first place [22]. Also, not everyone likes it necessarily. When added with thought and proper research, it could improve the programming course.

2.2.4 Microworlds

The archetypal example of an educational microworld is Papert’s *LOGO*, described in detail in his *Mindstorms* book [69]. LOGO uses a *turtle*, which is a robot, either physical or simulated, that can be driven around with simple commands. The robot leaves a trail, so pictures can be drawn by driving it around (Figure 2.2a). Papert describes a vision of children learning mathematics by talking to the computer in a *math language*, in order to learn mathematics in the same way one would learn French while living in France. By being immersed in a computer environment, children could naturally learn the language of the computer. He describes a conversation between children learning with LOGO (*playing*, really), who try to make a specific image. They try different things, and stop now and then to think if they can apply generalities they have learned (e.g. the ‘Total Turtle Trip Theorem’).

We think the LOGO turtle is still a great metaphor to use when learning programming. First, geometry is a nice field that can be explored using the turtle, and can tie in with formal mathematics classes. It is also a great first step to learning more complex geometric concepts such as trigonometry, which is used a lot when programming games. It is also possible to learn about methods that return a value, by equipping the turtle with some sensors, as with *Karel the Robot* (Figure 2.2b), created around the same time as LOGO [70]. With the turtle it is possible to make very pretty graphics, and the turtle can be animated, increasing motivation. It can be used in conjunction with robot classes teaching *LEGO Mindstorms* (aptly named after Papert’s book), which is already relatively popular at schools [53]. Finally, it has the potential to be used in two- or multi-player games. Therefore, we think it is a good requirement for our programming course.

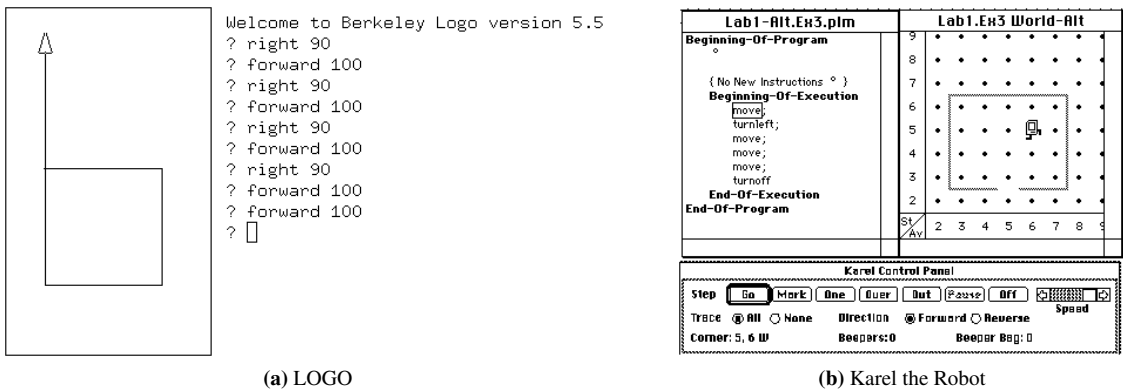


Figure 2.2: Two well-known microworlds: LOGO (a), and Karel the Robot (b). (Copyright University of California, Berkeley, and Richard Pattis, respectively)



Figure 2.3: Scratch is a visual programming language in which programs are always running. (Copyright MIT Media Lab)

2.2.5 Visualisation

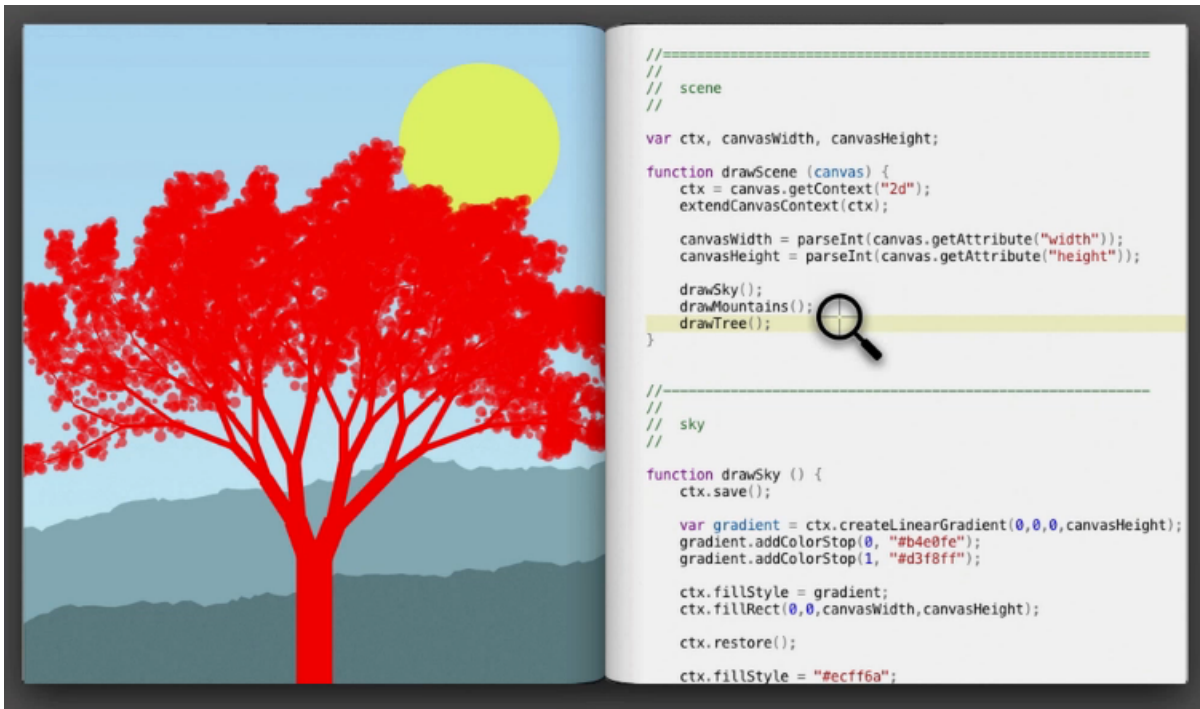
When talking about visualisation, we cannot help but mention *Scratch*, a popular educational system for children, with around 25000 new users each month [60, 5]. Scratch targets younger children, and it uses a graphical programming language (Figure 2.3). It focuses on creating 2D animations, and is also very suitable for making simple games. Users have a lot of freedom in what to create, and can learn a lot of programming techniques without directed teaching. An integral part of Scratch is the online community, where children can share their creations, and adapt the programs of others, so-called *remixing*. This online community opens up new ways of learning, such as learning communication skills in international collaborations.

Particularly interesting about Scratch is that they introduce a number of useful interface concepts. First of all, the program is always *live*, which means that there is no run button. Whenever the program is changed by the user, the new program is immediately used instead of the program that is currently running (level 4 liveness in Tanimoto’s taxonomy [81]). This results in an extremely short cycle between editing the program, and seeing the result, which is one of our requirements. Scratch is also what they call *tinkerable*, which means that it is easy to experiment. For example, by double clicking on any command runs it immediately. This, again, means there is a very short cycle between asking a question (*‘What does this command do exactly?’*), and getting the answer, allowing to explore the programming language rapidly. Brusilovsky et. al. call this a *zero-response-time compiler* [15].

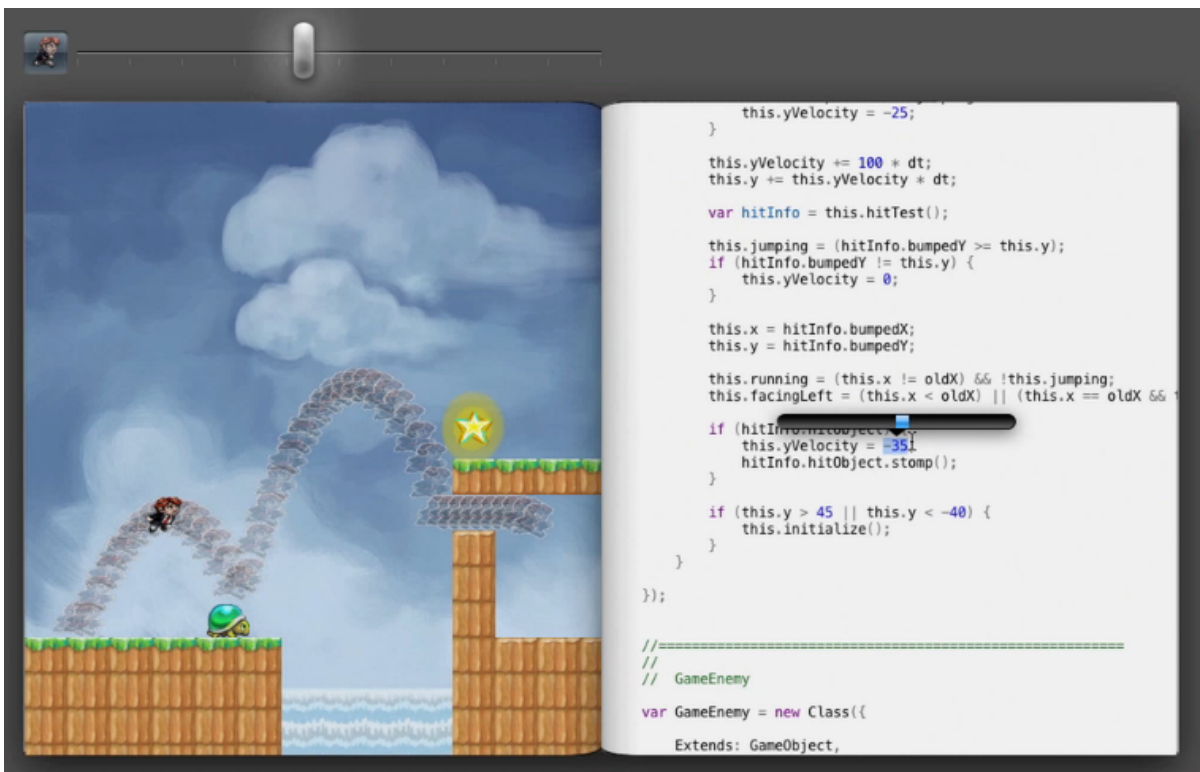


Figure 2.4: Scratch features a number of interesting interface concepts, such as variable monitors. (Copyright MIT Media Lab)

In Scratch, variables are visualised using *variable monitors* (Figure 2.4). Such a monitor shows the current content of a variable, and allows quickly experimenting with different values using a slider. This is reminiscent of interactive manipulation in *Mathematica* [91]. Another feature is the built-in sprite editor, enabling also the learning of



(a) Highlighting



(b) Abstracting over time

Figure 2.5: An interface prototype shown in *Inventing on Principle*. When highlighting a line of code, the corresponding image element is also highlighted, and vice versa (a). Time can be paused and reversed, and an abstraction over time can be shown, which is the trail of the character. Using code manipulation a numeric value can be adjusted on the fly, while the trail is simultaneously updated (b). (Copyright Bret Victor)

artistic skills, and increasing motivation. Finally, there are a number of deeper features that are worth mentioning, but which we will not discuss in detail, namely their approaches to errors and concurrency.

The most important previous research we have used, however, is the recent presentation *Inventing on Principle* by Victor [87]. His work is guided by a principle, which is that creators should have an ‘immediate connection with their creations’. As an example he demonstrates what is essentially a zero-response-time compiler, but with many additional features. For example, in his interface it is possible to hover over a piece of code, while the corresponding output on the canvas is highlighted (Figure 2.5a). Vice versa is also possible, hovering over a pixel on the canvas, while the corresponding line of code gets highlighted. This allows to quickly see which parts of the program correspond to which output. Another feature is to be able to select any numerical value, and change it by dragging (Figure 2.5b), which is somewhat similar to the sliders in Scratch.

The best part of the presentation is when he shows how to make games in this prototype. When changing the game behaviour, one would expect having to restart the game over and over again every time a statement is changed. This breaks the principle of the immediate connection, which is solved by allowing to pause the game, go back in time, and then edit the code. This caused the history to be replayed using the new program; every keyboard and mouse event is repeated, thus showing immediately what *would have happened*. He also shows what it means to abstract over time, which is rendering some object for every frame in the history (Figure 2.5b). This can give more insight into what is going on in the program, and allows to change the program while keeping an overview of the entire history.

When seeing this presentation, it occurred to us immediately that this can potentially help programming education tremendously. At the same time it was a nice opportunity to be the first to actually implement this in full, not just as a prototype. There are already a number of projects that include some of the ideas from this presentation [32, 88, 37, 46, 8], but especially complicated ideas such as abstracting over time have not yet been implemented inside a web browser. Therefore, this has become one of the primary requirements, and an important focus of this project. We discuss the details of his interface, and our adaptation of it, in detail in the next chapter. Still, we do not have any experimental data if this work is indeed beneficial for student achievement and motivation, which could be investigated in future research (Section 5.2.5). A literature review by Naps et. al. reveals that about one third of visualisations have a significant effect on achievement, and about 80% have a significant effect on motivation, so we can be relatively hopeful about the results [64].

Naps et. al. also present a list of best practices [64]. We have selected a couple of them which we think are particularly relevant to our project. Two of those are: ‘*Include execution history*’ and ‘*Support flexible execution control*’. These recommendations are aimed at visualisations, but could also apply to programming interfaces. This means that users should be able to step through the program, as is often done in a *debugger*. However, flexible execution controls means that they have to be able to step back as well, which is uncommon in debuggers (but not unheard of [31, 41]). A particularly nice implementation of this is the *Omniscient Debugger* by Lewis, which stores the complete execution history of a program [55]. It then allows to step through it very flexibly, and with some additional visualisations.

In the Omniscient Debugger, you can step forward or backward through the program, and the current line is always highlighted [55]. At the same time, there are a number of views on the screen, showing the stack and variables. The elements (variables, stack items) that are affected by the current step, are highlighted at the same time, giving a quick overview of what is happening. Users can click on any element in order to jump to the closest statement in which that element was altered. It is also possible to expand a variable, in order to see every value that was ever assigned to it, and jump immediately to a certain assignment.

We mentioned earlier that the notational machine model — the story or abstraction that describes how a program is executed — is important in acquiring knowledge about the programming language [76]. We believe that such a debugging tool may assist users in understanding how the program is executed, since each step is shown explicitly. Therefore, this is another requirement.

Some final considerations come from *Alice*, a language similar to Scratch, but with a three-dimensional world [19]. They make a few recommendations for programming languages for children. First of all, they recommend making built-in functions easier by using optional arguments, so that users do not have to use the full power of all functions at first. Luckily this is supported by Javascript. Another recommendation is to use animations when a state in the interface changes. They use an undo button as example, using an animation to show what changed. In the literature there seems to be evidence that some animation can be beneficial, if used cautiously [75]. Tversky

Omniscient Debugger 20.Nov.04 - com.lambda.Debugger.QuickSortNonThreaded

File Run Code Trace Filter Objects Debug Help Previous [Navigation Icons] Stamp[335] 99 Clock[0.033] 0.011 QuickSortNo:120

Threads [Navigation Icons]

```
<javawsApplicationMain_0> Lor
<Primordial_1>
<main_2>
<AWT-EventQueue-0_0>
```

Method Traces [Navigation Icons]

```
<QuickSortNon..0>.sort(0, 10) -> void
<QuickSortNon..0>.average(0, 10) -> 885
<QuickSortNon..0>.sort(0, 5) -> void
<QuickSortNon..0>.average(0, 5) -> 294
<QuickSortNon..0>.sort(0, 3) -> void
<QuickSortNon..0>.average(0, 3) -> 81
<QuickSortNon..0>.sort(0, 2) -> void
<QuickSortNon..0>.average(0, 2) -> 0
<QuickSortNon..0>.sort(0, 0) -> void
<QuickSortNon..0>.sort(1, 2) -> void
sort -> void
<QuickSortNon..0>.sort(3, 3) -> void
sort -> void
<QuickSortNon..0>.sort(4, 5) -> void
sort -> void
<QuickSortNon..0>.sort(6, 10) -> void
```

Objects [Navigation Icons]

```
int[11]_0
10 1227
9 1968
8 1233
7 1476
6 1725
5 984
* 4 243
3 492
* 2 735
1 0
0 1
<QuickSortNonThreaded_0>
array int[11]_0
<QuickSortNonThreaded_0>
array int[11]_0
<QuickSortNonThreaded_0>
array int[11]_0
```

Stack

```
QuickSortNonThreaded.main(St
QuickSortNonThreaded.sortNE1
<QuickSortNon..0>.sortAll()
<QuickSortNon..0>.sort(0, 10
<QuickSortNon..0>.sort(0, 5)
<QuickSortNon..0>.average(0,
```

Code [Navigation Icons]

```
return;
}

average = average(start, end);
middle = end; // This will become t

L: for (i = start; i < middle; i++) { // Start the pivot
    if (array[i] > average) { // Move all values
        for (j = middle; j > i; j--) {
            if (array[j] <= average) { // all
                tmp = array[j];
                array[i] = array[j];
                array[j] = tmp;
                middle = j; // The pivot point remains in t
                continue L;
            }
        }
    }

    sort(start, middle-1); // Do the bott
    sort(middle, end); // Do the top half here.

    return;
}

public int average(int start, int end) {
    int sum = 0;
    for (int i = start; i < end; i++) {
        sum += array[i];
    }
    return (sum/(end-start));
}
}
```

Locals [Navigation Icons]

```
start 0
end 5
* sum --
* i --
```

Locals [Navigation Icons]

```
start 0
end 5
* sum --
* i --
```

TTY Output [Navigation Icons]

```
----- QuickSortNonThreaded Program -----
-- Out of order: array[9]=1968 > array[10]=1725 --
-- 0 0 --
-- 1 1 --
-- 2 243 --
-- 3 492 --
-- 4 735 --
-- 5 984 --
-- 6 1227 --
-- 7 1233 --
-- 8 1476 --
-- 9 1968 --
-- 10 1725 --
-- 33ms --
```

this

```
<QuickSortNonThreaded_0>
array int[11]_0
```

From last: -53 stamps, -0.022secs First Line in: <QuickSortNon..0>.average(0, 5) -> 294

Figure 2.6: In the Omniscient Debugger different kinds of visualisations are shown, and elements corresponding to the current line of code are highlighted. (Copyright Bil Lewis)

et. al. suggest that animation can be beneficial if used to show something that naturally changes over time or has a causal relation [82]. We add these recommendations from Alice as requirements [19]. Another recommendation is to use number of full turns instead of degrees when using a spatial environment, but we agree with Papert that using degrees can give students insight into mathematics, i.e. geometry [69]. As a final remark we would like to recommend the work by Kelleher and Pausch, which gives a great overview of many more educational programming environments [50].

2.3 Final requirements

To summarise, we have a number of requirements, which we have tried to implement in our programming course for secondary school education. The list is ordered by priority, using the MoSCoW method [67]. In this method priorities are ranked using *MUST*, *SHOULD*, and *COULD*, where the first one is the most important. It also includes a negative priority *WON'T*, which we do not use here. Note that not everything on the list is actually implemented, so the requirements that we could not implement are a basis for future research (Section 5.2).

1. The programming language **MUST** be simple, real-world, and imperative
2. Students **MUST** be able to make games
3. A zero-response-time compiler **MUST** be implemented
4. The main interface concepts from *Inventing on Principle* **MUST** be implemented [87]
5. The course **MUST** use a problem-driven approach
6. The course **MUST** include automatically assessed exercises
7. The programming language **MUST** be a subset of Javascript
8. Debugging as seen in the Omniscient Debugger **SHOULD** be implemented [55]
9. The course **SHOULD** include the Canvas API
10. The course **SHOULD** include a robot microworld
11. Sometimes examples (strategies) **SHOULD** be given in exercises
12. Sometimes users **SHOULD** get a clean slate in exercises
13. The course **SHOULD** include ‘dares’, exercises in which programs are limited in length
14. The course **SHOULD** force the student to stop and think from time to time
15. The course **SHOULD** include a reference manual
16. Forcing to stop and think **COULD** be done by limiting the length of the program
17. Built-in functions **COULD** have optional arguments to make them simpler to use
18. Animation **COULD** be used (cautiously) to indicate (time- or causality-based) transitions
19. The course **COULD** include social networking features
20. The course **COULD** include features for student-teacher interaction
21. The course **COULD** include student-student assessment tools
22. The course **COULD** include student-teacher assessment tools
23. The course **COULD** feature two- and/or multi-player games
24. The course **COULD** use achievements, badges, ribbons, etc. (gamification)

*'19th century culture was defined by the novel, 20th century culture by cinema,
the culture of the 21st century will be defined by the interface.'*
— Lev Manovich

Chapter 3

Interface

In this chapter we discuss how our programming course works from the eyes of a user.¹ We do not discuss the actual implementation details, that is saved for next chapter. When we talk about our course, we are not talking about content, such as the actual exercises students can do. We have not developed large numbers of exercises we can discuss, only a few to test our approach with some children (Section 5.1). We are talking about the interface that enables this content, since new interfaces allow for new educational approaches.

When navigating with a web browser to the URL of our programming course, the homepage is shown (Figure 3.1). The first thing users see is an example game, which can be played immediately. This example game has been adapted from a Javascript tutorial by Mill [63]. The code of this game can also be changed immediately, demonstrating what a program looks like, and — if they are already more advanced — what the interface is capable of. Below this example are a number of *dares*, i.e. exercises in which programs have to be as short as possible. An orange arrow indicates where to start. Note that the final homepage will most likely be slightly different, such as with a menu for navigation, and buttons to login or register.

When the first dare is clicked, the editing interface is shown on top of the original page, as a modal window (Figure 3.2). We consistently use this approach when showing this interface, except for the example in the beginning. The rationale is that the original context is still shown in the background, so that users always can get back to the collection of dares they were selecting one from. If we would have chosen not to use a modal window, we would have to show the navigation between the dares somewhere else, and the editing interface is already quite large. We considered showing navigation at the top, as with *Code School* [3], but this would break consistency with the vertical visualisation of the collection of dares, which is nice for showing some more information such as high scores. Finally, we expect that users know this kind of behaviour, since it is very common when viewing pictures, especially on *Facebook* [4]. We follow most best practices for modal windows [84], except not using the escape button for closing (already in use for pausing time; Section 3.7), and not closing when clicking outside of the window (to prevent accidental clicks).

In the text of the dare there are some orange underlined words. Hovering over them with the cursor reveals some more arrows. This is a nice and unobtrusive way to literally point users in the right direction. We think this is a better way than using a tutorial that guides the user through an interface, since this way we allow them to explore the interface at their own pace, not forcing them to learn according to a specific path.

In Figure 3.2, only two tabs are visible at the top left, the *dare tab* and the *robot tab*, which is our microworld based on LOGO and Karel the Robot [69, 70]. Depending on the dare there can also be other tabs visible, namely the *canvas tab*, *console tab*, and *info tab*. In the rest of this chapter we discuss the editing interface in detail, including the different tabs.

¹Do experience this yourself! Point a modern web browser (Google Chrome, Mozilla Firefox, or Apple Safari) at <http://preview.jsdare.com>

jsdare

Make your own **games** by learning Javascript programming!



Anyone can learn how to write code like this.

GETTING STARTED

You learn programming by completing **dares**. These are short puzzles in which you have to copy the example, in as few lines of code as possible. They start simple, and become more difficult as you progress.

For now we only provide a number of **examples**. In the future we will provide some collections of dares to start with, and you will also be able to make and share your own dares. You can also play around in the **full editor**.

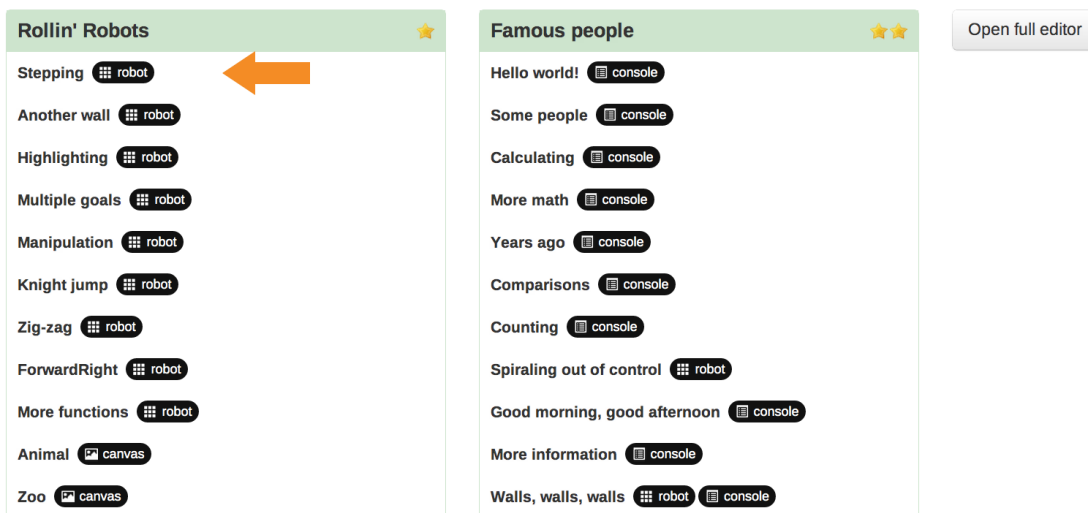


Figure 3.1: Current homepage. The orange arrow is animated, and shows where to get started.

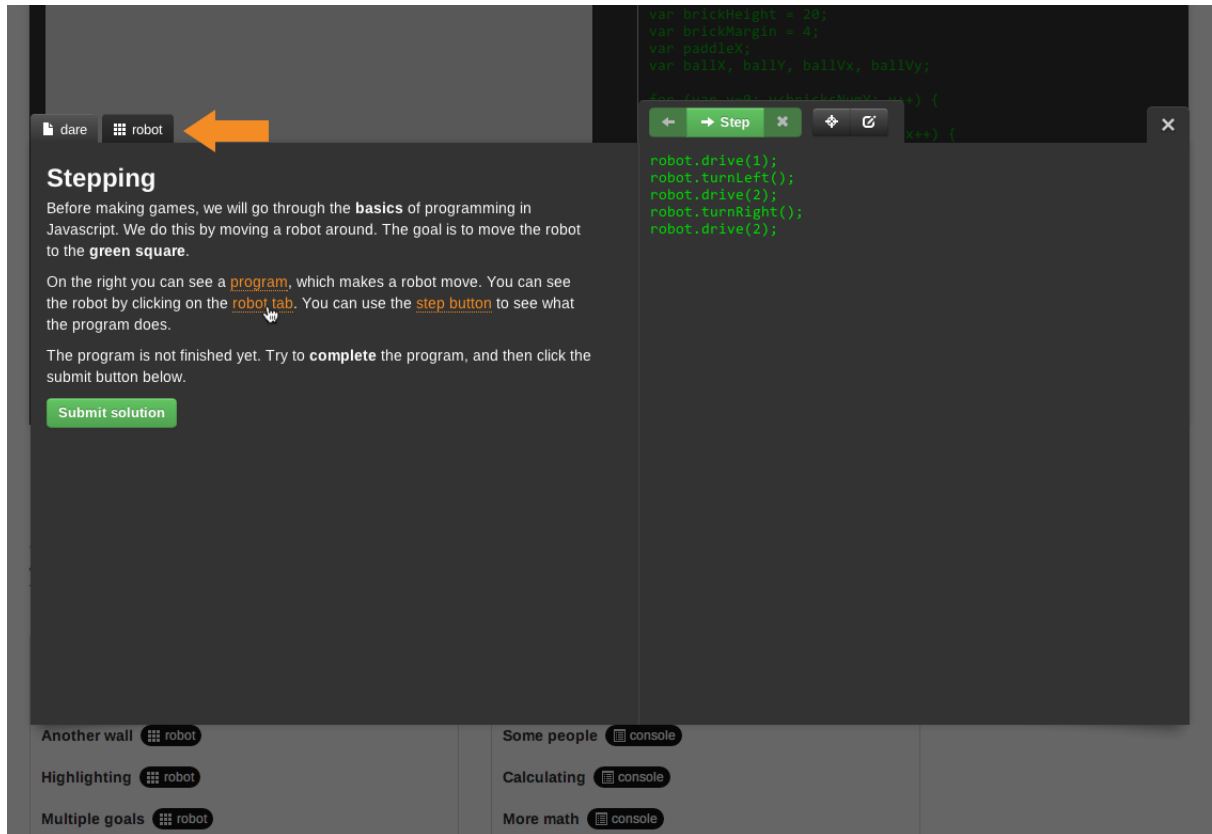


Figure 3.2: Opening the first dare. When hovering over orange text, an arrow is shown.

3.1 Basic editor features

The basis of the programming interface is the *code editor*, shown on the right (Figure 3.3). At every keystroke the program is parsed and run again: the *zero-response-time compiler* [15]. On the left the output is shown, if one of the three output tabs is selected (Section 3.9). This creates an immediate connection between the code and the output. However, it also creates an expectation of responsiveness, which is why we limit the execution time. Strictly speaking the program is only run again when the syntax tree changes, so this also makes the interface a bit more responsive. We look at the implementation details of this in next chapter.

When the program becomes longer, a vertical scrollbar is shown. A horizontal scrollbar is rarely shown, because we do not allow programs to exceed a certain width, which is less than the width that is visible in the editor. This way we enforce programs to be more readable, and in the case of dares, to not allow cheating by putting a lot of code on one line.

3.2 Language design

An important design decision is to use our own language specification, which we call *js--*. This language is a subset of the Javascript language, a powerful and modern language. Despite this, Javascript has a lot of *bad parts* too, as detailed in the book *Javascript: the Good Parts* by Crockford [20]. It has many pitfalls and bad design decisions, which makes it quite infamous. When avoiding these pitfalls, it is quite a good language, so we have removed most number of these difficulties in our subset.

For example, we have removed the feature of creating objects, since the syntax of inheritance is confusing, and the prototypical model itself is different from most other languages [20]. It is possible in our subset to use predefined objects, such as `console` and `canvas`. This can be considered a compromise between the imperative and real-



Figure 3.3: Overview of the editing interface.

world language goals, as it is practically impossible to use real-world APIs without using objects. Also, strings and arrays are in fact objects as well, and have methods such as `"Hello World!".substring(0, 10)` and `array.length`. These methods can still be used, but no methods can be added or altered. Usage of arrays is heavily regulated; only integer indices are allowed, in order to keep the language more in line with other programming languages.

Removing syntax for objects is one change from the original Javascript syntax, but there are several other differences. First of all, statements have to be separated by newlines, which is simply to enforce good programming practice. It also enforces a minimum length for a program, which positively influences dares (in which the length of the program is limited; Section 3.10). Simple statements, such as assignments and function calls, have to be ended by a semicolon. In Javascript itself semicolons can be left out, but this is considered very bad practice, because of the problems associated with so-called *automatic semicolon insertion* [20]. Also enforced is the style of having to put an opening bracket on the same line as, for example, an `if` statement. In many languages this is considered a matter of taste, although this can result in heated discussions. In Javascript, however, this is the only right way, as placing these characters on their own lines can actually in some cases lead to obscure errors, again due to the automatic semicolon insertion [20]. Therefore, this style is enforced in the grammar.

On the right hand side of assignments, within loop conditions, and within function calls, there are *expressions*, which return a value when evaluated. Expressions are largely unchanged compared to Javascript, except for the removal of some symbols, such as bitwise operators. It is also not possible to use newlines inside a statement, which means that every statement is restricted to exactly one line. More significantly, anonymous functions cannot be declared inside an expression, as we consider them to be too complex. This means that all functions are global, which has some benefits for other features, such as abstracting over time (Section 3.7).

Finally, a lot of other small things are removed: single quotes for strings (only double quotes are allowed, like in Java), hexadecimal and octal syntax for numbers (otherwise `011` would equal 9, which is confusing), `switch`-, `try-catch`-, and `with`-statements, ternary `if (a ? b : c)`, triple (in)equalities (`===`), inline regular expressions, some prefix operators (`++`, `--`, `~`), and postfix operators in expressions (but still allowed as separate statements).

3.3 Error messages

We do not use a console to show error messages in. Because there are no messages which refer to certain lines in the code, line numbers are omitted, reducing the visual clutter. Instead, all messages, such as errors, are shown inside the code editor, using both icons and text (Figure 3.4).

In the editor, there is a line on the left side, indicating the editing area. Errors are displayed by placing an exclamation mark icon left of this line, next to the line where the error occurred. Only when clicking this icon is the actual error shown using a text balloon below the code where there error occurred. Code can also be highlighted, if the information of where the error occurs exactly is available. The error message is hidden by default because while changing the program, it is parsed at every keystroke, thus generating many errors in intermediate states. Therefore it is useful if users can choose to display the error message only when necessary.

There are two types of errors: errors during parsing (such as a missing semicolon), called *syntactical* or *critical* errors; and errors during running (such as division by zero), called *runtime* errors. Both types of errors are shown identically inside the code, but the interface behaves slightly different for both types. When a syntactical error is encountered, the output on the left still shows the output of the last run (if any), even though the current program is not valid any more. Again, this is because the program is reparsed at every keystroke, and while editing the code it is useful to still see the previous output for reference. When a runtime error is encountered the output is shown until the point of the error, as expected. In both cases the brightness of the output is slightly reduced to indicate the presence of an error.

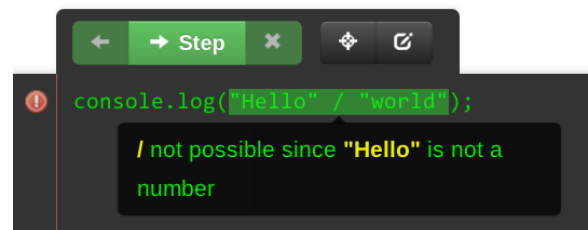


Figure 3.4: Error icons are displayed in the left margin; a helpful message is shown when clicked

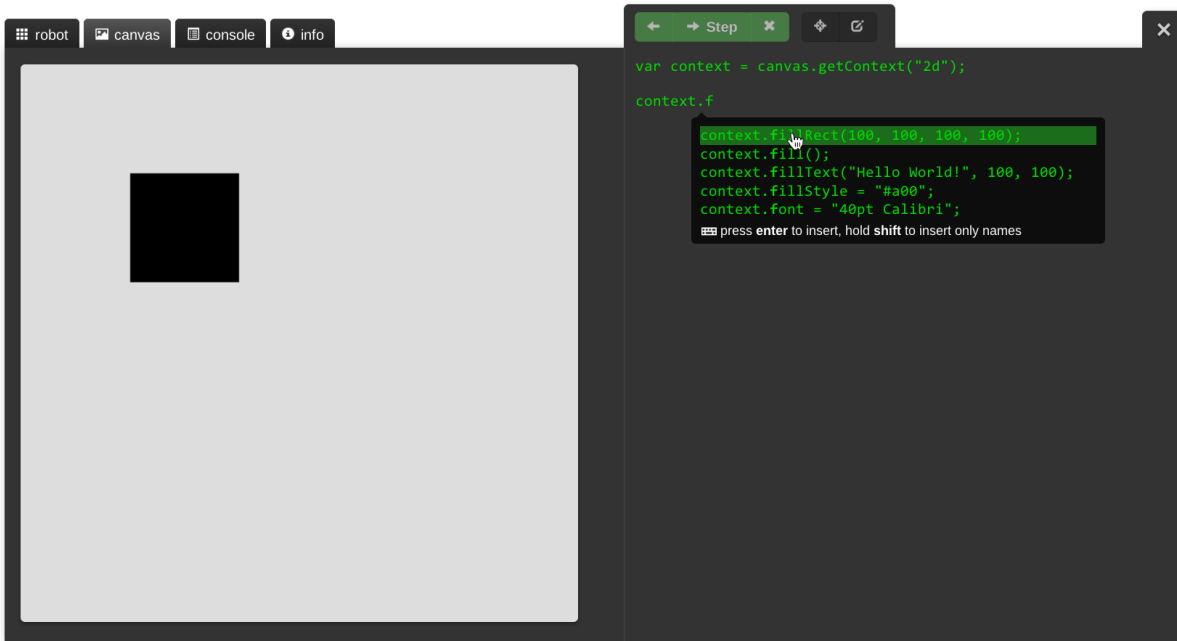


Figure 3.5: When using auto-completion, the statement is previewed in the respective output tab.

Defining our own language gives us full control of the execution. This means that we can also restrict the type system. Javascript is weakly typed, which means that it supports implicit type conversions. No types are specified beforehand for variables, but values themselves do have types. However, this type system sometimes leads to inconsistent and confusing behaviour [20]. For example, `5 * "5"` equals 25, but `5 + "5"` equals "55", as `+` is also the string concatenation operator. During runtime this is restricted by using our own functions instead of using the actual operators themselves, so we can enforce only numbers are used when using numerical operators. Furthermore, the numerical constants `Infinity` and `NaN` are also disallowed, as they are not useful in practice, and may be confusing (e.g. `Infinity - Infinity` equals `NaN` and `NaN == NaN` equals `false`). The operators `&&`, `||` and `!` are limited to booleans, as they can be used for other types as well in Javascript, unlike most other languages (e.g. `"" || 5` equals 5, `"a" || 5` equals "a"). Also, it is not possible to call a function without specifying all the arguments, as is allowed in regular Javascript. Whenever such a restriction is violated, a runtime error is thrown. Native Javascript errors are also caught and rethrown, so that we can change the error messages into something more informative.

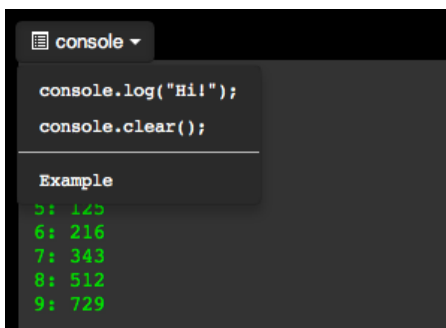


Figure 3.6: In an earlier prototype statements could be inserted from the tab instead of using auto-completion, but we decided to remove this method.

3.4 Auto-completion

Another basic feature is auto-completion. All global objects provide *examples* with each method and attribute. This example is used for auto-completion, so that when `console.` is entered, a box appears with an example for each of the methods of the global `console` object, such as `console.log("Hello World!")`. While hovering over these examples with the mouse, live previewing is used, as shown by Victor (Figure 3.5) [87]. The code is executed as if that example was actually inserted at that point, so that the effect of each method is immediately visible.

Before using this box that appears in the code, we experimented with different methods for auto-completion. For example, we tried integrating auto-completion inside the tabs that are used for selecting an

output (Figure 3.6). This has the advantage of always showing the console tab when previewing console methods, and the close coupling between the output and its methods might help in understanding how objects work. The disadvantage is that it is unclear where the code will be inserted exactly, whereas with the preview box inside the code, users explicitly insert the code at a certain point.

3.5 Stepping

We think that a step-through debugger can help users to understand the notational machine (Section 2.2.5). Most programming languages provide some sort of step feature as part of a debugger, either inside an IDE or as a separate program. However, it is usually one of many features, and may thus be easily overlooked by novice users. Also, debuggers often provide different step functions, such as *step into*, *step over*, and *step out*, which may confuse novice users.

In our interface the step feature takes a central role. Right above the editor there are three buttons with icons, and the middle button reads 'Step', and is the only clickable button. When clicking this button, the interface enters *stepping mode*, starting at the first statement. A message is shown at the current statement, indicating what is happening at that step. Examples are `a = 0`, declaring `factorial(n)`, and calling `console.log("Hello World")`. Not only statements are shown during stepping, but also intermediate expressions (Figure 3.7). This way, not only global structures are made explicit, such as loops, conditions, and functions, but also these local expressions. When in the current step a function call to an output object is made, such as `console.log`, then the corresponding element in the output is highlighted green.

Besides the step button there are two other buttons. On the right is the *close* button, which simply takes the interface out of stepping mode. On the left is the *step back* button, which takes the program to the previous execution step, as seen in the Omniscient Debugger [55]. This feature seems obvious, but is implemented in only few real-world programming interfaces. One reason may be that it is only possible to do this when the interface has a lot of control over the output, as this has to be put in a previous state as well. We retain full control over the output, so this is no problem. Our approach is to log the entire history of the program. This is usually expensive, but in our case the execution time is limited (Section 3.1), making it a reasonable approach.

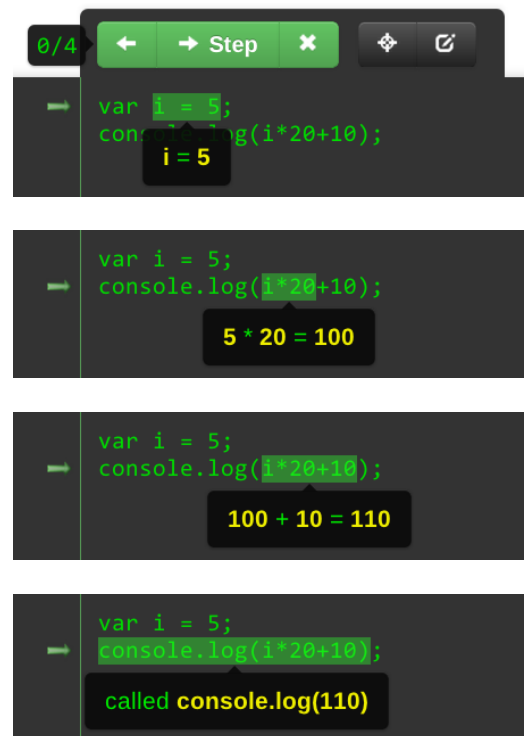


Figure 3.7: Intermediate expressions are also shown during stepping.

We deliberately left out advanced functions such as *step over* and *step out*. These are normally used for navigating the program while stepping, where the former lets users skip over a function call, and the latter makes the program execute until the function has returned. However, we provide another way to navigate the program. Next to the step buttons, a balloon is shown that indicates the current step, and the total number of steps. With the manipulation feature (which we look at in the next section), the current step can be easily changed, thus allowing users to navigate quickly to the desired point. An advantage is that the user does not need to understand new buttons such as *step over* and *step out*. Instead the more general manipulation feature is used for stepping as well. Another advantage is that this method is very visual; users can move back and forth while the output changes immediately, and the current step is directly highlighted in the editor. This is very different from the traditional method where they have to decide at every function call whether or not the function is worth stepping into or not, especially if this decision cannot be easily reversed by using a *step back* button.

Note that it is also still possible to change the program while stepping. This allows for immediately fixing problems while debugging, without having to restart the debugging process. Internally a new list of steps is generated when

the code changes, and the same step number as before is used. This may lead to changes in the step that is shown, such as when adding statements that are executed before the step currently shown. This is also why we show the current step number, as this explicitly shows that the step number stays the same when changing the code. If we would have used, for example, a slider to display and select the current step number, then this underlying mechanism would be less clear.

3.6 Highlighting and manipulation

The next features are slightly more advanced, and are especially useful when writing more complex programs. They are both presented in *Inventing on Principle* [87], although others have presented similar features (Section 2.2.5) [55, 91, 60].

The first is the highlighting feature. When clicking the highlighting button or holding a keyboard shortcut, users can move the mouse over lines of code, and corresponding elements are then highlighted in the output. For example, when they move over a `console.log` statement, the corresponding line is highlighted in the console, and when moving over a canvas drawing command, the corresponding shape is highlighted there (Figure 3.8a). It also works vice versa: when moving over an element in the output, the corresponding statement in the code is highlighted. When the highlighted statement is not visible on the screen, a scroll animation is used to make the statement visible. It is important that this is an *animation*, as this makes sure users do not lose track of what happens with the interface, but see the *transition* to the new state (Section 2.2.5).

This highlighting feature reinforces the connection between the code and the output, by making a visual connection. It may also help more advanced users to navigate their program, by being able to jump to the part of the program they want to edit simply by pointing at the corresponding element in the output. It is also possible to highlight higher-level structures, such as loops and functions. In that case, their entire contents are highlighted in the output.

The second feature is direct manipulation of code. When clicking the manipulation button or holding a keyboard shortcut, all the pieces of code that can be manipulated using the mouse are highlighted. For example, the boolean constants `true` and `false` are highlighted, and when clicked are toggled, and of course the result is visible immediately in the output. Strings that contain a CSS colour value are also highlighted [21], and when clicked a colour picker is shown, which allows users to edit the value, again with real-time updates of the output (Figure 3.9).

Numbers are also highlighted, and can be dragged to change their value (Figure 3.10). This is very powerful, as it shows in real time the *meaning* of a certain number. For example, when manipulating a number in a loop condition, you can immediately see lines in a console appear, or objects on a canvas. When manipulating a certain argument of a canvas drawing function, it becomes clear in an instant that the arguments represents, for example, the width of a rectangle (Section 5.2.4).



Figure 3.9: When manipulation is enabled users can change CSS colour strings using a colour picker.

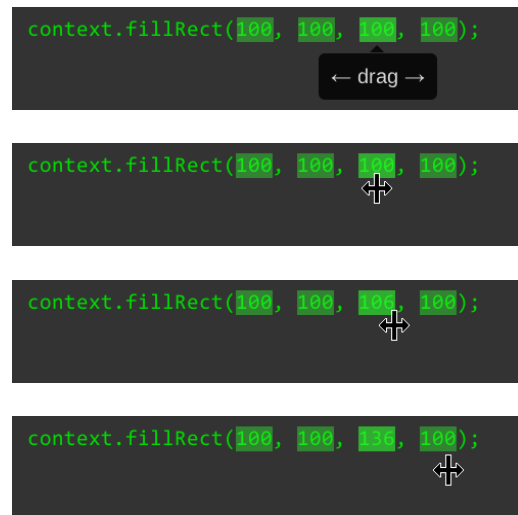
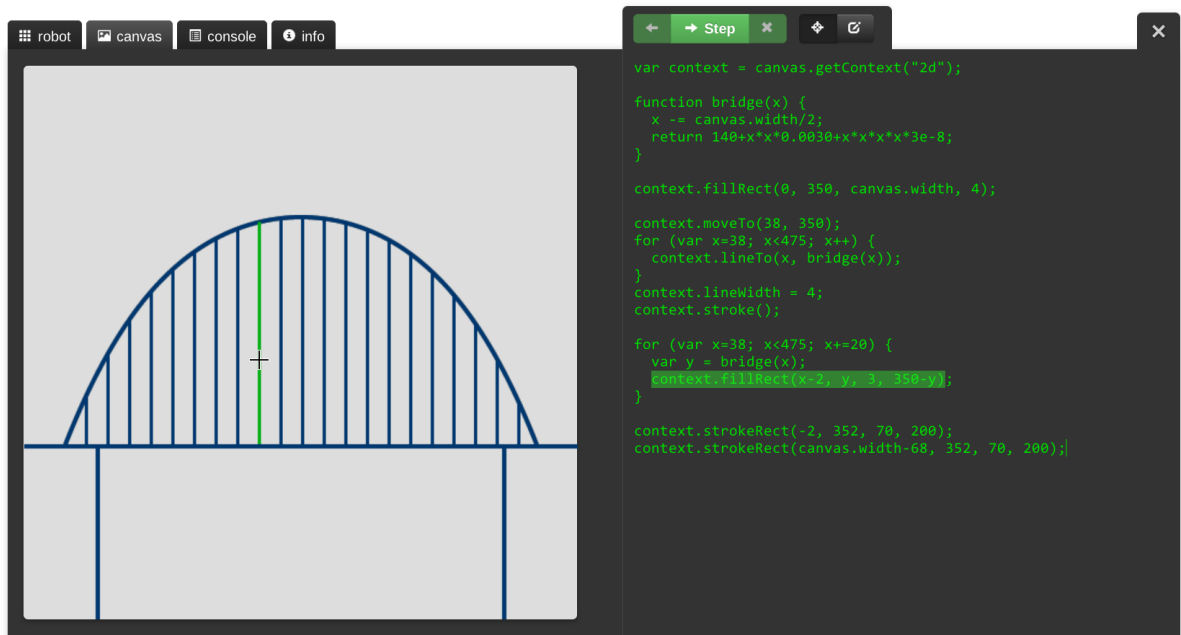
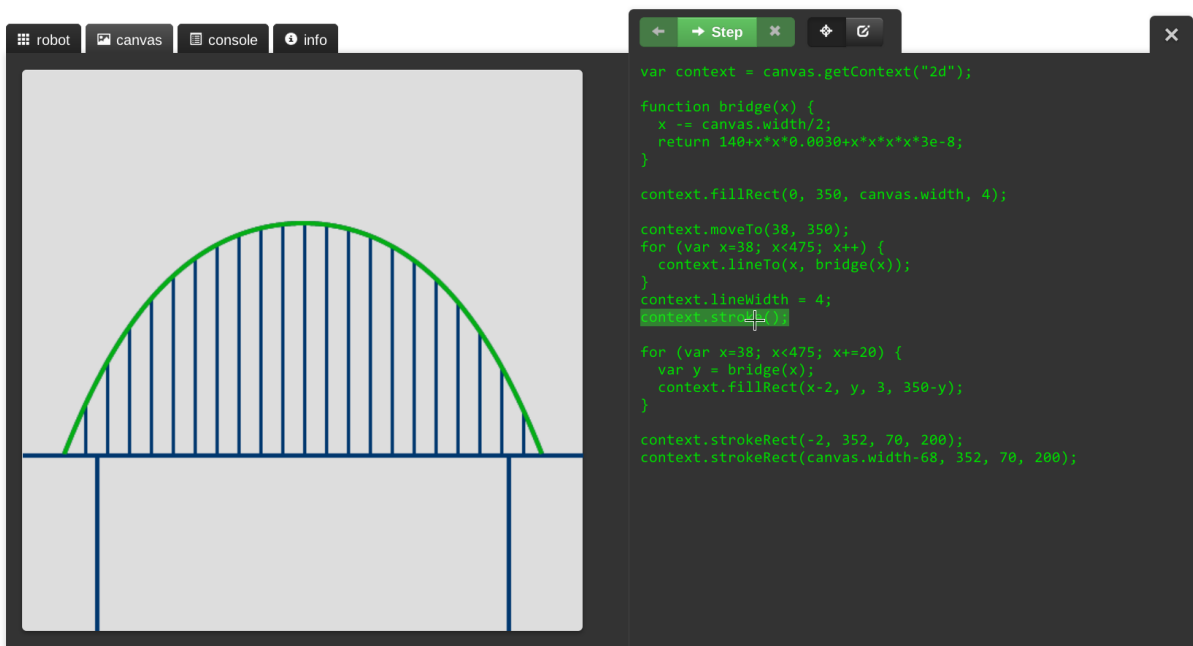


Figure 3.10: Manipulation can also be used to drag numbers to different values.



(a) Pointing at the canvas



(b) Pointing at the code

Figure 3.8: When highlighting a shape on the canvas, the corresponding line of code is also highlighted (a), and vice versa (b).

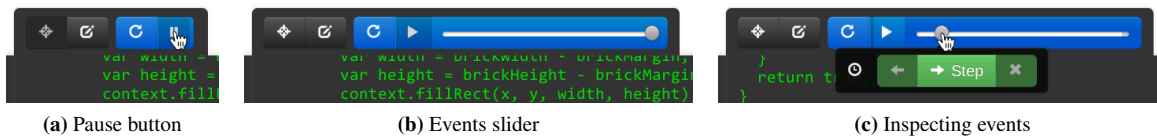


Figure 3.11: When events are used, two buttons appear, for restarting and pausing the program (3.11a). When clicked, a slider is shown (3.11b), which can be used to rewind time and step through specific events (3.11c).

3.7 Events

Being able to make small games is an important goal, as this can be very motivating. Therefore, it is paramount to be able to make programs *interactive*, by responding to a number of events. An event is an action generated outside of the program, such as when the mouse moves or keyboard is pressed, but events can also be generated by a timer that regularly issues an event. When an event is issued, the function that is associated with that event is called. That function gets passed one argument, the *event object*, containing information about the event, such as which key was pressed.

Mouse events are fired whenever the mouse is moved over one of the output tabs, or when a mouse key is pressed or released over an output tab. Keyboard events are fired when pressing or releasing keys, except while editing the code. Interval events are fired every number of milliseconds, as specified in the `window.setInterval` call. All calls to set function callbacks for events, are standard HTML DOM calls [44].

Whenever events are used in the program, two extra buttons appear in the toolbar (Figure 3.11a). The first one is to restart the program, which was pointless before since the output was always live, but with events it is useful. The second button is to pause the program. The step buttons disappear from the toolbar, since it is unclear which event would be used when clicking the step button. Instead, when the program is paused, a slider is shown, which can be used to move to a specific event in the history of the program (Figure 3.11b). When moving to a certain event, a balloon is shown with step buttons, allowing users to step through that event (Figure 3.11c). An icon is displayed in addition to the highlighting in the editor to indicate what kind of event was fired.



Figure 3.12: When going back in time, the dark overlay on the play button also moves back. When resuming from a certain position, the overlay also starts running from that position again.

While hovering over the slider, the knob moves to the mouse position, but when leaving the slider it returns to its original position. This allows for quickly scrubbing through the history of the program. When clicking, the knob jumps up and down, indicating that position has been fixed. This behaviour has been taken from another prototype by Victor [86]. Also, while scrubbing through history, the dark overlay on the play/pause button is moved as well, further reinforcing that moving the knob means moving back and forth in time. At the same time the function corresponding to that event is highlighted in the code editor. When clicking on the play button again, the program continues from the last selected position in history, and the dark overlay is animated from left to right, starting from the position corresponding to the selected event (Figure 3.12). This animation

from left to right while events are active, is to help users recognise that these buttons have to do with time, as in the literature it is shown that animations work well for time-based interactions (Section 2.2.5).

When changing the code, things get a little tricky. There are a couple of issues to take into consideration. First of all, we want to provide a direct connection between the code and the result, which means that we cannot restart the program whenever the code changes. After all, this would mean that if users change something that only has an effect after a number of events, then they would have to repeat those events, before being able to see the result. Therefore we discard all previous events, and only change the function definitions, instead of restarting the entire program. This way users can immediately see what the result is with those new function definitions. For example, in a platform game, a user can change the speed with which some enemy runs around. If the enemy is seen only

after half a minute, the user would not be able to see the changes until encountering this enemy again, in the case of restarting the program. However, if we only change the function definitions, and use these for the next events, then in the next event the speed of the enemy will be different, which would be immediately visible.

Another important consideration is that events depend on each other. This is a problem with the approach of changing only the function definitions, since this would make it possible to reach states in the program that cannot be reached when restarting the program. For example, in a platform game, one could lower gravity in order to reach a higher platform, and then increase gravity again. In the new program, with increased gravity, it would never be possible to reach that platform, but since the program has not been restarted, it is in a state in which the character is in fact at that higher platform. We do not do anything about this, since we believe that the usability is in this case more important than the problem of creating ‘impossible’ situations. A related problem is that when changing not the code inside a function, but ‘base code’, that nothing happens. This might be confusing for users, so in this case we flash the restart button, to indicate it should be clicked to reflect the changes (Figure 3.13). This is also done when editing a function that is called from base code, for the same reason.



Figure 3.13: The restart button flashes to indicate changes are not yet visible.

Yet another problem is that when changing the program and updating the function definitions, the old events in the history do not correspond to the new program any more. They cannot be used for stepping through the code, since the steps that are stored for them, may not even exist, if the user has deleted those lines. This is solved by removing all old, incompatible events from the history.

A different way to edit the code is to pause the time, and then edit the code. This causes the entire history to be replayed with the new code, and the changes are visible immediately. Because a potentially large number of events are executed at once, it is more performance intensive. Therefore not all history is recorded, but only a certain number of events. Again only function definitions are updated, because otherwise the program would have to restart again. The only exception is when no events have been discarded yet, and thus the initialisation of the program (the *base event*) is still in the history.

If the user changes the code when the program is paused, we do not want to change the events that have been fired, as this gets complicated. An example is when removing a line in which an event handler is set, which would cause all of those events in the history to become invalid. If only such events are in the history, the entire history would disappear immediately, which could be rather confusing for the user. Another problem arises when adding new event handlers in an existing event handler. In that case, additional events will have to be generated, but the history is only finite. What would happen in this case? One option would be to discard either old or new events, but this can be unwanted if the user was inspecting such an event that is discarded. The safest option seems to be to restrict when event handlers can be changed. In the current implementation users can only attach event handlers in initialisation of the program, thus minimising confusing behaviour. No functionality is compromised here, as program logic can be used instead of modifying event handlers, such as variables that indicate which event handlers should be active. Still, there is one case in which the problem still persists, which is when the history still contains the base event. The safest option seems to be to just keep the events as they are, as we already noted that invalid states may occur in other cases too. Like in other cases, we simply flash the restart button.

While running an interactive program, manipulation is enabled, but highlighting is not. It is only possible to use the highlighting feature when pausing. As said, the selected event is highlighted in blue in the editor, but also in the output. All highlighting occurs only within the selected event, so when one event is selected it is not possible to highlight elements in the output that were inserted in another event.

An additional feature, also suggested by Victor, is abstracting over time [87]. In his presentation this is only possible for predefined elements, but we present a method to do this for an arbitrary piece of code. When pausing an interactive program, and using the highlighting feature, lines appear next to each function. When hovering over such a line, the entire function is highlighted, not only for the selected event, but for all events in the history (Figure 3.14). When clicking on the line, this highlighting is preserved, allowing to edit this *trail in time* by changing the program.

We experimented with the way pieces of code can be selected for abstracting over time. First, we tried to let users make an arbitrary selection, but this has problems when updating this selection. You have to make decisions when to include new code into the selection, and when some code leaves the selection. For example, if a selected piece

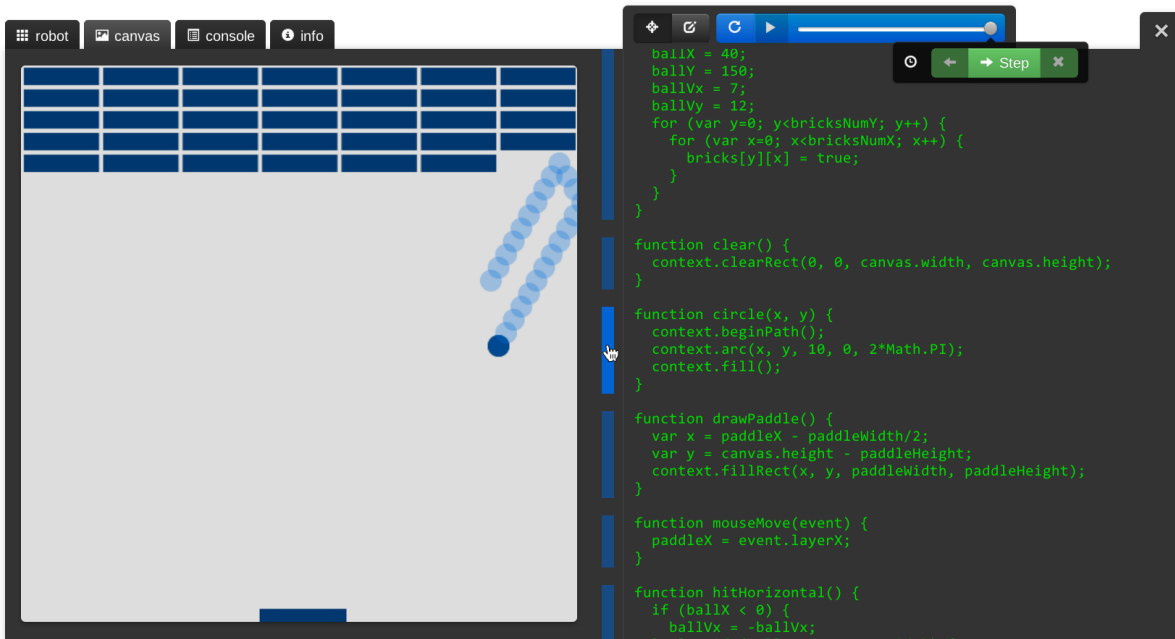


Figure 3.14: Abstracting over time works by selecting a line next to a function, and then that function is highlighted for every event in the history. Note that other elements on the canvas are also blue, since in highlighting mode the selected event is always highlighted in blue, and this program repaints the entire canvas on every tick.

of code is cut and pasted, should it still be selected? These decisions are *implicit*, hidden from the user. By only allowing entire functions to be selected, it is shown explicitly that no matter how you change that function, it will always be selected, until clicking on the line again. When moving a function, we simply find it again by the function name, which is unique.

3.8 Information tab

One of the tabs on the left side of the screen is the *info tab*, which shows additional information that is useful for learning programming (Figure 3.15). On the top of the info tab is a visualisation of the current scopes. There is always a global scope, and there may be multiple local scopes of variables declared inside functions. Of these local scopes only the latest scope is active, and this one is shown on top. The inactive scopes are depicted by a slightly darker colour. For the active scope all the variables and their values are shown. For the inactive scopes the list of variables are initially hidden, but can be revealed by clicking on the function name.

Of course, the list of scopes is updated accordingly when stepping through the code. If the current step alters a variable, it is highlighted in the corresponding scope. Also, when highlighting lines of code that change a variable, this variable is highlighted in the scope list as well. Vice versa, when highlighting a variable in the scope, all the statements in the code that change that variable are highlighted.

Below the scopes is a list of commands. These are all functions and attributes that correspond to a certain output, such as `console.log`, plus language features such as `for` and `+=`. When clicking a command a brief description is shown, including some examples. Stepping and highlighting are supported just as with variables in the scope list: the commands corresponding to the current step are coloured green, highlighting a line of code shows corresponding commands, and highlighting a command shows the statements and expressions in the code where that command is used.

robot canvas console **Info**

scope

This list shows the variables that are declared in your **program**, along with their values. At the beginning the only variables are those that we provide, such as **robot** or **canvas**. You can add your own variables and functions using **Var** and **function**.

```

global:
robot = [object robot]
canvas = [object canvas]
console = [object console]
document = [object document]
window = [object window]
Math = [object Math]

```

javascript

Below you find the basic constructs of the **JavaScript** language. Programs are executed from **top to bottom**, one statement after another. Use the **step button** to see in detail how your program is executed.

- ▶ **number**
- ▶ **string**
- ▶ **boolean**
- ▶ **var** (declaration)
- ▶ **=** (assignment)

(a) Scope information and Javascript reference

robot canvas console **Info**

canvas

The canvas is used to draw shapes on, and is the actual **HTML** element that is supported by most web browsers. This means that any program you write for this canvas can also be used **outside** of this environment, on any other site. It is also very suitable for programming **games**, by using events.

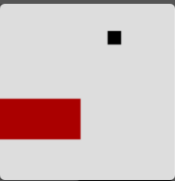
- ▶ **canvas.getContext("2d")**
- ▶ **canvas.width**
- ▶ **canvas.height**
- ▶ **context.fillRect(x, y, width, height)**

context.fillRect draws a filled rectangle on the canvas. The color set in **context.fillStyle** is used, by default this is black.

```

var context = canvas.getContext("2d");
context.fillRect(20, 40, 10, 10);
context.fillStyle="#a00";
context.fillRect(70, 70, 30, 60);

```



- ▶ **context.strokeRect(x, y, width, height)**
- ▶ **context.clearRect(x, y, width, height)**
- ▶ **context.fillText(text, x, y)**

(b) Canvas reference

Figure 3.15: The info tab has scope information (a) and a command reference. By scrolling down more commands are shown, such as canvas commands (b).

3.9 Output tabs

Currently there are three actual output tabs implemented, with the idea that such an output should be same as when programming in any other Javascript environment (Figure 3.16). The first two outputs, *console* and *canvas*, stick to this idea: both are commonly used in Javascript [44, 65]. The third one, *robot*, is specifically designed to make the learning process in the beginning easier, but breaks with our real-world requirement as it has a custom API.

The console tab reflects browser debugging consoles, available in all modern browsers. Those browser consoles are one-way only: there is no way to ask for user input through them, only to write strings to the console. This is different from many other languages which provide a synchronous input function which blocks the program until the input is given by the user. Javascript does not work this way; most functions are asynchronous and work with callbacks. On the one hand this makes it more difficult for novice users to write simple interactive programs, but on the other hand this works well with the interface model of instantly recompiling and running.

There is only one important console command, `console.log`, which prints a string to the console. This command is supported by all the browsers and other Javascript environments. Besides this, we implemented some other commands such as to clear the console or to set the colour of the text. These are just there for convenience and fun, and are not typically supported in real-life. However, a program would still work the same way without these commands, as they are only an addition to the `console.log` command, but do not enforce a different program architecture such as when using synchronous input commands.

The canvas tab uses an HTML5 `<canvas>` element, and uses the standard canvas API [45]. This makes it possible to draw simple shapes like rectangles, but also allows advanced users to draw complex curves and apply matrix transformations. Some methods are left out, however, most notably the `drawImage` command. The reason for this is that adding images would first require to be able to upload images to the server in order to be able to use them, which has performance and security implications, but more importantly it would require even more interface tools. Even when image uploads would be possible, using images in Javascript is relatively difficult, as it uses asynchronous callbacks before you can actually use the image. All in all, it seems too much trouble for an arguably fun but non-essential feature, so we decided to remove it altogether. In the future it might be possible to add some non-standard commands for adding images, and perhaps even an image editor as seen in Scratch [60] (Section 5.2.4).

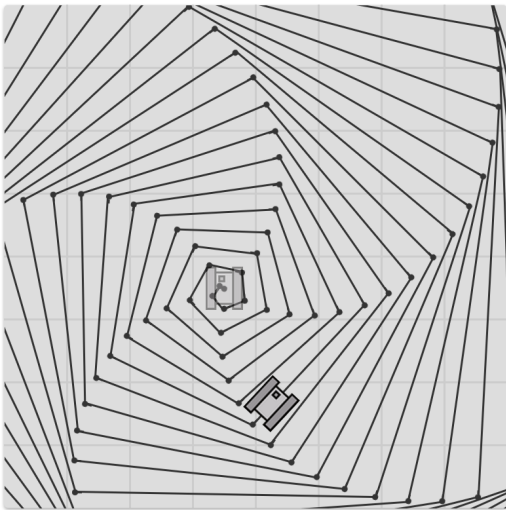


Figure 3.17: Without walls the robot can move around freely, like a LOGO turtle.

more predictable behaviour when programming, for example, a maze solving program. Besides adding walls, users can also click in the middle of a grid square, creating a *goal square* which is indicated by turning the square green. Goal squares can also be detected using a function, allowing for actual mazes to be created, where the robot can navigate the maze until a goal square has been found. Finally, the initial position of the robot is shown

Finally there is the robot tab, designed specifically for the purpose of teaching simple programming concepts. It is quite similar to the LOGO turtle, in the sense that there is a robot object that can move forward and backward, and turn [69]. Additionally, users can place walls in the space the robot moves in, and the robot has a function which detects these walls, similarly to Karel the Robot [70].

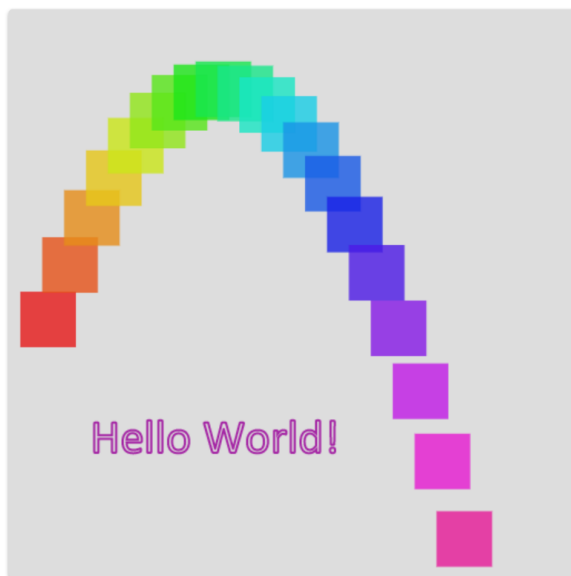
The robot tab can be used in two ways. Initially, there are no walls placed, which means that the robot can move around freely. In this mode the robot truly behaves as a LOGO turtle: it can move forward and backward for arbitrary distances, also fractional ones, and it can turn left and right with arbitrary angles. The path that the robot takes is drawn with lines, which enables users to create pretty patterns, just as with LOGO (Figure 3.17). It is possible to highlight lines on the path and see which commands are responsible for them, and vice versa.

Once the user starts adding walls, the driving behaviour is severely restricted (Figure 3.16c). The robot can only move integer distances, and turn right angles, forcing the robot to stay inside the grid, as with Karel the Robot. This gives the user

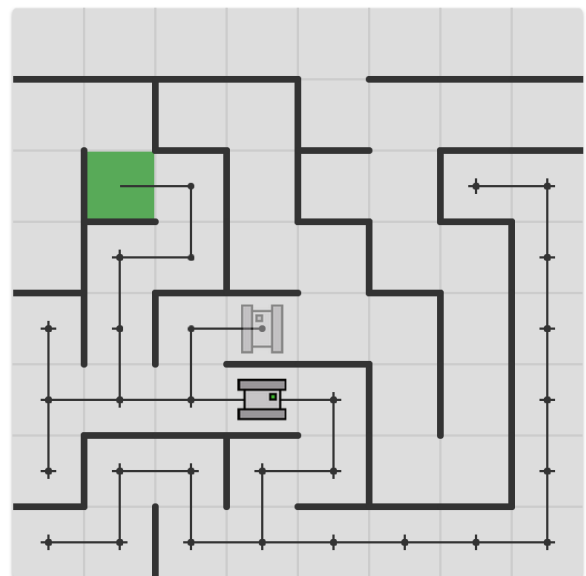
A colourful multiplication table:

1	2	3	4	5	6	7	8
2	4	6	8	10	12	14	16
3	6	9	12	15	18	21	24
4	8	12	16	20	24	28	32
5	10	15	20	25	30	35	40
6	12	18	24	30	36	42	48
7	14	21	28	35	42	49	56
8	16	24	32	40	48	56	64
9	18	27	36	45	54	63	72
10	20	30	40	50	60	70	80
11	22	33	44	55	66	77	88
12	24	36	48	60	72	84	96
13	26	39	52	65	78	91	104
14	28	42	56	70	84	98	112
15	30	45	60	75	90	105	120
16	32	48	64	80	96	112	128
17	34	51	68	85	102	119	136
18	36	54	72	90	108	126	144
19	38	57	76	95	114	133	152
20	40	60	80	100	120	140	160

(a) Console tab



(b) Canvas tab



(c) Robot tab

Figure 3.16: There are three output tabs: *console* (a) and *canvas* (b), which have the same APIs as the native browser versions; and *robot* (c), modelled after the LOGO turtle and Karel the Robot.

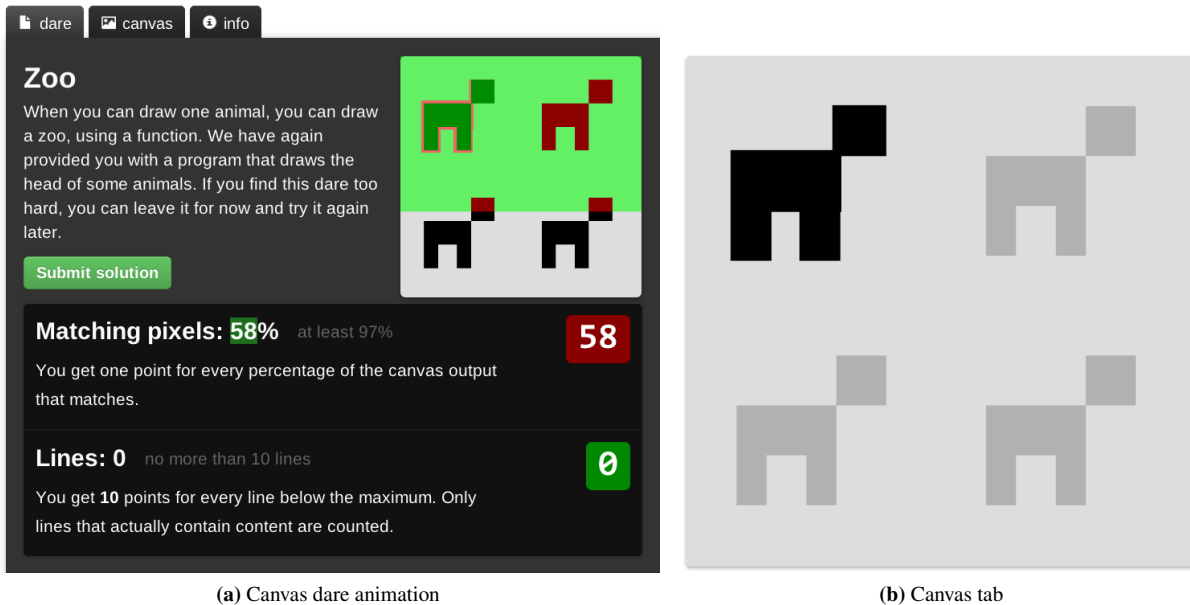


Figure 3.18: An animation is shown when submitting to indicate how the points are awarded (a). For reference the example is shown in a dimmer colour in the canvas tab (b).

at the beginning of the path, and can be dragged by the user to a different square. Moving the initial position and altering walls or goal squares causes the program to be executed again immediately. Note that when doing dares the maze cannot be changed.

Every time the path the robot takes changes, a new animation is played, which shows the robot navigating the path. It is this animation that makes the robot tab more suitable to teaching the basics of programming, as it makes it look like a lot of progress is made even when learning the simple commands. It also shows exactly how the path comes about, as the animation is a strong visual metaphor. Furthermore, it can create insight into how the program the user has written actually works. For example, a red or green light is shown on the robot whenever it checks whether it is standing in front of a wall or not.

3.10 Dares

In order to do directed teaching, we have devised a method for making programming exercises, that works well in conjunction with the output tabs, and `js---`. The idea is to limit the length of a program, similarly to Code Golf [1]. This way, users cannot write a long program that produces the requested output manually, but have to come up with solutions that include functions, loops, etc. We call these kinds of exercises *dares*. While there are of course also other kinds of exercises possible, we focus on dares, and have only implemented dares. In the future it might be possible to also include other methods (Section 5.2.3).

The dares we have implemented work similarly for the different output tabs. When doing a dare with the console, the objective is to simply copy the reference output. When submitting, the output of the submitted program is compared per character with the reference output. Canvas dares work in the same way; the reference image has to be matched per pixel. Usually it is not necessary to match it perfectly, but only by a certain percentage, which might differ per dare. Users get extra points for matching it better than the minimum. An animation is used to show how the scoring works, by overlaying on the example which pixels did or did not match, by using green and red colours (Figure 3.18a). Below the image there is a *scoring box*, showing a breakdown of the points. Note how the percentage is highlighted in green, just as with manipulation (Section 3.6). To prevent having to switch back and forth between the dare tab and the console or canvas tab, the reference text or image is shown in a dim colour inside the respective tab itself (Figure 3.18b).

Dares that use the robot output are slightly different. A maze is provided, which the user cannot change, and

the robot has to visit the goal squares in the maze (Figure 3.19). However, the robot does not have to follow the exact same path as the provided program, thus allowing for more creative solutions. Some goal squares may be optional, allowing the user to get extra points by visiting them. Again an animation is used, which highlights the goal squares that have been visited.

Points are awarded for each visited goal square, or each matching percentage, as long as those are above some minimum. The program cannot be longer than some maximum number of lines, and extra points are awarded for each line below that. Only lines that contain content are counted, so closing brackets, empty lines, and comments are discarded. An animation is used to briefly highlight every line that is counted, to give more insight in this calculation.

Now, in most programming languages it seems that cheating with this method is easy, just put the entire program on one line. This is where `js--` comes in, as it enforces newlines in the programming language. Every statement has to be on a newline, expressions cannot be split among lines, including strings and arrays. There is also a limit to the width of each line, to prevent making one long line with an array or string that contains all information necessary, and after that a small piece of code to execute the ‘program’ described in this array or string.

When designing a dare, one has to take care that the output cannot be generated by just using some long strings or arrays, even if they are within the width limit. With console dares, this usually means that the output should be relatively long, so that the overhead of writing a function or loop is outweighed by the number of individual log statements the program would otherwise need. With robot dares it just means a more complex maze, and with canvas dares more complex shapes. On the other hand, when designing a dare we should also take creativity into consideration. ‘Cheating’ is not always bad, and if a user finds some clever way to beating the system, then that should not necessarily be punished. Also, making the output too long or complex may put off users. The trick is to find a balance that makes obvious cheating impossible (otherwise users do not learn anything), but is still simple enough to be usable for novice users, and leaves room for some creativity.

An example is the robot dare in Figure 3.19. The maze has a certain pattern, so the dare can be solved by creating a function that takes as an argument the length it has to drive up and down. The path is complex enough that it takes a lot of lines of code if each command is typed in manually, but when using a function it is relatively easy to stay under the limit. Note that advanced users can write even shorter programs by using a loop and commands that check whether or not the robot is facing a wall. *Good for them!* We can only encourage users who learn more advanced programming to revisit the dares they have already solved, and try to make even better programs. This might give users a substantial boost in confidence and motivation.

In an earlier prototype we used a different representation for the amount of points. We showed a small Javascript program in which the amount of points was changed using an animation similar to manipulation (Figure 3.20). The idea was that by examining this small program, a user could get insight in how the points are awarded, and learn some programming at the same time. However, after some preliminary user testing we decided not to use this approach, as for novice users it was rather confusing. However, this or a similar approach could be used in other types of exercises, in which the calculation of points is more complicated (Section 5.2.3).

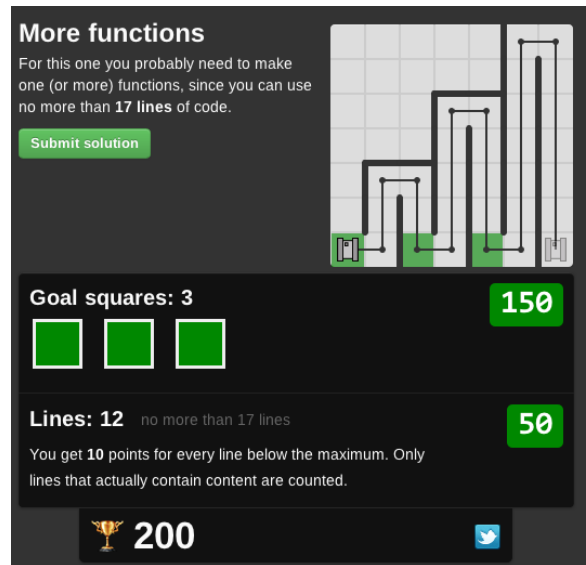


Figure 3.19: In a robot dare the robot has to visit goal squares. In this example, the pattern in the maze teaches functions with arguments.

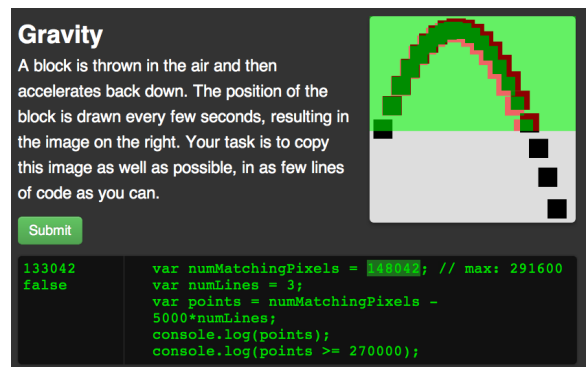


Figure 3.20: Previously we used a textual representation of points.

*'There is only a vision of how mankind should be,
and the relentless resolve to make it so. The rest is details.'*
— Bret Victor

Chapter 4

Implementation

In this chapter we look at the implementation details of the presented interface. All the software is implemented in Javascript, and the interface makes heavy use of the latest web standards, HTML5 and CSS3 [44, 21], which are supported by all modern browsers. For certain interface elements such as buttons and toolbars we use the *Twitter Bootstrap* library, and to abstract away from browser specific Document Object Model (DOM) implementations in Javascript, we use *jQuery* [83, 48]. We do not look at any server side details, as the current server implementation is very basic, and for a long time the software was developed even without a server.

The chapter is presented in four parts, which correspond to the four main *modules* in the software. First we look at the language implementation (Section 4.1), which contains the grammar, parser, and runner. A level above the language implementation is the editor, which includes the code editor, the toolbar, and a number of editor features (Section 4.2). Then there are the output tabs, all of which have some interesting implementation details (Section 4.3). Finally, we look at the highest level, which is the dares module (Section 4.4).

4.1 Language

The language module is responsible for parsing and running a program. For this two things are required: the actual code to compile and run, and a global scope which contains objects that the program can interact with. In this section we first look at the implementation of the *parser*, which converts the code into a *syntax tree*. The tree can then be compiled to *safe Javascript*, and be run using a *context* and *runner*. Finally, we look at some extra features built into the language module.

4.1.1 Parser implementation

The full grammar in Backus-Naur Form (BNF) can be found in Appendix A, but in this section we discuss some implementation details of the grammar. We use *Jison*, a Javascript port of the popular *Bison* parser library written in C++ [17, 30]. This tool generates Javascript lexers and parsers based on a grammar definition.

First of all, the lexer generated by Jison is used to tokenise the code, based on the primitives of the language, such as numbers, strings, operators, and keywords. Comments are immediately stripped from the input by the lexer. Newlines are not removed, as these are in fact used in the BNF grammar, since we restrict the usage of newlines as detailed in Section 3.2. After this, the parser is used on the tokens, generating a syntax tree.

When an error occurs during lexing or parsing, a syntax error is thrown. We do this by catching the errors thrown by Jison, and trying to make them more informative. First the lexer errors are handled, which unfortunately do not give much information, resulting in a plain '*Syntax error encountered*' message. This happens when the lexer cannot find any regular expression to match the upcoming text, for example when invalid characters are used. This is best circumvented by adding some dummy regular expressions to the lexer, in order to pass on the errors to the parser. For example, the single quotation mark can be added to the lexer, and when it is used we get a parser error

which tells us *which* token could not be matched, something the Jison lexer does not tell us. Then we can give an error suggesting, for example, to use double quotation marks instead.

Parser errors thrown by Jison contain a bit more information, such as the string near which the error occurs, the parsing tokens which were expected, and the last encountered token. This information is used to display a good error message. When there is only one token expected, this one is used in the error message, leading to messages such as *‘Invalid syntax encountered, perhaps some code near “console” should be put on a new line.’* when a NEWLINE token was expected. Then there are some heuristics based on the list of expected tokens and the current one, such as when the current one is NEWLINE, and the list contains `;`, then it is probably a forgotten semicolon. When the current one is FUNCTION or EOF, and `}` is in the list of expected tokens, then probably a closing block bracket was forgotten. The same works if `)` is in the list of expected tokens, and the current token is either `{`, `;`, or NEWLINE. The most common errors can thus be made much more informative.

```
function spiral(num) {
  for (var i=0; i<num; i++) {
    robot.drive(0.09*i);
    robot.turnLeft(73);
  }
}

spiral(40);
```

(a) Block location

```
Function spiral(num) {
  um; i++) {
    entering spiral(40) 9*i);
    robot.turnLeft(73);
  }
}

spiral(40);
```

(b) Line location

```
function spiral(num) {
  for (var i=0; i<num; i++) {
    robot.drive(0.09*i);
    robot.turnLeft(73);
  }
}

spiral(40);
```

(c) Text offsets, used for manipulation

Figure 4.1: Different types of line information are tracked for each node. For highlighting, the *block location* (a) is used; for stepping the *line location* (b); and for replacing values during manipulation (c) we use *text offsets*.

code — the initialisation code for interactive programs — that then a restart button in the toolbar starts flashing. In order to determine whether base code or only function definitions have changed, we can compare parts of the tree by serialising only these parts. For this serialisation we simply use compilation back to the original code in a normalised form, without whitespace and comments. By comparing these serialisations that only include base

4.1.2 Syntax tree

After successfully parsing a program, this generates a syntax tree. Every *node* in this tree has a unique *node ID*, which is generated by a counter that is incremented for each node. Each node also stores some line information (Figure 4.1). A multi-line range or *block location* is used for highlighting, so that for block statements the entire block is correctly highlighted. When stepping through the code, a special line representation is used, the *line location*. For example, when entering a function, not the entire function is highlighted, but only the name and arguments. Finally the exact start and end positions in the text are tracked (*text offsets*), which are used when changing the code dynamically, such as when using the manipulation feature.

The syntax tree also contains a number of lists of nodes, such as which nodes correspond to which lines in the code, which is used in the highlighting feature. It also stores a list for each different type of node, which is used to highlight certain literals when using the manipulation feature, and to call the right functions when using events.

Previously the syntax tree also allowed for some manipulations, such as inserting *hooks* before and after nodes. These hooks were functions that were executed before or after the node they were attached to was executed. This was used for highlighting, so that when highlighting a node, before and after that node functions were called which respectively enabled and disabled highlighting. However, this requires to run the program again whenever the highlighting changes. For highlighting we now use the lists of calls that all outputs store for the stepping feature anyway. Therefore we have removed the hooks from the syntax tree, as it required quite a bit of extra code in the code generation part.

To increase performance programs are only run when the semantics have not changed. For this, two syntax trees can be compared, and if they are the same then the semantics of the program have not changed.¹ In Section 3.7 we discussed that if a user changes the *base*

¹Note that this is only implication, not equivalence; if the syntax tree has changed, it is still possible that the semantics have not changed, but this is difficult to detect.

```

1 \begin{lstlisting}
2 function log(text) {
3   console.log("Log: " + text);
4 }
5 log("Hi!");

```

(a) Original code

```

1 function(jsmmContext) {
2   jsmmContext.increaseExecutionCounter(jsmmContext.tree.nodes[1], 3);
3
4   jsmmContext.tree.nodes[10].runFuncDecl(jsmmContext, "log", function(jsmmContext, args)
5     {
6       jsmmContext.tree.nodes[10].runFuncEnter(jsmmContext, args);
7
8       jsmmContext.increaseExecutionCounter(jsmmContext.tree.nodes[2], 2);
9
10      jsmmContext.tree.nodes[8].runFunc(jsmmContext, jsmmContext.tree.nodes[4].runFunc(
11        jsmmContext, jsmmContext.tree.nodes[3].runFunc(jsmmContext, "console"), "log"), [
12        jsmmContext.tree.nodes[7].runFunc(jsmmContext, jsmmContext.tree.nodes[5].runFunc(
13          jsmmContext, "Log: "), "+", jsmmContext.tree.nodes[6].runFunc(jsmmContext, "text")
14        ]]);
15
16      return jsmmContext.tree.nodes[10].runFuncLeave(jsmmContext);
17    });
18
19   jsmmContext.tree.nodes[13].runFunc(jsmmContext, jsmmContext.tree.nodes[11].runFunc(
20     jsmmContext, "log"), [jsmmContext.tree.nodes[12].runFunc(jsmmContext, "Hi!")]));
21 }

```

(b) Compiled code

Figure 4.2: For some simple input (a), the compiled safe code is quite complicated (a). Every operation is wrapped in a function that provides additional error checking, and records steps and calls.

code or function definitions, we can determine which parts of the tree have changed. One caveat is that in base code functions can also be called, and these have to be included when comparing base code, but we cannot statically infer which functions are being called. This information is therefore stored in the context object which we look at in Section 4.1.4, and used when comparing trees.

When using a new tree in the case that only a part of the program has changed, the node IDs of the other part of the tree have to be the same as before. The way we guarantee this is by using two different kinds of node IDs. In the base code the nodes are numbered `base-1`, `base-2`, etc. If the base code part of two trees is identical, then the same node IDs will be used for the nodes, since they are always numbered in the same way, deterministically. The nodes inside functions are numbered `functions-1`, `functions-2`, etc. Since the syntax tree can be replaced without running the program again, some bookkeeping is required; we must make sure that we only use node IDs when highlighting and in step messages, and never offsets or locations. Then the new tree can be used for updating the locations of elements in the editor, without requiring a rerun.

4.1.3 Compiling to safe Javascript

The syntax tree can be used to generate back the original code, be it in normalised form, but also to generate the *safe runtime*. For every Javascript operation a *wrapper function* is provided, which sanitizes the input and output, and restricts the operation if necessary.

Let us look at an example, detailed in Figure 4.2. First of all, the entire program is wrapped in a function that only takes a *context* object as an argument. This object contains everything needed for execution, such as a pointer to the syntax tree, a *scope* object for looking up variable names, a call stack for internal functions, and various other lists and counters. For each execution of the program a new context object has to be used, as it also stores information about the last run, such as a list of *step messages*.

On line 2 of Figure 4.2b, an execution counter is increased, always by the number of statements plus one, which is in this case 3. It is a simple measure to avoid infinite loops. The ‘plus one’ is to make sure empty loops are also counted. Much slower than simple statements such as variable assignment, are *external function calls*, to the console, canvas, etc. For each external function a *cost* attribute indicates the relative time it takes to process that call, and added to another counter every time such a call is made. These costs are precomputed, and the limit is also set beforehand. This is to ensure consistency when running programs: a program that runs on one computer should run exactly the same way on another, but the downside is that for fast machines the limit is too low. Finally, there is a third counter, that is increased when entering a function, and decreased when leaving one. This counter makes sure that there are not too many nested function calls, as this has negative performance implications.

On the other lines of Figure 4.2, we see the wrapper functions. Every function gets passed a reference to the context object, and return values from other functions. All values need to be passed by reference in order to be able to change variables, however, in Javascript primitive data types are passed by value by default. As there is no explicit syntax for pointers or passing by reference, we use a lot of small objects that contain the actual values, as objects are always passed by reference. This way we simulate pointers to primitive data types.

Also note that functions are defined as normal Javascript functions inside safe wrappers. Upon entering a function (line 5), a number of initialisations are done, such as creating a new scope containing the arguments passed into the function, and a pointer to the global scope. When leaving a function by using a return statement, or by getting to the end of the function (line 11), the local scope is closed again.

When using events, we only change the function definitions, so that the program does not have to be completely restarted (Section 3.7). For this, a different runtime is generated, which takes in a scope object, and returns a scope object with updated function definitions. The same generation code as for the safe runtime is used, with the exception that the main statement list skips over all statements except function definitions. The wrapper functions around the function definitions are also slightly different, as they should not throw an error when a function name already exists, as this will clearly be the case.

4.1.4 Context object

We already looked briefly at the *context* object in the previous section, in particular at some of the counters. The context object takes care of a number of other things as well. First of all, many wrapper functions add a message to the list of *step messages*. Such a message contains HTML text for displaying the step; a node ID for looking up the node in the syntax tree, in order to get the position; a string indicating what kind of position should be used (Section 4.1.2); and finally a number which indicates how many external calls have been made before that message, so that all later calls can be hidden when stepping through the program.

In order to keep track of variables, a special *scope* object is used. This object is used when looking up variable names, and declaring and assigning variables and functions. Furthermore, two lists of all the operations that have been executed are made, which contain external function calls, but also binary operators, variable assignments, and so on. These lists are maintained by an auxiliary object, the *command tracker*. The lists are used for the highlighting feature in the info tab, which highlights all the corresponding documentation when highlighting a line of code, and shows all corresponding lines of code when highlighting a piece of documentation. For the first direction, we have a list indexed by node ID, and for the other direction a list indexed by command ID. Similarly for the scope, all variable assignments, function calls, and function returns are logged by the *scope tracker* to be able to do scope highlighting on the info tab.

Highlighting of external function calls is done a bit differently. All outputs tabs have to store some information about each call, such as the step number associated with that call, which is used for only showing the calls up until the current step position. Because this is already stored at the output, we also store the node ID at the output, instead of keeping a list in the context. Whenever an element in one of the output tabs is highlighted, the corresponding node ID can be used directly to highlight the code in the editor, without enquiring the context object first.

A call always has only one corresponding node ID, but a node may have multiple corresponding calls. This is because we designed highlighting such that when highlighting an internal function call, all the external calls made from that function are also highlighted. Therefore we keep a list in the context of which node corresponds to which calls. Whenever an external function call is made, the call ID is put on the list of the node that contains that

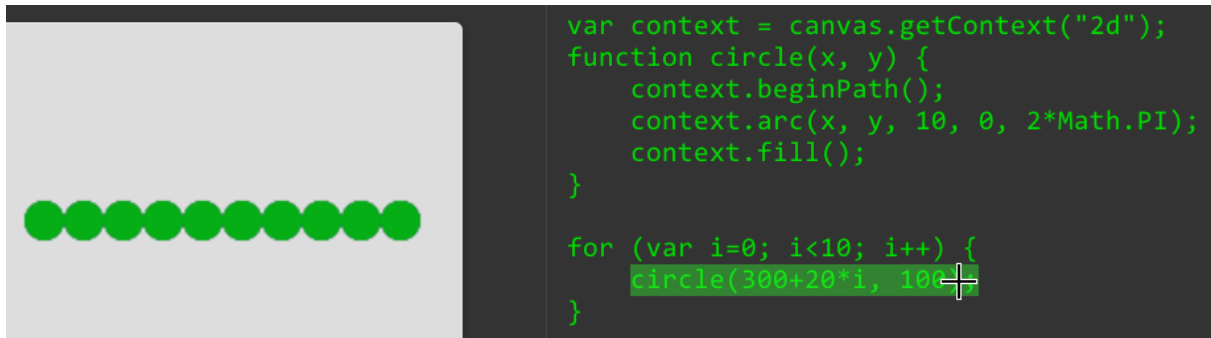


Figure 4.3: Since these circles are always highlighted together, they have the same call ID.

function call, and also of all the nodes that are on the internal call stack. The call ID itself is a serialisation of the node IDs on the call stack, as this uniquely identifies the possible highlights. Some calls can share the same call ID, but only if they will always be highlighted together (Figure 4.3).

4.1.5 Runner

One level above the running of programs using the context object, is managing different runs of one program. The *runner* object takes care of running the program when it changes, storing the current step and current event, running a function when an event fires, and maintaining the history of events.

When the program is first run — the initialisation of an interactive program or just the only run of a non-interactive program — it is treated as if it is an event, called the *base event*. In the runner, an *event object* is made, which contains the context object for that event, and the function name and arguments, which in the case of the base event are empty. The event is stored in the runner, and added to the list of events, the *history*. Initially, the step number is set to infinity. When clicking the step button, the step number in the runner is set to zero, and every subsequent click increments the number. When the end of the call list is reached, the step number is once again set to infinity. When the program is changed, and the program is not interactive, then the base event is run again. The step number stays the same (Section 3.5), unless the number of steps is now smaller than the step number.

When the program is interactive, events can come in, in which case the runner creates a new event object with the function name and arguments, and then calls this function with a fresh context object. If the user clicks pause, all subsequent events coming in are discarded. When the user moves over the history slider, the corresponding event is shown (Section 3.7). The editor toolbar queries the runner for information about the number of events, which events have errors, the current event number, and the current step number. Whenever buttons in the toolbar are clicked or whenever the slider is moved, the runner gets called, and updates the output tabs and editor with the new state.

The updating of a new state works as follows. The runner class has a reference to the main editor object, which ties multiple components of the interface together. Whenever the step number or event number of the runner is changed, by using the editor toolbar, the runner first checks these numbers to make sure they are valid. Then it calls the editor to signal that the state changed. The editor retrieves the required values from the runner, and calls the output tabs with the new step number and event number.

Now, every output tab has to maintain a list of events and calls itself. This is because this allows us to optimise the way calls are stored for each output differently. For example, the canvas only stores calls that actually draw something to the screen, and a serialisation of the canvas state when that call was made. This makes it much easier to do highlighting, as the state can simply be reinstated, after which the call can be made immediately. If we would store each call, then we would need to walk the entire list of calls to find out which properties have to be set before the drawing call could be made. The robot has completely different requirements, as it uses HTML elements for drawing the path, so a simple DOM call suffices for highlighting. When changing the step or event number, the canvas has to redraw based on the call list, whereas the robot can just make a few DOM calls to show and hide the appropriate elements.

Table 4.1: Synchronisation calls

method	description
<code>runnerChanged()</code>	The runner has changed state. Editor always calls <code>outputSetError</code> with the current error status, and <code>outputSetEventStep</code> with the event and step numbers.
<code>startEvent(context)</code>	Called before each run; a new event should be created. Always followed by <code>endEvent</code> .
<code>endEvent(context)</code>	Called after each run. Always preceded by <code>startEvent</code> , always followed by <code>runnerChanged</code> .
<code>clearAllEvents()</code>	Clear event list and reset outputs. Always followed by <code>startEvent</code> .
<code>popFirstEvent()</code>	Remove the first event from the list, all event numbers are decreased by one to reflect the new indices. Always followed by <code>startEvent</code> .
<code>clearEventsFrom(eventNum)</code>	Clear events starting from <code>eventNum</code> to the end; set the output to the state it was just before <code>eventNum</code> . Followed either by <code>startEvent</code> , or directly by <code>runnerChanged</code> in which case the new event number is <code>eventNum-1</code> .
<code>clearEventsToEnd()</code>	Clear event list, but do not reset outputs; current event gets number <code>-1</code> . Always called when the currently selected event indicated by <code>outputSetEventStep</code> is the last one; always followed by <code>runnerChanged</code> ; never followed by highlighting operations.

Because of these very different implementations, we decided to give the output tabs full responsibility over their calls. This means, however, that quite a bit of code needs to be replicated. After all, the different lists of events all have to stay in sync. Whenever the history in the runner changes, it calls the editor with some method, which in turn calls all the output tabs. The only other state between events is that of global variables, which is being tracked by the runner. A full list of synchronisation calls is outlined in Table 4.1.

When the program is not interactive, only the first four methods are called. When the program changes, everything is first cleared using `clearAllEvents()`, then the base event is run in between `startEvent(context)` and `endEvent(context)`, and finally `runnerChanged()` is called, so that the editor can update the toolbar controls, and call the `outputSetError` and `outputSetEventStep` methods of the output tabs. Note that the reason `outputSetError` is a separate method from `outputSetEventStep`, is that the former is also called in case of a parsing error to notify the outputs of this errors, while the latter is only used when running.

When using events, the other methods are also used. When an event is fired, the first three methods are called, unless the history is full, in which case also `popFirstEvent()` is called. When pausing, stepping and going back in time, only `runnerChanged()` is called. When resuming again, `clearEventsFrom(eventNum)` is called to erase the history after the selected event number. When the time is paused and the user edits the code, `clearEventsFrom(0)` is called to erase the history but still maintain the state as before the first event, and then the history is replayed by subsequent event calls. Finally, when the code is edited while the program is playing, the history needs to be erased completely, as none of the events correspond with the new code any more, so `clearEventsToEnd()` is called.

The runner gets notified when the user changes the program, as the editor object passes a new syntax tree to the runner. The runner then has to decide what to do with it, which is a rather complicated process (Figure 4.4). First of all, if the new tree is the same as the old tree, we replace the old one by the new one. This is necessary, even if they were semantically the same, since positions of nodes may have changed due to adding or removing of whitespace or comments.

If the tree is different, we look at interactivity. If the program is not interactive, the program is run again. If the program is interactive and the runner receives a new syntax tree, we have to take into account that we might need to update function definitions, events in the history, and perhaps set a flag that the restart button needs to flash. First we check if the program is paused. If not, then we only update the function definitions using the tree, and remove the entire history using `clearEventsToEnd()` (*path a*). This way new events will use the new function definitions. We also compare if any base code was changed, in which case we set the flag for flashing the restart

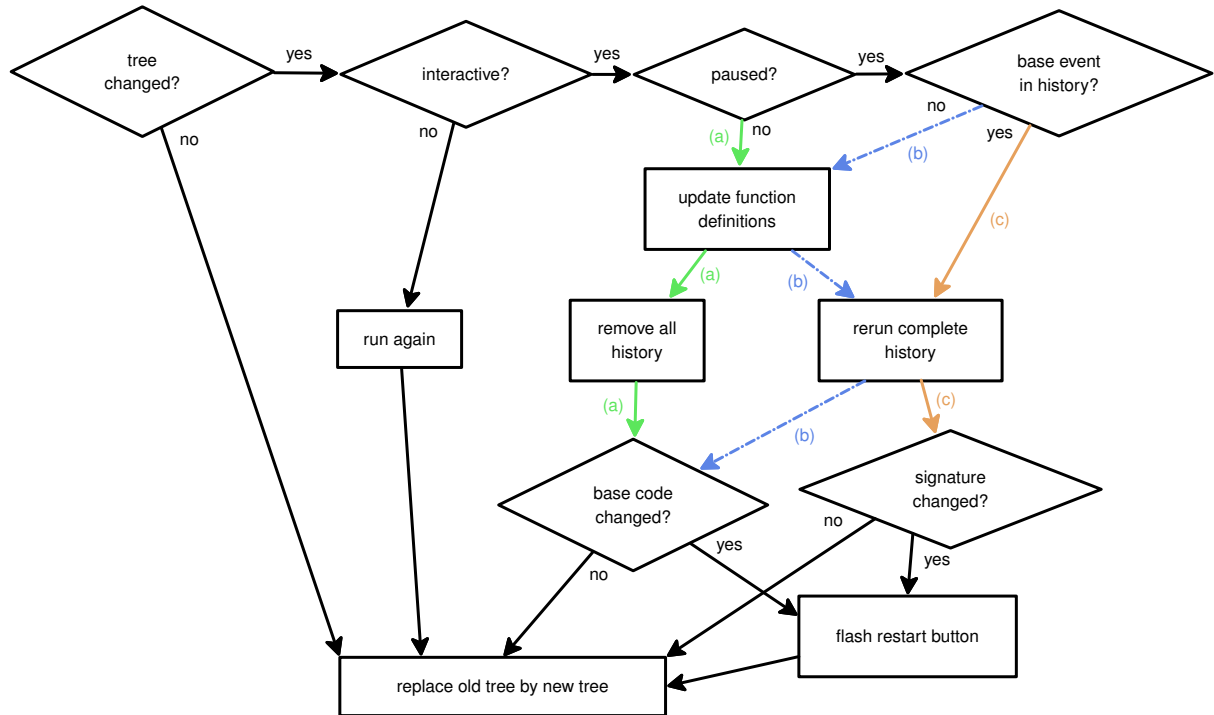


Figure 4.4: Flowchart of the decisions made by the runner when given a new syntax tree. When the program is interactive, there are three different paths of actions that can be taken based on whether the program is paused and whether the base event is still in the history.

button. If the program is paused, we check if the base event is still in the history. If not, then we update the function definitions and rerun all the events in the history (*path b*). Again, we also check if the base code has changed, for setting the flag to flash the reload button. If the base event is still in history, we rerun the base event with the new program, and after that all the other events in the history (*path c*). Since it is possible that the new initialisation adds, removes, or modifies event handlers, we check if the event handlers have changed (using a *signature*; Section 4.3.5), and if so, we set the flag for flashing the restart button.

Besides all this, the runner also has a number of auxiliary methods that are used by the editor, mainly as an interface to the context of the currently selected event. For example, it can return the function declaration node that is associated with the current event, so that the editor can highlight that function, as seen in Section 3.7. It can also return the call IDs of certain nodes in the syntax tree (in the current event), so that the calls can be highlighted in the output tabs. The same can be done for call IDs in *all* events, which is used for abstracting over time.

4.1.6 Extra language features

So far we have discussed most of the implementation details of the js-- language, but before we continue with other parts of the system, there are some final things to say about the language. First of all, there is the *editor helper* object. In order to keep some separation between the higher-level editor and the js-- language, we introduced this object that provides for some editor features, but operate in a language-specific way. This way, we could build support for other languages more easily in the future. First of all, the auto-completion feature as seen in Section 3.4 works this way. The editor provides the information of the current line that is being edited, and asks the editor helper whether or not there are examples to be shown for auto-completion. The editor helper then uses a custom parsing algorithm for the part of the line before the cursor, and use the last scope provided by the runner to look up examples. Another example is the manipulation feature, which we look at in more detail in Section 4.2.2.

In order to guarantee a good level of quality for the parser and runner, a unit test suite is used. Most of these test are quite standard; given some input they check for the correct output. Most of these tests are automatically generated using the existing parser and runner, and are then briefly hand-checked to verify if they are indeed correct. Besides

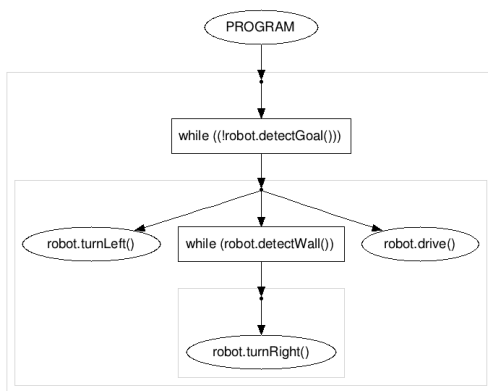


Figure 4.5: The syntax tree can be visualised using Graphviz dot.

these standard tests we use the fact that js-- is a subset of Javascript, so every program that runs in our subset, should produce the same result in Javascript. Using the DeMorgan dual we can check for errors in a similar way: every program that does *not* run in Javascript, should also produce some kind of error in js--.

We also built a couple of debugging tools while developing the system. The first one was a visual representation of the syntax tree. This was done by compiling the tree to the *DOT* graph language, which can be compiled into an image by *Graphviz dot*, or by using the *Google Chart API* online (Figure 4.5) [51, 35]. We also wrote a stress test for measuring performance, which we also used for determining costs of different output methods, as detailed in Section 4.1.3. We used a number of lists that could be printed to the browser console, such as all parser nodes, the final scope, the raw error message when the program contains an error, and the call stack when in stepping mode.

4.2 Editor

The editor module consists of three main components. First there is the *surface*, which takes care of rendering the large text box and elements such as highlights. Above the text box a *toolbar* is shown, which is another main component. These two components can be seen as the *view* parts of the model-view-controller pattern. The *editor object* can be seen as the *controller*, as it ties together the surface, the toolbar, and the runner from the language module, which can be seen as the *model*.

4.2.1 Surface

On the right side of the screen the code is shown in a large text box. On top of this text box different elements can be shown, such as errors, step messages, and highlights. All this together, we call the *editing surface*. The *surface object* is responsible for rendering the editing surface, while all events are passed to the editor object, which is where the actual application logic resides. The layout of the editing surface is fairly simple, as it is just one big HTML `<div>` element. Inside this there is a `<textarea>` element, which is the default way of using a text box in all browsers. Unfortunately this element does not support advanced features such as syntax highlighting, so the only way to reliably do this in the browser is to render all text manually. While there are existing libraries that do this, we decided not to put time into this arguably minor function.

The `<textarea>` element has a border on the left side, and icons can be shown next to it. This is done by placing elements in another `<div>`, called the *surface div*, which is placed on top of the `<textarea>`. Therefore, all elements put in there are positioned on top of the text. The top `<div>` can be scrolled vertically, instead of making the `<textarea>` scrollable. This is done to avoid having to recompute positions of elements in the surface div when scrolling, as this is necessary when the `<textarea>` would be scrollable. Recomputing positions is noticeably slow, and thus a sub-optimal user experience. Instead, we simply stretch the entire `<textarea>` when more lines of code are added, and shrink it again when lines are removed, which causes the top `<div>` to show a scrollbar. While scrolling, every character in the `<textarea>` will retain its exact offset compared to the top-left corner of the `<textarea>`.

When an error occurs, an icon is shown on the left side, and the line between this margin and the code turns red. This is a strong visual indication that something is wrong, and by scrolling up and down the error is found quickly. This is also why it is not possible to scroll horizontally, as this hides the error icon and line, so the user may not notice that something is wrong. In addition, the line length is restricted instead of wrapping lines, which makes the positioning of elements easier, and also enforces a good programming style (Section 3.2).

If the user clicks on an error or step icon, the message associated with that icon is shown or hidden in the editor. This is done by calculating the relative position to the top-left corner of the `<textarea>` using the character width and line height, which is trivial since we use a monospaced font. The code associated with the error or step is highlighted by adding a semi-transparent `<div>` element to the surface `div`. Underneath this highlight a balloon is shown with the message, which is also done by adding an element to the surface `div`. It is made sure that this balloon does not exceed the horizontal boundaries of the editing surface, as horizontal scrolling is not possible. When at an edge, the arrow pointing to the highlight is moved accordingly (Figure 4.6).

When the editor helper from the language module (Section 4.1.6) gives back examples for auto-completion, a similar box is shown. This *auto-completion box* shows the examples, if necessary with a vertical scroll bar. When moving the mouse over one of the examples, the editor object is called, which takes care of previewing that example. It is also possible to preview examples using the keyboard. For this, all arrow up and down events are intercepted, and used for moving the selection up and down instead of moving the caret in the `textarea`. Similarly, the tab and return events are intercepted in order to call the editor object, indicating that the current example should be inserted in the code. The same happens when simply clicking an example.

When highlighting is enabled, highlights are shown in the same way as when highlighting code for an error or step message. Also, a mouse move event handler is added to editing surface, so that when moving the mouse over the code, corresponding lines can be highlighted. This works by calling the editor object with the line and column coordinates, computed from the raw position coordinates as given in the mouse move event. The editor object then calls a method that updates the highlighted lines, and also calls the output tabs to highlight the corresponding calls, using call IDs provided by the runner.

As said before, updating positions is noticeable, so we want to avoid it as much as possible. Usually when the code changes, we need to update positions. However, keeping a key pressed can significantly decrease performance, so in this case we simply hide the entire surface until that key is released again. But this gives an annoying flickering effect when just pressing a key instead of holding it, since the surface is very briefly hidden and show again. Luckily, when a key is being held we get key down events for every subsequent character that is inserted after the first one, so we can just hide the interface at the *second* key down event.

4.2.2 Manipulation

Another feature is manipulation of literals, as seen in Section 3.6. The pieces of code that are highlighted when using manipulation are called *editables*. The functionality for this is somewhat scattered around, as the surface takes care of positioning the editables, there is some separate functionality for handling mouse events also in the editor module, and the language module takes care of how the values change when manipulating them (Section 4.1.6). We now look at the complete functionality of manipulation.

When enabling manipulation, the editor asks the language module for a list of editables. Each editable contains information about the position in the code, which is used for showing them on the editing surface, and for replacing the original code when an editable is interacted with. There are three kinds of editables. First of all, there is the *cycle editable*, which is currently only used for the boolean constants `true` and `false`. The idea is that every time such an editable is clicked, the next value from the list is inserted. In this case, the list only contains those two boolean constants, so you can toggle the value of a boolean this way.

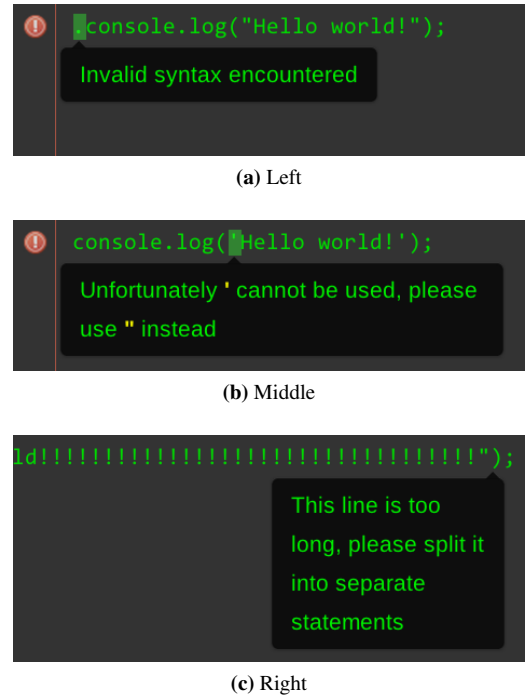


Figure 4.6: Balloons can be clipped to different sides of the surface, and the arrow pointing to the text is moved accordingly.

Another type is the *colour editable*. This is used for strings that contain a CSS colour value, and when clicked it shows a colour picker. For this we use an existing colour picker library by Lindekleiv [56]. The editor handles the interaction with the colour picker, and the language module makes sure the resulting value is put back in the Javascript string format.

The third type is the *number editable*. Any number in the code is highlighted, and can be changed by dragging to the left or right. Dragging to the left causes the number to decrease, and to the right to increase. Every time the horizontal mouse position changes a new number is calculated and inserted, updating the output tabs immediately. When the mouse is pressed down, the original value is stored, and whenever the mouse is moved, a function provided by the language module calculates the new number. The mouse move event is attached to the entire web page, not just to the editable itself, so that events are still being fired when moving the mouse outside of the editable.

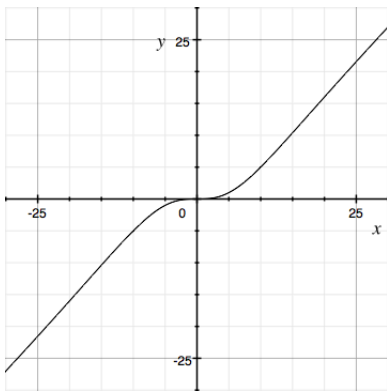


Figure 4.7: The function used for determining the delta based on the x-offset.

We experimented with the rate at which the original number is adjusted with respect to the dragging x-offset. The easiest way is to make this *delta* linear to the offset. This has the problem that it is hard to make either small or large changes; when using a linear slope around 1 or above, large changes are easy, but small changes require quite a steady hand, and lowering the slope makes it easier to find a specific value, but makes larger changes hard. A better function proved to be $\text{delta} = \text{offset}^3 / (\text{offset}^2 + 100)$ (Figure 4.7). This has a plateau at 0, which means that it is easy to snap back to the original value. After that it gradually approaches slope 1. This way it is easy to make precise changes if they are close to the original value, and when dragging further it becomes suitable for making large changes, although dragging becomes less precise. The order of magnitude of the change depends on the format of the original number. For example, 24 changes with step size 1, 2.4 with step size 0.1, and $2.4e20$ with step size $0.1e20$. This way the user can change the number format to get the appropriate step size.

When trying out ways to determine the format of the new number, our initial approach was to respect the user's choice of the number format as much as possible. We did this by stripping off and remembering the exponent part, and the position of the dot. What was left was simply an integer number consisting of the integer and decimal part of the original number. The idea was to calculate a new number from this, based on the x-offset when dragging, and then to restore the decimal and exponent parts.

The problem with this approach was that Javascript does not have arbitrary number precision, and for large numbers and small fractions it will automatically convert the format to an exponential format. For example, the number 1234567891234567891234 is automatically converted to the representation $1.234567891234568e+21$. Initially we tried to solve this by acknowledging this new format, adding the original exponent to it, and then trying to convert the format back to the original one. In practice this proved to be counter-intuitive and error-prone, so we discarded this approach.

A better approach is to respect the original number format less, but consistently convert the number when necessary, while using mostly internal Javascript functions for this conversion. First the number of significant digits is determined, and this is used for the initial conversion by using the internal function `toPrecision()`, which converts the number to an equivalent format with the same number of significant digits. Once in this format the number of decimal places is determined, and this — not the number of significant digits — is used as an invariant throughout. This means that whenever determining the new number format, we keep the number of decimals the same as of that of the normalised original value. Otherwise, numbers would be turned into unwanted formats a lot, such as $3e2$ instead of 30 , when starting with 5 as the original value, as it has only one significant number.

Besides this we use a few tricks to adhere to the user's format better, such as always using the number of decimals in the original number when the number is zero, e.g. 0.00 will be kept in that format even though it has no significant numbers. Also, the format of the exponential letter is recognised and used, as it can be either `e` (the default) or `E`. If in the syntax tree the number is contained in a `UnaryExpression` with a `+` or `-`, then this `UnaryExpression` is used instead of the number itself. This allows for taking the sign into consideration and thus prevents outputting `--10` or `+-10` whenever possible. Note that when the number is part of a `BinaryExpression` such as `5+3` the sign is not being changed, as it may be part of a formula, which we do not want to disrupt. For

example, when changing 3 in `sin(10-3)`, it may become `sin(10--10)`, which still respects the original format of `sin(10-x)`. Similarly, `+=` and `--` assignments are also not changed.

4.2.3 Toolbar

Above the editing surface we find the *editing toolbar*. The toolbar operates similarly to the surface, it just takes in information, displays it accordingly, and calls the editor object when necessary. The buttons are standard buttons from the *Twitter Bootstrap* library [83]. The toolbar consists of 3 parts, which we look at in this section.

The first part of the toolbar is the stepping part. The 3 step buttons are for stepping backward, forward, and closing the stepping mode (Section 3.5). Whenever a button is clicked, the editor object is called, which in turn calls the runner. It is also possible to click and press the backward or forward button, without releasing the mouse. A timer is set when pressing down the mouse, and when the timer fires an event, the editor object is called. Every time the timer fires, a new timer is created, with a slightly shorter timeout, to create an accelerating effect, until a minimum time is reached. When the mouse is released again, the timer is destroyed.

When stepping, a bubble appears left of the step buttons. This bubble shows the current step number, and the total amount of steps. When holding the manipulate button, the step number can be changed in the same way as when changing a number editable (Section 3.5), although the language module is not used for determining a new value.

The second part of the toolbar contains the highlighting and manipulation buttons (Section 3.6). They can also be pressed, after which the editor object will be notified. When hovering over one of the buttons, a small balloon appears showing the keyboard shortcut for that button. When holding the *control* key (or the *command* key, on the Macintosh), the highlighting function is enabled, and when releasing it again, it is disabled. Similarly, the manipulation feature is enabled while holding the *alt* key. The features are also disabled when the page loses its focus, because then the keyboard events do not fire any more. This is otherwise most noticeable when using the *alt-tab* or *command-tab* keyboard shortcut to change windows, because the subsequent alt or command release events are missed, so the corresponding features will be left on when switching back to the browser.

The third part is for using events (Section 3.7), and is hidden when not using them. The restart button has a flash animation defined in CSS [21], which is enabled when the base code of the program has changed. On the pause button a subtle dark overlay moves from left to right when running, which is done by placing a semi-transparent `<div>` on top of the button, and adjusting its width using a CSS transition. Simultaneously, a timer is set to fire exactly when the CSS transition ends, in order to reset the width to zero again, and restart the animation. When the button is clicked, or when *escape* is pressed on the keyboard, the program is paused. In some browsers, a lot of events are fired at once when pressing the escape button, so we wait until it is released before responding to key press events again.

The slider is made of a couple of `<div>` elements, styled using CSS shadows and rounded corners. Inside the slider we can highlight segments with a different colour, to show the currently selected event when hovering over the slider, and to show which events have errors (Figure 4.8). For this we again simply use some `<div>` elements with the corresponding colours and positions. When the mouse is moved over the slider, a bubble containing a step bar is shown, for which we simply use the exact same implementation as for the main step bar.

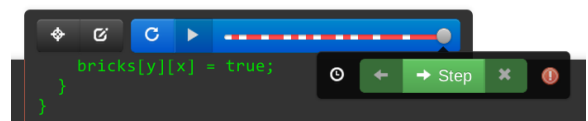


Figure 4.8: Segments inside the event slider can be coloured to indicate events with errors.

4.2.4 Editor object

The *editor object* can be viewed as the controller in the model-view-controller pattern. It receives callbacks from the surface object, the toolbar, the syntax tree, the runner, and the different output tabs, and also calls these objects when necessary. For example, when the user changes the code, the surface calls the editor object, which then parses the code using the language module. Then it checks if the resulting syntax tree contains an error, in which case the toolbar is disabled, the error is shown on the surface, and the output tabs are told that there is an error. If

there is no error, the new syntax tree is passed to the runner, which then updates the program, or even the entire history. While this happens, editor functions are called, which in turn call functions of all the output tabs, so that they can keep their history in sync (Section 4.1.5). Finally, the runner tells the editor object that it is done, and the editor object tells the outputs that there is no error, and what the current event and step numbers are.

To initialise the editor object, we have to provide a language module; output tab objects; HTML elements in which to create the surface and toolbar; and a runner object, which contains a number of settings and the global scope. Besides for running, the editor object uses the runner and tree from the language module for getting node IDs and call numbers for highlighting. It also uses the most current tree for looking up locations in the text corresponding to node IDs. When using events and abstracting over time (Section 3.7), it stores the function names of the functions that are highlighted, and updates these highlights at every change of the code. It calls the editor helper from the language module to build editables when using the manipulation feature.

An auxiliary object used by the editor object is the *code helper*. This is used for converting offsets to line and column numbers, and vice versa. A number of methods are available in the code helper, such as looking up text for a line-column range, and determining the leftmost and rightmost columns (ignoring spaces) for a multi-line range.

Finally, it is possible to set a callback in the editor object, which is called every time the code is changed. In that case, the new program is passed as a string to this callback. This is used to save the program on the server whenever a small change is made, although we do throttle requests to the server so that only every few seconds a request is made.

4.3 Output tabs

On the left side of the interface, there are the *output tabs*. Each output tab has a corresponding Javascript object which is made part of the global scope in `js--`, and registers these calls in a list. When using events, the output tabs are notified when a new event starts or ends, and also when the event list changes (Section 4.1.5). The output tabs each keep their own list of events, since they all work in different ways, allowing to optimise this list for the specific output tab.

We look at the implementations of the different output tabs, respectively *console*, *robot*, and *canvas*. Then we discuss the *info* tab, which provides some additional information, despite not having any methods exposed in the language. We also look at some output objects that are not represented as tabs, but are part of the output module anyway, *input* and *math*.

4.3.1 Console

The *console* output tab is used for displaying lines of text. There are only three methods that can be used with the console: `console.log()`, `console.clear()`, and `console.setColor()`. Only the first one is a standard method that is supported by the debugging consoles in most major browsers. The second one is used for clearing the console, and is especially useful when making console games. The third method sets the colour of the text that is added after calling it. It is relatively easy to implement this console using basic HTML elements and some additional Javascript, so in the future this can be released as a small library that people can use if they want to run their code outside our environment, and still be able to use the non-standard methods (Section 5.2.4).

In our implementation, we have to do a bit of additional bookkeeping for implementing the event history efficiently. At the core there is a `<div>` element, which itself contains another two `<div>` elements. The first one contains *old lines*, which are the lines that do not correspond to any events in the current history, because those events have been removed already. Therefore, we do not keep track of these individual lines, and they are grouped together. Then there is the `<div>` of *current lines*, each of which is created by some call in one of the events in the history. Whenever a specific event or step is shown, all the lines are hidden first. Then the old lines are shown all at once. After that we traverse all the calls in the event history until we reach the current event and step. Every `console.log()` call has an associated line element, which is shown when traversed. When encountering a `console.clear()` call, all lines, including old ones, are hidden again. An optimisation is to store at every event where the last clear call was, so that not the entire history has to be traversed.

Every time the first event of the event list has to be popped, the lines associated with that event can be moved to the list of old lines, and all references to the lines can be removed. However, this is a relatively expensive operation, and while running the output tab should perform best. Therefore, we delay this operation until selecting a specific event and step number, which only happens when pausing the program. This is done by keeping a *mirror* string, which contains just the HTML of the current console. At every `console.log()` call, the mirror string is updated as well, which is a cheap operation. At the start of every event, the mirror string is copied for that event, which is also cheap. Also, a reference to the first added line in that event is stored. When pausing and selecting a specific event, we look up the oldest stored event. Then we remove all the lines before the first line added in that event. The old lines are then directly updated using the copy of the mirror string stored with that event. This way, the old lines are pruned when paused, so that we do not have to do expensive HTML updating operations while running.

In order to prevent the user from adding an excessive amount of HTML elements, which slows down the interface, the number of visible lines are counted. A problem with this, however, is that `console.clear()` only hides the currently visible lines, as they could be made visible again when selecting a specific event or step. If a program generates a lot of lines, but also clears the console in between, this should be allowed, for example to make console games. For this to work, the old lines that are not in the event history any more should be pruned regularly, to avoid having too many HTML elements on the page, even though they are hidden. Therefore, the procedure as described before is run every once in a while in addition to when the user pauses the program, as a form of *garbage collection*, if you will.

Besides the currently shown lines, the console also stores the current colour for each event. However, we do not store calls to `console.setColor()` explicitly. The colours are already set for each line element, and there is no need for knowing when exactly the current colour was changed. Only when returning the state of the console to that of an earlier event, when the `clearEventsFrom` method is called (Section 4.1.5), should we know the colour, which is why it is simply stored per event.

When highlighting lines in the console, we store call information with the lines themselves using the jQuery Data API [48]. This allows us to look up the event number and the index of that line in the call list for that event. There, in the call list, the node ID is stored. The node ID is used for calling the editor, which then highlights the node in the code.

Finally, the console has two extra features. The first one is showing a *target console*, which is used for dares. It shows what the console should look like for matching dares, so that the user can try to copy this. For this, another `<div>` element is used, which sits behind the actual console, in a slightly dimmed colour. This way, the user can place lines over the target console. The other feature is auto-scrolling. Whenever a `console.log()` call is made, the console automatically scrolls to the bottom. When the user scroll the console manually, this is temporarily disabled, until the user scrolls back to the bottom.

4.3.2 Robot

Another output tab is that of the *robot*. Just like the console it heavily uses HTML elements for displaying its content. This is done instead of using a `<canvas>` element, which allows us to draw arbitrary shapes, as with HTML elements we can use native browser functions, such as CSS animations [21], and mouse events. The event history works similarly to that of the console: a list of calls is maintained for every event, and each call contains a reference to the HTML element. This is used for showing or hiding elements when selecting a specific element and step number, and for highlighting calls. Each element also has some information associated with it using the jQuery Data API [48], used for highlighting the other way around, just as with the console. All elements are placed into the area in which the robot can move around, called the *robot environment*.

The HTML elements that are used in the robot output tab, are for the path of the robot. Whenever the robot moves using the

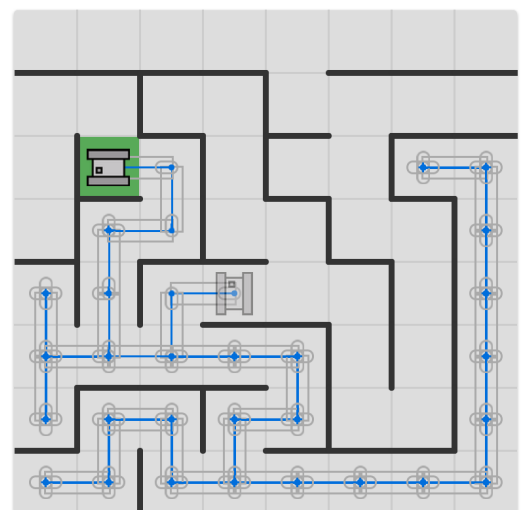


Figure 4.9: When highlighting, the path elements that capture mouse events are made bigger. Here the outlines are visualised, but normally they are transparent.

`robot.drive` method, a line is drawn. This line is essentially a long `<div>` element with a black background colour, positioned and rotated using CSS properties. We chose the width of this element to be visually pleasing, but for highlighting we found that it is too small. Therefore, we use in fact two nested `<div>` elements, the outer one used for detecting the mouse when highlighting and thus having a larger width, but is transparent (Figure 4.9). The inner one is the actual line segment. Using CSS we make sure that when highlighting, the currently highlighted line segment is always positioned on top of all the other line segments, so that the focus is not lost unexpectedly.

When the robot turns using `robot.turnLeft()` or `robot.turnRight()`, we draw a small dot with a line. This line indicates the direction the robot faces after the turn, thus showing that the robot has turned even when after that turn it again turns in place. The HTML element is built similarly to that of the line, with an outer `<div>` for detecting the mouse when doing highlighting, and two `<div>` elements inside for the dot and the line. When dots and lines are on the same level, i.e. both not highlighted or both highlighted, then the dots are always drawn on top.

Besides having references to HTML elements, the call list (associated with each event) also refers to segments of animation. There are two objects taking care of robot animations, the *animation object* and the *animation manager*. For each new animation an animation object is created, which stores the animation in a convenient way, and can play any segment of the animation. For this we use a timer in conjunction with CSS transitions. We calculate the time it should take for the segment to play, and set a CSS transition for that time. Then the browser will take care of animating that particular segment. At the same time a timer is set to fire after the same duration, so that then the next segment of the animation is played.

The animation manager takes care of creating and deleting animation objects, and playing the appropriate parts. Whenever the code changes (or actually, the syntax tree), a new animation object is filled with calls, thus creating a new sequence of animation segments. This sequence is serialised into a string by the animation object. The animation manager checks whether or not this string has changed, and only starts playing the new animation when it actually did. This means that while changing the code, but not the path of the robot, the currently playing animation continues, instead of restarting from the beginning every time the code is rerun.

We decided not to implement specific measures for pruning the path if events are being discarded, but to always show the entire path. Since there are also no methods for clearing the path as with the console, this means that the path will never shrink. To contain the number of HTML elements on the page, a counter is used, and an error is thrown once it reaches some threshold. This means that it is not possible to write a program with which the robot can be driven around indefinitely, but it does mean that the path is consistently shown. We think this is the preferred behaviour for novice users, which is what the robot environment is aimed at.

When doing robot dares, the user does not have to copy the path from the example, such as with the console and canvas. Instead, a number of goal squares are shown in green, which the robot should touch by driving over them. When not doing a dare, goal squares can be placed or removed by clicking on them. They are, again, just `<div>` elements with some CSS styling. In the same way the user can add or remove walls on the grid by clicking on them.

When there are walls placed, the robot is restricted to drive on the grid, which means that only integer distances can be used, and right angles. When in this mode, an exact algorithm is used, which does not use trigonometrical functions. This way we can unambiguously check whether or not the robot is standing in front of a wall, by using the `robot.detectWall()` method. It is also useful for throwing errors when the robot actually runs into a wall. In the same way we check goal squares for `robot.detectGoal()`, and to check if the robot has visited a goal square for dares. When there are no walls placed, the robot can drive around freely, and the position of the robot is calculated using trigonometrical functions.

The user can also drag the initial position of the robot, which is shown in a dim colour. When dragging it, a mouse move event is attached to the robot environment, and the offset of the initial mouse press compared to the top-left corner of the robot image is stored. At every mouse move event, the image is moved so that it stays under the mouse cursor, until either the mouse is released again, or leaves the robot environment.

Every time the user changes the robot environment, by changing walls, goal squares, or the start position of the robot, the `outputRequestsRerun()` method of the editor object is called, which causes the program to be run again. It is also possible to set a callback to be run every time this state is changed, for example to save the new

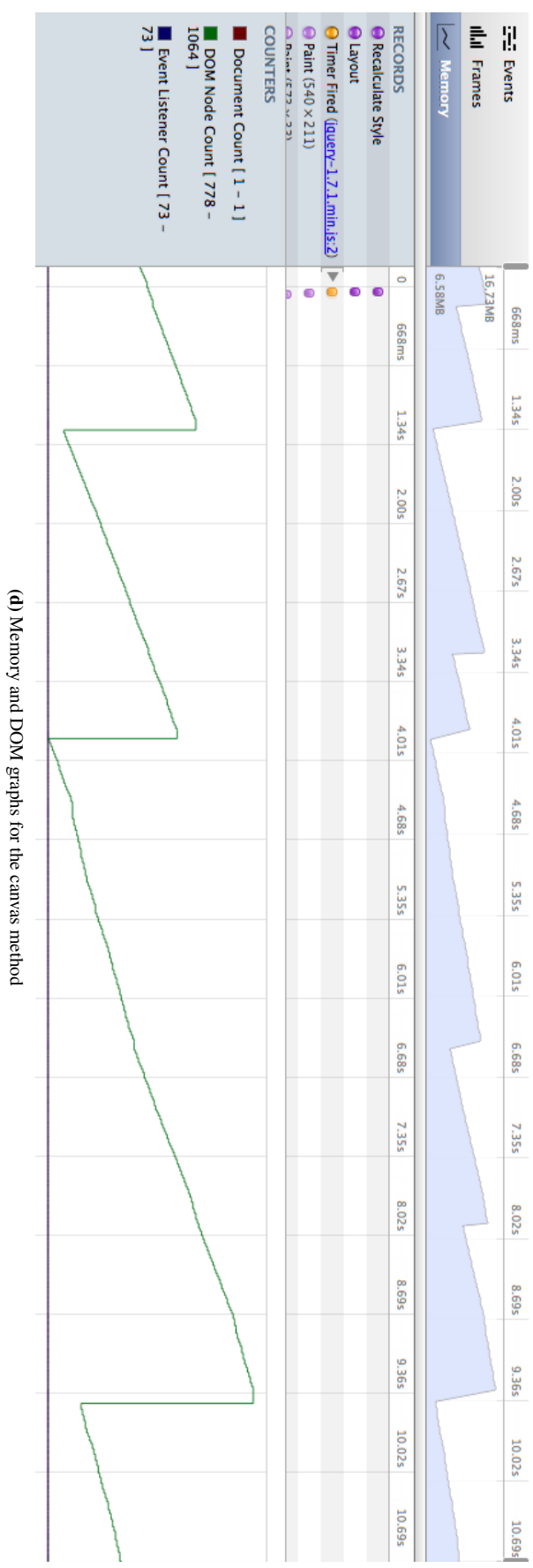
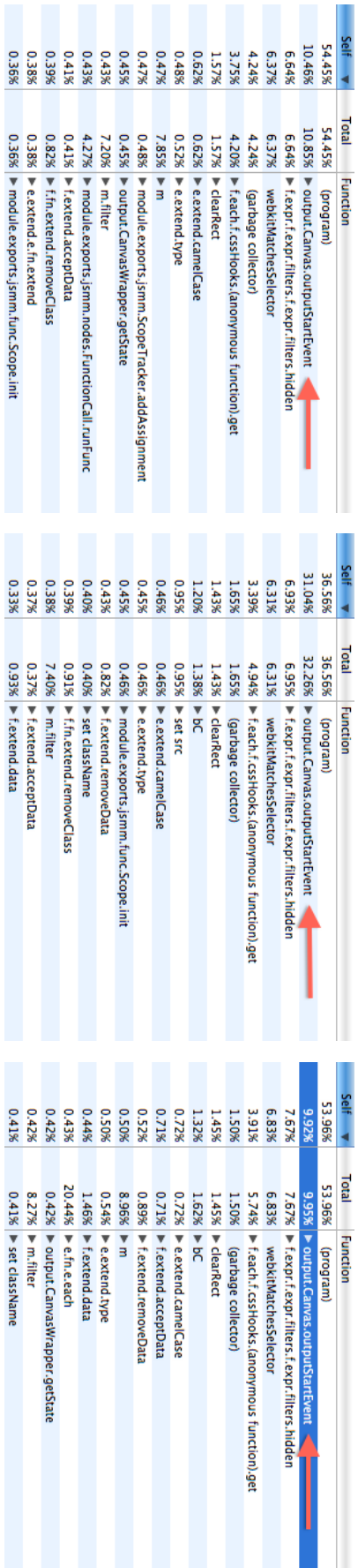


Figure 4.10: We did our own testing of the different methods for saving and restoring the visible state of the canvas.

state, just as with the editor (Section 4.2.4). The state is then serialised into a string, and passed to the callback. The string can be passed to the robot output tab again, to reinstate the walls, goal squares, and start position.

Unfortunately, because no universal Javascript standard exists for something like the robot environment, it cannot be used outside of this interface directly. However, we have written the robot environment as a separate module that does not use any other parts of the system. This means that in the future we could release it as a stand-alone library, allowing users to use it on any website (Section 5.2.4).

4.3.3 Canvas

With the *canvas* output tab users can draw arbitrary shapes and text. It can be used to make Javascript games, which is one of the goals of this project. It complies with the HTML5 Canvas API, although only a subset is supported [45]. Some basic properties of the canvas object are `canvas.width` and `canvas.height`, but for most operations a two-dimensional context must be requested, using `var context = canvas.getContext("2d");`. Then this context object can be used for drawing shapes and text. The only reason for doing it this way is to comply with the standard API.

Because the canvas is commonly used for building games, we put a lot of effort into optimisation of canvas calls during runtime. When possible, we defer calculations until the program is paused, to minimise delays during execution. Besides this, it should be possible to show content of the canvas for every event and step in the current history, and to resume execution after a certain event. Furthermore, we should be able to highlight any call in the history, both by highlighting the node in the code, or by hovering over the canvas itself.

The simplest way to do this, is by storing every call for every event. This way, it is always possible to reconstruct the image at a certain point. But when events are thrown away, we do not need all this information any more, it is sufficient to know what the state of the canvas was before the first event. This state comprises of two parts: the *visible part*, or the image; and the *invisible part*, or the property values of the context object. For the invisible part, we built a *canvas wrapper* which is used instead of the actual canvas API. It can serialise and de-serialise the canvas properties, allowing us to save and restore the invisible state. Most properties can just be read using the canvas API directly, but one exception is the *path*. This is the current sequence of line operations, before it is actually drawn to the canvas. The canvas API does not provide any methods to explicitly read or set the path, so the only option is to store all calls. When restoring the path, we can then call `context.beginPath()` to reset the path, after which we iterate over the list of calls.

For the visible part, there are essentially three ways to save and restore the actual image. The first one is copying a canvas onto another `<canvas>` element, the *canvas method*. The second one is copying a canvas onto a Javascript image object, the *image method*. The third one is storing the pixel data in an array, the *pixel method*. Baulig has done an analysis of these three methods, for both reading and restoring an image [12]. This shows that the canvas method works best for reading the image, and both the image and canvas method work best for restoring it. For us, reading efficiently is most important, so the canvas method would be preferable.

Self	Total	Function
58.43%	58.43%	(program)
6.70%	6.70%	webkitMatchesSelector
4.83%	5.03%	▶ output.Canvas.outputStartEvent
4.53%	4.53%	▶ m.selectors.filter.PSEUDO
4.01%	5.82%	▶ f.each.f.cssHooks.(anonymous function).get
3.52%	3.52%	▶ f.expr.f.expr.filters.f.expr.filters.hidden
1.45%	1.45%	(garbage collector)
1.16%	1.16%	▶ clearRect
1.06%	1.86%	▶ f.extend.data
0.98%	1.03%	▶ bC
0.82%	0.82%	▶ e.extend.e.fn.extend
0.58%	9.12%	▶ m
0.58%	0.58%	▶ e.extend.camelCase
0.54%	0.54%	▶ f.extend.acceptData
0.51%	0.96%	▶ f.extend.removeData
0.46%	8.52%	▶ m.filter
0.44%	0.46%	▶ e.extend.type
0.42%	0.46%	▶ bC

Figure 4.11: Reusing canvas elements in a circular buffer, while using the canvas method for each element, clearly performs best.

We did our own analysis to confirm this using a real world case: the game that we use on the homepage (Chapter 3) [63]. We ran this game for 10 seconds while running the Javascript profiler built into Google Chrome [34], the results of which are shown in Figure 4.10. The lists show how much time is spent inside the body of each function. Both the pixel and canvas method score best at around 10%, while the image method indeed performs much worse at around 30%. However, the implementation of the canvas method creates a new `<canvas>` element for each event, as can be seen by the line that displays the DOM node count in Figure 4.10d. When reusing old elements in a circular buffer, the result is much better, doubling the performance (Figure 4.11). The performance of the canvas saving method is further improved by an order of magnitude by only saving it for a certain number of events, and using the call lists of other events to reconstruct the image for a specific event. How often

the canvas is stored, is a trade-off between runtime and pausing performance. Saving once every thirty events makes the function disappear from the top of the list, without noticeably affecting performance when scrolling through the event history when the program is paused.

With this implementation three things are saved for each event: the invisible state, a number which signifies the offset to the event that contains the last visible state, and the call list. For the call list we only save calls that actually draw something, not just change the invisible state. For each call we save the method and its arguments, as well as the invisible state at that point. This makes it very easy to highlight calls given some call IDs, as we can simply look up the calls, set the invisible state, change the colour, and make the call. By doing this, the highlight is always drawn on top, which is another advantage as it even highlights calls that have by then become hidden.

Highlighting the other way is a bit trickier. The idea is to somehow map canvas coordinates back to node IDs. One way this can be done is to manually implement all canvas methods to work on a two-dimensional array of node IDs, and use this besides calling the actual canvas methods to draw shapes on the screen. This is a very error-prone and cumbersome method, as we have to exactly copy every detail of the canvas API. Instead, we use the canvas API itself for this. The trick is to use a second canvas that we call the *mirror canvas*, which is hidden from the user. The exact same calls as made to the normal canvas are also made to the mirror canvas. Every call is identified by a unique colour, and when drawing to the mirror canvas this colour is used instead of the original colour. (Figure 4.12). This way, by looking up a colour value in the mirror canvas for using the mouse coordinates, the original call can be found again, so that the entire shape of that call can be highlighted in on the canvas, and the associated node can be highlighted in the editor.

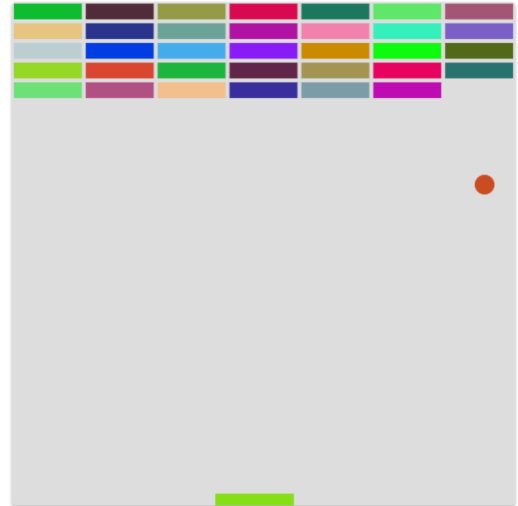


Figure 4.12: The (normally hidden) canvas mirror uses different colours to map mouse offsets back to node IDs.

There are two problems with this approach. The first one is that there is a finite number of colours, and a seemingly infinite number of potential calls. However, in practice there are only a finite number of calls allowed, as the program is terminated when it runs for too long, and we only allow highlighting of the current event. Even when there are not enough colour values, it is not too bad to recycle previously used values, as this only means that in the rare cases that this happens, the wrong lines are highlighted. This is not ideal, but not a huge problem, given that we have never yet encountered this in practice.

Another problem is more pressing, which is that the canvas mixes colours that are on the edge of a shape. This is called *anti-aliasing*, and makes lines and edges look smoother, but poses a problem when actually using the colour for something else than a visual indication. When just using the call number in red-green-blue format, we did encounter this problem, as the wrong nodes and shapes were highlighted. A solution to this is to at least use very different colours for every new call, making it less likely that mixing two colours results in a colour that already identifies another call. In theory it is possible that colours get mixed up because of this, but in practice we have not encountered this, so hopefully it is sufficiently rare.

Just as with the console output tab, it is possible to show a *target canvas* when using dares, which is put behind the actual canvas in a slightly dim colour. This way, the user can match the target image pixel by pixel.

4.3.4 Info

The info tab shows the list of scopes corresponding to the current event and step, and a command reference (Section 3.8). For visualising the scopes it uses the scope tracker provided by the context object (Section 4.1.4). The scope tracker takes care of storing and retrieving which variables contain what value at each step. The info tab can use this information directly to visualise the variables. Enough information is provided to allow for highlighting in the same way as with the output tabs. The command reference shows a predefined list of

commands. It can be specified per dare which commands to show, and which examples to show for each command. For highlighting the command tracker is used, also provided by the context object.

4.3.5 Input

The input object is not strictly a tab, but is part of the output tabs module. It provides global event methods (keyboard and interval), and can add mouse event methods to output tabs. As soon as the user program defines an event handler, the runner is notified that the program is interactive. A *signature* is passed to the runner, which is a string that contains which events handlers are defined, including the function names. This is necessary since the runner needs to know when the event handlers change in order to flash the restart button in rare cases (Section 4.1.5). Whenever a jQuery event comes in [48], it is first adapted for use in js—, which means that most properties are removed. Then it is passed to the runner, which calls the appropriate function with the adapted event as only argument.

When adding mouse events to an output tab, event handlers are only actually attached when they are being used in the program, since mouse events are relatively expensive. This is also why mouse move events are throttled: when the first mouse move event comes in, it is passed to the runner right away, but after that we wait a number of milliseconds before sending the next event to the runner. If in the meantime multiple events come in, only the last one is used.

Since event handlers can only be set in the base event, the input object keeps track of whether or not the current event is a base event. In subsequent events an error is thrown.

4.3.6 Math

The math object is mostly just a wrapper for the standard Javascript `Math` object. This object provides a number of mathematical operations and constants. These operations are stateless, which makes it trivial to wrap them, since we do not need any history. However, there is one exception, which is the `Math.random()` function. Of course this function is supposed to generate a random number at every call, but this leads to some unwanted behaviour. If the output of a program relies on this function, it would change every time the user changes some code. This would disturb the process of quickly testing something, as it is unclear whether behaviour is due to the change, or due to the randomness.

Therefore, we have implemented a deterministic random number generator. For this we use a recommended approach by Marsaglia [61], a simple *xorshift random generator*, which works as follows: $x \wedge= (x \ll 13)$; $x \wedge= (x \gg 17)$; $x \wedge= (x \ll 5)$; , with any non-zero starting value for x . This sequence is run for every subsequent random number, which is then converted from a signed 32-bit integer to a float between zero and one. This method is extremely simple, yet it has traverses over all 32-bit integers except zero (period $2^{32} - 1$), and passes almost all randomness tests. For each event the last value of x is stored, so that when resuming from an older event the same sequence is again produced.

4.4 Dares

Dares are exercises in which the program is limited in length. Dares are grouped in *collections*, which contain a number of dares. In a collection we show whether or not a dare has been completed using a green background, and the personal high score for that dare. There are three types of dares corresponding to the three output tabs: *robot goal dares*, *console matching dares*, and *canvas matching dares*. These types share some code, in particular for animations.

For now, we do not verify the scores on the server-side. This is because the canvas element can behave differently on every computer, and even between browsers. For example, some fonts may not be installed on all computers, or the implementation of drawing algorithms may be slightly different, causing rounding errors. Also, the Canvas API is still in flux, so there can be deliberate differences between implementations. So it is not possible to generate a reference image on the server, and then compare it with the image generated in the browser, as they may differ

so much that it is not possible to replicate the image in the browser in the same amount of lines. The only way to properly solve this is create a completely custom implementation of the Canvas API, so that we can guarantee the consistency of all operations across different environments. This is very involved, however, and will probably harm performance.

Therefore we compare the programs in the browser. This way the reference image is generated in the exact same environment, so it is always possible to get the full amount of points by replicating the image exactly. We simply send the amount of points to the server, which makes it easy for users to cheat by manually making a request to the server. This is discouraged by simply not showing a public list of high scores, thus providing no incentive to cheat. This could also potentially help novice users in being less anxious while learning programming, since they will not notice it as much when they are under performing. For those users who do like the competitive aspect, we could implement discussion boards for each dare, in which solutions and scores would be shared.

When the user submits a program, an elaborate animation is shown to provide clarity as to how the points are awarded. For this a special *segmented animation* object is used, to which animation segments can be added. Each segment contains a function callback, a speed, and a number of times the callback should be executed. Every time a callback is called, the number of times it has been executed before is given as an argument. For example, when showing the animation for the number of lines in a program that has ten lines, a segment is added specifying the callback should be called ten times. The segmented animation object also allows to jump to the end of the animation, which means that all remaining callbacks are called with their maximum number as argument. This puts the interface in the state it would be after running the entire animation.

All dares share the logic for giving points and animating the number of lines of the program. The robot goal dares check how many goals are visited, animates this accordingly, and gives points if the minimum number of visited goals is reached. The console dares compare the reference output and program output character for character. An animation is used for showing which characters match. For this a canvas is used on top of the console, on which red and green rectangles are drawn, taking into consideration the console font, and special characters such as tabs. Canvas dares work similarly, with a different canvas on top of the canvas in which the example is shown. A canvas pixel matches if the Euclidean red-green-blue distance between a reference pixel and the program pixel is below some threshold. There is also a special mode to ignore colours, which can be used to encourage users to experiment with colours, which could be motivating.

'Teaching is just a different way of learning. The critical difference is that the teacher gets paid.'
— Unknown

Chapter 5

Evaluation

5.1 User evaluation experiment

In August 2012 we performed a small-scale user evaluation experiment with 7 participants, ages 10 to 17. They were recruited by asking colleagues and acquaintances if they would have their children participate in our experiment. Some children had some minor programming experience, but none of them had any experience with Javascript. For one hour they tested the software in a field trial, and afterwards the children were asked to fill out a questionnaire. It should be noted that one of the children was dyslexic, which is difficult when using a text-based programming environment. A staff member assisted her in reading and typing.

5.1.1 Method

Now, a study with only 7 participants cannot be used for a proper comparison with other courses, so we did not have a control group at all. Furthermore, there was most likely some serious selection bias, as we did not make a random selection of children from some school. Comparing whether a certain course improves learning requires a study of at least one semester, and preferably longer, since it is a long-term effect. We did not have the time to do that, so the evaluation is aimed to discover things *within* the course. We tested if there are any bugs in the software, both obvious ones (e.g. glitch in positioning, storage issues, etc.), and problems with interaction. According to Nielsen, a small group of users can be used if the goal is to uncover usability problems, since a majority of problems can be found by testing with only 5 or 6 users [66].

To test if the children were able to learn some programming in an hour, with minimal instruction, we created a set of dares that teach the basics of the robot environment, and then teach functions (first without, then with arguments). We made notes of what happened and what problems the children ran into. We also measured how long they spent at each dare, to compare them with each other. The experiment is a *formative evaluation* using a *field trial*, using the classification of Lockee et. al. [11]. A formative evaluation is a suitable technique for getting feedback during implementation, so we can use the results for further developing the software.

Note that we consciously decided not to use a well-known technique such as think-aloud testing in this experiment, or constructive interaction (pairs of children collaborating) which could work even better [9]. First of all, such a setting would take significantly more time as we would have to study the children individual or in pairs. Second, the computer classroom setting is more natural, since the programming course will most likely be used like that at schools. Finally, it would be harder to get parental permission for videotaping, which we did not require in our classroom set up.

In order to prepare for the experiment, we tested the software for a couple of minutes with some fellow students, as suggested by Nielsen [66]. This was done to make sure that there were no major issues that would prevent the experiment from running smoothly, and to get an initial idea of how users interacted with the software. We did not encounter any major issues.

We followed the recommendations given by Hanna et. al. for doing usability testing with children [40]. Before the experiment, we emailed all the participants with a description of what we were going to do, to set their expectations correctly. We also sent an informed consent form for both the parents and children to sign, detailing that participation would be anonymous and voluntary. During the experiment we gave as few instructions as possible, tried to spend equal amounts of time with each participant, and encouraged them when they accomplished something. We made sure that the experiment itself did not last longer than an hour, and afterwards we provided some drinks and snacks.

In the first couple of dares we introduced the basic Javascript syntax and the interface. All tasks were to simply navigate the robot by manually giving commands for each action, instead of using functions and loops. We introduced the robot tab, info tab, stepping, highlighting, manipulation, and errors by using big orange arrows. We introduced programming itself by first providing an almost working program that the user had to complete, and finally moving towards a blank slate, so the user had to write it from the beginning. We also gently introduced the concept of minimising the number of lines by explaining the concept and awarding points for fewer lines, but still only setting a rather loose limit.

We then introduced the concept of functions by giving mazes with certain patterns, the navigation for which can be put in functions, and setting the line limit so that functions had to be used. Besides giving a basic explanation of functions and a reference to the info tab, we also provided the first such dare with a pre-built function. In the dare after that the user had to do it from scratch. We repeated the same pattern for functions with arguments. Finally there were two dares with the canvas, first to get a taste of real-world programming, and then to apply functions again. We also made a second set of dares in case some finished the first set early, which contained mostly console dares.

After the experiment, we asked the children to fill out a questionnaire, because this allowed us to get quite a bit of additional information without costing much time [33]. First we had some scaled questions in which the participant had to circle a number from 1 to 5. These questions were to get some indication of how they perceived the course. Then there were four open questions, asking what they did and did not like about the course, what they would want to learn in a programming course, and if they had any suggestions to make the course better. After filling out the questionnaire we had a brief discussion (5 minutes) of what they thought of the experiment.

5.1.2 Results

First we discuss the observations we made during the experiment. The children seemed to enjoy our course, and some even got through the entire set of dares we prepared for the hour, and started with the ‘backup dares’ already. The *Code Golf* method of limiting the length of programs also seemed to work. With the function dares they sometimes started out writing out the program like they used to do, and then discovered it was too long. Then they would *stop and think* about how they could make it shorter, and then they would use functions.

We also discovered some bugs, the most notable of which was when the text size in the browser was decreased. This introduced some major interface glitches. We did not discover the reason for the bug during the experiment, but luckily the interface was still usable enough to continue. Another thing we found out while observing, was that it is useful for teachers to have some indication of which dare the children are doing. For example, we could always show the name of the dare on the screen, which is now hidden when selecting another tab.

One interesting observation was that the children did not really use stepping, highlighting and manipulation much, even after having explicitly learned about them. This is disappointing, since we hoped that these tools may help the learning process. However, we also noted that in cases where it was very useful to use manipulation, such as with the canvas dares at the end, some started using it, and others also used it after suggesting this to them. We occasionally had to point out things or give hints, such as what might be wrong with the program (*‘Try clicking the error icon’*), how to use functions (*‘How did you use functions for in the previous dares?’*).

Another observation was that a lot of children still tried to simulate the program in their head, counting the number of squares the robot had to move, while they could easily have tried it right away. One child asked if it is possible to include a grid on the canvas for easier placement of shapes, while he had no problem placing the shapes using the manipulation tool.

We also found that they had quite a few problems with the first dare that introduced functions (*Knight Jump*), as

Table 5.1: Completion time of dares

dare	type	# finished dares	average time ^a	S.D. ^b
Stepping	introduction	6 ^c	2.56	1.48
Another wall	introduction	7	2.89	2.00
Highlighting	introduction	7	2.08	1.03
Multiple goals	introduction	7	3.11	1.68
Manipulation	introduction	7	5.53	3.59
Knight jump	functions	7	11.43	6.30
Zig-zag	functions	6	15.42	9.32
ForwardRight	functions	5	3.90	2.33
More functions	functions	4	6.50	3.85
Animal	canvas	3	6.48	3.61
Zoo	canvas	2	8.68	5.80
Hello world!	second set of dares	2	5.57	4.06

^a Times in minutes^b Standard deviation^c One machine crashed at the beginning, and the timing data about the first dare was lost.**Table 5.2:** Questionnaire quantitative questions

question	average ^a	S.D. ^b
Do you think that you learned some programming today?	3.71	1.91
Do you think that learning programming is useful?	4.57	1.69
Did you like learning programming with ‘jsdare’?	4.14	1.77
Would you like to do this course at school?	3.71	1.58
Would you recommend this course to your friends?	4.14	1.69
Do you think the course was logical?	4.00	1.69
Do you think the information in the course was helpful?	4.14	1.6

^a Participants answered on a scale from 1 (*‘not at all’*) to 5 (*‘absolutely’*).^b Standard deviation

many seemed to fail to see the repetition in the maze at first. Although the subsequent dares also took a lot of time, it looked like they understood the repetition better. In Table 5.1 it looks like after introducing functions the completion times got better, but we have to note that those who actually finished more dares did better on the dares in which we introduced functions as well, as they all had the same time limit of one hour in total. Still, even if we look at the individual data as seen in Appendix B, it looks like the subsequent dares in which we introduce function arguments take less time.

Then the questionnaire. A summary of the quantitative results can be found in Table 5.2. Unfortunately there are no anomalies that we can use for further developing the course. The results are predominantly positive, but this is meaningless without a control group. In questionnaires the results are usually positively biased [33].

The qualitative part of the questionnaire was more useful. Some representative positive comments were: *‘It’s very simple to use and you get into it very quickly. Also, having the graphics on the side was very nice.’* *‘You used what you learned to create different things, it felt more like a game because of the competition element.’* There were also some critiques: *‘It did not give the “best”/“shortest” answer, so you couldn’t learn from your mistakes.’* *‘I thought that towards the end it started to become tedious.’* *‘It was getting too hard.’* When asked what they would learn in a programming course, three answered ‘games’, two ‘actual robots’, and one ‘search engine’. Suggestions for improvement included: examples of how this is used, easier instructions, harder assignments for experienced students, more initial guidance, and more variation in the dares. In the discussion we had after the questionnaire they also suggested showing the maximum amount of points for each dare, more clarity in how points are awarded, tracks behind the robot instead of lines, and marking visited goals.

5.1.3 Conclusions

We can cautiously call the user evaluation experiment successful. The zero-response-time compiler seemed to invite experimentation, and the limited length of programs caused some children to stop and think at the function dares. We discovered a number of bugs, which we will fix for the final release. We also got some suggestions that are simple to implement and may contribute to the usability, such as showing which goal squares were visited, and designing dares with some more variety. The fact that when asked what they wanted to learn in a programming course, the most heard answer was making games, confirms that we are on the right track. Two students wanted to do programming with physical robots, which we already suggested would be perfect to do alongside this course (Section 2.2.4). Finally, they all seemed to enjoy it, and had no problems to get started.

The fact that they did not use the interface elements such as stepping and highlighting often is slightly disappointing, but also understandable since the programs were not yet complex enough to do extensive debugging. The children did seem to like using the manipulation button with the canvas dares, which is consistent with the notion that for most dares the extensive features were simply not useful enough. We did not test going back in time at all, since we did not introduce them to events. The fact that they did not always experiment with the program but instead reasoned about it in their heads, is also hard to judge. On one hand this seems to be a failure of the interface, while at the other hand it may indicate some kind of transfer of knowledge, of trying to understand how it works. Or it may simply be that this is how they are used to think, and that after working with this interface for a longer period of time, this will change.

The complaint that we did not show the ‘best’ answer, is a tough one. When discussing with the children after the questionnaire, they all thought it would be a good idea to show this. However, this is almost impossible to do, especially when we let users generate their own dares. Finding the shortest program is a problem that can only be solved by (intelligently) trying out all possible programs, which does not scale. Relying on a human to is prone to mistakes, and also does not scale. An elegant solution, we think, is not to try to display the *best* score, but simply a *good* score. When a user creates a dare, we can use the score that the reference program gets as this ‘good’ score. It is then possible to get a score below this good score, for example when the program is slightly longer than the reference program, or if the image matching is not perfect. However, it could also be possible to do better than the reference program, simply by writing a shorter program. The extra points can then be seen as bonus points, while the students would get some indication of how well they are doing.

It was still necessary to give some help every now and then. Especially when getting started with functions, we gave some guidance on what to try, and where to look. Besides this we did not tell them in person what a function is exactly. This confirms that it is still important to have a teacher, be it more for guidance and coaching than standing in front of a whiteboard and explaining things. We still believe, however, that it is paramount to give some explanation of general concepts, after having tried them in practice. This can deepen understanding of the concepts. We think that this is the right order: first learn why you need it, by encountering the problem in practice; then learning how to use it; and only then learning about the general principle. Because of the limited role of the teacher, large parts of the course could be done at home. Social networking functions for student-student and student-teacher interaction could help as well.

5.2 Reflection

In this section we reflect on what we did, and provide pointers for future research. First our successes. We have successfully implemented a programming course using both old and new ideas. Our programming language `js--` is indeed simple, real-world, and mostly imperative. It is a subset of Javascript, still with functionality to access objects, but with syntax for creating objects and inheritance removed. We have indeed implemented a zero-response-time compiler, many interactive features from *Inventing on Principle*, and debugging tools as seen in the *Omniscient Debugger* [87, 55]. We implemented a modern microworld inspired by LOGO and Karel the Robot [69, 70], and we support the HTML `<canvas>` element for making pictures and games [44]. We have introduced a form of Code Golf adapted to programming exercises [1]. Our exercises are problem-driven, and force users to stop and think when their program does not fit the allowed number of lines. A comprehensive reference manual is included inside the course, and can be adapted to the current exercise.

Next we discuss a number of ideas for future research in detail.

5.2.1 Program performance

In our current implementation a lot of information is stored for each run (Chapter 4). All the steps are stored in order to make the step buttons work, calls are stored inside the output tabs, and scope and command information is stored by the respective trackers. All these calls to store this information make up a large part of the performance overhead. More memory is allocated which can cause operating systems to swap other programs to disk more easily. Moreover, since Javascript is garbage collected, an additional performance penalty is issued in the form of slower garbage collection.

All this overhead is created while most of the time it is completely unnecessary. Only when actually using stepping or highlighting it is necessary to generate all this information. Only information about the state before and after each event needs to be stored in order to be able to go back in time and resume from an earlier event. It would therefore be useful to not store all this information when running normally, especially when playing a game, and only to generate this information when highlighting or stepping. An even better implementation would allow for abstraction over time (which is essentially highlighting) while only storing information needed to highlight the abstracted functions. This would allow for efficient manipulation of values while abstraction over time is used.

Another performance improvement would be to use web workers [43]. This is a novel browser API supported by a number of web browsers, which allows for a separation between the user interface and background tasks. Normally the user interface blocks when Javascript is running, thus greatly limiting the amount of time we can spend running programs. In our current implementation the time limit is set so that it still runs relatively smoothly on older computers, while on newer computers the extra performance is not being utilised. When using web workers we could increase the time a program can run, which would allow users with faster computers to run more complicated programs, while on older computers these more complicated programs would run rather slow, but at least the interface would still be usable while waiting for the program to finish. When every run blocks the entire interface, the time limit has to be very low because otherwise typing text would be very slow and thus annoying. If the interface is not blocked, the time limit can be increased to one or two seconds for older computers, which still maintains a relatively direct connection between the code and the result, and is not annoying because the user can still type during computation.

We have found that the CSS animations we use for the restart button and the pause button (Section 3.7) can cause a lot of computations to be made. Even on modern computers this can use more than 30% of the power of one CPU core, just for a simple animation. While this should eventually be fixed by browser vendors, we can mitigate this problem by using old-fashioned GIF animations.

5.2.2 Social features

On the requirements list there were a number of requirements that have to do with online interactions. One option would be to integrate the course with existing social networks, such as Facebook and Twitter [4, 6]. This would allow students to share their accomplishments with their friends, and teachers to share exercises they have made. Another feature would be to include discussion boards per dare, or per collection of dares. This way students can help each other if they are struggling with some exercises. Games could be assessed by asking other students and teachers to rate them or comment on them. Gamification could be included, which means that students could get achievements, badges, and ribbons for their achievements. For schools it would be useful to have provisions for classroom teaching, such as manual assessment by teachers or classmates, an integrated grade centre, monitoring tools for teachers, and private discussion boards.

5.2.3 Types of exercises

At the moment the only type of exercises we use are dares, exercises in which the program has to be as few lines as possible. It is possible to design different types of exercises, and also different types of dares themselves. If we start with looking at dares, there are two main types, the robot goal type in which visiting some goal squares is the objective, and the matching type in which the console or canvas output has to match the reference output. It is also possible to make a type of dare which would call a number of functions, and compare the results with a reference output, like unit testing. It can be done by comparing the return value of a function, or by looking at the

output of one of the output tabs after calling a function. The reference output can also be hidden, which makes the challenge of creating a good program harder.

We could also have exercises that do not limit the length of the program, but some other attribute. An obvious metric would be the number of steps, since this is a direct measure of performance. Then there are other, more complicated metrics that could be used, such as cyclomatic complexity. Or a constraint on certain types of statements, such as a maximum number of loops, or a minimum number of array declarations. Most importantly, we must make sure that we can visualise these metrics, and that they are relatively easy to explain.

Another type of exercise would be to create a program that plays some kind of game against an opponent. The opponent can then be a random other player who also programs his or her solution live, or a tournament style set up in which your game plays against a lot of opponents, or just a predefined opponent. Points can then be awarded based on how well the program does. This can be measured by awarding points for completing certain objectives in the game. It is not necessary that it is a zero-sum game, in fact, it would be interesting to explore cooperative games.

5.2.4 Interface features

Highlighting is already quite powerful, but can be improved on further. For example, in-line documentation can be shown when highlighting a line, with a link to the built-in reference manual for more information.

At the moment it is possible to manipulate booleans, numbers, and strings that contain a valid CSS colour representation. It would be interesting to explore other possibilities. An example would be strings that contain a canvas font definition, for which a specialised font editor could be shown. Or for more complex canvas calls a helpful visualisation could be shown. For expressions a graphical formula editor could be shown. For simple for loops a tool could be used which allows to quickly explore other types of iteration, such as counting down instead of up, or in certain steps.

In Section 3.9 we discussed the problems with including images in programs. Luckily, images can be represented in Javascript as strings, using base 64 encoding. This means that we could make an integrated image editor, which would support editing images in place, and uploading of existing images. There are still some problems, as the API for base 64 encoding is still somewhat difficult, and the length of strings is currently limited by the maximum width of a line. Another option would be to provide a fixed set of images, and integrate an image browser, such as done by *Khan Academy* [8].

At the moment we visualise variables using a list of scopes, in which each variable is shown using just its last value. There are many, many more methods of visualising variables and their values [71, 64]. We could incorporate some of the most effective visualisations into the course.

In order to enhance real-world applicability of the learned skills, it would be nice to be able to export the Javascript code written in our course to a self-contained web page. This way students can see what their code would look like outside of our editor. In order to do this we would need to make the robot tab into an independent library. Perhaps the same should be done with the console, so that users can keep using the `console.clear()` and `console.setColor()` functions.

5.2.5 Experimental research

We have only done a small-scale field study, and a simple questionnaire. In order to determine whether our course improves achievement and motivation, a much larger experiment could be done. Such an experiment should test different parts of the course individually, such as the robot environment; dares; highlighting and manipulation; pausing and abstracting over time; and debugging. This should be done with a control group that learns similar programming concepts.

If the course is published, we could gather data to see which dares are difficult or easy, just as we did in our experiment. We can measure the time it takes users to complete each dare, and how much time they spend on it afterwards, for example to improve their score. We can also try to group different solutions for each dare using tree comparison algorithms, to gain insight how users learn.

'Program or be programmed.'
— Douglas Rushkoff

Chapter 6

Conclusions

In this thesis we have presented a website for learning programming, aimed at high school children, essentially a programming course. In this course we have brought together a number of user interface ideas into one system. The danger when attempting a project with such a wide scope is that you can end up attempting to do a lot of different things, but doing none of them very well. We feel that even though we attempted to bring together a relatively wide variety of ideas, the end result is a coherent integration.

Giving students the opportunity to freely play with programming by making games and other interesting programs, can be very motivating for them, and a great way to learn basic programming. The popularity of visual programming languages and LEGO Mindstorms kits confirms this. But when it comes to real-world programming, it is much more harder to explore what is possible. We believe that by using our method of dares one can teach programming while allowing students to explore creatively within the bounds of the given exercise. This way, students can build deep knowledge and intuition before explicitly being told about the syntax and strategies. We think this can be much more motivating than telling them what to do in a conventional way. Instead it is a challenge of solving problems, and learning what is needed to solve these problems on the way.

While this problem-driven approach requires much less work from a teacher, we believe that it is still important for teachers to lecture about formal computer science concepts afterwards. This is important, both for grounding their newly gained skills in explicit knowledge, and for communication, as it is hard to talk about ideas without knowing their names. Along the way the knowledge of students can be tested by having them designing simple games, a highly motivating and open-ended way of learning. When making games and driving robots around in certain patterns, students will discover that there are certain rules governing these computations. This is a great way of applying mathematics and physics to practical and motivating problems.

The interface that we have presented enables very fast exploration, allowing students to get a feel for how certain computations behave. The stepping and highlighting functions might greatly contribute to the mental model students have of how a program is executed. While developing the interface, we ourselves discovered that it was often very useful to quickly try some algorithm, and then play with numbers to get an idea of how it behaves, for example to test the deterministic random number generator. The information tab can be quickly used to look up syntax and semantics, and the scope visualisation can give even more insight in the way a program is executed.

We believe that a basic understanding of programming can be empowering, not only to children, but to everyone. It is a skill that is becoming more important in our society every day, as our dependency on machines grows. But there is also a deeper importance. Knowing what is going on inside all these machines allows people to understand what is and what is not possible; the benefits of technology, the dangers of technology. The deep connection between our machines, universal truths, beautiful nature, and ultimately, ourselves. This is something that should be taught at every school, at every level. We are grateful and proud to have made a humble contribution.

Acknowledgements

There were many people who contributed to this project in one way or another. First and foremost, my supervisor, Irina Voiculescu. You helped me focus when needed, and gave a lot of suggestions, most importantly the idea to use a maze in which a robot drives around. Also thanks for helping recruit some children for the evaluation experiment, and not to mention the number of pages you have proofread. Thanks to Suzanna Marsh, who also did a great job in recruiting participants, and helped me out during the day of the experiment. Thanks to the children themselves, who were willing to be ‘at school’ for an hour during their vacation! Thanks to everyone who has taken the time to test my prototypes and read my drafts, sometimes leading to long and interesting discussions. In particular I would like to thank Mark IJbema, Roan Kattouw, Trevor Parscal, Niluka Satharasinghe, Alex Bolton, Jan Kazemier, Frans Schreuder, Nathan Mol, and Femke Hoornveld. For the Requirements exam in Hilary term I interviewed four teachers about programming education, which I used as a starting point for this thesis. A special thanks to you, as I have directly used your ideas when designing my programming course. For the same reason I would like to thank everyone who has published ideas which I have used in this project. I would like to say thanks to my friends in Oxford for a great year — I hope we will see each other soon; and to my friends in Groningen, especially the awesome volunteers at Science Center North, who give children with interests in science and programming a place to be themselves and explore their interests, without actually having the financial resources to do so. Thanks to my parents and grandparents for lots of love and support, allowing me to pursue my dreams. And finally, many thanks to Femke for enduring my being away for a year — I’ll be home soon!

Bibliography

- [1] Code golf. <http://codegolf.com/>. Retrieved August 17, 2012.
- [2] Codecademy. <http://www.codecademy.com/>. Retrieved August 17, 2012.
- [3] Codeschool. <http://www.codeschool.com/>. Retrieved August 17, 2012.
- [4] Facebook. <http://www.facebook.com/>. Retrieved August 17, 2012.
- [5] How many new scratch users register each month?
<http://stats.scratch.mit.edu/community/usersbytime.html>. Retrieved August 17, 2012.
- [6] Twitter. <http://www.twitter.com/>. Retrieved August 17, 2012.
- [7] Pedagogical changes in the delivery of the first-course in computer science: Problem solving, then programming. *Journal of Engineering Education*, 87:313–320, July 1998.
- [8] Khan Academy. Khan academy computer science.
<http://www.khanacademy.org/about/blog/post/29417655743/computer-science>, August 2012. Retrieved August 17, 2012.
- [9] Benedikte S. Als, Janne J. Jensen, and Mikael B. Skov. Comparison of think-aloud and constructive interaction in usability testing with children. In *Proceedings of the 2005 conference on Interaction design and children*, IDC '05, pages 9–16, New York, NY, USA, 2005. ACM.
- [10] Jeff Atwood. Please don't learn to code.
<http://www.codinghorror.com/blog/2012/05/please-dont-learn-to-code.html>, May 2012. Retrieved August 17, 2012.
- [11] B. Barbara Lockee, M. Moore, and J. Burton. Measuring success: Evaluation strategies for distance education. *Educause Quarterly*, November 2002.
- [12] Daniel Baulig. High performance ecmascript und html5 canvas, April 2011.
- [13] Robert M. Bernard, Philip C. Abrami, Eugene Borokhovski, C. Anne Wade, Rana M. Tamim, Michael A. Surkes, and Edward Clement Bethel. A meta-analysis of three types of interaction treatments in distance education. *Review of Educational Research*, 79(3):1243–1289, 2009.
- [14] Michael Bloomberg. Tweet. <https://twitter.com/MikeBloomberg/status/154999795159805952>, January 2012. Retrieved August 17, 2012.
- [15] P. Brusilovsky, A. Kouchnirenko, P. Miller, and I. Tomek. Teaching programming to novices: A review of approaches and tools. In *Proceedings of ED-MEDIA 94—World Conference on Educational Multimedia and Hypermedia*, 1994.
- [16] Mary Elaine Califf and Mary Goodwin. Testing skills and knowledge: introducing a laboratory exam in cs1. *SIGCSE Bull.*, 34(1):217–221, February 2002.
- [17] Zach Carter. Jison. <http://zaach.github.com/jison/>, 2010. Retrieved August 17, 2012.
- [18] Alex Qiang Chen. Web evolution. Master's thesis, University of Manchester, 2008.

- [19] Matthew Conway, Steve Audia, Tommy Burnette, Dennis Cosgrove, and Kevin Christiansen. Alice: lessons learned from building a 3d system for novices. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, CHI '00, pages 486–493, New York, NY, USA, 2000. ACM.
- [20] Douglas Crockford. *JavaScript: The Good Parts*. O'Reilly Media, Inc., 2008.
- [21] Cascading style sheets. Collection of Working Drafts. Retrieved August 17, 2012.
- [22] Sebastian Deterding. Gamification: designing for motivation. *interactions*, 19(4):14–17, July 2012.
- [23] L. Doherty and V. Kumar. Teaching programming through games. In *Technology for Education, 2009. T4E '09. International Workshop on*, pages 111–113, August 2009.
- [24] Christopher Douce, David Livingstone, and James Orwell. Automatic test-based assessment of programming: A review. *J. Educ. Resour. Comput.*, 5(3), September 2005.
- [25] E. Drever. *Using semi-structured interviews in small-scale research: a teacher's guide*. Using research series. Scottish Council for Research in Education, 2003.
- [26] Albrecht Ehlert and Carsten Schulte. Empirical comparison of objects-first and objects-later. In *Proceedings of the fifth international workshop on Computing education research workshop*, ICER '09, pages 15–26, New York, NY, USA, 2009. ACM.
- [27] John English. Experience with a computer-assisted formal programming examination. *SIGCSE Bull.*, 34(3):51–54, June 2002.
- [28] M.C. Escher. *M.C. Escher: The Graphic Work: Introduced and Explained by the Artist*. Taschen Basic Art. Taschen America LLC, 2001.
- [29] M. Feldgen and O. Clua. Games as a motivation for freshman students learn programming. In *Frontiers in Education, 2004. FIE 2004. 34th Annual*, volume 3, pages S1H/11–S1H/16, October 2004.
- [30] Free Software Foundation. Bison — gnu parser generator. <http://www.gnu.org/software/bison/>. Retrieved August 17, 2012.
- [31] Free Software Foundation. Running programs backward. <http://sourceware.org/gdb/current/onlinedocs/gdb/Reverse-Execution.html>. Retrieved August 17, 2012.
- [32] Frogatto. New features in the frogatto editor. http://www.youtube.com/watch?v=ri614C_Buwg, April 2012. Retrieved August 17, 2012.
- [33] B. Gillham. *Developing a Questionnaire*. Real World Research. Bloomsbury, 2000.
- [34] Google, Inc. Chrome developers tools. <https://developers.google.com/chrome-developer-tools/>, 2012. Retrieved August 17, 2012.
- [35] Google, Inc. Google chart tools. <https://developers.google.com/chart/>, 2012. Retrieved August 17, 2012.
- [36] Michael Gove. Speech at the british educational training and technology show. <http://www.education.gov.uk/inthenews/speeches/a00201868/michael-gove-speech-at-the-bett-show-2012>, January 2012. Retrieved August 17, 2012.
- [37] Chris Granger. Light table — a new ide concept. <http://www.chris-granger.com/2012/04/12/light-table---a-new-ide-concept/>, April 2012. Retrieved August 17, 2012.
- [38] Sacha Greif. Please learn to code. <http://sachagreif.com/please-learn-to-code/>, May 2012. Retrieved August 17, 2012.
- [39] Patricia Haden. The incredible rainbow spitting chicken: teaching traditional programming skills through games programming. In *Proceedings of the 8th Australasian Conference on Computing Education - Volume 52*, ACE '06, pages 81–89, Darlinghurst, Australia, Australia, 2006. Australian Computer Society, Inc.

- [40] Libby Hanna, Kirsten Ridsen, and Kirsten Alexander. Guidelines for usability testing with children. *interactions*, 4(5):9–14, September 1997.
- [41] Alan Harris. Performance tuning, configuration, and debugging. In *Pro ASP.NET 4 CMS*, pages 229–256. Apress, 2010.
- [42] Ken Hartness. Robocode: using games to teach artificial intelligence. *J. Comput. Sci. Coll.*, 19(4):287–291, April 2004.
- [43] Ian Hickson. Html living standard. Web Hypertext Application Technology Working Group (WHATWG), <http://www.whatwg.org/specs/web-apps/current-work/>. Retrieved August 17, 2012.
- [44] Ian Hickson. Html5: A vocabulary and associated apis for html and xhtml. W3C Working Draft, <http://www.w3.org/TR/2011/WD-html5-20110525/>. Retrieved August 17, 2012.
- [45] Ian Hickson. Html canvas 2d context. Editor’s Draft, <http://dev.w3.org/html5/2dcontext/>, March 2012. Retrieved August 26, 2012.
- [46] Daniel Hooper. Codebook. <http://danielhooper.tumblr.com/post/19313911658/codebook>, March 2012. Retrieved August 17, 2012.
- [47] Petri Ihantola, Tuukka Ahoniemi, Ville Karavirta, and Otto Seppälä. Review of recent systems for automatic assessment of programming assignments. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*, Koli Calling ’10, pages 86–93, New York, NY, USA, 2010. ACM.
- [48] jQuery Foundation John Resig. jquery. <http://jquery.com/>, 2012. Retrieved August 17, 2012.
- [49] Randolph M. Jones. Design and implementation of computer games: a capstone course for undergraduate computer science education. In *Proceedings of the thirty-first SIGCSE technical symposium on Computer science education*, SIGCSE ’00, pages 260–264, New York, NY, USA, 2000. ACM.
- [50] Caitlin Kelleher and Randy Pausch. Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Comput. Surv.*, 37(2):83–137, June 2005.
- [51] E Koutsofios and S North. Drawing graphs with dot. *ATT Bell Laboratories*, 1993.
- [52] Douglas Kranch. Teaching the novice programmer: A study of instructional sequences and perception. *Education and Information Technologies*, 17:291–313, 2012.
- [53] Pamela B. Lawhead, Michaele E. Duncan, Constance G. Bland, Michael Goldweber, Madeleine Schep, David J. Barnes, and Ralph G. Hollingsworth. A road map for teaching introductory programming using lego mindstorms robots. In *Working group reports from ITiCSE on Innovation and technology in computer science education*, ITiCSE-WGR ’02, pages 191–201, New York, NY, USA, 2002. ACM.
- [54] Scott Leutenegger and Jeffrey Edgington. A games first approach to teaching introductory programming. *SIGCSE Bull.*, 39(1):115–118, March 2007.
- [55] Bil Lewis. Debugging backwards in time. In *Fifth International Workshop on Automated Debugging*. arXiv, September 2003.
- [56] Olav Andreas Lindekleiv. jquery ui color picker widget, 2012.
- [57] Ian Livingstone and Alex Hope. Next gen. Technical report, Nesta. Retrieved August 17, 2012.
- [58] Noel Llopis. Lechimp vs. dr. chaos. <http://gamesfromwithin.com/lechimp-vs-dr-chaos>. Retrieved August 17, 2012.
- [59] Ju Long. Just for fun: Using programming games in software programming training and education — a field study of ibm robocode community. *Journal of Information Technology Education*, 6, 2007.
- [60] John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. The scratch programming language and environment. *Trans. Comput. Educ.*, 10(4):16:1–16:15, November 2010.
- [61] George Marsaglia. Xorshift rngs. *Journal of Statistical Software*, 8(14):1–6, 7 2003.
- [62] Richard E. Mayer. The psychology of how novices learn computer programming. *ACM Comput. Surv.*, 13(1):121–141, March 1981.

- [63] Bill Mill. Canvas tutorial. <http://billmill.org/static/canvastutorial/>. Retrieved August 17, 2012.
- [64] Thomas L. Naps, Guido Rößling, Vicki Almstrum, Wanda Dann, Rudolf Fleischer, Chris Hundhausen, Ari Korhonen, Lauri Malmi, Myles McNally, Susan Rodger, and J. Ángel Velázquez-Iturbide. Exploring the role of visualization and engagement in computer science education. *SIGCSE Bull.*, 35(2):131–152, June 2002.
- [65] Mozilla Developer Network. Console api reference. <https://developer.mozilla.org/en/DOM/console>. Retrieved August 17, 2012.
- [66] Jakob Nielsen. *Usability Engineering*. Morgan Kaufmann Series in Interactive Technologies. AP Professional, 1994.
- [67] International Institute of Business Analysis, K. Brennan, and International Institute of Business Analysis Staff. *A Guide to the Business Analysis Body of Knowledge (BABOK Guide), Version 2.0*. IT Pro. International Institute of Business Analysis, 2009.
- [68] M. Overmars. Teaching computer science through game design. *Computer*, 37(4):81–83, April 2004.
- [69] Seymour Papert. *Mindstorms: children, computers, and powerful ideas*. Basic Books, Inc., New York, NY, USA, 1980.
- [70] Richard E. Pattis. *Karel the Robot: A Gentle Introduction to the Art of Programming*. John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 1981.
- [71] Arnold Pears, Stephen Seidman, Lauri Malmi, Linda Mannila, Elizabeth Adams, Jens Bennedsen, Marie Devlin, and James Paterson. A survey of literature on the teaching of introductory programming. In *Working group reports on ITiCSE on Innovation and technology in computer science education, ITiCSE-WGR '07*, pages 204–223, New York, NY, USA, 2007. ACM.
- [72] D. N. Perkins and Fay Martin. Fragile knowledge and neglected strategies in novice programmers. In *Papers presented at the first workshop on empirical studies of programmers on Empirical studies of programmers*, pages 213–229, Norwood, NJ, USA, 1986. Ablex Publishing Corp.
- [73] Robert Pierce and Manuela Aparício. Resources to support computer programming learning and computer science problem solving. In *Proceedings of the Workshop on Open Source and Design of Communication, OSDOC '10*, pages 35–40, New York, NY, USA, 2010. ACM.
- [74] Jan Paul Pasma. Requirements. Unpublished report made for the Requirements exam at Oxford University, 2012.
- [75] Lloyd P. Rieber. Animation in computer-based instruction. *Educational Technology Research and Development*, 38(1):77–86, 1990.
- [76] Anthony Robins, Janet Rountree, and Nathan Rountree. Learning and teaching programming: A review and discussion. *Computer Science Education*, 13(2):137–172, 2003.
- [77] Eric Schmidt. Mactaggart lecture. <http://www.guardian.co.uk/media/interactive/2011/aug/26/eric-schmidt-mactaggart-lecture-full-text>, August 2011. Retrieved August 17, 2012.
- [78] Zed Shaw. Please don't become anything, especially not a programmer. http://learncodethehardway.org/blog/MAY_15_2012.html, May 2012. Retrieved August 17, 2012.
- [79] Michael Spivey. Geomlab. <http://www.cs.ox.ac.uk/geomlab>. Retrieved August 17, 2012.
- [80] Elizabeth Sweedyk, Marianne deLaet, Michael C. Slattery, and James Kuffner. Computer games and cs education: why and how. In *Proceedings of the 36th SIGCSE technical symposium on Computer science education, SIGCSE '05*, pages 256–257, New York, NY, USA, 2005. ACM.
- [81] Steven L. Tanimoto. Viva: A visual language for image processing. *Journal of Visual Languages & Computing*, 1(2):127–139, 1990.

- [82] Barbara Tversky, Julie Bauer Morrison, and Mireille Betrancourt. Animation: can it facilitate? *International Journal of Human-Computer Studies*, 57(4):247–262, 2002.
- [83] Twitter. Bootstrap. <http://twitter.github.com/bootstrap/>, 2012. Retrieved August 17, 2012.
- [84] UXMovement. Modal windows best practices. <http://uxmovement.com/forms/modal-windows-best-practices/>, 2011. Retrieved August 17, 2012.
- [85] Willem van der Vegt. The codecup. *International Olympiad in Informatics workshop*, 2006.
- [86] Bret Victor. Up and down the ladder of abstraction — a systematic approach to interactive visualization. <http://worrydream.com/LadderOfAbstraction/>, October 2011. Retrieved August 17, 2012.
- [87] Bret Victor. Inventing on principle. CUSEC, 2012.
- [88] Cedric Vivier. Live scratchpad. <http://neonux.github.com/LiveScratchpad/>. Retrieved August 17, 2012.
- [89] W3Techs. Usage of javascript for websites. <http://w3techs.com/technologies/details/cp-javascript/all/all>. Retrieved August 17, 2012.
- [90] Denise Woit and David Mason. Effectiveness of online assessment. In *Proceedings of the 34th SIGCSE technical symposium on Computer science education*, SIGCSE '03, pages 137–141, New York, NY, USA, 2003. ACM.
- [91] Wolfram Research, Inc. Interactive manipulation. <http://reference.wolfram.com/mathematica/guide/InteractiveManipulation.html>. Retrieved August 17, 2012.

Appendix A

Language

name	regular expression
digit	[0-9]
alpha	[a-zA-Z_]
alphanum	[0-9a-zA-Z_]
whitespace	[\f\r\t\v\u00A0\u2028\u2029]+
linecomment	([/][^\\n]*) (#[^\\n]*)([/][^\\n]*)
multicomment	([/][^\\n]*)([/][^\\n]*)
skip	{whitespace} {linecomment} {multicomment}
newlines	{skip}*([\\n]{skip})*+
fraction	"."{digit}+
exponent	[eE][+-]?{digit}+
number	(([1-9]{digit}*) "0"){fraction}?{exponent}?
string	["][^\\n]*([\\n][^\\n]*)*["]
reserved	(("undefined" "null" "break" "case" "catch" "default" "finally" "instanceof" "new" "continue" "void" "delete" "this" "do" "in" "switch" "throw" "try" "typeof" "with" "abstract" "boolean" "byte" "char" "class" "const" "debugger" "double" "enum" "export" "extends" "final" "float" "goto" "implements" "import" "int" "interface" "long" "native" "package" "private" "protected" "public" "short" "static" "super" "synchronized" "throws" "transient" "volatile" "arguments" "NaN" "Array" "Object" "RegExp" "toString")(!{alphanum})) ("jsmm"{alphanum}*)
invalid	"' "~" ">>=" "<<=" ">>" "<<" " "&" "===" "!== "?" :" "\$" "\""
skip	{whitespace} {linecomment}
plusis	"+=" "-= "*= " /= "%=
compare	"==" "!=" ">=" "<=" ">" "<"
plusmin	"+" "-"
mult	"*" " /" "%"

Table A.1: Lexer rules

Jison first tokenises the input by using a number of lexer rules (Table A.1). Then it parses the input using the following parser rules (in *Backus-Naur Form*):

```

<program> ::= <programStatementList> <eof>
           | '\n' <programStatementList> <eof>

<programStatementList> ::= <empty>
           | <programStatementList> <commonStatement> '\n'
           | <programStatementList> <functionDeclaration> '\n'

```

```

<statementList> ::= <empty>
| <statementList> <commonStatement> '\n'

<commonStatement> ::= <simpleStatement> ';'
| <blockStatement>
| <returnStatement>

<simpleStatement> ::= <assignmentStatement>
| <varStatement>
| <callExpression>
| <identExpression> <plusmin> <plusmin>

<assignmentStatement> ::= <identExpression> '=' <expression>
| <identExpression> <plusis> <expression>

<varStatement> ::= 'var' <varList>

<varList> ::= <varListItem>
| <varList> ',' <varListItem>

<varListItem> ::= <name>
| <name> '=' <expression>

<returnStatement> ::= 'return' ';'
| 'return' <expression> ';'

<expression> ::= <andExpression>
| <expression> '||' <andExpression>

<andExpression> ::= <relationalExpression>
| <andExpression> '&&' <relationalExpression>

<relationalExpression> ::= <addExpression>
| <relationalExpression> <compare> <addExpression>

<addExpression> ::= <multExpression>
| <addExpression> <plusmin> <multExpression>

<multExpression> ::= <unaryExpression>
| <multExpression> <mult> <unaryExpression>

<unaryExpression> ::= <primaryExpression>
| <plusmin> <unaryExpression>
| '!' <unaryExpression>

<primaryExpression> ::= <literal>
| <identExpression>
| <callExpression>
| <arrayDefinition>
| '(' <expression> ')

<literal> ::= <number>
| <string>
| 'true'
| 'false'

<identExpression> ::= <name>
| <identExpression> '.' name
| <identExpression> '[' <expression> ']'

<callExpression> ::= <identExpression> '(' ')'
| <identExpression> '(' callArguments ')'

<callArguments> ::= <expression>
| callArguments ',' <expression>

```

```

<arrayDefinition> ::= '[' '['
  | '[' <arrayList> '['
<arrayList> ::= <expression>
  | <arrayList> ',' <expression>
<blockStatement> ::= <ifBlock>
  | <whileBlock>
  | <forBlock>
<ifBlock> ::= 'if' '(' <expression> ')' '{' '\n' <statementList> '}' elseBlock
<elseBlock> ::= <empty>
  | 'else' <ifBlock>
  | 'else' '{' '\n' <statementList> '}'
<whileBlock> ::= 'while' '(' <expression> ')' '{' '\n' <statementList> '}'
<forBlock> ::= 'for' '(' <simpleStatement> ';' <expression> ';' <simpleStatement> ')' '{' '\n' <statementList> '}'
<functionDeclaration> ::= 'function' <name> '(' ')' '{' '\n' <statementList> '}'
  | 'function' <name> '(' functionArguments ')' '{' '\n' <statementList> '}'
<functionArguments> ::= <name>
  | <functionArguments> ',' <name>

```


Appendix B

User evaluation experiment data

Table B.1: Questionnaire scores

#	Age	Learned	Useful	Like	At school	Recommend to friends	Logical	Helpful information
1	16	1	5	4	4	4	4	4
2	13	3	5	4	4	3	5	4
3	13	5	5	5	4	5	5	5
4	13	3	4	2	2	5	3	4
5	15	5	4	5	4	4	5	4
6	15	4	4	4	3	3	3	3
7	9	5	5	5	5	5	3	5
Average		3.71	4.57	4.14	3.71	4.14	4.00	4.14
S.D. ^a		1.91	1.69	1.77	1.58	1.69	1.69	1.6

Participants answered on a scale from 1 (*'not at all'*) to 5 (*'absolutely'*).

^a Standard deviation

Table B.2: Questionnaire answers (part 1)

#	<i>'What did you like about the course?'</i>	<i>'What did you not so much like about the course?'</i>
1	It's very simple to use and you can into it very quickly. Also, having the graphics on the side was very nice.	No complaints
2	Well laid out, quite fun.	Website crashed twice. ^a
3	It was good and got progressively harder. I thought the (distance) was good.	I did not like anything for it was all good.
4	I liked to be able to understand how it works	It was getting too hard.
5	You used what you learned to create different things, it felt more like a game because of the competition element.	It did not give the 'best'/'shortest' answer so you couldn't learn from your mistakes.
6	I like that there were easy options to copy each set of instructions called functions.	I thought that towards the end it started to become tedious. Perhaps more exciting levels.
7	The playing with programming	Nothing

^a This was already a known bug.

Table B.3: Questionnaire answers (part 2)

#	<i>'What would you most like to learn in a programming course?'</i>	<i>'Do you have any suggestions to make the course better?'</i>
1	How to programme a real robot.	Maybe some more initial guidance as it wasn't immediately clear how you could change the code
2	Programming, preferably.	Add more dares? Add login?
3	I would like to learn to programme something like Tetris.	Maybe put in a harder course for more experience people.
4	How to make games like hang man.	Make the instructions easier to read and understand.
5	How it is used in modern day technology, games, even a real robot e.g. LEGO Mindstorms or a turtle bot	Hands on like LEGO Mindstorms or turtle bot, examples of how it is used.
6	I would like to learn about how search engines are programmed to co-operate with the instructions you are asking it to do.	I think that to make it better there could be more variation in the levels.
7	Perl	A free login account

Table B.4: Completion time of dares (part 1)

#	Stepping	Another wall	Highlighting	Multiple goals	Manipulation	Knight jump
1	^a	70	139	197	593	1004
2	84	91	84	131	117	166
3	116	121	59	183	163	581
4	291	167	188	349	509	457
5	124	153	114	83	169	733
6	132	208	111	193	281	1059
7	175	403	177	169	491	802
Average	153.67	173.29	124.57	186.43	331.86	686
S.D. ^b	88.67	120.04	61.88	100.77	215.49	378.26

^aTimes in seconds.

^a One machine crashed at the beginning, and the timing data about the first dare was lost.

^b Standard deviation

Table B.5: Completion time of dares (part 2)

#	Zig-zag	ForwardRight	More functions	Animal	Zoo	Hello world!
1	1203					
2	1244	157	294	394		483
3	443	115	192	270	345	185
4						
5	309	265	532	503	696	
6	1513	221				
7	839	411	541			
Average	925.17	233.8	389.75	389.00	520.50	334.00
S.D. ^a	559.41	140.07	230.71	216.54	348.00	243.69

^aTimes in seconds.

^a Standard deviation