Graceful Language Extensions and Interfaces

by

Michael Homer

A thesis
submitted to the Victoria University of Wellington
in fulfilment of the
requirements for the degree of
Doctor of Philosophy

Victoria University of Wellington
2014

Abstract

Grace is a programming language under development aimed at education. Grace is object-oriented, imperative, and block-structured, and intended for use in first- and second-year object-oriented programming courses. We present a number of language features we have designed for Grace and implemented in our self-hosted compiler. We describe the design of a pattern-matching system with object-oriented structure and minimal extension to the language. We give a design for an object-based module system, which we use to build dialects, a means of extending and restricting the language available to the programmer, and of implementing domain-specific languages. We show a visual programming interface that melds visual editing (à la Scratch) with textual editing, and that uses our dialect system, and we give the results of a user experiment we performed to evaluate the usability of our interface.

Acknowledgments

The author wishes to acknowledge:

- James Noble and David Pearce, his supervisors;
- Andrew P. Black and Kim B. Bruce, the other designers of Grace;
- Timothy Jones, a coauthor on a paper forming part of this thesis and contributor to Minigrace;
- Amy Ruskin, Richard Yannow, and Jameson McCowan, coauthors on other papers;
- Daniel Gibbs, Jan Larres, Scott Weston, Bart Jacobs, Charlie Paucard, and Alex Sandilands, other contributors to Minigrace;
- Gilad Bracha, Matthias Felleisen, and the other (anonymous) reviewers of papers forming part of this thesis;
- the participants in his user study;
- David Streader, John Grundy, and Laurence Tratt, examiners of the thesis;
- and Alexandra Donnison, Amy Chard, Juanri Barnard, Roma Klapaukh, and Timothy Jones, for providing feedback on drafts of this thesis.

Contents

Chapter 1

Introduction

Grace is an object-oriented imperative programming language in development intended for education. Grace aims to be used in introductory computer science courses and to incorporate accumulated knowledge about teaching while avoiding pitfalls found in contemporary languages commonly used for these courses.

In this thesis we investigate extensions to the Grace language allowing the programmer to express their intentions in a manner suitable to their understanding of the task at hand. For novice programmers complex ideas can be presented simply and departures from the expected actions restricted. For more experienced programmers these extensions provide power within a concise structure. We do not focus on Grace's educational claims, but instead aim to design features of general utility that meet the goals of the language.

The contributions of this thesis are:

- An object-oriented pattern-matching system embedded into the language with minimal syntactic and semantic change. This system includes both a concise and readable syntax for writing pattern-matching expressions and a fully object-oriented design for patterns themselves, incorporating both those patterns built into the language

and user-defined patterns. This contribution was published in the 2012 Dynamic Language Symposium [74].

- A module system based on objects, providing the desirable attributes of modules by building on existing common language semantics. This contribution was published in a paper presented at DYLA 2013 [70].

- A system of *dialects*, language variants able to provide extensions and restrictions of the base language to the user, built using our module system and pattern matching. We show how this system supports both pedagogical sublanguages and powerful domain-specific languages within a single language, and present case studies to evaluate its breadth and power. This contribution was published in a paper at ECOOP 2014 [71].

- A novel editing environment integrating both drag-and-drop visual editing and conventional textual editing of code in one, with bidirectional transition between the two and supporting multiple user-defined dialects. We present the results of a user experiment evaluating our design. This contribution was published in VISSOFT 2013 [72] and 2014 [73].

While the papers cited above have additional authors, the principal work and authorship was by the author of this thesis.

The structure of this thesis is as follows. The next chapter describes related work in all of the contribution areas and programming pedagogy generally. Chapter 3 introduces the Grace language overall. Chapter 4 describes our pattern-matching system and Chapter 5 our module design. Chapter 6 presents our dialect design and a variety of case studies to evaluate it. Chapter 7 shows our visual-textual editing environment and the results of our experiment evaluating the system. Chapter 8 describes Minigrace, our self-hosted compiler for Grace, and Chapter 9 concludes. Within each of the contribution chapters separate sections (Sect. 4.7, Sect. 5.6,

Sect. 6.5, and Sect. 7.6) explicitly contrast and position our work among related work.

For ease of reference the author will use the royal pronouns throughout this thesis.

Chapter 2

Related Work

In this chapter we describe other work related to the contributions of this thesis. This chapter will provide an overview of the related work; each contribution chapter includes a section explicitly contrasting our work with others'. The next section gives an overview of programming education research generally, and the following sections address work in the areas of our contributions. Section 2.2 discusses pattern-matching systems, Section 2.3 describes module systems, Section 2.4 notes research on language variants and domain-specific languages, and Section 2.5 presents other visual programming interfaces.

2.1 Programming language education

Although we do not make claims about Grace's educational suitability, we must be mindful of the nature of the language in our contributions, and place them in the corect context. Here we summarise relevant educational literature and describe past educational programming languages. A broader overview of work on programming languages or tools for novices in general is given in the work of Pears *et al.* [148], while Robins, Rountree, and Rountree provide an overview of work on learning programming [166] and Carter *et al.* [23] summarise work on programming students' motiva-

tion and programme structure.

The most common report of research on teaching programming to novices is that they perform unusually poorly compared to other programmes in the university, often in combination with an unusually high rate of top grades in the same course (a *bimodal* distribution) [107, 165]. The second most common is that the authors have found a miracle cure by using/not using static types, objects, classes, graphics, arithmetic, Java, Python, C++, their new language, or their new tool, and that everyone should immediately switch to this approach; this part of the literature is backed up by anecdotes and limited quantitative studies.

The remainder of this section is broken into three parts: the next addresses research in educational psychology with relevance to programming education, while Section 2.1.2 describes pedagogical approaches, and Section 2.1.3 educational programming languages.

2.1.1 Educational psychology

Educational psychology studies human learning and cognition. Much work has been done in the area with children and undergraduate students in particular, although relatively little on programming language education in itself. Much research in the area has application to teaching in general, including teaching programming, but some concepts in particular stand out as important for programming education: *transfer of learning* is the application of skill learnt in one area to learning another related skill [150, 164], while the distinction between expert and novice problem-solvers is also instructive.

Transfer can be divided along multiple lines [164]. A key division is the distance of transfer: *near transfer* applies learning about one domain to a domain whose essential stimuli are similar to the first, and *far* or *distant transfer* applies concepts to a domain much different than the original. In the context of programming education, moving from one language to

another is likely to be near transfer [142], but the division is contextual. Another division is between *low road* transfer, where skills are applied in the new domain without effort because the skill has been developed to a very high level, and *high road* transfer where explicit connections must be made between concepts [150].

An *expert* in a domain has substantial experience in that area and is able to conceptualise it at a high level, while a *novice* does not have this experience, and is likely to view problems in the domain at a low level, with many small problems they must solve individually. An expert is able to achieve low-road near transfer, and high-road distant transfer; a novice struggles to achieve any transfer at all. Becoming a genuine expert is thought to require thousands of hours of experience, but other plateaus are thought to exist, and in particular novices can reach the point of competence and begin to achieve easy near transfer with several hundred hours of experience with careful teaching [35, 150]. Early in learning, however, a novice will experience no transfer at all, and sees no benefit from having learnt even a closely related skill to a low level.

For learners to achieve transfer they must be taught the concepts in a fashion that facilitates transfer [150]. Without such teaching the knowledge tends to be *inert*: it can be applied within its original context, but learners will not generalise from that context to apply their knowledge elsewhere. Perkins and Martin found that students learning to program would learn language constructs inertly, and so had difficulty applying their knowledge to the act of programming, notwithstanding that the distance of transfer is minimal in this case [149], while Dyck and Mayer found that without transfer-focused teaching learners of BASIC would master the syntax of the language, but struggle more with semantics than those taught with transfer [41]. An assumption in much teaching is that (near) transfer will occur automatically by the low road, but research has not borne this assumption out in practice [150], instead finding that instruction must be tailored to assist high-road transfer; in the literature, this tailoring to target high-road

transfer is called *bridging* [41].

In the context of teaching programming, the first domain may be a single programming language and the second a different language; they may also be different ways of working within the same language, or even the syntax and semantics of a single language. To transfer their skills from one domain to the other students will need a combination of sufficient experience and bridging instruction, or else there will be no transfer and the second domain will need to be taught essentially from scratch. This finding has implications for the design of educational languages: because an educational language explicitly expects learners to move on to other languages afterwards, the language must support the learner for long enough to allow them to build sufficient competence that they can successfully transfer their skills to another language.

2.1.2 Programming pedagogy

Papert [143] proposed a discovery learning format for teaching programming based on Piaget's theory of cognitive development [153, 154, 155, 156]. In Papert's model, students learn a programming language in a similar conceptual fashion to how children learn natural languages. In the discovery format learning is almost entirely self-directed and learners are provided with an interactive system (the Logo language and environment) to work with where changes can be readily perceived. A student learns by experimentation, gradually forming a mental model of the programming environment based on their observations of the effects of their actions; proponents claim that this approach leads to both programming competence and general problem-solving ability that can be applied to other domains (including non-programming domains and mathematics). Experimental teaching did not show the results predicted by Papert [101, 35], although some contemporary work found that an approach with more explicit bridging showed more success [38, 27]. After much work on teach-

ing mathematics with Logo failed to bear results, interest waned in the late 1980s. Little work on programming education with a true educational psychology focus has occurred since; the computer science education literature of the 1990s and 2000s to date has largely been filled with designers, authors, and fans proposing that their language or approach is best, and experience reports that do not account for confounding factors [187, 108, any SIGCSE proceedings]. Nonetheless, some interesting results have been presented in recent years, and we will discuss these now before moving into discussion of proposed educational languages.

Garner *et al.* [57] report on the prevalence of different errors in novice code in Java. The most common error in their categorisation was "basic mechanics" (largely typographical in nature – omitted brackets, misnamed variables or classes, and similar). The next most common problems all dealt with high-level understanding of the task at hand, and misunderstandings and misapplications of procedural constructs followed. Mechanical errors dominated for students at all levels. The authors suggest targeted interventions aimed at the common errors and using the distribution of errors to guide the length of instruction on particular topics.

Cardell-Oliver [21] suggests using automated metrics to diagnose student code, and to report results and suggestions from those metrics to students as formative feedback to help them improve. Well-known metrics can measure program size, efficiency, correctness, and style automatically, and outlying results can be reported to the student with guidance on how they might improve their programs. The same metrics can guide course design by determining where students are actually having problems, which may differ from the instructor's supposition. The author also identifies limitations in past studies using single quantitative measures of performance, and of grading models in common use that may cause the bimodal distribution sometimes found in programming courses; in both cases, the author proposes the use of objective software metrics instead.

Robins [165] proposes a model of student attainment in which a chain

of dependent concepts in computer science causes student performance to diverge: either the student understands early concepts, and continues to perform, or encounters difficulty at some point and is unable to catch up; the author proposes that the dense dependency chain of learning programming concepts is unique. This *learning edge momentum* is a rich-get-richer model: a student who is performing well at a given point in the course will likely attain competence at the next concept introduced, while one who has not understood the last concept will likely not understand the next. This effect compounds during the course, leading to a bimodal distribution. Robins proposes this model as an explanation for the high rate of high grades that often accompanies the high failure rate, and provides citations from other educational work to substantiate the model as a reasonable explanation of observed facts, as well as to counter the alternative explanation that there are distinct populations a priori able or unable to program. The author notes that the model suggests an especial effort and focus be applied at the start of a programming course to put as many students as possible on the positive path.

Murphy and Thomas [129] note implications of psychological research on self-conception of intelligence for computer science education: in particular, views on whether applied intelligence is fixed or malleable and able to grow. The authors note that experimental results in psychology may in particular provide partial explanations for the tendency of some students to stop completely when faced with difficulty, as well as the observed gender disparity in the field, and suggest an approach for changing students' views so that all have a *growth* mindset to achieve more effective instruction. They also note that the views of the instructor may also have an effect.

Caspersen and Bennedsen [24] propose a design for an introductory object-oriented programming course based on educational theory. Their approach is based on explicitly forming patterns for programming activities through structured repetition, using worked examples in particular as a focus of the course content. The authors note providing multiple such

examples as beneficial, along with varying the surface structure of the examples and having students describe the programs to themselves. The authors report success in running the course they describe, but have only anecdotal evidence for that success at this point.

The ACM and IEEE have collaborated on curriculum guidelines for computer science, programming, and software engineering programs since 1968 [2]. The latest edition, CS2013 [88], requires that students be prepared to work in multiple paradigms and multiple disciplines, and provides specific guidance for introductory courses. The guidelines note explicitly that "it is important that students do not perceive computer science as only learning the specifics of particular programming languages" [88, p41]: students must learn the base concepts through programming, and be prepared to apply those concepts in other languages or contexts. The report also notes the explicit trade-off that must be made for an introductory language:

> The use of a language or environment designed for introductory pedagogy can facilitate student learning, but may be of limited use beyond CS1. Conversely, a language or environment commonly used professionally may expose students to too much complexity too soon.

A language or tool designed for introductory courses will need to tread this line carefully, supporting learners to the point that they are able to use their knowledge in other contexts.

2.1.3 Educational languages

Many educational programming languages and systems have been designed, and some have achieved wide currency for a time. BASIC [93] was a language originally designed for beginners, but not necessarily for programming education; the designers' goal was to open up programming to users in non-computing fields. BASIC was originally unstructured, but

later gained procedures, and was widely used in various forms for teaching for many decades. Early versions of the language included only GOTO-based control flow and two-letter variable names. BASIC in particular was often scorned as a poor choice of introductory language despite its popularity, with claims that its use led to poor programmers. Many different implementations with slightly different syntax, semantics, and features became widespread with the rise in minicomputers, and Microsoft's Visual Basic is still a popular language for real-world programming today [180].

Pascal [195, 87] was one of the most successful educational languages, in widespread use for teaching from the 1970s through 2000s. Pascal is an imperative, statically-typed, procedural programming language, with one of the key goals being to teach structured programming, a then-recent development. The original working Pascal compiler was written in Pascal itself, and the needs of that compiler also influenced the language design. While Pascal was explicitly intended for education, it also gained use outside of teaching. This use was controversial both with the designer and others, who found the simplifying limitations of Pascal unsuitable for real-world programming [94].

SP/k [67] is a family of languages designed to teach structured programming: there are several subset languages SP/1 through SP/8, each of which introduces new constructs building on the previous subset. SP/k was designed as a subset of PL/I and achieved some use in the 1970s and 1980s.

Lisp [121] was not built as an educational language, but rather simply as a practical notation for writing programs for limited machines. Lisp is an impure functional language, with all control flow by prefix function calls. Lisp is a homoiconic language: its representation of data structures and code are the same, built of *s-expressions* written in parentheses; this homoiconicity led to advanced macro systems that manipulate program code. Many variants of Lisp have existed and continue in use today. Lisp has been proposed for educational use because of its syntactic simplicity [49]:

there is only one syntactic form, only one kind of bracketing construct, and only one kind of control flow. The argument is that by having only one form to learn students are less confused and make fewer errors, freeing them to focus on the essential components of learning.

Logo [143] is a Lisp derivative explicitly aimed at education. Logo's design has roots in mathematics and psychology research, and pioneered in particular the "turtle graphics" paradigm, where an on-screen image or tangible robot follows instructions to move around the world, leaving a trail behind itself. Logo is one of the few proposed educational languages with genuine educational research behind it, which was discussed earlier.

Racket [49, 182] is another Lisp derivative explicitly proposed for teaching. Racket is integrated with DrRacket, a specialised integrated development environment for teaching. DrRacket includes a number of features to help novices, including a stepping debugger, various metadata overlays, and educational language variants. Racket supports a system of language variants that allow giving students at different stages of learning access to different functionality and different error messages, and includes a proposed sequence of variants to use for teaching.

Scratch [162] is a visual language for teaching aimed at children. All editing in Scratch is by drag-and-drop: the programmer drags a jigsaw-piece tile out of the toolbox and puts it onto the work area or into a space in an existing tile. The language is integrated with the editor and a persistent microworld where different sprites can be placed and given code to control their movements, dimensions, and actions. Scratch is an imperative language with some structured components, and inherently supports concurrency. Research has shown some success in drawing children to engage with computers [53, 17, 115], but has also seen difficulties in learners moving on to other languages [123, 124].

Alice [36] is another visual language for teaching, aimed at a slightly older audience than Scratch. Alice includes a graphical microworld filled with three-dimensional sprites, which can be manipulated and moved in a

three-dimensional space. As in Scratch, code editing is by drag-and-drop, although with a larger number of menus than Scratch. Alice is primarily an event-driven programming system, where code executes in response to a condition being true within the microworld, but it includes most common structured-programming features as well.

SOLA [120, 119] is a simplistic objects-first prototype-based imperative programming language intended to teach the concepts of object orientation to novices, drawing from principles set out by McIver and Conway [130] for their GRAIL language. SOLA is a textual language fully integrated with an editor, and is structured so that a user can interrogate the structure of their objects at any time. The authors propose that by limiting learners to only the basics of object orientation they will be forced to learn those concepts well without being distracted by accidental complexities of the language, and present a small study [119] to support this claim; this study separates out reading, writing, and modifying code, which has not been commonly done in the recent literature.

Stefik and Siebert [173] describe Quorum, an "empirically-designed" introductory language, and perform empirical studies to compare novice accuracy in their language, a "randomly-generated" language called Randomo of their own design, and several other languages. The authors study keywords and constructs for "intuitiveness" with novices, and select the most intuitive terms to incorporate into Quorum. Randomo replaces the keywords of Quorum with random non-alphanumeric symbols from the ASCII table. The authors also investigated readability of larger program constructs, finding that Quorum performed best at this task. Finally, they conducted an experiment where they presented students with the same programs in Quorum, Perl, Randomo, Java, Python, and Ruby, along with a textual description, in order that participants would learn the syntax and semantics of the language; they then asked participants to produce programs performing particular tasks in that language. The authors reported that users in Java and Perl performed no better than in the randomly-

generated language, while users of Quorum, Ruby, and Python performed comparably and better than random. These studies did not involve any structured instruction of participants and so are not directly comparable to real teaching, rather representing untrained readability with targeted reproduction.

2.2 Pattern matching

A pattern matching system is a structured way of categorising at runtime data entities in a program based on some characteristic, and branching on the basis of that category [181]. In an object-oriented language, that entity being matched will be an object, but pattern matching is most common in functional languages where some combination of records, types, and primitive values can be involved in matching. The syntax, semantics, and role of pattern matching differ significantly between languages.

Matching is the basis of entire models of program organisation [85]. Where in object-oriented code dynamic dispatch on the receiver allows executing different code for different kinds of data by placing the code in the object, a language with strong pattern-matching facilities allows the program to be structured in the opposite manner: code dealing with different types of data is placed outside the entity in question, after the fact, and potentially with all the related cases together. These different models of organisation are usually found in different languages; our work (in Chapter 4) is on permitting both models within a single (object-oriented) language, so we will discuss both functional and object-oriented related work here with a particular focus on how they enable (or hinder) these models.

We define two broad approaches to pattern matching that influence both the syntax and semantics of the resulting system: matching can be what we will term *inherent*, meaning that patterns are a special built-in construct functioning in predetermined ways, or *exherent*, meaning that the

precise behaviour of matching is user-defined, with the language providing a structure.

In an inherent matching system, such as in Scala [46], OCaml [102], or Haskell [117], specfic matching syntax can be provided that expresses the full range of possible patterns. These patterns are embedded deeply in the language. The semantics of matching is generally determined by what leads to (perceived) good syntax that appears naturally in the language. Patterns are not first-class entities in themselves.

In an exherent matching system, such as in Newspeak [58], OMeta [192], and F# [176], patterns are first-class entities and can have arbitrarily complex behaviour. As a result of this flexibility, however, a fully-integrated syntax is more complex or impossible. The semantics of patterns align with the rest of the language.

In an exherent matching system the programmer can write code that is polymorphic in patterns, such as a method or function that accepts patterns and retains them to be applied later. With an inherent system this polymorphism is not generally possible, as patterns are not first-class entities. The programmer can, however, often build a first-class object or function that simulates this behaviour, in effect building an exherent system on top of the inherent system of the language. The reverse behaviour, building an inherent system on top of an exherent one, is not generally possible as inherent systems innately depend on language syntax support. In this way an inherent system has some additional flexibility that an exherent system does not, albeit the flexibility to simulate an exherent system.

Languages using both inherent and exherent approaches exist, although inherent matching is more common and precedes exherent techniques, existing in very early functional languages. Some languages integrate aspects of both approaches.

Pattern-matching systems often draw from mathematics the concepts of partial and piecewise functions, particularly in functional languages. A

partial function is a function that is defined only for some inputs; the set of inputs for which the function is defined is called its *domain*. A *piecewise function* is one with different bodies (implementations) defined for different sub-domains. Piecewise functions can be seen as combinations of multiple partial functions, one for each constituent domain. In mathematics, the domain is a set, but in programming languages how a domain is defined, what it means, and how it affects the program varies dramatically.

Our pattern-matching design draws from Scala [46] (principally an inherent matching system) and Newspeak [58] (principally an exherent matching system), so we first focus particularly on these languages. We outline the design of these and other pattern-matching systems here, and contrast them with our approach in Section 4.7.

2.2.1 Scala

Scala includes a powerful inherent pattern-matching system, with explicit syntactic extensions supporting it [46]. The syntax of patterns is a distinguished sub-language, designed to make for concise patterns.

Scala matching is primarily based around the match-case construct:[1]

```
x match {
  case 1 => "one"
  case "two" => 2
  case y : Int => "scala.Int"
  case (a, b, c) => "A triple: " + a + "," + b + "," + c
}
```

The match-case construct defines a series of partial functions, combining them into a piecewise function. Each instance of the **case** keyword introduces a partial function definition with the domain determined by the constraints that immediately follow: “1”, “y : Int”, etc; the behaviour of

[1] As with many Scala programs, the typechecking of this example is perhaps-unexpectedly context-dependent. A complete program incorporating the example verbatim is included in Appendix B.1.

each pattern is built into the language. These definitions are valid within the **match** block only, which is a special parsing context. A block of cases returning values can also be provided to methods taking single-argument (first-class) functions as arguments; these cases must cover every possible input. The matching syntax also permits "guards": additional boolean tests that must pass for a case to succeed. Guards are written after **if** and before the **=>** of the case block.

Scala supports user-defined "case classes" with additional matching functionality. A case class is an ordinary class whose parameters are retained and publicly-available. Case classes are explicitly declared as such and can be both instantiated and used in pattern matching:

```
case class Point(x, y)
...
x match {
  case Point(0, y) => "On the y axis at " + y
  case Point(x, 0) => "On the x axis at " + x
  case Point(x, y) => "On neither axis at " + x + "," + y
}
```

The constructor parameters of case class instances can be decomposed, matched against, and accessed during pattern matching. Like other classes, case classes can inherit, and instances of case classes can have additional methods and fields. Case classes as patterns are distinguished by always having parentheses after their names; if there are no constructor parameters to match, the case name is followed by empty parentheses: `case Nil() => "nil"`.

"Extractor objects" are a generalisation of case classes, allowing any object to permit matching and decomposition. Extractor objects are first-class objects and allow some exherent matching behaviour. An extractor object has a method "unapply" which returns either a list of decomposed values from a successful match, or a Boolean indicating success or failure with no decomposed values. Extractor objects can have arbitrary semantics

for determining whether to match or not, and what values to extract; they have no fixed relationship to other extractor objects. Using an extractor object has the same syntax as a case class, including parentheses.

We contrast our design with Scala explicitly in Section 4.7.1.

2.2.2 Newspeak

Newspeak [58] includes an object-oriented pattern-matching system designed around message passing (also known as method calls). In contrast with Scala, Newspeak's is a fully exherent matching system: patterns and classes are objects, and matching does not involve distinguished syntax or altered semantics but rather ordinary method invocation. All patterns implement doesMatch:else: and all matchable objects case:otherwise:[2].

To perform a match the programmer sends a case:otherwise: message to the target object they wish to match. The first argument to case:otherwise: is a pattern object, and the second is a block of code to execute if the match fails. The target object double-dispatches to the pattern object's doesMatch:else: method, passing itself and that same block.

In the ordinary case the pattern then asks the target to match: the pattern, and the target responds by sending a type-specific message back with some state that it wishes to be recognised by; for example, a point could respond by sending the x:y: message, while a string would simply send itself in the string: message. A pattern implements the messages sent by the objects it wishes to match, or handles all messages through the default doesNotUnderstand: protocol. Any unhandled message causes a "not understood" exception, which is interpreted as a failure. The matching protocol is shown in Figure 2.1.

At any of these points either a target object or a pattern can deviate from the protocol. An object that does not wish to be matched at all can reject the initial case:otherwise: message and opt out of pattern-matching altogether.

[2] In Newspeak, method names can have multiple parts; these are written using colons.

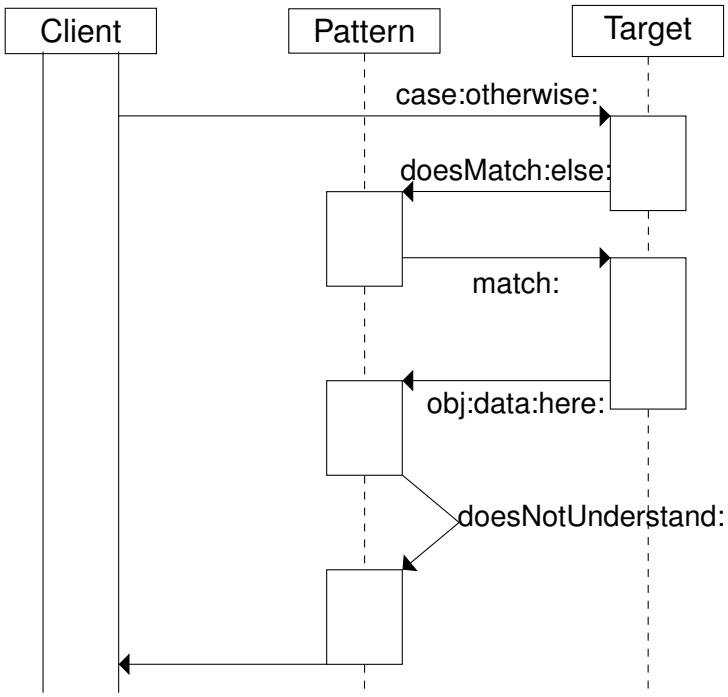


Figure 2.1: Matching sequence in Newspeak. The obj:data:here: message is target-type-dependent, while doesNotUnderstand: is optional.

Complex patterns are built using object composition, including nested decomposition of object state. A pattern with any desired semantics can be written by implementing a suitable doesMatch:else: method, but a library of suitable combinators and pattern components is included in the system. There is no distinguished syntax for patterns; as an exherent system, Newspeak relies on its (relatively flexible) syntax for building pattern objects, and method invocation for performing the match. As a result, some patterns in Newspeak are more verbose than they might be in an inherent matching system such as Scala's. The compensation is that the full power of Newspeak's language is available for defining what should and should not match, and that patterns can be used in any of the ways that other first-class objects can.

We contrast our design with Newspeak explicitly in Section 4.7.2.

2.2.3 Other object-oriented languages

Inherent matching

Thorn [11, 12] makes heavy use of pattern matching, with extensive inherent syntax supporting matches as part of many of the language constructs, including the control structures and clausal function definitions. Thorn offers a wide array of built in data types, each with corresponding matching destructors, and a set of algebraic pattern combinators. A Thorn class's formal parameters are used to initialise instance objects, and they can also be extracted if an object is matched against. Thorn includes special patterns (and syntax) for matching with regular expressions, arrays, bitpatterns, and XML.

Fortress [170], another large and flexible language, has a relatively modest range of pattern matching features, including destructuring of objects and tuples. Fortress matching is also primarily inherent, although patterns may be lifted to objects. Like Thorn, patterns in Fortress may be used freely in definitions, rather than solely in matching constructs.

OCaml includes an inherent pattern-matching system through special syntax [102]. The programmer can match against any object, but the only available patterns are algebraic data types. Matching is based on the implementation site of the target object: matching is a slight generalisation of a standard instance-of check supported in many languages.

Machete [66] (a forerunner to Thorn) is an extension to Java introducing pattern matching in a conventional inherent way: adding a match statement, a rich library of built-in patterns, and a separate type of "deconstructor" declaration for objects that extract values when matching. While Machete is primarily an inherent system, and pattern semantics are embedded in the language, as part of its embedding into Java it does permit limited exherent extensions through these deconstructors. As in Thorn, Machete includes special patterns for matching XML, regular expressions, and arrays.

OOMatch [163] is a more radical language design that fully integrates

pattern matching inherently into Java, providing clausal function definitions and multi-methods. OOMatch semantics are a dramatic departure from those of Java, although most Java programs are upwardly compatible.

Exherent matching

OMeta [192] is an object-oriented language for pattern matching based on Parsing Expression Grammars (PEGs) [51]. PEGs by nature are an exherent approach to pattern matching; pattern elements are first-class and are combined together to build larger patterns, including through aliasing and nesting. A program in OMeta is a PEG that recognises a given class of inputs and optionally decomposes an input into a structured result, or rejects the input. OMeta models grammars as traditional object-oriented classes and productions as methods. A particular goal of the language is to permit extending existing grammars in a structured fashion. OMeta extends PEGs to support matching arbitrary data, but many of its features remain aimed at conventional text parsing.

Lua includes an exherent text pattern-matching library with compositional first-class patterns built on PEGs [81]. Patterns can be sequenced, combined, disjoined, and negated through object composition. The library requires no additional syntactic support, but is geared specifically towards matching text, not arbitrary objects.

MatchO [189] is another extension to Java that provides a flexible pattern library but does not need (or provide) any specialised syntax above that of Java — although syntax can be supplied by invoking a generated parser inline. MatchO is an exherent system, but is able to obtain some benefits of an inherent system through this parsing approach, albeit with a very stark delineation between code in the base language and in a pattern. In MatchO, patterns are necessarily first-class, given that they are implemented as a normal Java library, but MatchO does not generalise patterns to pattern combinators.

2.2.4 Functional languages

Many if not most functional languages include substantial pattern-matching functionality, and are often based around matching as a mechanism for program organisation [85]. In most cases matching is principally on the type (or implementation) of an entity, and may be in the form of partial function definitions.

Most commonly, functional languages support inherent matching. These languages frequently borrow from mathematics the concept of partial functions and use that as the basis of the language. Haskell is a representative example of this approach which we discuss in the next subsection.

Some functional languages support exherent matching, either in addition or instead of inherent matching. We will discuss F#'s and Racket's interesting approaches to leveraging the existing nature of functional languages to support first-class patterns and matching below.

Haskell

Haskell, in common with most other functional languages, provides excellent syntactic and semantic support for patterns [117]. This pattern matching follows an inherent approach and is built in to the language: patterns are not first-class elements that can be operated on. Patterns are either embedded deeply into the language, like the historical `n + k` patterns, or correspond exactly to matching data type definitions.

All functions in Haskell can be partial or defined piecewise using pattern matching:

```
fib :: Int -> Int
fib 0 = 1
fib 1 = 1
fib (n + 1) = (fib (n − 1)) + fib n
−− Alternatively:
fib n = (fib (n − 1)) + (fib (n − 2))
```

This function returns an element of the Fibonacci sequence [157]: 1 if the argument is 0 or 1, and for any other argument the sum of the previous two elements.

Numeric patterns are built in to the language, but user-defined data types can also be matched against:

```
data List a = Nil | Cons a (List a)
length :: List a -> Int
length Nil = 0
length (Cons x xs) = 1 + (length xs)
```

This function computes the length of a linked list recursively, with Nil as the terminator. Matching here is exactly on the definition site of the target: only a Cons arising from this definition will be matched.

Special cases for some common data constructors are built in, like the ":" constructor for the built-in list type and "()" for tuples. The syntax is quite concise and allows specifying, in particular, recursive function definitions easily. Recursive functions are a very common occurrence in Haskell code, so the pattern matching system optimises for them.

Piecewise function definitions are in fact syntactic sugar for a single function containing a use of the built-in **case**...**of** syntax, which performs pattern matching in exactly the same way. In either case, the pattern specifies which values or types should be matched and optionally destructures them, and finally a value must be returned. A pattern cannot be passed as an argument to another function or stored in a variable.

Proposals to add reified patterns to Haskell have come from both Tullsen [184] and Reinke [161]. Tullsen represents patterns as reified partial functions, and then pattern combinators are function combinators; larger structures such as case statements and clausal function definitions are then built on top of these. Reinke's "lambda-match" proposal extends this approach to combine individual cases into whole match statements with combinators. Wadler proposed "Views" [191] as a way to support programmer-defined matching and destructuring of abstract types in func-

tional languages, which Peyton Jones made more concrete in proposing view patterns for Haskell [152]. A view pattern is a function that matches an abstract value and returns a concrete data type that can be further matched-against [152]. None of these proposals became part of the overall Haskell language.

F#

The F# language supports exherent matching against abstract structures via active patterns [176], as well as the conventional inherent matching common to functional languages. An active pattern is a function with a structured name where that name indicates one or more alternative cases. Active patterns support first class patterns in this way, as F# has first-class functions. The language also provides a range of pattern combinators, which in this case are specialised function combinators. F# builds in support for the common cases most functional languages include in their inherent matching systems.

Although they can support partial matching, active patterns support exhaustive matching particularly well, because a single succinct pattern function matches and destructures every alternative, returning a result indicating which case of the pattern is matched. In contrast, Scala and Newspeak require individual matching and extract functions for each case. An example total pattern match (adapted from the cited paper) is:

```
let (|Cons|Nil|) l =
  if nonempty l then Cons(hd l, tl l)
  else Nil
```

This function definition defines two patterns Cons and Nil. When matching against one of F#'s built-in lists and suitable definitions, these patterns are total, with the matching behaviour defined by the code in the body of the definition and what it returns. These patterns could be used to compute the length of a list recursively:

```
let rec length l =
  match l with
    | Cons(x, xs) ->
      1 + length xs
    | Nil ->
      0
```

This definition follows the example given for Haskell above exactly, but the patterns are defined by user code and are first-class entities. A complete program incorporating these definitions is given in Appendix B.2.

Racket

The Racket Scheme dialect also includes an extensible exherent pattern-matching facility [181]. Uniquely amongst all the designs presented here, Racket's powerful macros enable the language to be extended without any changes to its core implementation.

Also uniquely amongst the functional languages we have discussed, Racket makes full use of its functional nature in its pattern system. Patterns are represented entirely by ordinary functions that examine and classify their arguments, and these functions are addressed using the ordinary mechanisms of the language for accessing its first-class functions. Most patterns desugar to application of a single function with surrounding boilerplate code. Syntax for user-defined patterns is written in the same macro format as built-in patterns, so even these built-in patterns are not privileged in any way. Racket's system is thus fully exherent, in contrast to other functional languages.

We discuss Racket further and contrast its system with ours in Section 4.7.3.

2.3 Modules

A module system provides a way of organising code at a high level, where different modules have a degree of conceptual independence from each other. For example, different modules may have their own namespaces, be in separate files, or represent a barrier against access to code.

While most languages support some sort of module system, they vary dramatically in how they do so. In particular, some languages represent modules as first-class entities of the language, while in others they are a grouping of components that are less than first class. We term the latter "packages"; packages provide less power than first-class modules, as a package cannot (for example) be aliased at run time, but are more common. In most cases module systems are something of an afterthought in the design of a language, and so are often special cases with behaviour different than other aspects of the language, and semantics that must be learned separately.

Our module design in Chapter 5 is inspired particularly by aspects of Python's and Go's module systems [159, 60]. We outline the design of these and other module systems here, and contrast them with our approach in Section 5.6. We will first examine systems where modules equate in some fashion to first-class objects, and then look at package systems. Finally, we give a brief overview of work on obtaining access to other systems through a module system.

2.3.1 Classes and objects as modules

Python

Python [159] supports modules with separate compilation, which become objects at run time. All top-level definitions in a source file are attributes of the module object. Python's **import** statement includes the qualified name of the module as a sequence of dot-separated identifiers, which are

mapped onto a filesystem path: **import** x.y.z resolves to a file z.py inside the directory x/y. The source file is loaded at runtime and the resulting singleton object is bound to the imported name. This is the only way that a Python object can be created without a class. The **import** statement may include an optional as clause, which gives a local name to the imported module.

When the qualified name of a module includes multiple levels, as above, each intermediate layer is itself a module. The module x is defined in the file x/__init__.py. When a module's qualified name includes a dot all modules along the chain are imported from left to right, and the leftmost component of the name is by default bound in the local scope so that the same qualified name used to import the module can be used to access it. To make that name available, each intermediate module has a new field added referring to the next in the chain: after **import** x.y.z x has a y field, and x.y has a z field; these changes are globally visible. This mutation is necessary for the design to work, and possible because Python objects are mutable by default. The module object is a real object that can be aliased and passed to other code.

As Python is dynamically typed, there is no type information present in Python modules. Python only enforces encapsulation by metaprogramming.

We contrast Python's approach with ours in more detail in Section 5.6.1, including discussion of some drawbacks of this approach we seek to avoid.

Newspeak

Bracha *et al.* [15] describe modules as classes in Newspeak. In this language a module definition is a top-level class, whose instances are termed "modules" and are immutable. Classes can be nested, and the code in a class can access external state in three ways: lexically, from an outer scope; from an argument provided at instantiation-time, or from a superclass. A module definition has no lexically-surrounding class and so must be passed all

modules it will use encapsulated in a "platform" object, or else obtain them through inheritance.

Dependencies are not bound statically in the module source; instead, the dependencies of a module can depend on instantiation-time arguments, so a module may be provided a different implementation of a dependency in different programs, or in different parts of the same program. A module can pass on the dependency mapping it was given, or create its own. Every module can have multiple instances and can be inherited from. We contrast Newspeak's approach with ours in more detail in Section 5.6.2.

Explicit module construction

Ungar and Smith [186] describe how the Self language can support modular behaviour by prototype-based object inheritance. Self does not include a distinguished "module" concept; rather, everything is an object and some objects may be used as modules. Like other objects, these module objects must be accessed by sending a message to another object or by creating them locally. Any object may both inherit and be inherited from, and inheritance of environment objects subsumes the role of lexical scope, allowing an object to present a customised picture of the world to code defined within. The Self system as a whole works in terms of "worlds", where objects are created and continue to exist in the world until destroyed, even persisting when the program or programs are not running. There is no inbuilt concept of a module. To move (or copy) objects between these worlds, a "transporter" [185] is used. The transporter does have a concept of a module, which it uses to produce the correct behaviour. The programmer annotates individual slots (fields or methods) with the module they belong to, and potentially with instructions for how each should be treated. An entire module, spanning many living objects, may then be moved to another world to be used there, permitting code reuse. Any object can be used as the basis for module.

In AmbientTalk [37, 188] modules are written as explicit objects and

loaded by asking a "namespace object" for them, which maps directly onto the filesystem according to a configuration set up earlier. Module import implicitly creates delegation methods when required, allowing modules to be imported into a local scope belonging to any object in the system. Importing a module can occur at any point in the code by calling a method on the namespace object.

Kang and Ryu [90] formally describe a proposed module system for JavaScript. JavaScript itself does not have any support for modules, but they are often simulated by objects or functions in the global scope, which can lead to naming conflicts. Kang and Ryu's module system extends the language with an explicit `module` declaration, creating a new namespace and binding a reference to it, which may be traversed to obtain explicitly-exported properties defined in the module. They show that their system safely isolates private state, but allows both nesting and mutual recursion. The view of a module presented to the outside is an object, although the implementation of the "module" declaration desugars to multiple objects and closures to give effect to encapsulation rules. The semantics are complicated by JavaScript's idiosyncratic scoping and visibility rules.

2.3.2 Packages

In contrast to the above languages, in which modules are first-class and have a run-time existence, we now look at some designs that provide what we call "packages": grouping of components that are less than first-class or which do not exist at runtime at all. We present Go in particular as a package system we will draw on in our design, and then present the development of important package systems chronologically.

In the Go language [60], a package may comprise several files, all of which explicitly declare themselves part of the same package. The definitions in these files are combined and may be accessed by clients using Go's **import** statement. Once imported, the module's public interface is

available through a dotted name, as in Python, but the module itself has no run-time existence. Go's import syntax uses opaque strings, which in practice are interpreted as filesystem paths. Associated tools are able to interpret import paths as URLs and use them to fetch and install modules from remote locations when required. Our approach draws from Go in some respects, and we contrast Go's approach with ours in more detail in Section 5.6.3.

Alphard [197] incorporates an early explicit module system aimed at enabling program verification. Implementation and interface are separated, so that verification needs only to confirm that the implementation behaves in accordance with the specification.

CLU [106] includes modules that encapsulate the implementation of a desired behaviour. These modules may interact with each other through public interfaces, which fully characterise the module. A module may be replaced by another implementation with the same abstract interface. A language construct permits collecting operations together into an abstract interface.

Standard ML [114] includes a module system built around functors. ML modules make heavy use of static types to achieve their goals. Modules bind environments, types, and functions together, and may have multiple different instantiations. A module gives a single name to a combination of behaviours and state. The module system is powerful but includes many constructs with subtle interactions. In particular, modules can have almost arbitrary type-dependent behaviour giving different effects at different points where the module is used.

The BETA language itself does not include a module system, instead supporting modularisation through an integrated programming system [96]. Programmers can split their programs however they wish, because subtrees of the abstract syntax tree can be maintained and compiled separately, and then combined. Modules are not first-class entities in BETA, but the modularisation of any arbitrarily complex component is possible.

Modules in Modular Smalltalk manage the visibility and accessibility of names [194]. Modules are not objects and do not exist at runtime. Instead they define a set of bindings between names and objects. They can be used to group collaborating classes, and provide a local namespace for their own code to refer to while making only the module itself globally accessible.

Modula-3 [22] includes both "interfaces" and "modules". An interface is a group of declarations without bodies; a module may export implementations of part of an interface, and may import an interface to gain access to its elements. A module may provide definitions of some, all, or none of the elements of the interfaces it exports, but will have unqualified access to definitions made elsewhere. A single interface may have its implementation spread across multiple separate modules. Only the interface needs to be referenced or understood by other code.

Scala [138] includes the concept of "packages", which can contain classes, traits, and objects, but not any other definitions, and are imported by their qualified name. Scala also has "package objects", which are specialised objects that can be declared inside a package to augment it with other definitions, such as methods. Definitions from another package can be imported directly into a scope or be accessed through a qualified name.

Java also includes "packages", which are weaker than those in Scala; they serve to subdivide the namespace of classes, as well as providing a level of visibility between public and private. Strniša *et al.* [174] propose and formally describe a module system for Java in which a module is an encapsulation boundary outside these namespace packages. A module is able to define the interface available to the outside world and to import other modules into scope with their interfaces. These modules also allow combining otherwise-incompatible components in separate areas of the program by enforcing a hierarchy on access.

Racket [182] supports packaging multiple functions into a single unit, which it calls a module. Racket modules are imported using the require function, which takes a string as input (generally interpreted as a relative

file path). Modules are defined either explicitly using the module form or implicitly by the variant of the language a file is written in. Typed Racket includes a require/typed function, which assigns local types to the functions and structures imported from the module. These types are enforced as contracts. The module itself does not have any run-time existence; instead require simply makes its contents available to the importing code.

Szyperski [177] argues that both modules and classes are essential components for structuring object-oriented programs. He defines a module as a "statically-instantiated singleton capsule containing definitions of other items (objects, methods, types, classes) in the language, capable of exposing some, all, or none of its contents to the outside and providing unrestricted access to the contents internally". A class is a template for constructing objects, which may inherit from other classes and have many instances. While class instances (objects) exist at run-time, a module is a purely static abstraction serving to separate code. One module may import another, obtaining access to the public contents of the other module through a qualified name. This mechanism is separate from both the inheritance and instantiation mechanisms supported by classes, but accurately reflects the intention of the programmers of both modules. Because modules have no run-time existence, however, it is not possible to parameterise code with them, and an additional language mechanism is required to define and import them. Szyperski argues that modules as he describes them are beneficial for program design because they enhance encapsulation and structure, and for programming practice because they allow separate compilation.

2.3.3 Foreign objects

One role that module systems sometimes play is mediating access to outside resources, such as code written in other languages. The module system in that case provides a barrier beyond which lies the "outside" world where the language semantics may not hold entirely, as well as a way of

representing what from the outside should be accessed. We will aim to enable some of these behaviours, and so present relevant related work here.

F# accesses external data sources with "type providers" [175]. A type provider defines a way of accessing a data source outside the program — like a database or web service — and integrating it with the program as though it were an integral part of the system. Type providers are fully integrated with the IDE: when accessing a web service, for example, auto-complete menus will appear with the different actions or sub-fields available from that service in that particular context. The F# compiler statically generates binding code according to the definitions in the type provider and what is obtained from the external source, and ensures that type information from the remote source is fully propagated into the program.

Newspeak's foreign function interface defines "alien" objects [127], which come from outside Newspeak code but appear to their clients exactly like ordinary Newspeak objects. The implementation of an alien object is unknown and undefined, but it understands and responds to messages sent to it, and knows how to use its own representation to implement its own behaviour. These alien objects are not necessarily obtained through the module system.

Major dynamic languages, such as Python [159], Ruby [168], Perl [151], and Lua [82], allow dynamically-loaded code using the implementation's internal API to provide methods or objects. The foreign function interfaces of some statically-typed languages permit accessing code written in other languages, such as with the Java Native Interface [105] and Haskell's FFI [117].

2.4 Dialects and domain-specific languages

Dialects are extensions or restrictions of the language available to a programmer. A dialect may be a domain-specific language, a subset of a larger language, or a modification of the behaviour of particular program source.

Domain-specific languages (DSLs) are special-purpose languages using the vocabulary of a particular application domain to represent logic or data from that domain. Domain-specific languages can be contrasted with *general-purpose languages*, which intend to allow working in any domain.

Domain-specific languages can themselves be divided into *internal* domain-specific languages, which are embedded in a host language to some extent, and *external* domain-specific languages, which are independent of other languages [112]. An external DSL is completely unrestricted in the syntax and semantics it can admit, while an internal DSL must compromise with the host language. Internal DSLs, however, get some degree of interoperability with general-purpose code for no or little cost. These characterisations are not absolute, and some systems blur the distinction between the two. Our focus is on internal DSLs, but we will also describe notable external DSL systems (Section 2.4.9).

The dialect design we present in Chapter 6 builds on our module design and shares goals with Racket. We outline the design of Racket and other systems for language subsets and domain-specific languages here, and contrast them with our approach in Section 6.5.

2.4.1 Racket

Tobin-Hochstadt *et al.* [182] describe "languages as libraries" in Racket. Racket supports multiple language definitions through the use of the language's "Advanced Macrology" [33] to translate the input source text down to core Racket, adding new functionality, or even replacing the language syntax and semantics along the way. These language variations can be used in the manner of a library.

Racket (then DrScheme) reintroduced the concept of using multiple "language levels" for teaching [50], which was originally from SP/k [67]. Racket's levels are intended to be moved through in sequence with gradually increasing power: earlier levels restrict functionality that novices

will not need to use, and provide more informative error messages and suggestions based on their knowledge of what the programmer can write, preventing them from stumbling too far from what they know. While Racket languages have broad power, the defined language levels form a sequence building up to an unrestricted language.

Racket supports variant languages in two ways: through Lisp-style macros, and by replacing or augmenting the parser. In the first approach, the language (re)defines macros to be used by the programmer, who obtains them by declaring the language they will be using. Racket macros are very powerful and able to integrate closely with the IDE, such as by telling the IDE to provide visual indicators that certain identifiers represent the same actual variable. As is common in Lisp-like macros, a macro can prevent, repeat, or reorder the evaluation of terms, and manipulate them to have different meanings altogether. As a Scheme derivative, Racket has *hygienic* macros, which are guaranteed not to interfere with any identifiers in scope where the macro is used, but they are free to treat their arguments in any fashion they please.

Racket also supports variant languages by replacing the "reader", the code turning input text into S-expressions. With a language defined in this way there is no need for the source code to resemble base Racket at all; the Racket distribution includes an implementation of Algol-60 by this mechanism [178]. When replacing the reader the language author essentially writes a parser and code generator from scratch, and uses only the Racket IDE and any available libraries they wish to bind to in the language.

We discuss Racket in more detail, and contrast its approach with our design, in Section 6.5.1.

2.4.2 Scala

Scala [138, 167] includes several features supporting domain-specific languages. The language syntax permits methods acting like built-in structures and operators with many levels of precedence and associativity. Scala implicit parameters allow an argument to be passed without naming it, determined by the type. A method parameter can be labelled as **implicit**, and if that parameter is omitted the compiler will search the local scope for a definition with the correct type to pass to the method. Scala also supports implicit conversions lifting expressions to user-defined types. The following code uses both implicit features to allow strings to "say" themselves in a toy DSL:

```
import scala.language.implicitConversions
import scala.language.postfixOps
abstract class Speaker {
  def act(s : String)
}
implicit object TerminalSpeaker extends Speaker {
  def act(s : String) = println(s)
}
class MagicString(val s : String) {
  def say(implicit f : Speaker) =
    f.act(s)
}
implicit def makeMagicString(s : String) = new MagicString(s)
// Now the programmer can write:
"Hello" say
// and the program will print "Hello"
```

The **implicit def** defines a function that can be called automatically by the compiler when it would otherwise report a type error. In this case the function lifts a String to a MagicString, with an additional method say. The say method has an implicit parameter with type Speaker: when the

programmer calls say and does not provide the parameter, the compiler finds TerminalSpeaker in scope and passes it implicitly. With both of those defined, "`"Hello"` say" will print "Hello" to the terminal. In combination these features allow domain-specific languages that are aware of the context in which they are used. Scala's treatment of syntax and semantics is determined by the static type information it has available.

Scala also includes powerful macro features [47, 18] integrating the compiler and runtime. There is no formal "dialect" system in Scala, although similar functionality can be built using other constructs of the language. Scala mirrors have the ability to perform both run-time and compile-time reflection, and these can be used to implement domain-specific languages with similar ability to those in Racket, including the ability to defer some processing until run time, although with the same fundamental syntax.

We contrast our design with Scala in Section 6.5.2.

2.4.3 Ruby

Internal domain-specific languages (DSLs) are common in Ruby, supported by particular language features [52]. Two common strategies for Ruby DSLs involve using the language's open classes, and using per-instance dynamically-bound evaluation.

Open classes permit modifying third-party classes — including built-in objects — to add new methods, enabling users of the DSL to write, for example, 3.years.ago to represent a time. These modifications are globally visible, and work only so long as they don't conflict with other modifications.

The second strategy depends on dynamically-bound block evaluation using the method instance_eval, which is defined on all objects. This method allows one to execute a block of code inside the context of another object as though the block were written inside the object's definition, and thus with access to methods defined in the object. Any identifiers used inside the block will be found dynamically inside the object, so the author can

```
class Greeter
  def initialize
    @greeting = "Hello"
  end
  def say(s)
    puts s
  end
end

TheGreeter = Greeter.new()
def greetingdsl(&block)
  TheGreeter.instance_eval(&block)
end

greetingdsl do
  say @greeting
  say "World"
end
```

Figure 2.2: A trivial DSL using instance_eval in Ruby.

create a suitable DSL simply by creating an object with methods they want to expose and (optionally) a top-level method as the interface to the DSL.

Figure 2.2 shows a simple Ruby DSL. The Greeter class defines a method say and an instance variable @greeting. The greetingdsl method accepts a block as an argument, and evaluates that block inside an instance of the Greeter class (the ampersand converts the block to and from a first-class value). The programmer enters the DSL by calling the greetingdsl method and passing in a do...end block: the block can access both the method and the instance variable inside the Greeter instance, and so is inside the trivial DSL the class defines. The language syntax permits reasonably fluid code to be written in this way. Different DSLs may be used at different points by evaluating code inside different objects.

We contrast our design with Ruby in Section 6.5.3.

2.4.4 Haskell

Haskell is also used to define domain specific languages [3, 89]. These DSLs typically use Haskell's type classes to embed themselves in the language. Existing functions and operators become part of the DSL by defining type-class instances for the language representation — whether that representation is the data the DSL consumes, or a reflexive representation of the program itself. Static type information directs which functions are actually executed for a particular expression, often based upon the calling context (i.e. the expected return type). A programmer can temporarily enter the domain of a DSL simply by declaring the return type of their function.

DSLs can also be embedded in Haskell using the language's **do** notation (which provides quasi-imperative syntax for writing code inside a monad). This syntax can be used for DSLs in two ways. In the simplest form, the user writes code inside a **do** block that simply calls the functions of the DSL to perform computation. Another form, known as the *free monad plus interpreter* pattern, instead builds up a syntax or operation tree, which will then be executed by another function. Again, the static type information dictates which functions are run for which part of the program. The DSL may look natural in use. For example, code using a simple "turtle graphics" language might look like this:

```
prog :: LogoProgram ()
prog = do
  forward 100
  turnRight 45
  color red
  forward 100
  ...
```

Static type information is crucial to the semantics of Haskell DSLs (as it is in Haskell programs generally). We contrast our design with Haskell in Section 6.5.6.

2.4.5 Cedalion

Cedalion [112] is a language for defining domain-specific languages. Cedalion aims to promote "language-oriented programming", a programming style in which many DSLs are used in combination, with a new language defined for each subdomain spanned by the program. Lorenz and Rosenan, Cedalion's designers, define four kinds of language-oriented programming system: internal DSLs, where a DSL is implemented within a host language (as in our design), external DSLs, where the DSL is a separate language with its own compiler or interpreter (as in Xtext, Section 2.4.9), language workbenches, which combine tools and an IDE to present external DSLs as though they were internal (as in Spoofax, Section 2.4.9) and language-oriented programming languages, like Cedalion, which they claim permit the most useful range of languages. We classify Cedalion as an internal DSL system, as there is a single semantic model in use.

All Cedalion languages are interoperable because they share the same host language. Many languages may exist simultaneously in the program, supported by a special "projectional editor" [190]: the abstract syntax tree is edited, rather than textual source, following the tradition of Gandalf [62] and Mentor [39]. A Cedalion language defines a display grammar for that syntax tree, rather than a parsing grammar for text. These languages may exactly resemble the notation used by domain experts with few constraints.

We contrast our design with Cedalion in Section 6.5.5.

2.4.6 Converge

Converge [183] supports embedding domain-specific languages through a compile-time metaprogramming facility. Converge supports both the equivalent of a straightforward macro system, where code is generated from a quasi-quoted block of source, and embedding whole blocks of an arbitrary language within the program. These DSL blocks or quoted DSL phrases are parsed at compile time according to a user-defined grammar,

and translated into an abstract syntax tree of the base language by more user code, also executed statically. The generated AST is treated as though the corresponding code were written at that point. Because the generated code is inserted at the point of writing, a DSL can permit references to existing entities in scope, while being hygienic and avoiding any name conflicts by default. The DSL to use for a given block is identified by a preceding tag in the source identifying the applicable translation function.

The execution semantics of a Converge DSL are always those of the underlying Converge language. Multiple DSLs can exist in a single program and their outputs can interact in the same way that other Converge code can. There is no built-in support for embedding passages of one DSL in another, although a language providing that effect can be defined. Converge fits near to the language-oriented programming paradigm, but does not require a special editor as Cedalion does.

2.4.7 Wyvern

Wyvern [131] supports nested domain-specific languages within Wyvern code [139, 140]. A DSL block is identified by indentation or predefined quoting, and parsing is type-directed: which language a DSL block is written in is determined by the type to which its result is assigned.

Within top-level Wyvern code a block of DSL code can be embedded within paired quoting characters. Alternatively, a special identifier "~" indicates that a DSL block follows, and the next line begins the DSL block, which must be indented relative to its predecessor; the DSL continues as far as the indented block. The result of the DSL code is inserted at the site of the quoted expression or "~"; the DSL block will be parsed according to the **type** required at that point, and corresponding base Wyvern code generated. Wyvern expressions can be embedded within a DSL block to allow accessing outside state or methods.

We contrast our design with Wyvern in Section 6.5.4.

2.4.8 Protean operators

Ichikawa and Chiba [78] propose *protean operators*, a system of operators for type-directed parsing of internal domain-specific languages, and describe ProteaJ, their system embedded in Java. With protean operators the expected type in a given context is used to determine the parsing rules, as in Wyvern, but unlike Wyvern the code to be parsed under different rules is written inline and without any visible indicator that it is a different language. The authors achieve this by permitting the definition of families of operators returning a particular type, including the empty operator "" (juxtaposition). Each character in a context where a protean operator type is expected is treated as a separate token, and user-defined code executes for each operator to build up a result of the correct type. These DSLs can only be used where the expected type of an expression is statically unambiguous.

2.4.9 External domain-specific languages

The related work above deals with *internal* domain-specific languages, those that are embedded within a host language. An alternative approach is *external* domain-specific languages: those that live outside a host language and have outside tooling to make them work. Some systems, notably Cedalion above, blur the line between the two. Our work is on internal language variants, but we will describe some major external systems briefly here.

Xtext

Xtext [48, 43] is a toolkit for building domain-specific languages, integrated into the Eclipse IDE. In Xtext, a new language is defined by first specifying the grammar of the language and then separately the behaviours that should be attached.

To define an Xtext language the author must define the syntax in Xtext's declarative grammar language, and then write an "inferrer" in imperative code to translate the language defined in the grammar to an implementation language (typically Java). The inferrer can process the syntax tree in any way it chooses and generate arbitrarily complex code; optionally, the language can allow embedded Java expressions and types. Because Xtext is integrated with Eclipse, defining a language also creates syntax highlighting and code completion support for the language.

We contrast Xtext with our design, and show an example Xtext language, in Section 6.5.7.

TXL

TXL [32] is a source-to-source preprocessing transformation system where passages of one or more languages can be embedded in a single file, and then all transformed into a target language. The target language is not necessarily the language of any of the input, although the intent is that passages of a domain-specific language are embedded in a program in a general-purpose language. Each embedded language is entirely separate, but — with careful design — multiple languages may interact safely. Because TXL languages are processed entirely outside the host language they have no innate knowledge of, or access to, the surrounding program, and the language author must account for any features of the target language that may affect correct behaviour of the program, possibly including parsing and computing scope relationships, or else accept that some embeddings may be incorrect or unsafe.

Stratego/XT

Stratego/XT [16] is a program transformation tool intended to take input source code and produce modified source. The Stratego language expresses rules for rewriting syntax and other rules for specifying when and how

particular rewritings should be applied, either by static rules or through dynamic context-sensitive evaluation. The accompanying toolset runs Stratego programs over input source and includes specialised tools and languages for advanced transformations. A single overarching transformation may involve code in several different domain-specific languages. Stratego/XT is intended to work on a single transformation of the input, rather than on embedded passages as in TXL, but a language could be defined with either effect.

Kats and Visser [91] describe the creation of domain-specific languages using the Spoofax workbench, which incorporates Stratego/XT. A programmer may describe the constructs of their language and their intended behaviours, and then write their application logic within the program domain using a customised IDE with autocompletion and syntax checking as they type. A Spoofax DSL definition includes an ordinary grammar and transformation description, with additional information for further IDE integration of the created DSL itself.

Perl

Perl [151] "source filters" [118] allow arbitrary rewriting of the source code of a Perl program when it is executed. These filters are written as Perl code in another module, accessed with **use** name::of::filter. The filter code is given the entire remaining text of the program, and is able to transform it programmatically before execution. This functionality is a built-in feature for Perl programs at runtime.

Source filters are aided by various other modules that assist in transforming by providing parsing and other functionalities. Source filters can provide very different languages, ranging from simple replacements of keywords with words from other languages [171, 29] to embedding passages of entirely different languages [193, 172] or supporting different programming models, up to and including Shakespeare [64]. Alternatively, rather than changing the language a filter can provide checking for common errors [28]

or instrumentation [30]. By doing total textual substitution, source filters can have unrestricted effects, as in Racket dialects. An unusual feature is that some source filters will rewrite the source code in place to correct perceived errors or provide a different representation.

2.4.10 Pluggable checkers.

As well as extending the language a dialect or DSL author may want to check for additional errors or report existing errors differently. Frameworks for adding additional, optional error detection are called *pluggable checkers*. We briefly mention relevant work in this area here.

JavaCOP [116] is a framework for implementing pluggable type systems in Java. This framework provides a declarative language for specifying new type rules and a system for enforcing those rules during compilation. JavaCOP rules may enforce, for example, that a parameter must not be null, or that a field is transitively read-only. A dialect can enforce these rules as well, but is also able to enforce broader constraints by extending or limiting the constructs available to the user of the dialect.

The Checker Framework [144] is a mature library that provides similar functionality to JavaCOP, with better support for overloading and some other Java language features, in part by using an only-partially-declarative syntax. Imperative rules provide more power to the Checker Framework than JavaCOP at the expense of concision.

2.5 Visual interfaces for novices

Several visual programming systems for novices or non-programmers have been developed, with varying approaches. In this section we present related work in this area, and we will contrast them with our approach in Section 7.6.

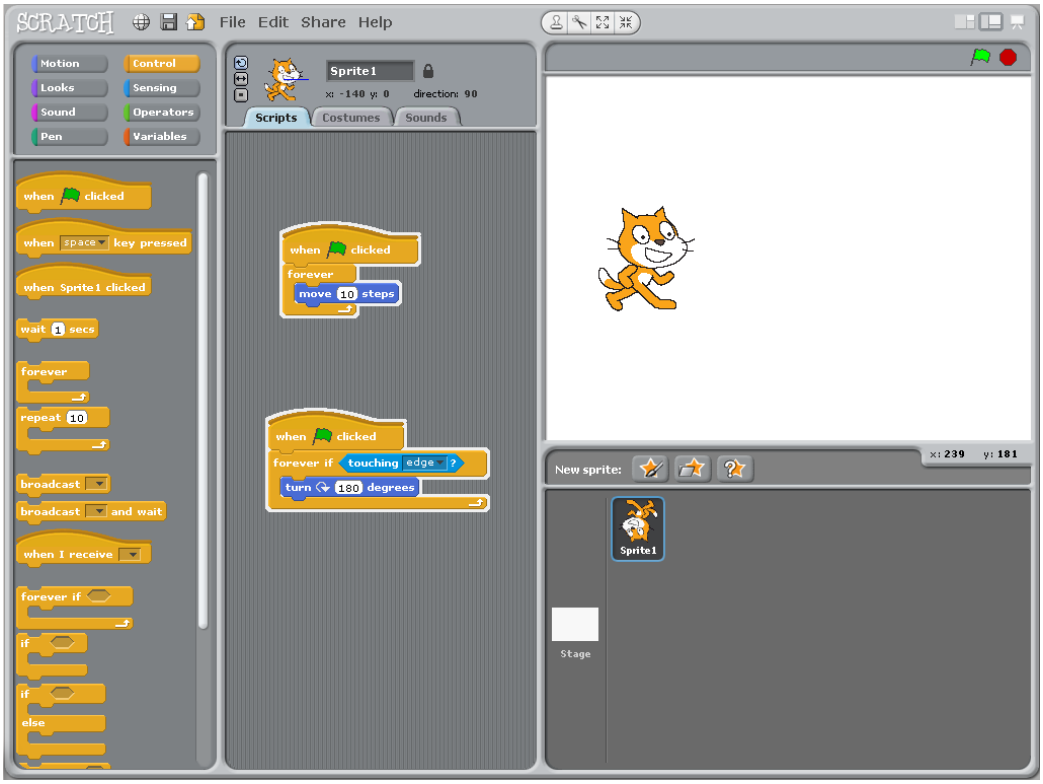


Figure 2.3: Screenshot of a simple program running in the Scratch interface.

2.5.1 Scratch

Scratch [162] is a wholly visual drag-and-drop programming environment with jigsaw puzzle–style pieces, aimed at novices and children. A simple program in the Scratch interface is shown in Figure 2.3. Scratch programs manipulate a persistent microworld; the Scratch environment also includes a persistent graphical area which may contain multiple "sprites", each of which has its own independent code associated with it and may move, draw, or display messages from itself. The Scratch language follows a concurrent event-driven model, where many pieces of code may be executing at once, in the same or different sprites.

Scratch takes full advantage of its purely-graphical nature; the shape of each tile maps exactly to where it is syntactically valid, and some tiles

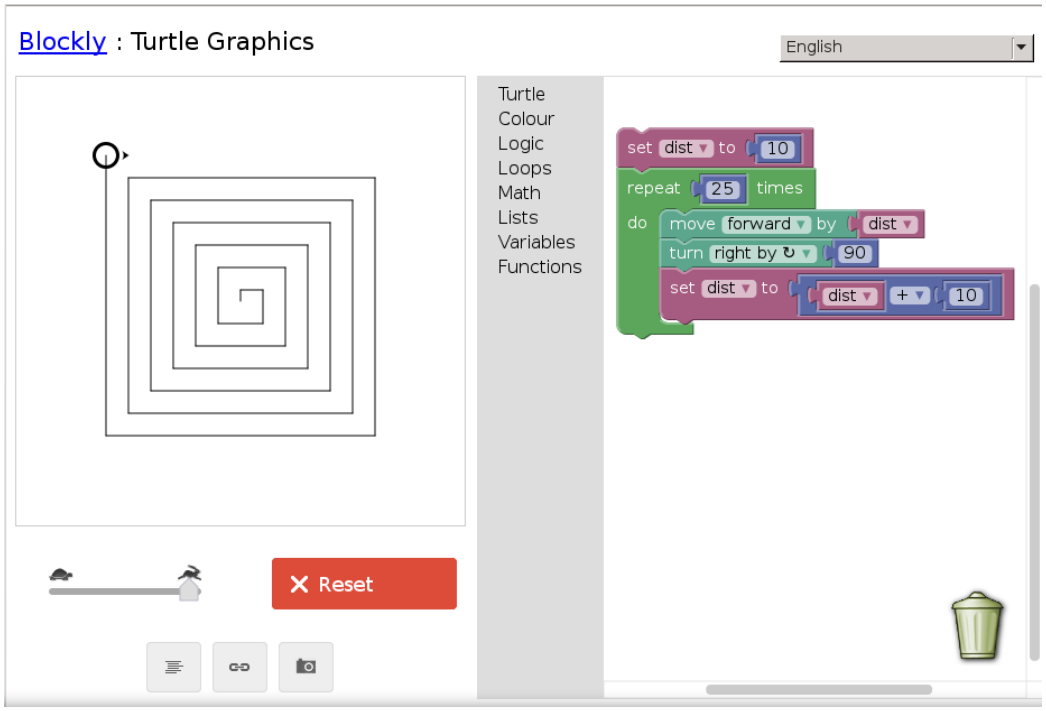


Figure 2.4: Screenshot of a simple program running in the Blockly interface.

combine what would be multiple concepts in most languages into a single element. Consequently, Scratch code does not have a textual form and cannot be "written". Scratch has been found useful for motivating new programmers to begin exploring the ideas of programming, and inspired our work.

2.5.2 Blockly

Blockly [10] is very similar in ethos to Scratch, but incorporates multiple variant languages which can be extended with JavaScript code, and lacks the persistent world of Scratch. Blockly runs entirely in a web browser. A simple program in the "turtle graphics" variant is shown in Figure 2.4.

The user can export their Blockly program to JavaScript or Python code that has essentially the same behaviour as the original Blockly program.

There is no way to reimport exported code back into Blockly.

Blockly supports extending the language with new tiles, and its included demonstrations use this ability. The Blockly interface is embedded in a host application (within a web page), and the host can define additional tiles along with how they should behave. Both the tiles and the behaviours are defined in JavaScript, the language of the host environment.

2.5.3 Codemancer

Codemancer [111] is a game designed to teach programming concepts. Within the game, programs are treated as magic spells, and the player assembles the correct spell to solve a problem out of "runes" (tiles) which they drag into place in the desired sequence to write a procedural program solving a task. A sample program in the interface is shown in Figure 2.5. Runes primarily move the player character around the game world or perform in-world actions, but also include control-flow constructs such as conditionals and loops. In each level the user makes one program that has a single flow of control and a linear definition.

2.5.4 Lego Mindstorms

Lego Mindstorms [99] is a combination software and hardware system for building and controlling small robots built out of specialised Lego bricks. The main environment used for Mindstorms programming uses a drag-and-drop (in some versions, point-and-click) interface. The language focuses on specifying sequences of responses to sensor stimuli, and includes explicit control-flow constructs which are shown through physical layout.

Lego Mindstorms takes its name from Papert's book [143] in which he proposed microworld-based learning. Lego Mindstorms contrasts with the other microworld languages and environments we discuss in that it controls a physical robot in the real (macro) world. The robot has interchangeable components chosen by the user that can vary dramatically in capability.

Figure 2.5: Screenshot of a program in Codemancer from a video demonstration of the software: http://youtu.be/590fFcwIcms?t=2m50s.

2.5.5 Calico

Calico [9, 20] is a multi-language IDE for introductory programming built on top of Microsoft's Dynamic Language Runtime. A visual language called Jigsaw has been built especially for this environment. Jigsaw uses puzzle pieces and drag-and-drop similar to Scratch. A key part of the Calico environment is that code written in one of the supported languages can be accessed from another; as a consequence, a Jigsaw program can run code written in (for example) Python, Ruby, or Scheme as a library function. Jigsaw's execution model is based on translation to Python, and a Jigsaw program can be exported to Python code. Some Python programs can also be converted into Jigsaw code. The Jigsaw syntax does not match Python's, instead using tiles like "repeat n times" for a looping construct and "let" for variable assignments, but the language does map onto Python code. The authors intend to allow Jigsaw code to be translated automatically into (but not from) several textual languages [9]; only Python support appears to be

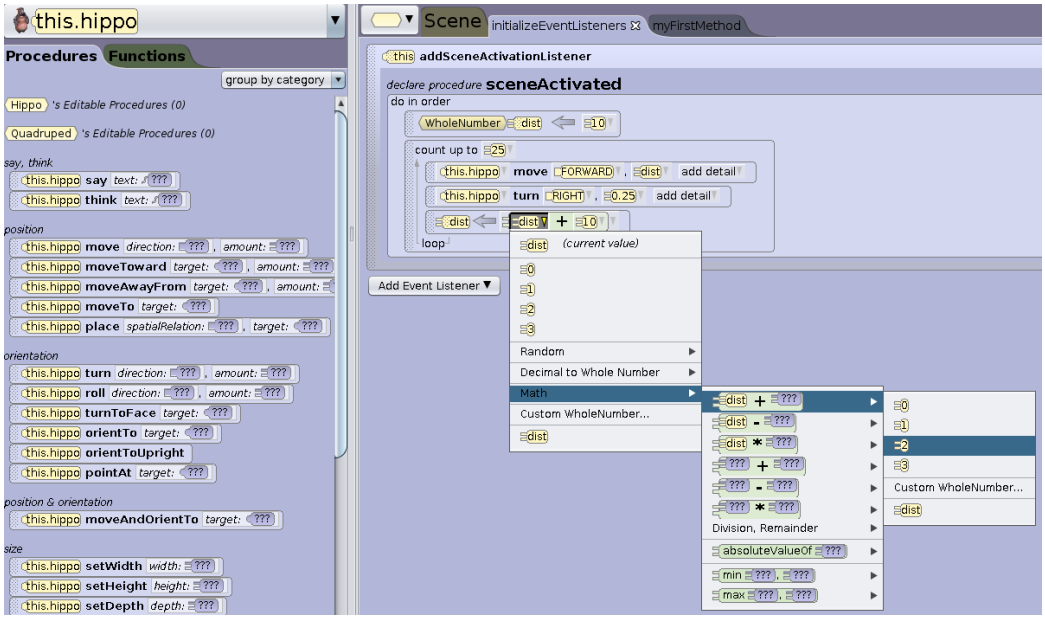


Figure 2.6: Screenshot of a simple program in the Alice interface.

implemented at present. Calico provides several graphical, microworld, and real-world robotics libraries that can be accessed from any supported language, including Jigsaw.

2.5.6 Alice

Alice [31] is a 3D microworld language manipulated by drag-and-drop. The Alice IDE allows users to drop 3D models into the world and associate logic with them. Each object in the world is also an object in the object orientation sense, and can respond to outside events and messages. All code is strongly statically typed. Code editing is by drag-and-drop and menu selection; there is no concrete text, although recent versions of Alice can also export code to Java. Some code is shown in the Alice interface in Figure 2.6.

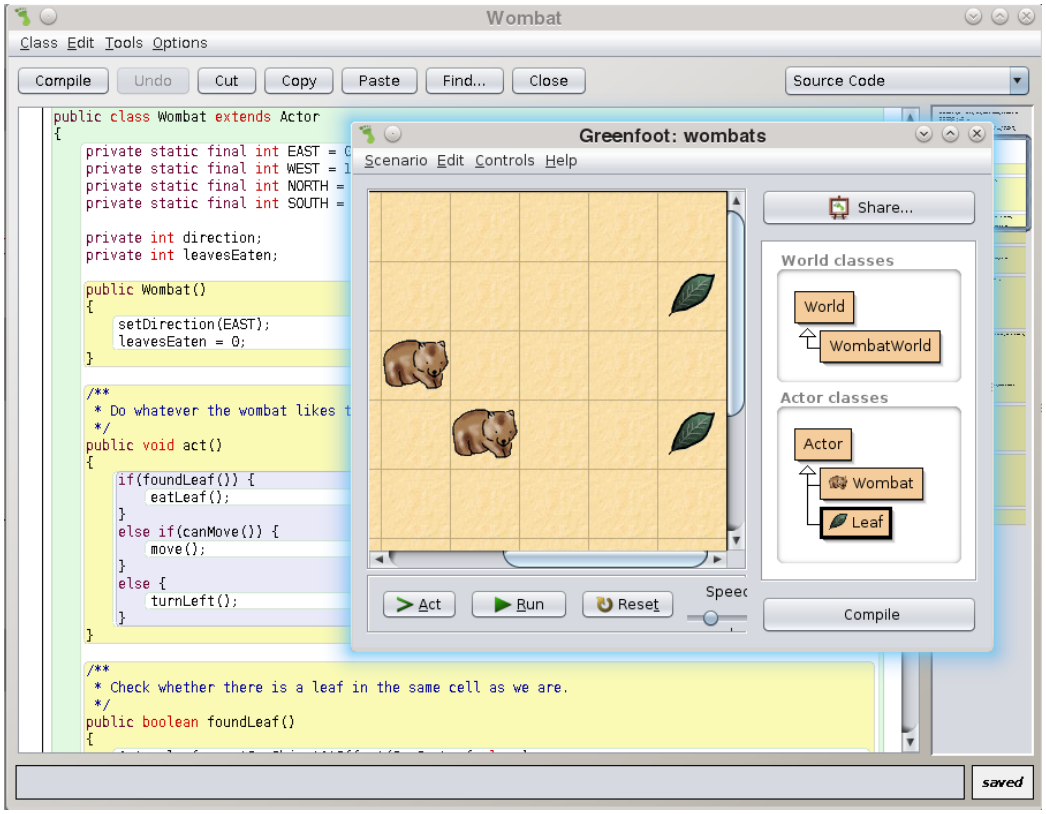


Figure 2.7: Screenshot of code and microworld in Greenfoot.

2.5.7 Greenfoot

Greenfoot [95] is an IDE for a subset of Java, presenting a graphical microworld based on the Actor model. The Greenfoot tutorial program is shown in Figure 2.7. Users write textual source code, but many high-level concepts are available as built-in methods of the world, or are predefined for all actors. Code is written and accessed only with textual Java syntax, which users must learn assisted by common IDE features.

2.5.8 TouchDevelop

TouchDevelop [75] integrates an essentially textual language with an IDE aimed at touch-screen usage. The IDE avoids most use of textual input by

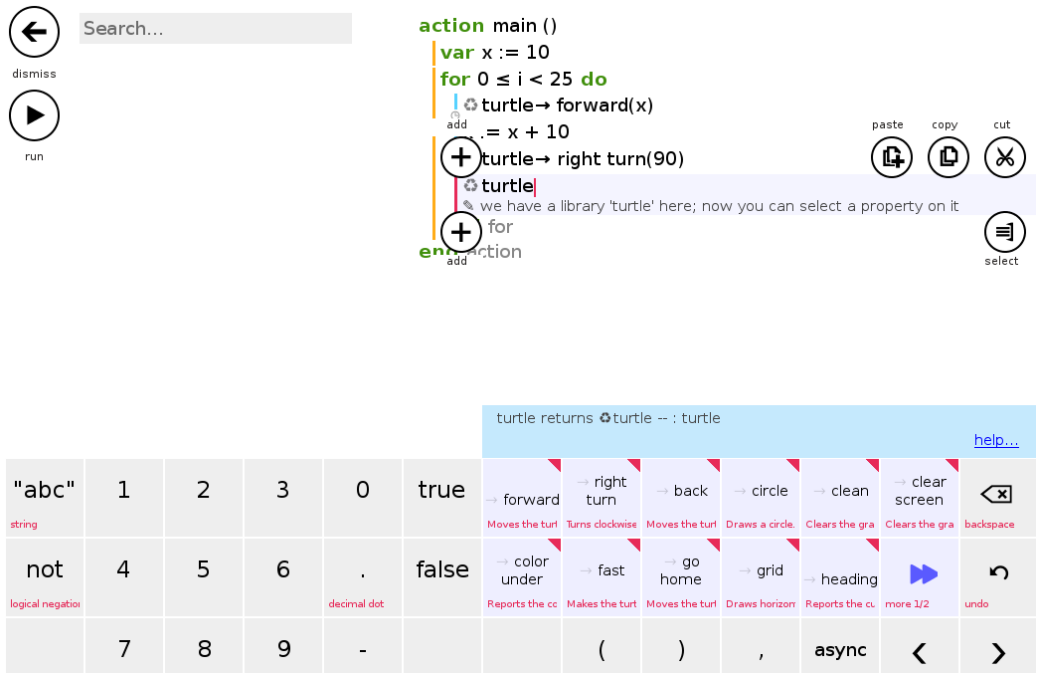


Figure 2.8: Screenshot of code editor in TouchDevelop.

having the user manipulate the syntax tree itself: the user touches where they want to change and the IDE presents them with a list of options they can put there. When the programmer adds a new element the system will prompt them to fill in any required arguments, like the condition of a loop. A simple "turtle graphics" program is shown being edited in Figure 2.8.

The syntax is reasonably conventional, although symbols are used to mark method calls and some aspects, such as comments, are shown only by typographic features. Programs are always shown textually with light visual annotation, and editing always corresponds essentially to a textual insertion or deletion.

The interface is designed to be used on tablets and mobile phones, as are the resulting programs, but they may also be used without a touch screen.

2.5.9 Droplet

Droplet [40] is a combined visual-textual editor that treats block-oriented visual display as "syntax highlighting" for text, and can transition between the two displays. Droplet supports multiple languages through writing CoffeeScript or JavaScript code defining blocks and a parser. Droplet aims to be useful both for transition from visual to textual languages and as a general editor on mobile devices.

2.5.10 Graphical overlays on programs

Our work will incorporate some graphical overlays showing relationships between different parts of a program, so we also briefly present related work on that topic.

Spreadsheets represent a common form of dataflow programming performed primarily by non-programmers. Frequently, spreadsheet applications can present information on dependencies between cells by overlaying graphical indicators (such as arrows and cell highlights) on top of the base spreadsheet display. Grigoreanu *et al.* [61] investigated the efficacy of these overlays in aiding end users to find errors, finding that they were useful but that users often did not deploy them. Igarashi *et al.* [83] proposed additional transient views involving animation and other features to help users gain more information from these overlays. Burnett and others [19, 4] have applied techniques from specialised visual languages to spreadsheet overlays, with particular focus on displaying smaller subsets of the available information to help users follow along.

DrRacket [182] is an IDE for the Racket language, a dialect of Scheme aimed at education. The editor is purely textual, but includes an overlay system linking definitions with their usages with arrows when the mouse is hovered over a term in the editor.

Omar *et al.* [141] describe "active code completion", where pop-up "autocomplete" menus are augmented with additional behaviour. For

example, in a context where a colour expression is known to be required, the popup can present a colour picker and generate code corresponding to the chosen colour; in another context, the popup can indicate expressions on values available in scope that correspond to the correct type, and show where these values are found. All completions in their system are type-directed.

Chapter 3

The Grace Language

Grace [7, 8] is a new imperative object-oriented programming language aimed at education, with the goal of permitting a variety of teaching styles within a single language. This chapter provides an introduction to the Grace language and its features.

3.1 Goals of Grace

Grace is intended to be used for introductory programming courses. Consequently, it aims to look familiar to instructors who know other object-oriented languages, while supporting the individual teaching styles and sequences that instructors have. In Grace it is possible to write statically-typed code, dynamically-typed code, or a mixture of the two, and move from one to the other. A course could begin with objects, classes, functions, or procedural code, and move from one approach to another while staying within the same language.

To satisfy these goals each part of the language must be designed so it does not interfere with all the other parts. Grace does not include a large number of new features, or a new paradigm for programming: instead, existing features that have shown themselves useful in existing languages are brought together and integrated. The following sections describe the

features of the language, with a focus on those necessary to understand the work in this thesis, and with examples of parallel functionality in other languages.

One particular goal of the language is to eliminate *incantations*. An incantation is a piece of code present to satisfy the requirements of the language, but which does not have any particular role in meeting the programmer's intention for their program. A common example of an incantation is Java's:

```
public static void main(String[] args) {
```

which must appear in every runnable program (and inside a class declaration, another incantation). Calling these incantations is not to imply that they are meaningless: to the contrary, every part of the above incantation is meaningful. In fact, that single line includes six distinct and meaningful concepts: **public**, **static**, **void**, main, String, and []. Changing any one of these will cause the program not to run, but explaining to a novice seeing their first program the meaning of a static method is nontrivial, particularly when they have never seen any other kind of method before. The temptation then is simply to tell the student to ignore the incantation; this choice may even be worse, because then the student really does learn to treat it as an incantation. Grace aims to avoid all of these pieces of boilerplate code in order to allow users to focus on the essential complexity of their program, and instructors to introduce concepts in the order they desire when students are ready for them; Grace's syntactic and semantic choices derive from that goal.

3.2 Variables and literals

Grace includes two kinds of variable declaration: a mutable variable declared with **var**, whose binding may change during the program, and an immutable definition declared with **def**, whose binding is fixed. Local variables can be declared and used like this:

```
var x := 1
def y = 2
var z
x := x + y
z := x * y
```

After this code, x has the value 3, and z is 6. Attempting to update the value of y would cause an error. Note that **var** declarations and assignment statements use :=, while the **def** declaration uses =. In Grace, all assignments to things that may change use :=, and all fixed definitions use =, because the language designers feel it is important to emphasise that these are different. A **var** declaration need not initialise the variable immediately, but it will be a fatal error to read from an uninitialised variable. Variable names must start with a letter and can contain letters, numbers, and the apostrophe character. The same variable name cannot be declared again within the scope of an existing definition; this restriction is the "no shadowing rule".

In the example above we used literal numbers in the source code. Grace permits both number and string literals as special syntactic forms:

```
def ten = 10
def half = 0.5
def binary = 2x110110
def hex = 0xfacade
def message = "Hello, world!"
```

Numeric literals are in decimal by default, and may include a fractional part. Literals in non-decimal bases from 2 to 36 are written with the base (in decimal) before an x and the digits in that base thereafter. The base 0 is a synonym for 16. String literals are enclosed in double quotation marks. Strings can also include interpolated code, which is evaluated and its value inserted at the point of the interpolation:

```
def half10 = "Half of ten is {ten * half}"
// half10 is now "Half of ten is 5"
```

Strings can be concatenated together with the ++ operator.

3.3 Objects in Grace are closer than they appear

In Grace, everything is an object, by which we mean a combination of identity, state, and behaviour: there are no "primitives" or variables that do not refer to objects. The number 1 is an object, and the numeric literal 1 refers to that object.

As well as the objects constructed by numeric and string literals, a programmer can construct their own objects with an *object constructor* (or *object literal*):

```
object {
  def x = 1
  var y := 2
  method increment(by) {
    y := y + by
    return y
  }
}
```

Mutable and immutable fields are declared with exactly the same syntax as local variables, using **var** and **def**. The initialisation expressions of fields are executed inside the object when it is constructed.

Methods are declared using the **method** keyword. Methods accept parameters and return the value of the last expression in their body, or the value explicitly returned with a **return** statement. Parameter names may not shadow any other variable in scope. Any parameter can be silently discarded by naming it "_".

An object constructor creates a new object every time it executes, and returns a reference to that object which can be saved, returned, or passed as a parameter, just as any object reference can be. Inside an object, the **self** keyword is always a reference to itself. Objects can be nested, and the surrounding object obtained through the **outer** keyword.

3.4 Methods

To access a method of an object we use a *method request*, which is a method call; we will be using these terms interchangeably in this thesis. The usual form of a method request is to write a reference to the object, a dot, the method name, and the argument list:

```
pt.setCoordinates(x, y)
```

Some common cases are optimised. When the method is on the current object or a surrounding object, the receiver (the part before the dot) can be omitted, as can the dot. When there are no arguments, the argument list can be omitted. When there is exactly one argument and it is "self-delimiting", like a string, the parentheses can be omitted.

```
print(1 + 2)
window.show
greeter.greet "World"
```

The term "method request" is used because the language designers feel it is important to stress that the agency of determining what to execute belongs to the receiver object, and that the more common "call" indicates that the caller has chosen what to do; they avoid "message send" to avoid confusion with network messages.

There are some other special kinds of method request: operators and field accesses.

3.4.1 Operators

In Grace, operators (like the +, *, ++ used earlier) are also methods, and accessed by method requests. An operator method is simply a method whose name is entirely operator characters[1]. An operator method defines

[1]Exactly which characters these are is not defined: our implementation (Chapter 8) supports the ASCII "keyboard operator" characters and members of the Unicode categories "Symbol, Mathematical" and "Symbol, Other".

an infix binary operator. A request for an operator method is written between its receiver on the left and its argument on the right:

```
2 * 3
"Hello" ++ "World"
def x = object {
  method !@^*(other) {
    print "Hello, {other}."
  }
}
x !@^* "world"
```

The four standard arithmetic operators +, −, *, and / have their standard precedence and associativity. There are no precedence rules for non-arithmetic operators: it is a syntax error to mix different operators without explicitly parenthesising to disambiguate them. All operators are left-associative with themselves: their left-hand side is fully evaluated first, and they are given the single minimal expression immediately to their right as their argument.

Grace also supports unary prefix operators, such as boolean negation !. A prefix operator is a method defined with the name prefixX, where X is a valid operator name. The method defining the prefix operator ! is called **prefix**!:

```
def a = object {
  method prefix! {
    return "Hello"
  }
}
print(!a) // -> "Hello"
```

Prefix operators bind more tightly than other operators but less tightly than other method requests.

Grace supports only prefix unary operators and infix binary operators in order to avoid building in any detailed concept of operator syntax and

precedence. With these choices the operators can always be differentiated solely by where they are found in the source: when seen immediately after a term they **must** be a binary infix operator, and when not immediately following a term they **must** be a unary prefix operator.

3.4.2 Field accesses

Reading a public field from an object uses exactly the same syntax as a zero-argument method call: x.y. Writing to a public mutable field uses Grace's standard assignment syntax:

```
x.y := 4
```

In both cases, real methods can be defined to behave in this way, or to override an existing field. A reader method is simply a zero-parameter (nullary) method of that name, while a writer method is one with := on the end of its name:

```
method y:=(value) {
  validate(value)
  realY := value
}
```

3.4.3 Multi-part method names

Method names in Grace can include multiple parts with separate argument lists in between. Multi-part methods are defined in exactly that way:

```
method check(x)between(min)and(max) {
  (x >= min) && (x <= max)
}
```

This method name has three parts, "check", "between", and "and". Each part has its own parameter list. This method could be called in the same way:

```
check (5) between (3) and (7)
```

As for single-part names, self-delimiting single arguments do not require parentheses, but all other argument lists do.

When a method name is long it may be convenient to split the request over multiple lines. The programmer can do so provided that they indent the second and subsequent lines of the request to indicate that the line is a continuation of the previous, or have the next part immediately follow the end of the argument list for the previous part:

```
check(5)
  between(
    3
  ) and(7)
```

3.4.4 Visibility

Grace allows setting the visibility of fields and methods with annotations. Annotations use the syntax **is** annotationName(arguments). The annotations for visibility are public and confidential.

A method in an object is public by default, but can be made *confidential* so that it is only available within the object and inheritors:

```
def x = object {
  method foo(a) is confidential { ... }
  method bar { foo(5) }
}
x.bar     // OK
x.foo(5) // Error – 'foo' is not accessible
```

Confidential methods cannot be accessed on any other object but **self**.

A field in an object is confidential by default, and can be made *public* so that it is accessible from the outside:

```
def x = object {
  def a is public = 1
  var b := 2
}
print(x.a) // OK
print(x.b) // Error — 'b' is not accessible
```

The term "confidential", rather than the more common "private", was chosen because the language designers feel that novice programming language learners would otherwise be confused by the slightly different meaning of "private" between Grace and Java.

In the case of a mutable field the programmer may want to allow reading of the field, but not writing; in this case they can annotate the field readable:

```
def x = object {
  var b is readable := 2
}
print(x.b) // OK
x.b := 2   // Error — 'b:=' is not accessible
```

Other user-defined annotations are possible; an annotation is syntactically a method request to any method that is in scope.

3.5 Blocks

A key construct of Grace is the *block*, also known as a lambda or first-class function. A block is written in braces and constructs an object representing a piece of code that may be executed in the future. A block can have parameters and is executed using its apply method.

```
def block = { print "Hello" }
block.apply // Prints "Hello"
def greeter = { name -> print "Hello, {name}" }
greeter.apply "world" // Prints "Hello, world"
```

The code in a block is nested inside the surrounding lexical scope and can access any methods or variables defined there, including whatever **self** is in that scope.

A block may be executed zero, one, or many times. This fact is key to defining some of Grace's basic control structures. In combination with multi-part method names, Grace avoids having any built-in control structures at all. Instead, if()then()else, for()do, and while()do are methods.

```
if (x < 0) then { print "Negative" } else { print "Non−negative" }
for (1..10) do { i −>
  print "I can count to {i}!"
}
while { x > 0 } do {
  print "I can count down to {x}"
  x := x − 1
}
```

if()then()else accepts a Boolean as the condition and two blocks: one to apply if the condition is true, and another to apply if the condition is false. The condition expression is evaluated only once, and only one of the other blocks is applied.

for()do accepts an iterable object and a unary block, and applies the block once for each element in the iterable, passing the element in as an argument each time. The iterable expression is evaluated once, and will be asked for each item in turn; the block is applied many times.

while()do is the most unusual-looking of these methods. Unlike in most languages that use braces, the condition of the loop is written in braces as well as the body: because the condition of a while loop is evaluated many times, it must be a block, as blocks are what provide deferred and repeated execution. The method accepts two arguments, both of which are blocks, and applies them alternately until the condition block returns false.

3.6 Classes

Grace classes are syntactic sugar for nested object and method declarations: they are objects just like any other, and they do not represent types. A class declaration is written:

```
class foo.bar(x, y) {
  method sum { x + y }
}
```

This declaration is exactly equivalent to the unrolled form:

```
def foo = object {
  method bar(x, y) {
    object {
      method sum { x + y }
    }
  }
}
```

Class declarations exist to allow teaching about classes, and to make some kinds of code shorter. It is never necessary to use a class, as objects and methods are exactly equivalent, and types (Section 3.8) are not coupled to classes.

3.7 Inheritance

Any object constructor in Grace can inherit using the **inherits** keyword. Following the keyword must be an expression resolving to the construction of a fresh object; a class instantiation is one such expression, but any request of a method that always returns an object constructor suffices. Because the object constructor must be tail-returned it is always clear locally when an object can be inherited from:

```
method superobj(x') {
  // This method always returns an object
  // constructor as its final action, so
  // the method can be inherited from.
  return object {
    method x { x' }
  }
}
def a = object {
  inherits superobj(5)
}
a.x // 5
```

An inherited method can be overridden in the sub-object. All requests for that method will then execute the overriding method. To access a method from the parent object that has been overridden the programmer can use the **super**.x syntax inside the object. **super** is only valid as a receiver, and cannot be used in any other context.

During inheritance, all methods are defined in their final forms in the object. Any initialisation code in field declarations and ordinary code in the body of an object in the inheritance chain is executed from top to bottom in the super-most object first, and so on through to the final sub-object. If any of that code requests a method on **self**, the final overridden version will be executed. **self** always refers to the same object identity throughout the process, as in most object-oriented languages; it is never possible to observe a parent object directly or to see methods being defined.

3.7.1 Chained inheritance

To define a method that calls another method with different parameters, but which can still be inherited from, an additional degenerate object can simply be inserted in the intermediate method:

```
method midobj(n) {
  object {
    inherits superobj(n / 2)
  }
}
def a = object {
  inherits midobj(10)
}
a.x // 5 again
```

This device allows more complex inheritance chains while still making clear locally what can and cannot be inherited from.

3.8 Types

Types in Grace are optional, gradual, and structural. They are *optional* because they can be omitted; they are *gradual* because they can be partially present, checked at run time when not enough is known statically; and they are *structural* because they depend only on the interface of an object and not its implementation.

Grace supports two kinds of type annotation. A variable, field, or parameter can be annotated with a type after a ":", and the return type of a method can be given after a "−>", written immediately after the parameter list.

```
def s : String = " "
method add(x : Number, y: Number) −> Number { x + y }
```

A type is declared using the **type** keyword:

```
type Greeter = {
  greet(n : String) −> Done
}
```

The above declares a type called Greeter, which requires that any object that belongs to it have a method called greet accepting a single argument,

which is a String, and returning Done. Done is the type of a statement or action that has no meaningful return value; all objects belong to the Done type, but it has no useful methods. The name Done was chosen, rather than the more common Void because the designers wished to stress that the type was not in fact uninhabited. As with all types in Grace, Done can be omitted from the declaration to use purely dynamic typing. An anonymous type is written with the **type** keyword and braces:

```
method halfsqrt(x : type { sqrt −> Number }) { x.sqrt / 2 }
```

There is no "cast" operation in Grace: when an object needs to be treated as a different type than it is currently known to be, and simple subtyping does not suffice, the programmer must either pass the object through dynamically-typed code or explicitly use the pattern matching we designed and present in Chapter 4.

Grace follows the standard co- and contra-variance rules for subtyping: in order for B to be a subtype of A, the corresponding methods in B must return subtypes[2] of the return types of the methods in A, and their parameters must be supertypes of the parameter types in A.

As Grace is gradually typed there is another type, Unknown, representing a type which is not known statically. The unknown type is statically compatible with any type: a value of type Unknown can be assigned to a variable of any type and this assignment will be permitted statically, but may cause an error at run time.

Both when defining a type and when writing a type annotation the programmer may use type operators to construct a type out of existing types. The available operators are &, |, and +.

The & operator is type intersection: A & B contains those objects that belong to both A and B. The objects in this intersection must have all the methods from both types with compatible signatures.

The | operator gives the *untagged variant* type: A | B contains those objects that belong to either A or B (or both). Given a reference of type A | B, only

[2] As usual, types are regarded as subtypes and supertypes of themselves.

those methods in both A and B with mutually-compatible signatures can be called. An object with only these methods does not belong to the type, however; to belong to A | B the object must belong to at least one of the types on their own.

The + operator is method intersection: A + B is the type with all the methods common to both types. A + B is a supertype of A | B, but an object that belongs to A + B is not necessarily either an A or a B. While practical need for + is rare, it is included for completeness.

3.8.1 Generic types

Grace types can have generic parameters written in <>:

```
type List<T> = {
  add(e : T) −> Done
  get(n : Number) −> T
}
var listOfStrings : List<String>
```

Methods can also have generic parameters, also written in <>:

```
method greet<T>(x : T) { ... }
greet<String>(name)
```

A method may have multiple generic parameters. In a multi-part method name the generic parameter list is always written on the first part. Generic parameters are reified and retained at run time.

To permit gradual typing, generic type arguments may be omitted from a method request. In this case, they are populated with Unknown at run time. Either all generic arguments must be provided or none, but any or all can be explicitly given as Unknown if desired.

3.9 Pattern matching

Grace supports an object-oriented pattern-matching system. We present this system in Chapter 4.

3.10 Modules

All Grace code is written inside a module. Grace modules are objects, so methods and fields can be defined at the top level. Modules can import other modules to allow splitting a program into multiple parts. We present the module system of Grace in Chapter 5.

3.11 Dialects

Grace supports a system of language variants called dialects. Dialects allow extending or restricting the language available to users, principally to support different teaching styles and stages of progress. A module declares the dialect it is written in using a **dialect** declaration on the first line. We present the dialect system of Grace in Chapter 6.

Chapter 4

Patterns as Objects[1]

4.1 Introduction

While Grace is an imperative, object-oriented language, an important aim of the design has been to give instructors the freedom to choose their own teaching sequence, by reducing dependencies between features in the language and its libraries. The language supports many different approaches within the object-oriented paradigm, but the ACM CS2013 curriculum regards both object-oriented and function-oriented programming as core topics, and requires computer science programmes to cover both dynamic dispatch and pattern matching [88]. To support teaching in this function-oriented approach Grace must support programs organised in a more "inductive" style: as collections of functions or procedures that make decisions based on the types or values of their arguments.

In this chapter we present our design of an object-oriented pattern-matching system for Grace. Our system is fully object-oriented, but embedded naturally into the language. We seek to minimise the language support required to integrate patterns, but to present an interface as typical as possible. We want users and instructors to be able to define their own

[1] This chapter expands upon a paper [74] published in the 2012 Dynamic Language Symposium.

patterns and use them equally with patterns we define, but without needing to understand a complex protocol. It is balancing these conflicting goals that makes the contribution of this chapter, and we believe we have found the design that sits best within these constraints. While our motivation is to include this functionality in Grace, the system we present could be applied in similar languages.

This chapter explains how we solved the problem *gracefully*. Our approach draws on well-known techniques: modelling patterns and cases as partial functions, reifying those functions as first-class objects, and then building-up more complex patterns from simpler ones using pattern combinators. This results in flexible pattern-matching and case statements that incorporate a programmer-extensible range of pattern matches, including matching against constants, matching against an object's type (that is, its method interface), and also binding a variable of the new type, matching against the value of a variable or expression, and "destructuring" an object to extract its components, which requires the cooperation of the object in determining what those components should be.

All of this is presented in a conventional pattern-matching syntax and implemented using two localised language extensions: treating blocks (lambda expressions) as *partial* functions, and binding variables in nested patterns. In terms of the vocabulary we set out in Section 2.2, we have designed an exherent system, where patterns are first-class entities, but with the syntactic simplicity of an inherent system, bridging these approaches with the minimum intrusion onto the language.

Our design for pattern matching was inspired by, and significantly based upon, the pattern-matching designs in Scala [46] and in Newspeak [58]. The "look" of our match-case construct is derived from Scala, while the underlying framework draws from Newspeak, although there are significant departures from both in order to form a coherent matching system.

This chapter presents three elements of the design of this pattern-matching feature: Section 4.2 briefly describes our conceptual model of

patterns and matching; Section 4.3 describes how we have embedded pattern-matching syntax into the language, using only a single localised extension to the syntax of blocks; and Section 4.4 presents the reification of the conceptual model as objects supporting that syntax, based on unifying partial function objects, pattern objects, and pattern combinator methods. Section 4.5 shows the integration of pattern-matching with Grace's type system, necessary for the system to be integrated into the language itself.

Section 4.6 describes a generalisation of pattern-matching to all type annotations in the language. In Section 4.7 we discuss the particular design decisions we made, the roads that we did not take, and how our choices compare to those made in other languages (notably Scala and Newspeak).

4.2 Conceptual model

In common with Newspeak, we treat a case statement as a combination of partial functions — functions that are defined on a restricted domain of inputs, as described in Section 2.2. A request to apply a partial function must supply an argument. If the argument is in the domain of the function then the function executes using that argument and returns a result, but if the argument is outside the domain, the function fails and is not executed.

We represent both the partial function itself and its domain as objects. Because Grace already has a representation of total functions as objects (blocks), we extend these to partial functions. A series of method requests ascertains whether the argument is in the domain, applies the function, and returns the result. This representation of partial functions allows pattern matching to be added with minimal disruption to the language.

As Grace is gradually typed, even without pattern matching it is possible to write a type on a block parameter and to attempt to apply the block to an argument of a different type. If such an application occurred at run-time, the program would report a type error and terminate. To add partial functions we extend blocks with a `match` method to also allow for a *non-fatal*

indication of a type mismatch. The match method returns either a result indicating that the argument was outside the domain of the function, or the result of the function encapsulated in an object indicating a successful match.

To permit matching on values, such as numbers and strings, we generalise the annotation of a parameter's *type* to a *pattern*: all types are patterns, but patterns can also represent individual objects or values, sets or ranges of values, bit patterns, or any other criteria that can be defined in code. We represent patterns as a nested composite object structure, using the same match method as for blocks.

Patterns are permitted as annotation only on the single argument of a partial function block. A *destructuring pattern* written there can extract values from an object, declaring additional patterns that the components must match, and potentially binding the extracted values to names. These are the *only* language-level impacts that pattern matching and case statements place on Grace: the extension of type annotations into pattern annotations, and the ability of patterns to bind additional parameters. Unlike in other languages, there are no macros, no rewriting, no additional control structures, and nothing special about destructuring.

4.3 Graceful patterns

This section describes how patterns appear to the Grace programmer. The syntax is explicitly conventional, familiar to programmers of Scala, F#, Haskell, and other languages, and involved only minor extensions to Grace.

4.3.1 match()case()...case

The programmer's interface to pattern matching is the set of match()case() ...case methods, with the same form as other Grace control structures. The

method takes as its first argument the *target* of the match, and as succeeding arguments some number of matching blocks. These matching blocks are ordinary Grace blocks that have been extended to incorporate a pattern literal as the parameter list; the body of the block contains the code to be executed when the pattern matches. This is best explained using an example, which we will use through the next sections:

```
match(expr)
  case { 0 −> "zero" }
  case { n : Number −> "Number less than {n+1}" }
  case { s : String −> "String \"{s}\"" }
  case { x −> error "Unexpected value {x}" }
```

This match expression first evaluates expr to obtain an object obj, and then attempts to match the patterns in sequence, executing the first case block whose pattern matches obj. The whole match...case returns the return value of the executed block, and no further matches are attempted. We designed this syntax carefully in order both to fit into the existing language and to be "obvious" in meaning in the simple cases, particularly to a reader familiar with pattern-matching in other languages.

4.3.2 Matching blocks

A matching block has the syntax of an ordinary block literal, enclosed in {}, but also incorporates a pattern literal before the −>, where the parameter list would be.

The syntax for patterns is a strict superset of that for the parameter list of a single-parameter block: all single-parameter blocks are matching blocks. In fact, a matching block satisfies both the interface of a block (with apply) and of a pattern (with match). The model of execution for matching blocks is described in Section 4.4.8; for now, they simply encapsulate both a pattern and some corresponding code.

4.3.3 Literal patterns

In the example above, the first pattern is the literal `0`, which matches the number object 0. All numeric and string literals can be used as patterns, and match themselves (they are what we term *autozygotic*).

If `expr` evaluates to `0`, including as the result of some calculation, the pattern will match and the match expression will return `"zero"`.

4.3.4 Type patterns

The second pattern `n : Number` matches when `obj` has type `Number`, but also has the effect of binding `n` to `obj` within the body of the block. This syntax is identical to that for declaring a block with a single parameter of type `Number`, a deliberate choice to present a consistent syntax that may be understood by readers who do not know the pattern-matching system. Within the body of the matching block `n` will have type `Number`, exactly as when applying a block outside of matching.

The third pattern is similar, but matches when `obj` has type `String`. Because Grace is gradually typed, static types like these may be used in code that is otherwise dynamically typed. Because Grace types are structural, when used as patterns they match whenever all of the methods in the type are found in the object with appropriate signatures. An anonymous type can be used to check for only a particular method:

```
case { x : type { asString −> String } −> x.asString }
```

In fact, any piece of pattern or type syntax can appear after the : here; a type expression simply means a pattern matching exactly the members of that type. This behaviour will be described in detail in Section 4.4.

4.3.5 Variable patterns

The final pattern introduces a new parameter named `x`; the pattern always matches (it is *irrefutable*) and has the effect of binding `x` to `obj`. Again, this has

the same syntax and behaviour as a block with a single untyped parameter.

The syntax for a variable pattern is simply any identifier in "left-hand-side" context: either before a ":" as in type patterns, or on its own as a default case as in the example. Variable patterns can also be used inside a destructuring pattern (Section 4.3.8). An identifier written in this context is always a variable pattern, and never refers to any variable in the surrounding scope that might already exist. We address referring to preexisting variables in Section 4.3.10.

4.3.6 Wildcard pattern

To write a pattern that always matches but does not bind a parameter, the wildcard identifier _ may be used:

```
case { _ -> error "Unexpected value"}
```

The wildcard pattern is useful in situations where an irrefutable pattern is necessary, but binding the target of the match to a new variable name is not required. The _ explicitly indicates that the value is intended to be accepted, but discarded, just as it can be used as a name for superfluous parameters to methods or blocks. _ is useful not only as a "default" catch-all case, but to perform a type match without saving the result, or as part of destructuring (Section 4.3.8).

4.3.7 Combinators

Patterns can be combined using the pattern combinators & and |. The pattern a & b matches when patterns a and b both match:

```
type X = { x } // the type with method x
type Y = { y } // the type with method y
match (val)
  case { o : X & Y -> "Point ({o.x}, {o.y})" }
```

while a | b matches when either pattern a or pattern b matches:

```
match (val)
  case { _ : Number | String | Boolean −>
    "A value of a built−in type"
  }
```

This syntax again corresponds exactly to an existing Grace feature: & and | are used to construct intersection and union types, respectively, and where both branches refer to types the semantics are identical. In our X & Y example, o must both be matched by the patterns X and Y and conform to the type X & Y (which is **type** { x ; y } — having both an x and a y method). Within the body of the block, o has the corresponding type X & Y.

4.3.8 Destructuring

Patterns can also extract data from the matched object for binding or further matching. We call this use a *destructuring match*; it requires that the matched object cooperate by providing a method that exposes the necessary data (this method is not visible in the syntax).

```
match (astNode)
  case { nd : ASTString("") −> "Empty string" }
  case { nd : ASTNumber(n) −> "The number {n}" }
  case { nd : Operator("+", ASTNumber(0), y) −> "Just {y}" }
  case { nd : Operator("+",
            m : ASTMember(name : String,
                    Identifier("self"), y))
      −> "self.{name} + {y}" }
  case { nd : Operator("+", x, y) −> "Adding {x} and {y}" }
```

Destructuring matches can be nested arbitrarily deeply. Each subpattern can use the full pattern syntax, including literal patterns and combinators, variable and type patterns, or other destructuring matches. The same variable name may not be bound at different points in the match. Instead, repeated bindings will be a static error under Grace's "no shadowing"

rule (Section 3.2).

In this example we see matching against the literal patterns `""`, `"+"`, `"self"`, and 0, the variable patterns n, y, m, and x, the type pattern name : String, and general destructuring patterns ASTString, ASTNumber, Operator, ASTMember, and Identifier. Every kind of pattern appears directly inside a destructuring match at some point.

The pattern as a whole will only match if the outer pattern matches, and the values the object destructures to under that pattern also match the inner patterns. If an outer pattern matches but some of the inner patterns do not, the entire match fails and the next matching block will be tried. This simplifies code over the non-destructuring version. Without destructuring, nested match...case or if statements would require some combination of repeated code, many methods each dealing with some part of the matching tree, and non-local returns to achieve the same behaviour, compared to largely-declarative and shorter code using destructuring matches. Because of this, we consider the added syntactic and conceptual complexity of destructuring itself to be worth the cost.

4.3.9 Predicates

While a programmer can define a custom pattern with any semantics they want, a common case will be to perform some test and either succeed or fail on the basis of its result. We can aid this case by providing a simple way of creating a pattern with these semantics from a predicate.

We define a prefix ? method on blocks to lift them to patterns in this way. If b is a block implementing a predicate (that is, having a single parameter and returning a Boolean), ?b is a pattern that matches exactly when that predicate is true. These *predicate patterns* can either be used inline in a match or saved as pattern definitions:

```
def OddNumber = ?{ x : Number −> (x % 2) == 1 }
match(n)
  case { _ : ?{ x : Number −> (x % 2) == 0 } −> print "{n} is even!" }
  case { _ : OddNumber −> print "{n} is odd!" }
```

The prefix ? method has essentially this definition:

```
method prefix? {
  object {
    inherits BasePattern.new
    method match(o) {
      def matchResult = outer.match(o)
      if (!matchResult) then { return FailedMatch.new(o) }
      if (!matchResult.result) then { return FailedMatch.new(o) }
      return SuccessfulMatch.new(o)
    }
  }
}
```

The block is first matched, and then the result tested to determine the success of the predicate. Consequently, blocks used as predicates can themselves be partial functions, as in the example above.

4.3.10 Using arbitrary expressions to obtain patterns

There is a potential ambiguity in this pattern syntax. If a bare identifier such as d is used as a pattern, in a context where the identifier d is already bound, does it indicate a variable match (which always succeeds and binds d to the object being matched), or does it indicate that the object already bound to d should be used as a pattern? We avoid this ambiguity by requiring that the latter case be written with parentheses: (d).

Similarly, matching against the result of a method request with parameters would look similar to a destructuring match: both consist of a name and then some expressions in parentheses. The syntax treats this as a de-

structuring pattern. To obtain a match against the return value of a method, it may similarly be enclosed in parentheses:

```
case { t : (sum(values)) -> "The correct total, {t}." }
```

Where the result of an operator expression, such as `a + b`, is to be used as a pattern to match against, again we parenthesise it to avoid conflicts with the pattern syntax. For the same reason, matching against a type requires a `:` in the pattern, rather than simply the type name.

We chose this arrangement, essentially requiring the ordinary syntax to be offset in a pattern context rather than offsetting the pattern syntax at all times, because the overwhelmingly more common case will be to match against fixed patterns rather than the results of expressions. We were also able to preserve compatibility with the ordinary block parameter syntax and semantics, which already have their own context

4.4 Patterns as objects

As mentioned in the introduction to this chapter, our aim was to provide the — in most ways quite conventional — facilities described in the previous section by leveraging the existing features of the language, making only minimal extensions. In this section we describe the way that we reified our conceptual model as Grace objects in the implementation, and how we represented the different syntactic and conceptual elements in an object-oriented fashion.

4.4.1 Patterns as an object framework

Here we describe how we implemented the conceptual model as an object-oriented framework. Most Grace programmers will not need to know about this implementation, needing only to deal with the surface syntax, but an author who wants to create new kinds of pattern will be able to do so by understanding the implementation.

A pattern object is an object that has a match method: match takes as an argument the *target* of the match and returns an object of type MatchResult. MatchResult is implemented by two classes: SuccessfulMatch, which inherits from true, and FailedMatch, which inherits from false. Because MatchResult objects inherit from the Booleans, the match method of a pattern may be used as a condition:

```
if (pattern.match(obj)) then { ... }
```

MatchResult objects have two methods, in addition to those of Booleans. The result method returns the object that was matched. In a simple pattern the result of a SuccessfulMatch will be the original target of the match, but in a user-defined pattern it may be more specific. The bindings method returns a list of values that are bound to the variables of the pattern as an effect of a successful match; the bindings of a FailedMatch are always empty. These classes are instantiated with SuccessfulMatch.new(target, bindings) or FailedMatch.new(target), with the effect of associating the relevant parameter with the corresponding field. The matching protocol is shown in Figure 4.1.

Pattern objects are used to represent patterns at run-time. These objects are related by the inheritance hierarchy shown in Figure 4.2. Because patterns are objects they are first-class elements of the language; they can be named, passed around, copied, and composed, and the matching model functions on that basis.

4.4.2 Irrefutable patterns

An *irrefutable* pattern is one that always matches any object, that is, it cannot fail. Irrefutable patterns are supported in our model, in which the first step is to ask the pattern to match an object. In this respect we make a conscious departure from Newspeak's approach: because Newspeak does what in some respects is the "correct" object-oriented thing to do by first asking the object if it wishes to be matched, Newspeak supports only unmatchable objects (objects that cannot be matched by any pattern), and not irrefutable

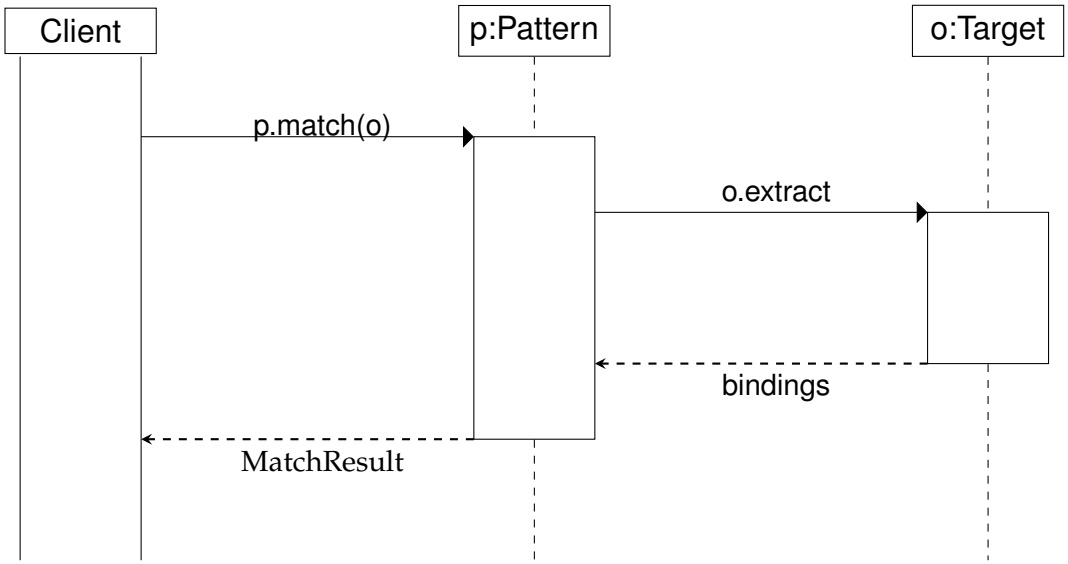


Figure 4.1: Matching sequence in Grace. Extraction is performed only for destructuring matches.

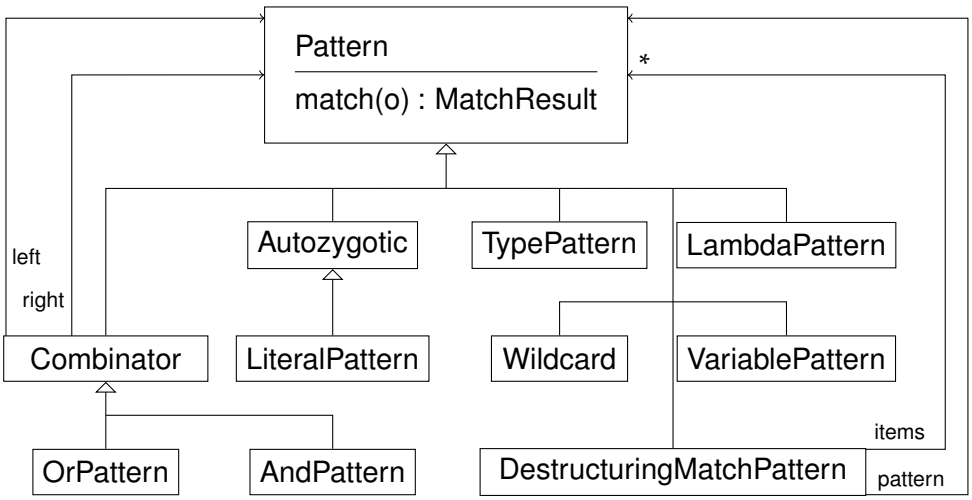


Figure 4.2: Hierarchy of built-in pattern objects.

patterns. By definition, it is only possible to permit one of these in the same system: either there is a pattern that matches any object, or an object that is never matched by any pattern, but never both.

We made a deliberate decision to support irrefutable patterns. We did not find a use case for unmatchable objects on the user level; rather, we found it more useful to permit explicitly matching and discarding, or matching and binding, any object given.

Wildcard pattern

The simplest pattern is the wildcard pattern, which corresponds to "_" in the pattern syntax. The wildcard pattern always matches, and does not bind anything: the matched value is simply discarded. The match method immediately returns a successful match.

```
def wildcardPattern = object {
    method match(o) {
        SuccessfulMatch.new(o, aTuple.new)
    }
}
```

The wildcard pattern creates no bindings and does not modify the target, and so returns a SuccessfulMatch with an empty tuple of bindings and with the target of the match passed on unchanged.

Variable pattern

The variable pattern also always matches, but includes a single binding: the given target of the match. It applies no tests and no transformations to the object, immediately returning a successful match with the correct binding.

```
class VariablePattern.new(name) {
  method match(target) {
    SuccessfulMatch.new(target, aTuple.new(target))
  }
}
```

The variable pattern corresponds to a variable name in the pattern syntax: { a −> a * 2 } constructs the pattern object VariablePattern.new("a"). The target of the match is passed on unchanged, but unlike in the wildcard pattern is also used as the single binding to create from this pattern.

4.4.3 Combinators

Pattern combinators are represented by pattern objects that hold the argument patterns, and use them somehow to establish their own match result.

& Combinator

The & combinator is represented by an AndPattern object. An AndPattern conjoins two patterns, ensuring that they both match.

AndPattern is an instance of the *Composite* structural design pattern [56, p.163], so it is itself a Pattern. An AndPattern contains other patterns as components and uses the components recursively for matching, without knowing anything about what they are:

```
class AndPattern.new(pattern1, pattern2) {
  method match(target) {
    def match1 = pattern1.match(target)
    if (!match1) then {
      return match1
    }
    def match2 = pattern2.match(target)
    if (!match2) then {
      return match2
    }
    def b = match1.bindings ++ match2.bindings
    SuccessfulMatch.new(target, b)
  }
}
```

Here, the component patterns are both applied to the target of the match: if either fails, the AndPattern immediately returns the failure. When both component patterns successfully match, the AndPattern returns a SuccessfulMatch whose bindings are the concatenation of the bindings of the component matches. The job of actually inspecting the target object or extracting bindings is left to the subsidiary patterns, and the AndPattern needs not even look at the object it has been given.

| combinator

The other obvious combinator on patterns is disjunction, represented in the syntax by | and as an object by the OrPattern, which combines two patterns, and succeeds if either one of them succeeds:

```
class OrPattern.new(pattern1, pattern2) {
 method match(o) {
  if (pattern1.match(o)) then {
   return SuccessfulMatch.new(o, aTuple.new)
  }
  if (pattern2.match(o)) then {
   return SuccessfulMatch.new(o, aTuple.new)
  }
  FailedMatch.new(o)
 }
}
```

The OrPattern has a dual structure to the AndPattern and is also a composite, leaving the task of inspecting the target object to the component patterns, this time short-circuiting to success when a pattern matches and failing otherwise.

Unlike the AndPattern, however, the OrPattern cannot return bindings. This is because the caller cannot know which of the component patterns succeeded, and hence does not know which variables will be bound. We considered returning the intersection of the bindings from the two components, but the transformation ceases to be fully syntax-directed at that point: knowledge of the entire scope of the pattern in use is required to know what to do with each binding. We also found relatively few use cases where this was helpful, and these cases can be covered by a custom pattern if required. Providing no bindings gives a simpler implementation and explanation, and is consistent with the rule that identifiers may not be repeated in parameter lists.

4.4.4 Types

We represent Grace types by objects with a method match(o) that returns a SuccessfulMatch if the argument o has a conforming type, and a FailedMatch

otherwise. In both cases, bindings is empty. Thus, types are also patterns. In the next example we use a type as a pattern to ensure that o has a value method, and then request it.

```
type Valuable = { value −> Number }
if (Valuable.match(o)) then {
    total := total + o.value
}
```

As a consequence, types can be used as patterns in match-case expressions. A pattern like z : Valuable combines a type-match with a variable pattern that binds a value. This is represented using the AndPattern, so a case of the form:

```
case { z : Valuable −> ... z.value ... }
```

results in the construction of the pattern

```
AndPattern.new(VariablePattern.new("z"), Valuable)
```

This pattern will succeed when the target of the match has the methods defined in the Valuable type, and will result in the variable z being bound to the target in the body of the block.

4.4.5 Autozygotic patterns

Numbers and Strings are patterns that match themselves, or more precisely match those objects to which they are equal: they are *autozygotic*, as defined in Section 4.3.3. This means that numeric and string literals within the pattern syntax have exactly their ordinary meaning: they refer to the corresponding Number and String objects. These objects have a match method of the form:

```
method match(o) {
    if (self == o) then {
        return SuccessfulMatch.new(o, aTuple.new)
    }
    return FailedMatch.new(o)
}
```

User-defined objects can have the same method to make themselves autozygotic. An autozygotic object can be useful for representing a signal or enumeration value, which can then be matched against directly.

4.4.6 Destructuring patterns

A destructuring pattern extracts some of the (conceptual) state of the object it matches, and attempts to match it against other patterns. The destructuring pattern written in the syntax as:

```
Point(x, 0)
```

is translated into the DestructuringMatchPattern object:

```
DestructuringMatchPattern.new(Point,
    aTuple.new(VariablePattern.new "x", 0))
```

This pattern combines another pattern (here, Point) which must match the object as a whole, and a tuple of other subpatterns which must match the destructured values in order for the whole pattern to match.

We will work through the implementation of this, the most complicated built-in pattern of the system, below, but the basic structure of the match is simple: first, attempt to match the target against the top-level pattern. Then obtain the destructured values from the target through its extract method, and attempt to match them with the subpatterns pairwise. If any match fails, the overall match fails; otherwise, the match succeeds with all the accumulated bindings.

How does this proceed? The match method of DestructuringMatchPattern is:

```
class DestructuringMatchPattern.new(pat, componentPatterns) {
  method match(o) {
    def m = pat.match(o)
    if (!m) then {
      return m
    }
    var bindings := aTuple.new
    for (componentPatterns) and (m.bindings) do { cPat, bindObj ->
      def inner = cPat.match(bindObj)
      if (!inner) then {
        return FailedMatch.new(o)
      }
      bindings := bindings ++ inner.bindings
    }
    SuccessfulMatch.new(o, bindings)
  }
}
```

We will go over this code in pieces.

```
class DestructuringMatchPattern.new(pat, componentPatterns) {
  method match(o) {
```

Our DestructuringMatchPattern constructor has two parameters: pat, the "top-level" pattern expected to match the entire object, and componentPatterns, a tuple of other patterns to match against components. Like all patterns, DestructuringMatchPattern has a match method with a single parameter.

```
    method match(o) {
      def m = pat.match(o)
      if (!m) then {
        return m
      }
```

First we attempt to match the target, o, using the pattern we were given. We save that result into m to be used later, and then test whether it is true (a SuccessfulMatch) or false (a FailedMatch). If it is not successful then the

destructuring pattern cannot succeed, so we immediately return the failure.

```
def m = pat.match(o)
...
var bindings := aTuple.new
for (componentPatterns) and (m.bindings) do { cPat, bindObj ->
    def inner = cPat.match(bindObj)
    if (!inner) then {
        return FailedMatch.new(o)
    }
```

Having successfully matched the top-level pattern we continue on to examine the sub-patterns. We declare a variable to hold a tuple of bindings to return, which is initially empty.

Next we want to look over two things at once: the list of sub-patterns we were given (componentPatterns) and the bindings returned from the top-level match (m.bindings). The top-level pattern determines what the extracted values from the object are, and returns them as its bindings. The pattern may examine the object directly, ask the object to provide its contents, or simply fabricate some values for a particular purpose.

For each component pattern and extracted value, we try to match the pattern (cPat) against the extracted value object (bindObj). If this match fails, the entire destructuring match also fails, so we short-circuit out and return a FailedMatch.

```
for (componentPatterns) and (m.bindings) do { cPat, bindObj ->
    def inner = cPat.match(bindObj)
    ...
    bindings := bindings ++ inner.bindings
}
SuccessfulMatch.new(o, bindings)
```

Having successfully matched the component pattern, we move on to accumulating any bindings from it. Bindings occur most commonly from variable patterns and other destructuring patterns; in our example, the

VariablePattern for x will create one binding. On the other hand, a type or autozygotic pattern creates no bindings — "0" will return success when matched against itself, but will not try to bind any variables. We concatenate all our accrued bindings together to be returned.

After examining all the component patterns, if we did not already return a failure then the match as a whole succeeds. We return a successful match on the given object, including the bindings we accumulated. There can be arbitrarily many bindings, depending on how many variables are used in the pattern and sub-patterns, or there can be none at all if particular components were matched directly.

Nested destructuring patterns

Destructuring patterns can be nested in the surface syntax, and exactly the same nesting manifests in the object hierarchy. Each layer of nesting constructs a new DestructuringMatchPattern, and is treated in the same way as other patterns. If a nested pattern binds many variables, these will be carried through to the result of the outermost destructuring match. The following pattern syntax:

```
Pair(Pair(x : Number, 1.0), p : Pair(y : String, z))
```

thus results in a SuccessfulMatch object having four bindings, one for each variable named, in the left-to-right order they appear in the syntax.

4.4.7 Destructuring types

To simplify a common case we provide an extension to type patterns allowing them to destructure almost automatically. A type containing an extract method returning a tuple implicitly supports destructuring matching. The run-time type pattern object will implement destructuring, with the object itself being asked to provide the destructured values. Because the object conforms to the type, the extract method is known statically to exist.

For example, given some Point objects, a programmer may want to match those with a subset of coordinates, or extract and bind the coordinates. The programmer could write a pattern themselves to do so, but such a pattern would be largely repeated boilerplate. Instead they can arrange their types and objects so that it happens for them. For example, given the type:

```
type Point = {
  x -> Number
  y -> Number
  extract -> Tuple<Number,Number>
}
```

and the class:

```
class aCartesianPoint.at(x' : Number, y' : Number) {
  def x = x'
  def y = y'
  method extract { aTuple.new(x, y) }
}
```

we can perform a destructuring match using the Point type pattern:

```
match (pt)
  case { p : Point(x, 0) -> "The point ({x}, 0)" }
```

to match all points on the x axis.

To support this behaviour the DestructuringMatchPattern adds an additional protocol step: in the case where the pattern itself does not provide bindings in its MatchResult, it will examine the object for an extract method and substitute the return value of this method for the bindings given by the pattern. This step is necessary because type patterns in general produce no bindings — p : Pair is also a valid pattern, and it should not try to bind the internal state of the Pair object in question.

4.4.8 Lambda patterns and match...case

All of the patterns expressible in the pattern syntax described in Section 4.3 are represented as objects. Programmers can also construct pattern objects directly, and mix them with the pattern objects generated from the pattern syntax.

Having shown how patterns are represented as objects, we can explain how match...case is implemented. The case parameters are LambdaPatterns, which combine a pattern, representing the domain of the function, with a "plain" block of executable code, representing the body. The match method of a LambdaPattern first attempts to match the pattern. Only if the match succeeds does it attempt to execute the block with the accrued bindings. The return value of the block is used as the result of the SuccessfulMatch.

```
class LambdaPattern.new(pattern, block) {
  method match(obj) {
    def result = pattern.match(obj)
    if (!result) then {
        return FailedMatch.new(obj)
    }
    def returnValue = block.applyWithArguments(result.bindings)
    SuccessfulMatch.new(returnValue, aTuple.new)
  }
}
```

A lambda pattern may also be used as a sub-pattern when side effects are desirable during matching, or when a simple way of computing the result of a match is required. As mentioned earlier, we extended Grace so that all single-parameter blocks were implicitly matching blocks; now we see what that means. A matching block is a special kind of Lambda pattern which implements both the interface of a block (with apply) and the match method shown here.

The match-case method tries to match each LambdaPattern in turn until one succeeds. This behavior is equivalent to that of the | combinator — try

to match each pattern in turn, returning the first success. The two-clause match()case()case method would look like this:

```
method match(val) case(b1) case(b2) {
  (b1 | b2 | { _ −> error "match−case was not exhaustive" })
    .match(val).result }
```

In practice, our implementation does not work this way for reasons of efficiency and simplicity, but the fact that it could demonstrates the basic compositionality of the Pattern design.

4.5 Types and patterns

How does pattern-matching mesh with Grace's optional, gradual type system? All of the patterns seen so far should be able to be statically typed. Most importantly, whenever a variable is bound in a match, either at the top level or via destructuring, we should be able to give it a static type. A condition required for the matching system to be integrated into the overall design of the Grace language was that the matching process should not need to descend into dynamically-typed code when the surrounding context was statically-typed; this restriction ruled out some approaches.

In this section we sketch how our approach permits types to be given to objects during matching; we will not attempt to prove correctness and we do not claim that the types assigned will be the most precise possible, simply that they are good enough to be useful, and that the type given to a variable matched by a type pattern is that type. Because these types are applied only within eventual variable bindings, it is only the ability to derive a valid static type which is important: having already matched at run-time, they are guaranteed to pass any dynamic check for the type they were given afterwards. Static type-checking in Grace is the responsibility of the dialect in use, and can be customised; we attempt only to show how sensible typings may be derived within the default structural system of the base language.

In particular, in this section we will give the types of pattern objects and the MatchResults returned from their match method. We will also show how combinators affect typing, the role of destructuring matches, and show where static typing and variant types can guarantee that matches are exhaustive.

The semantics and syntax of pattern matching are independent of static types, and we seek only to provide an overview for the curious reader. A reader who is not interested in how pattern objects may be typed can proceed to Section 4.6 on page 102.

4.5.1 Pattern and MatchResult

The Pattern and MatchResult types are generic, parameterised over the types of the result and the bindings:

```
type Pattern<R,T> = {
  match(o : Object) -> MatchResult<R,T>
}
type MatchResult<R,T> = {
  result -> R
  bindings -> T
}
```

In untyped code, these parameters are instantiated with type Unknown, but patterns may also declare types for themselves.

The simplest pattern that assigns a type is a type pattern itself. A variable associated with a type pattern has the corresponding type, so the variable n in the pattern n : Number is given type Number. The pattern object would instantiate the type parameter R to Number in this case.

In the case:

```
case { p : Pair -> "Pair ({p.left}, {p.right})" }
```

the new variable p has static type Pair, making the operations p.left and p.right statically type safe. The pattern object for Pair would have the type:

```
type PairPattern = {
  match(target : Object) ->
    MatchResult<Pair,Tuple<>>
}
```

The second type parameter to MatchResult is the empty tuple because this pattern binds no variables.

Similarly, user-defined patterns define their match method to return a result value of a particular type. This type must be correct, or the pattern code itself would have caused a type error. The result type R can then be given to the variable in question just as for a type pattern.

4.5.2 Destructuring

Destructuring matches may bind new variables, which are given types from the tuple type returned by the bindings method. These types in turn are derived from the types of the internal patterns written in the source, which are passed as generic type parameters to the constructor of DestructuringMatchPattern: in essence, the destructuring itself can be ignored as far as typing goes, and it behaves exactly like any other pattern match once transformed into objects. For example, given the case:

```
case { p : Pair(x : String, y : Number) -> ... }
```

the pattern generated is a variable pattern p and a DestructuringMatchPattern representing the Pair(...). The MatchResult returned from that pattern has type:

```
type Example = {
  result -> Pair<String, Number>
  bindings -> Tuple<String, Number>
}
def pat = ... // The DestructuringMatchPattern given above
def mr : Example = pat.match(p)
```

The result type of the destructuring match, and so the type given to p inside the body of the block above, is Pair<String, Number>. x and y will be a String and a Number respectively, as expected; the bindings method will return a tuple containing the values of these types to assign. The Example type is a subtype of MatchResult:

```
def mr' : MatchResult<Pair<String, Number>, Tuple<String, Number>> = mr
```

and the destructuring pattern itself has type:

```
def pat' : Pattern<Pair<String, Number>, Tuple<String, Number>>
```

At all times a valid type can be given to every value in use: the only place where "casting" occurs is inside the implementation of type patterns String, Number, Pair, which are built in and inherently check the types of their targets. In this case the pair's generic parameter types match those of the bindings tuple, but in general they can differ arbitrarily.

If a type is given to a variable inside a destructuring match, that type holds outside the destructuring pattern as well. The **only** point where a new type is assigned is where a pattern is given on the right-hand-side of a : character.

4.5.3 Combinators

When combinators are used, the types of variables become more complex. The & combinator gives the variable the types from both patterns. In the following example, o conforms to both X and Y, so both x and y methods may be requested:

```
type X = { x } // type with x method
type Y = { y } // type with y method
match (val)
  case { o : X & Y -> "Point ({o.x}, {o.y})" }
```

Grace's type system uses the notation X & Y for the type that conforms to both X and Y, so this is consistent with the base behaviour of a block, and

we can say that o has type X & Y. If X and Y are not types but instead general patterns, it is the intersection of their result types (R) that is important.

By contrast, the pattern X | Y matches when either X or Y does. In this case, only methods that are common to objects matching *either* X *or* Y may be requested in statically type-safe code. The type of such an object is the *untagged variant type*, also written X | Y. All objects that have type X and all objects that have type Y will also have the untagged variant type X | Y; these are exactly the objects that the pattern X | Y will match. Again, where X or Y is not a type itself, it is the result type R which is unioned.

4.5.4 Exhaustive matching

Untagged variant types also serve another role. A match-case expression can be statically determined to be exhaustive when the target of the match has a variant type, and all branches of the variant have associated cases. A warning can be given both for non-exhaustive matches, which may have unintended behavior, and for unreachable branches of the match:

```
var x : Number | String | Boolean := ...
match (x)
  case { n : Number −> ... }  // Doesn't execute anything
  case { b : Boolean −> ... } // if x is a String.
match (x)
  case { n : Number | Boolean −> ... }
  case { s : String −> ... }
  case { _ −> ... } // Unreachable!
```

Particularly in student code, it can be useful to report errors for missed or impossible cases. The ability to do so is a natural consequence of the structure of pattern objects, when used with variant types. An instructor who wants more stringent static checking could construct a dialect that provides it, using the dialect system described in Chapter 6.

4.6 Generalising patterns

The pattern-matching design described so far is largely discrete, fully implemented, and incorporated into the main Grace language. This section discusses a further generalisation of patterns to encompass all types in the language.

Because Grace is gradually-typed, types must be checked at runtime when dynamically- and statically-typed code mix. This mixing can only happen at method boundaries and when assigning to variables. These checks are equivalent under pattern-matching to requesting the `match` method on the type pattern, and in fact this is what we implemented in our compiler Minigrace (Chapter 8) to avoid redundant copies of the type-checking algorithm. We can extend this usage: by permitting any pattern to be used in type position, and enforcing at runtime that the pattern matches, an arbitrarily powerful user-defined type system can be enforced by the language.

For example, a square-root method might wish to ensure that its argument was always positive. We can define a pattern to represent this, as shown in Figure 4.3. The pattern has the same form as a pattern for use within the ordinary pattern-matching context.

More complicated features, such as full (dynamically-checked) dependent types, can be implemented in the same way using custom pattern objects (static checking can be implemented using our dialect system described in Chapter 6; these features complement one another). This `PositiveNumber` pattern is in fact built in through the unary prefix `>` operator on numbers, so the `sqrt` method could be defined as:

```
method sqrt(n : >0) { ... }
```

While this kind of test could be written in manually when required, abstracting the implementation away inside a pattern leads to a more declarative reading of the code, and avoids errors that may occur from repeating the tests. This approach is also in line with how such types are

```
def PositiveNumber = object {
  inherits basicPattern.methods
  method match(o) {
    if (!Number.match(o)) then {
      return FailedMatch.new(o)
    }
    if (o > 0) then {
      return SuccessfulMatch.new(o, aTuple.new)
    } else {
      return FailedMatch.new(o)
    }
  }
}
method sqrt(n : PositiveNumber) {
  n ^ 0.5
}

sqrt(−1) // Produces a runtime type error
```

Figure 4.3: Example of a pattern used as a (dependent) type.

generally written in languages which have them built in [122, 133, 147], while having arbitrary computational power.

This generalisation is under consideration to be used as the underlying semantics of the language, with the previously-explained gradual structural typing as the system presented to users and generalised types forbidden by the standard dialects. Minigrace supports these uses of patterns as an extension.

4.7 Discussion and comparison with related work

In this section we will compare our design to those used in other languages, discuss alternative approaches we considered, and further motivate the inclusion of pattern matching in Grace.

While object orientation and pattern matching are often contrasted, even

in purely object-oriented code there are times when a programmer needs to know the type of an object, such as when operating on elements of a heterogeneous collection. This is one reason why Java has an instanceof operator, and even Smalltalk programmers sometimes ask an object if it accepts a particular message. Unfortunately such a construct is awkward to use, often requiring type casts and redundant checks. Odersky argues that pattern-matching is simpler and clearer in many circumstances that would otherwise require the overhead of the Visitor pattern [46, 135]. He also argues that pattern matching is a natural way to handle different kinds of exceptions [136]. For these reasons, and also because instructors and students should be able to compare programs that achieve the same goals using pattern matching and polymorphic dispatch, we designed a pattern-matching framework for Grace.

In our conceptual model, a case statement is a series of partial functions, represented by a block, and a pattern is an object with a match method, which might use other patterns in a composite fashion. To arrive at this model of matching we needed to eliminate a great many conceivable models with superficial attraction, which turned out to be flawed after deeper consideration; we will discuss some of these rejected approaches in Section 4.7.7. Many approaches that can work in a purely dynamically-typed language do not have a viable static typing, for example, while some that work with pure static typing cannot work for dynamically-typed code, but any approach in gradually-typed Grace needed to support both. Other approaches lead to matching protocols that are difficult to follow or understand, such as those relying on nested blocks. We designed and even implemented some of these models before we were able to eliminate them. The final model, of reified partial functions, addresses the shortcomings of the alternatives while being readily embeddable into the language.

Grace's pattern matching was inspired by, and significantly based upon, the designs for pattern matching in Scala [46] and in Newspeak [58]. The "look" of our match-case construct is derived from Scala (an inherent system,

within the terminology we set out in Section 2.2), while the underlying framework draws from Newspeak (an exherent system), although there are significant departures from both in order to form a coherent matching system.

4.7.1 Scala

Scala includes a built-in pattern matching syntax with which an object may be matched against several patterns:[2]

```
x match {
  case 1 => "one"
  case "two" => 2
  case y: Int => "scala.Int"
}
```

x **match** is Scala's syntax for calling a method **match** on the object referred to by x. The block in braces defines multiple partial functions, one for each **case** keyword, and is a special case in the syntax (in fact, match is not a real method call at all).

In our design, however, each case is an independent block, both syntactically and semantically, and encodes a single partial function. match()...case is a real method request, and no special syntactic form is required. Many patterns, including all of those above, look identical in both Scala and Grace, although the semantics is drastically different; given this similarity we view our design as at least no worse than Scala from the perspective of an end user of matching. Scala also includes further special-case patterns, such as a :: as for list decomposition, which we do not define syntax for.

Patterns in Scala are mostly special cases with their own treatment: Scala's is an inherent matching system, meaning a pattern is not a first-class entity and has special semantics defined by the language. In contrast, our system defines all patterns as first-class objects, and matching occurs by

[2] An extended version of this example is presented in Appendix B.1.

requesting the match method on that object, all preexisting concepts within Grace. Our patterns are thus a smaller extension to the language, despite sharing a similar syntax, which we believe makes our design superior to Scala's from the perspective of the language designer and implementor.

In Scala, different types of pattern are statically transformed to different checks and there is no single consistent behaviour. Matching on numbers and strings is essentially syntactic sugar for the corresponding if-then-else conditionals, while other patterns have their own entirely different behaviour.

For user objects Scala's pattern matching is geared around "case classes": classes that export their constructor parameters, generally representing data types. Case classes are analogous to our type destructuring patterns, but rather than magically expose parameters for a special kind of class we included an explicit extract method that will be part of the matched object itself, so that the object chooses what it exposes, and may even change what it exposes over time, while a Scala case class is fixed at construction time.

Because the extract method is part of the type, all objects conforming to the type are known to contain it. As Grace is structurally typed, multiple implementations of the same type are possible, and they may choose to expose different values while still conforming to the same interface and matching the same type. In Scala, only instances of the same case class can be matched in this way. Constructor parameters are not automatically exposed in our design: the object may expose all of the parameters, some of them, or none, and they are not automatically made public fields as they are in Scala.

Scala also supports "extractor objects", which expose some values programmatically through an unapply method. Extractor objects correspond approximately to custom pattern objects in our design, and behave like a generalisation of case classes. The unapply method of the extractor object returns the list of bindings the extractor object should create, or a Boolean if there should be no bindings, wrapped in an Option monad. In either case,

the reference to the extractor object must be written with parentheses. In our design, a pattern is free to return bindings or not, and a Grace programmer can build explicit patterns with whatever combination of tests and bindings they wish. In defining a custom pattern our system has at least the power of Scala, and more in certain respects because all patterns are first-class; consequently, we view our design as superior from the perspective of a pattern author as well.

Scala has a typical inherent matching system, with some object-oriented extensions. The semantics of matching are inconsistent with the rest of the language, being a collection of special cases. Our design allows a reader to apply their knowledge about the rest of the language to the semantics of matching without surprises, as well as to define their own patterns and extensions that can be used exactly on a par with the built-in constructs. Because Scala's patterns are not first-class they must always be written directly into source code, within one of the delimited areas where the pattern language is accepted. In our system, arbitrarily complex patterns can be built up using combinators and pattern expressions over time: for example, a pattern could be constructed that matched any element of a given list simply with a loop and the | combinator:

```
def l : List<Number> = ...
...
var pat := l.first
for (l) do { element ->
  pat := pat | element
}
...
match (o)
  case { el : pat -> ... }
```

Such a construction is not possible in Scala, nor in any other inherent matching system, because in these systems patterns cannot be named and retained as they can be in our design. Despite this additional power, the

system we have designed allows common patterns to be expressed with the same syntactic directness as Scala.

4.7.2 Newspeak

Our notions of first-class pattern and pattern combinator were developed from Newspeak's design for pattern-matching [58], illustrated in Figure 4.4. Newspeak arguably does the "right" thing (or at least the pure object-oriented thing) in that the target of the match is always in control of the protocol: Newspeak's initial matching message "case:otherwise:" is sent to the target of the match, passing the pattern as an argument. This is the reverse of our design, in which the target is passed as argument to the match method requested on a pattern. In Newspeak, the default response to that initial message is to double-dispatch back, i.e., to send a "doesMatch:else:" message to the pattern asking that it match itself against the target. In theory, this gives the target complete control of the matching process. In practice, when we looked at the use-cases for matching, we noted this default method was rarely overridden: almost every object proceeded directly to the double dispatch.

This led us to a deliberate choice where we differ from Newspeak. While Newspeak permits objects to determine whether they can be matched or not, this power comes at the expense of the ability to define patterns that always match (what we called *irrefutable* patterns earlier (Section 4.4.2)). Any pattern-matching system can support at most one of irrefutable patterns and unmatchable objects: a system with both would run into difficulty when the two meet. We found irrefutable patterns to be significantly more useful. In particular, our wildcard pattern (written _) is not possible in Newspeak.

As well as wishing to support irrefutable patterns, we found Newspeak's matching protocol overly complicated. Comparing the Newspeak protocol in Figure 4.4 and our protocol in Figure 4.5 we see significantly

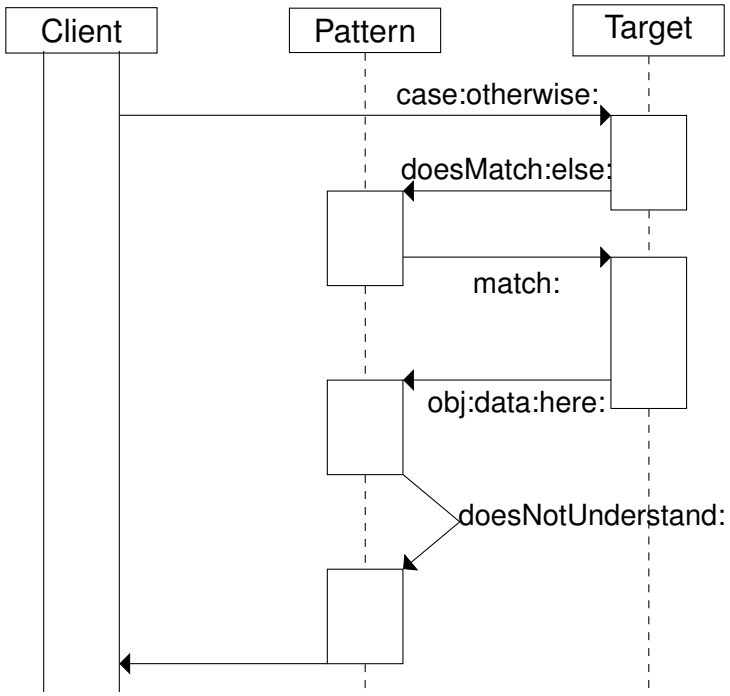


Figure 4.4: Matching sequence in Newspeak

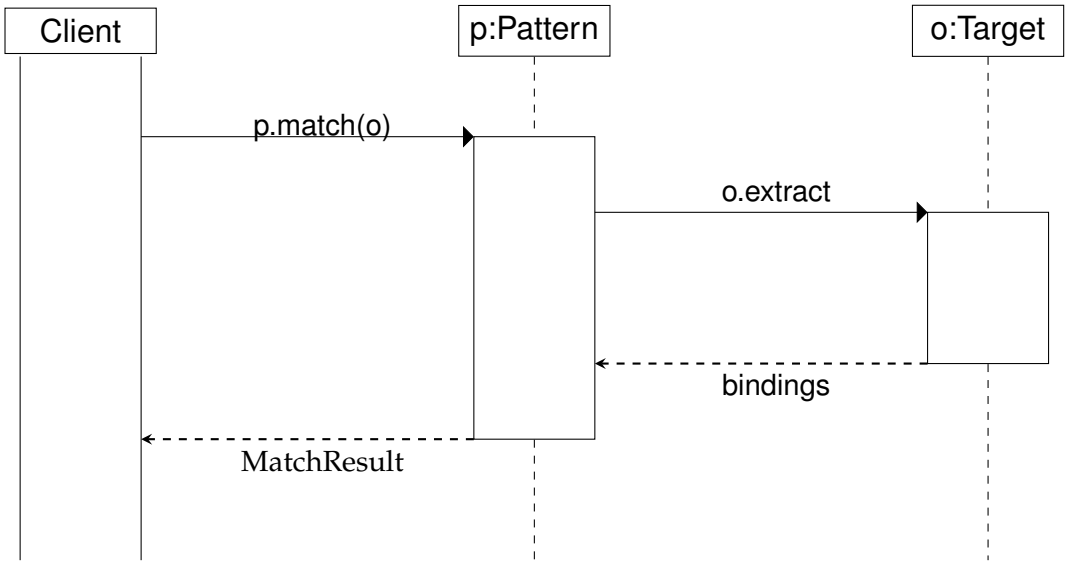


Figure 4.5: Matching sequence in our design.

fewer method calls in our design, including no double-dispatch bouncing between objects, without losing any more expressive power than the irrefutable-unmatchable tradeoff just mentioned. We view this simplicity as an advantage for anyone attempting to understand or debug the matching system.

The centre of Newspeak's protocol is the doesMatch:else: message: this supports composite patterns and pattern combinators similarly to our match method. As its name implies, doesMatch:else: takes a block that is evaluated when a match fails. In contrast, our match returns either a SuccessfulMatch or a FailedMatch object; this is similar to the way in which Scala uses an Option monad. Because our MatchResult objects inherit from Booleans, the match method can even be used directly in the existing if-then-else construct of the language:

```
if (pattern.match(obj)) then {
  ...
} else {
  ...
}
```

In this way control flow is always clear and consistent, even in the presence of matching.

At the bottom of the Newspeak pattern-matching protocol, abstract type tests and destructuring are supported by a second double-dispatched message. To support matching, essentially all Newspeak objects must respond to "match: pat" messages by sending a message characterising the object and its state back to the "pat" parameter: a point might send the two-parameter message "x:y:", passing its coordinates as the arguments to the message, whereas a string may simply send itself as the sole argument to a one-parameter message "string:". Patterns then implement the message that will be sent by objects that they match — other messages raise a "does not understand" exception, which is interpreted as a failed match. This design is similar to Blume *et al.*'s proposal for matching based on first class

cases, rather than first class *patterns* [13].

Compared with Newspeak, all levels of our matching are carried out via the same match method, which is uniformly implemented by all kinds of patterns, from self-matching literals and reified types to pattern combinators and blocks representing whole cases. Both our design and Newspeak's need syntactic extensions to support patterns and variable binding: our design also needs the primitive reified types (the "type pattern" objects) to be supplied by the runtime system, rather than using doesNotUnderstand:.[3] In spite of this disadvantage, we consider our pattern matching protocol, shown earlier in Figure 4.5 more straightforward than Newspeak's protocol.

One of our goals was to allow users and instructors to define their own patterns with relative ease. We believe our system does so to a much greater extent than Newspeak: the author of an ordinary object to be matched need not interact with the matching system at all, while the author of a pattern need implement only one method which returns a value after linear flow of control. The pattern author does not need to understand double dispatch, nor implement multiple methods, nor handle method-not-found errors. We view our system as an improvement over Newspeak's for pattern authors.

Newspeak patterns are not fully integrated into the surface syntax in the way ours are. By combining a suitable syntax with a model of reified partial functions, connected by a purely syntax-directed transformation, we constructed a novel system with both visual and conceptual elegance. Through supporting this "inherent matching"-style syntax our design eases the burden on an end user of matching compared to Newspeak, despite the underlying system having all the power of Newspeak's exherent design.

For the reasons given above, we claim that our system improves on Newspeak's syntax and Scala's semantics, and so successfully bridges the inherent and exherent matching divide, while including new features and

[3]Type patterns could alternatively have been implemented via reflection, but this was both unsupported at the time and an unhelpful abdication of responsibility — delegating to reflection is simply pretending to avoid a language extension.

(a)

```
(match e_1
  [(and (? number?) x) e_2]
  [_ e_3])
; Becomes
(let ([tmp e_1]
      [f (λ () e_3])
  (if (number? tmp)
      (let ([x tmp]) e_2)
      (f))))
```

(b)

```
match (e_1)
  case { x : Number −> e_2 }
  case { _ −> e_3 }
```

Figure 4.6: (a) A simple Racket pattern-match invocation and its macro expansion [181]. e_N are arbitrary Racket expressions. (b) The equivalent Grace match for semantic comparison.

consideration of the target audience of Grace.

4.7.3 Racket

The Racket Scheme dialect also includes an extensible pattern-matching facility [181]. Uniquely amongst all the designs presented here, Racket's powerful macros enable the language to be extended without any changes to its core implementation. While our design requires semantic support for partial functions and reified types, and syntactic support for destructuring, the remainder of our design avoids reflection, unlike Racket, and uses straightforward, object-oriented design techniques.

Figure 4.6 shows a simple use of Racket's **match**, and what the macro expands to for that use; the pattern match is exactly equivalent to some temporary bindings and an if statement, and produces these as the code that will be executed. For more complex matching the analysis and generated code is more complicated, but the pattern match is always translated to equivalent imperative code. The "?" form allows using an arbitrary boolean function as a pattern predicate, similar to (and inspiring) our predicate

patterns that lift blocks to patterns.

Racket's approach is fully within the Lisp tradition, and may well be the ideal design within that tradition, while Grace is a pure object-oriented language with different needs. We view the semantics of our macroless design as simpler to understand. Our pattern syntax fits into Grace at least as well as Racket's matching fits into its syntax. Overall, Racket and Grace have similar goals but occupy different language niches, and their matching systems reflect that difference; we believe our system is most suitable to the constraints of the Grace language.

4.7.4 Gradual and optional typing

As well as being in an object-oriented language, our design differs from most pattern-matching designs in that Grace is gradually and optionally typed, while most languages with pattern matching are strongly statically typed, typically via some mix of inference and explicit declarations. (The outliers here we have already discussed: Racket, which is also optionally typed, and Newspeak, which is purely dynamic). These languages' pattern-matching facilities are generally tied tightly into their type systems. In fact, this is as true for Racket as it is for Scala: objects are matched and destructured based on their defining class. This is true even in OCaml, which also has a structurally-typed object system, but which supports pattern matching only on algebraic data types, not on objects [102]. In contrast, our design matches according to the semantics of Grace types, and hence only on the publicly visible interface of an object — its "duck type" if you will — which is completely decoupled from that object's implementation. This behaviour is necessary for matching behaviour to be semantically consistent in a structurally-typed language; as our goal was to have matching fit into the underlying semantics of Grace, we needed to design in this behaviour, which we achieved through the reified (structural) type pattern objects we described.

4.7.5 Matching as monads

In essence, our design uses an object-oriented implementation of the exception monad: a pattern may either match, and return a value, or fail and prevent subsequent computation from being attempted within that branch. In either case a MatchResult object encapsulates the result. Both success and failure can be detected, and a result extracted, or ignored — a failure is "soft", but in itself prevents subsequent execution within the monad unless the failure is caught and recovered from. The | combinator, for example, continues calculation after a match failure, while the & combinator returns the error. Patterns are composable and can be sequenced. Custom patterns can have arbitrary behaviour, providing the "programmable semicolon" effect of a monad. To this extent, we have defined an imperative, object-oriented syntactic form and semantics for a particular monad.

4.7.6 Future work

We have considered a number of further extensions to our pattern-matching design. One relatively straightforward extension is to support matching against regular expressions. The simplest implementation would make use of an external regular expression library such as Perl-compatible regular expressions (PCRE) [65]. In the absence of regular expression literals, we can define an operator to convert a String into a RegExp object. To fit into the pattern-matching framework, all the RegExp object need do is to support the protocol described by the Pattern type.

Another possible extension is to allow patterns to be used anywhere there is a type annotation, which we discussed in Section 4.6.

A more ambitious extension is to incorporate matching and destructuring of sequences, as in combinator parsers. From one perspective, parsers, especially combinator parsers [100], are rather similar to matching: parsers either complete successfully and return a representation of the parsed input, or they fail. In other words, parsers are partial functions. The key difference

between patterns and parsers is that while patterns match against whole objects, parsers typically parse an *input sequence*, and a successful parse may consume some, all, or none of the remaining input. To support parsing, we extend MatchResult to maintain the representation of the unparsed input: the sequence parser combinator ∼ starts the right-hand parser when the left-hand parser finishes. What is interesting about this embedding is that alternation and parallel parser combinators correspond exactly to the "and" and "or" pattern combinators. The resulting language is similar in many ways to OMeta [192] — an object-oriented language for parsing — because the parsers are integrated into the matching facility, rather than simply being a stand-alone library. Lua's text pattern-matching library is built on Parsing Expression Grammars in a similar style, without any syntactic support, but Lua matches only against text, not arbitrary objects [81].

4.7.7 Alternative approaches

We considered a number of alternative approaches before arriving at the design presented here. All of them had some conceptual flaw that made them unsuitable for Grace or in general, but some took a substantial amount of consideration before that became apparent, and a few even made it as far as preliminary implementation. We present some of these designs in brief here, and explain why they were not chosen.

Nested function unrolling

This approach reached almost total implementation in Minigrace, more than any other alternative we considered. It used essentially the same syntax as we have described, but relied on nested blocks performing different stages of the match.

The fundamental construct was a method on patterns:

```
matchObject(target)matchesBinding(successBlock)else(failureBlock)
```

If the pattern matched the target object, it would apply the successBlock, passing in any bindings that should be accrued from the match. If the pattern did not match, it would apply the failureBlock. In either case the method would return the value returned by the block it applied. By passing blocks like this the approach has commonality with Newspeak. In Newspeak, however, failure is indicated by an unsuccessful message send (i.e., a method lookup error), which is interpreted by the matching infrastructure; we did not like this repurposing nor the fact that it was badly-typed, and so introduced a failure block instead.

A top-level matching block would be translated into an ordinary block accepting a single parameter, whose body constructed all the nested blocks required for the match (including one for the actual body of the matching block) and began the process. Each layer of block would handle one step of the overall match; a single simple pattern would generally unroll to three or four layers of block.

Let us consider a very simple case. Given a match-case statement like this:

```
match(x)
  case { 1 | 2 -> print "Small number" }
```

the unrolling transformation would create nested blocks like these:

```
{ target ->
  1.matchObject(target)
    matchesBinding { print "Small number" }
    else {
      2.matchObject(target)
        matchesBinding { print "Small number" }
        else { FailedMatch.raise }
    }
}
```

This pattern binds no variables; if it did, each of them would be given as a parameter of the block given as matchesBinding. The else block in the

first layer contains the right-hand side of the alternation, as it is only used when the left-hand side did not match, and the else block inside that raises an error indicating that the overall match fell off the end.

Note that the body of the initial matching block is repeated in both branches; in this particular case the block could have been pre-allocated, as it does not depend on any state, but in general the body will rely on variables bound during the match, and so must be lexically inside the scope of those variables. With a more complicated pattern like

```
{ a : Pair(x : Number | String, y : 1 | 2) -> print "{x},{y}" }
```

the unrolling must repeat not only the body, but the entire match process after Number | — the tests for 1 | 2 and the bindings of a and y — on both sides in order to have the correct variables bound at the end. Notwithstanding the repeated code, this transformation is simple to implement in the compiler, which need only alias or copy the AST nodes.

We implemented this design and it substantively worked. There were two significant issues that ultimately led us to move on to the design presented in this chapter. The first was simply one of comprehensibility: the nesting quickly became intractably deep, and the repetition of branches made the expansion very hard to follow. The other was one of typing: giving correct static types to all of the intermediate variables involved in the expansion, and ensuring that they were assigned values of the correct type, was a challenge.

The combinatorial explosion of blocks was not a problem for the compiler, although it was inefficient, but it was a significant burden to anybody trying to understand the matching process. We aimed, as a minor goal, to make the system simple for the language designer and implementer, but would be prepared to trade that off for better experience for end users and pattern authors, so this flaw was not in itself fatal. End users would see no effect from this approach, so they were not an obstacle; pattern authors, however, would also need some understanding of the unrolling, both to aid in debugging their pattern and in some cases to implement it correctly.

We felt that the unrolling was potentially too complex for that purpose. Blocks would be nested very deeply, and in the case of combinators would even be repeated in different branches of the match. A relatively simple destructuring match with two arguments, one of which was another destructuring match, required over 60 blocks in total. An optimising compiler could remove many of the nested blocks to reduce the inefficiency, but the semantic model was less simple in execution than it had seemed in the abstract.

Typing was more of a concern. While Grace is gradually-typed, and dynamically-typed code is both possible and acceptable, we did not want our expansion to discard type information when it was present. At some points it became necessary to drop into dynamically-typed code to allow the system to proceed: for example, during destructuring we would need to bind (and even use) temporary variables before knowing their types and construct blocks with parameters that might not be satisfied; as a consequence, these internal variables needed to be given the dynamic type `Unknown`. We considered discarding type information in this fashion undesirable, and descending into dynamically-typed code to perform matching within otherwise statically-typed code was a bar to integrating matching into the core Grace language.

A further issue was simply that the act of implementing the method `matchObject()matchesBinding()else` was a chore: an author of a custom pattern needed to understand how the expansion worked, which was complex, and was relied upon to implement the actual behaviour of the matching correctly by applying one or other block and returning the correct result. While in itself that was not enough to render the system unworkable, it was a factor in combination with the other issues. We wished to minimise the burden of understanding on pattern authors as far as possible, and requiring that they implement the correct execution of blocks, rather than simply linear control flow, was an obstacle to that goal, as well as potentially leading to mistakes that they would have difficulty tracking.

One advantage the unrolling approach has over our eventual design is that variable bindings simply fall out correctly: our design does not permit binding variables on either side of an Or combinator, but unrolling automatically permits the variables bound in every branch of the pattern to be used in the body, and would raise an error if any variable not universally defined were used, entirely by the ordinary semantics of the language (admittedly, this error would be within the generated code, and some care would be required to ensure a useful error message was produced). We considered this as a point in favour of unrolling, but eventually determined that it was an uncommon use case that could be manually compensated for where required.

This design is plausible and functions correctly, with a transformation directly into ordinary code, but it was not suitable for the overall environment of Grace. Nonetheless, it may be appropriate in another language with different constraints.

Dedicated pattern language

Early on, we considered an approach where pattern-matching was an innate concept of the language, and a dedicated sublanguage with its own semantics implemented matching. `match()...case` was a keyword construct in this scenario, and the case blocks were not ordinary blocks in any way. In this the approach is similar to Scala's.

Within the pattern language certain concepts had a known meaning, such as numeric and string literals and type names. The compiler would simply know what it meant to match against a number and perform that directly. From the user's perspective, matching would always "do the right thing" as far as their intentions went: all branches would be checked and the correct variables bound, particular syntax could support special cases like tuples or list construction, and all the common cases would be covered.

As well as Scala this approach is taken by many, if not most, languages integrating some sort of pattern matching, so it was certainly worth con-

sidering. The fundamental objection we have to it is precisely that it is a special case of the language, and no part of it is first class in any sense. User-defined patterns, if they exist at all, are definitively "less good" than their built-in counterparts, and the patterns themselves are not first-class in an object-oriented sense: we cannot name them, pass them around, or copy them, at least not without further special casing in the system. Polymorphic applications of patterns are not possible or are unnaturally difficult. In addition, another keyword construct must be added to the language, which the user cannot mimic and which is another piece of semantic and syntactic baggage to learn.

Piecewise functions everywhere

In many functional languages pattern-matching is an integrated part of the function dispatch system, behaving analogously to the multimethods of some object-oriented languages and systems [1, 14, 25]. In Haskell, for example, one can define a function in pieces, with different bodies for different arguments, or with destructuring, which will be pattern-matched when the function is called:

```
fib 0 = 1
fib 1 = 1
fib n = (fib (n − 1)) + (fib (n − 2))
```

In Haskell this is syntactic sugar for a single definition using the **case**... **of** construct, but to the user it appears as a piecewise function definition, as might be used to define a Fibonacci sequence function in mathematics. Other definitions allow defining cases for each member of an abstract data type (approximately, method overloading on parameter types). We considered allowing this either for all methods or only for blocks, with a special syntax. In this way pattern matching would devolve entirely onto method application.

While piecewise function definition can be useful, it would have been a substantial addition to the semantics of Grace, and likely to be difficult

to explain to novice programmers. A programmer can obtain this effect in our approach by writing a pattern match immediately inside the body.

Performance

Our pattern-matching system involves the construction of many objects, particularly matching blocks and match results. Where matching occurs inside a tight loop and there are very many branches to the match, the construction of these objects and the resulting garbage collection is likely to have a performance impact not found in other designs. Efficiency is explicitly not a goal of Grace, so we do not consider this impact to be significant in general. The allocation of blocks is an equal problem for `if()then()else` statements, and considered a reasonable price for the flexibility the overall approach to control structures brings.

In specific cases where matching is to be used in this way the program can be structured carefully to avoid excess allocation, such as by pre-generating matching blocks outside the loop. An optimising compiler could detect uses of matching and optimise the code without affecting the semantics, by performing that restructuring automatically or transforming patterns with well-known semantics (such as strings, numbers, and combinators) into other constructs. Our compiler does not attempt this optimisation, but we have experimented with various optimisations with promising results. The introduction of matching has no performance impact on code that does not use matching.

4.7.8 Application

Pattern matching is used heavily in the dialect for writing dialects described in Section 6.2.3 and the dialects using that dialect. Patterns allow identifying the nodes of large trees that interest the programmer concisely and clearly, which would otherwise require more complex manual inspection, and most importantly allow exposing a simple pattern-based interface to the dialect's end users.

4.8 Conclusion

Pattern matching is no longer the preserve of advanced functional languages and regular-expression-based scripting languages. Mainstream programming languages incorporate pattern-matching functionality, and the ACM Computer Science curriculum requires students to understand pattern matching as a mechanism for program organisation. Consequently, Grace should incorporate a pattern-matching system.

This chapter describes how we incorporated pattern matching into Grace, without disrupting (and even while leveraging) the language's dynamic object-oriented nature. We applied well-known principles — reified partial functions, patterns modelled as first-class objects, and complex structures constructed through pattern combinators and the Composite pattern — to add comprehensive pattern-matching functionality with minimal extension to the language. While our design is for Grace, the approach we took relies only on common features that are also applicable in other object-oriented languages that want to incorporate pattern matching.

Chapter 5

Modules as Gradually-Typed Objects[1]

5.1 Introduction

In object-oriented languages, objects and the classes that generate them are the primary unit of reuse — but objects and classes are typically too small a unit for software maintenance and distribution. Many languages therefore include some kind of package or module construct, which provides a namespace for the components that it contains, and a unit from which independently-written software components can obtain the components they wish to use.

Grace needs a module system, but the language should not be complicated any further than necessary in introducing that system. In this chapter we present a module system built entirely from preexisting concepts and constructs of the language. We also show how our system satisfies many desirable attributes of a module system in general.

[1]This chapter expands upon a paper presented at DYLA 2013 [70].

5.1.1 What is a module?

As an educational language, Grace does not need as elaborate a module system as might be required in an industrial-strength language, but it does need a module system of some kind. The module system must support the different applications of modules that students may need to learn, and permit programs to be organised in a useful way for students, instructors, and other programmers. We were influenced by the criteria set out by Szyperski [177] on the role of modules and classes. As well, we noted the work of Littman *et al.* [109] showing that novice users find debugging multiple-module programs difficult when they do not understand the dependencies of the modules. We also wanted to use the module system, or a variant of it, to implement the dialects described in Chapter 6.

From these general ideas we set out these specific requirements for a module system for Grace:

R1. Separate compilation: each module can be compiled separately.

R2. Foreign implementation: it should be possible to view packages implemented in other languages through the façade of a Grace module; the client code should not need to know that the implementation is foreign.

R3. Namespaces: each module should create its own namespace, so maintainers of a module need not be concerned with name clashes.

R4. Sharing: clients should be able to share the objects provided by a module.

R5. Type-independent: because Grace is gradually typed, the module system cannot depend on the type system, but the module system should support programmers who wish to use types.

R6. Controlled export: some mechanism should be available to hide the internal details of a module's implementation.

R7. Multiple implementations: it should be possible to replace one module by another that provides a similar interface, while making minimal changes to the client.

R8. Explicit dependencies: code that uses a module *depends* on that module: these dependencies should be explicit so that a reader may follow the dependency chain and flow of execution.

Many of these requirements will hold for other languages as well; only R5 is especially particular to Grace, although industrial languages may well have additional requirements.

Grace meets these requirements by representing modules as objects. Combining these module objects with Grace's gradual structural typing provides a wide range of functionality. This approach to the design of module systems has been influenced by Python and Newspeak. The next section describes the design of our system, and Section 5.3 shows how the design meets the requirements above. Section 5.4 describes how our system extends to support package management functionality appropriate to the needs of Grace. Section 5.5 outlines extensions to the system and potential future work, and Section 5.7 concludes.

5.2 Modules as objects

A Grace module is a piece of code that constructs an object. This *module object* behaves like any other object; in particular, it may have types and methods as attributes, and can have state. A module corresponds to a source file: the module object is created as if the entire file were inside an **object** {...} constructor. Here is a complete, if simple, module:

```
def person = "reader"
method greet(name) {
    print "Hello, {name}!"
}
greet(person)
```

Executing this module will print "Hello, reader!" and construct a module object with the greet method in it.

That module object has ordinary Grace object semantics, meaning that a module can also bundle classes or object constructors, as well as methods:

```
class Greeter.new(greeting) {
    method greet(name) {
        print "{greeting}, {name}!"
    }
}
def standardGreeter = object {
    method greet(name) {
        print "Hello, {name}!"
    }
}
```

A module can contain many methods, objects, classes, fields, and types, or none.

The effect of this is similar to modules in Python [159] (discussed in Section 2.3.1), but uses existing language features instead of adding a new concept that must be explained and has its own idiosyncratic behaviours. We will contrast our approach with Python's explicitly in Section 5.6.1.

Making modules objects, rather than introducing a new feature, interacts constructively with other aspects of the language, such as gradual typing.

5.2.1 Importing modules

To access another module, the programmer uses an import statement:

```
import "examples/greeter" as doorman
```

The string that follows the **import** keyword must be a literal; it identifies the module to be imported. From the perspective of the language this string is opaque; our current implementation treats it as a relative file path. The identifier following **as** is a local name that is bound to the module object

created by executing that file. If we assume that `"examples/greeter"` refers to the first simple module shown above, then a name doorman is introduced in the local scope, bound to an object with a greet method.

In a given program, a module is executed only once. Every import of the same path within a program will access the *same* module object. A module can maintain state, and that same state will be used by every client of the module.

To take advantage of static type checking, a variant of the import statement allows the programmer to specify the type that the imported module should meet:

```
import "examples/greeter" as doorman : BasicGreeter
```

As types in Grace specify an interface, not an implementation, this asserts that the imported module object must satisfy the interface defined by the BasicGreeter type; if it does not, an error occurs. This error occurs either at compile time or at bind time, depending on the implementation, but always before any of the code of the importing module runs. The type could be imported from another module, or be defined by the client:

```
type BasicGreeter = {
    greet(n : String) -> Done
}

import "examples/greeter" as doorman : BasicGreeter
```

An alternative implementation of the same type may be chosen by changing the import path string. When no type is specified in the import statement, the type of the module is inferred from its implementation.

5.2.2 Gradual typing of modules

Through careful use of modules, imports, and type specifications, a system can be configured in a range of different ways covering many of the common purposes of modules.

Inferring module types

We can import a module with its original types, whatever those may be, using:

```
import "x" as x
```

Parameters and return values with the Unknown type in `"x"` will also be Unknown in the importing module, while uses of statically-typed parameters and return values will be statically checked in the importing module.

Discarding module types

We can ignore any type information specified in the module `"x"` using:

```
import "x" as x : Unknown
```

In this case, providing an argument to a method of x that does not match the declared type of a parameter, or requesting a method that is not defined, will not be reported as a static error, and will not prevent the importing code from being compiled and run. Grace's gradual typing means that such errors will be caught dynamically, if and when the offending code is executed.

Specification conformance

We can define the interface that we want a module to support separately, in another module, which allows for multiple implementations of the same interface. We can also check that a module provides the features that we expect:

```
import "xSpec" as xSpec
import "xImpl" as x : xSpec.T
```

Here, `"xSpec"` is a module defining the type T, like a Modula-2 definition module [196]. The type of the implementation module `"xImpl"` is required to conform to the type xSpec.T. A different implementation of xSpec.T can be selected by changing just the second import statement.

Asserting local conformance

A module that is intended to satisfy a type defined elsewhere can assert its own compliance with that type. Suppose a module intends to satisfy the type T defined in `"xSpec"`. It can enforce that type conformance itself:

```
import "xSpec" as spec
assertType<spec.T>(self)
```

Using the library method assertType<T>(o) causes the compiler to statically check that o has type T, so this code asserts that the current module object has the type T imported from spec.

assertType itself has a simple definition:

```
method assertType<T>(_ : T) {}
```

By declaring a type parameter T, and taking a parameter of that type, assertType forces the typechecker to establish the conformance of the given object to the given type. **self** refers to the module object itself at the top level of a module, so the programmer can cause as many checks as they like to occur against different types they wish to implement.

Type ascription

A module can perform "type ascription" when importing other code:

```
type ExpectedType = { ... }
import "somemodule" as x : ExpectedType
```

Here the module imports `"somemodule"`, but is explicit about the type it assumes that module will satisfy — which will often be a supertype of the type of the object actually supplied by the outside module. Future changes to the module `"x"` that invalidate that assumption will yield an error at the import site; this puts the "blame" in the right place.

Type ascription also imposes the constraint x : ExpectedType on code that uses x in the importing module. If any code in the importing module tries to use x in a way that conflicts with that constraint, it will receive a static error. This is true even if the implementation module `"x"` uses

dynamic typing, and some or all parameters and return types are Unknown; a programmer may develop a module gradually starting from dynamic code and either moving to static or not, while using a consistent target interface in client code.

5.2.3 Recursive module imports

Module imports cannot be recursive: A module A cannot import a module that directly or transitively imports A.

This restriction is a deliberate choice. For Grace's target audience — novice programmers — we believe that cyclic or recursive imports are most likely to indicate a program structuring problem.

This choice also means that we can fully create and initialise all imported modules before the importing module. In this way we avoid problems caused by partially-initialised modules, which novices are likely to have difficulty understanding. By the time a piece of code has access to a reference to a module, that module has executed in full, any side effects have occurred, and all its fields are set.

While cyclic *imports* are prohibited, modules may recursively *use* one another, just like any other pair of objects. This mutual usage can be accomplished by the programmer explicitly providing one of the modules with a reference to the other as an ordinary method parameter. The programmer can ensure that initialisation is completed appropriately, by whatever means are appropriate to their program (such as setting a field).

Our module design keeps its thumb firmly on the scales to push users towards the "right" choices to make, given the nature of the language, but takes care not to exclude users from taking a different path if they wish. In particular, because an imported module is just an object, any other object can be substituted and given the same local name without changing any other part of the importing module.

5.3 Design rationale

In Section 5.1.1, we laid out the requirements for modules in Grace. Grace objects already satisfy many of these requirements. Top-level objects cannot capture variables or outside state, so they can be compiled separately (requirement R1); they create a namespace accessible through "dot" notation (R3); the same object may be referred to by many other objects (R4); they are gradually typed (R5); and they provide controlled export to clients (R6). Through careful design of the import mechanism and our existing gradual structural typing we were able to achieve the ability to use foreign implementations (R2), and to substitute one implementation for another (R7), while the **import** construct itself ensures that dependencies are explicit (R8). Using objects as modules avoids introducing another concept into the language, supporting Grace's design principle of building a small language.

We considered an alternative design in which modules were classes, as they are in Newspeak [15]. Using classes would offer some advantages: modules would be instantiable and could be parameterised over objects and types. In the style of Newspeak, we could have omitted the **import** construct in favour of providing the module with an explicit `platform` parameter containing its dependencies. We eventually rejected this approach because we wished to make dependencies as explicit as possible in the code using them, because we wanted to avoid formulaic incantations, and because Grace is primarily based on objects (with classes derived from objects), rather than on classes (instantiating objects), as is Newspeak.

We also considered a variant of the current design in which a file containing a *single* top-level declaration of an object, type, or class was a module, while a file with *multiple* declarations was an object with the attributes thus declared. We rejected this option because of the extra complexity, the lack of uniformity, and the need for special-case behaviour, all of which would need to be explained to students.

5.4 Package management

Many modern languages include a package manager and repository built into their distribution; examples include Perl's CPAN [179], RubyGems [169], and LuaRocks [128]. A user can install new libraries for the language using these tools. "Package manager" is the standard term for such tools, but the term "package" is unconnected to the concept of packages in languages like Java or of the sort we discussed in Section 2.3.2: instead, a *package* is a discrete library that can be installed. Within this section we will use the term exclusively with this meaning. We can leverage our module system, and in particular the flexible import paths, to construct a suitable package management system.

Grace has a unique set of constraints for such a system. Most end-users of the language will be students, who have little need for complex packaging, but the most important audience is course instructors. Instructors frequently have their own libraries they wish students to use, and in Grace may well have customised dialects as well. Instructors are busy and may not have the time or inclination to follow instructions for packaging up their library code.

It is important that both installing libraries is easy (for students), and that making a library accessible through the system is easy (for instructors), and so the barrier to entry in both directions must be very low. A particular instructor's library may only be used for a single course, or even a part of a single course, at a single institution, and may be of no use to other users.

5.4.1 Identifying packages

Most existing systems define their own global namespace of package names, which library authors register on a first-in-first-served basis. This approach makes for short and memorable names to begin with, but raises some issues in the long term. Most "good" names will be taken early on, forcing later authors into more and more creative names. Abandoned libraries continue

to "squat" on their name forever; while these libraries may be determined and deleted somehow, doing so leads to a repetition of the first problem while also creating confusion in which package is actually referred to by the recycled name. In addition, given the user base of the language, many low-quality packages may have their names registered by students who then move on from the language.

A suitable package management system for Grace, then, must have a very low barrier for publishing, and must ensure that a particular package can be readily identified while not requiring a new global namespace of package names. It must nonetheless be possible to retrieve a known package to install easily. The combination of ready publishing, globally-unique names and ready retrieval led us towards using URLs to identify packages.

A URL in a well-known protocol like HTTP can be dereferenced to retrieve a file. Publishing such a file is generally a mere matter of copying the file into place on a web server. URLs are guaranteed to be globally unique by the existing domain registration and DNS systems, and by the simple fact that it is possible to retrieve a particular file from them.

We can reinterpret some import paths as describing URLs, and then provide a tool to retrieve and install libraries from the internet. Minigrace includes these behaviours as an extension, and it is proposed for adoption into the language specification.

This approach does have some limitations compared to other systems because of the trade-offs we made in the context of Grace. Many package managers have some support for versioning of packages, which this approach does not. Where referring to a particular version may be necessary, however, a version number can be placed in the URL itself, along with any other desired conventions. URLs are also not necessarily permanent: domain names can expire, or the files can be changed, but these problems are no greater than for any other system that retrieves packages from the origin server. There is also no explicit support for signing or verifying

distributed packages. For the target audiences of Grace, however, these limitations are less important than for an industrial language.

5.4.2 Finding packages

The key choice here is to reinterpret an import path as a URL, simultaneous with the interpretation already given by the implementation. Minigrace already maps the string onto a local path: for example,

```
import "x/y" as y
```

will look for a file ~/.local/lib/grace/modules/x/y.grace (among other locations determined by the installation configuration). While maintaining this interpretation, the package manager can also interpret the import path as a URL, and attempt to retrieve it when requested. The compiler need have no knowledge of this interpretation: the package manager can retrieve the relevant file and store it in the same place the compiler will look for it. The package manager determines whether the import path is a candidate to be mapped to a URL, and if so constructs the appropriate HTTPS URL corresponding to the import path.

5.4.3 Installing packages

We have implemented this system into a tool called gracepm, which is included in the Minigrace distribution. The package manager supports two major modes: install, which retrieves and installs a particular module and its dependencies, and satisfy, which ensures that all dependencies of a given Grace source file are met. A worked example in more detail is given in Appendix C.

5.4.4 Publishing packages

Earlier we also said it must also be easy for a package author to publish their library, and from here we see how they can do this: they simply put

their source code on the web, and it is immediately accessible to all users, without any packaging or registration step. We consider this to be as simple as it is reasonably possible for publishing a package to be, and in line with the target audience of Grace.

Because dependencies are already manifest in the source code, no additional configuration or metadata is required from the publisher, and the famous "don't repeat yourself" principle [76] is fully satisfied. The package manager can examine code for **import** statements to determine which dependencies are required. The source code and metadata can never become desynchronised because they are one and the same, eliminating a common class of errors in existing systems. In particular, the author can never accidentally add a code dependency without adding it as a packaging dependency as well. This result flows from our design requirement that dependencies be explicit (R8).

Delegating package names to the URL system resolves both the uniqueness problem and accessibility. By incorporating a domain name, uniqueness is delegated onto the domain-name registration system and then the owner of the domain (just as Java uses reverse-DNS names for packages by convention, and for the same reasons). Interpreting the path as an HTTPS URL maps it onto an already-defined access mechanism. Webspace is easy to come by, particularly for instructors at institutions, and web access is ubiquitous. It is worth noting that only published libraries need use this URL format for their imports: local code can continue to use local-only names, or any names supported by the user's implementation.

5.5 Extensions and future work

The module system design that we have presented is open to a number of extensions. In particular, it is possible to interpret the import string in various ways, without affecting the rest of the language. Moreover, because a module presents itself as an object, its internal implementation

and behaviour are hidden. In this section we present some preliminary experiments and discuss future extensions that exploit these features.

5.5.1 Foreign objects

We can access code written in other languages, or behaving in unusual ways, by compiling it appropriately and then importing it in the ordinary way. This functionality is found in various forms in many dynamic languages.

Objects that have been imported from a source outside the universe of Grace code are called "foreign objects". From the perspective of client code, there is no difference between an import that returns a foreign object and an import that returns an ordinary module object. Internally, a foreign object may construct new objects or classes "on the fly" to represent the resources it provides, and it may access other libraries available on the implementation platform.

Because these foreign objects present themselves as ordinary objects to Grace code, all of the ordinary facilities of objects and modules are available for use with them, including type specification and ascription during import (as described in Section 5.2.2), aliasing, and using them as parameters. A foreign (perhaps optimised) module may be substituted for a Grace implementation used with a type specification in exactly the same way that another Grace implementation could be so substituted.

The implementation of the foreign object itself is tied to the ABI of the language implementation: it must present objects in the format the run-time system requires and accept method requests using the protocol the run time uses. The code may need to marshal and unmarshal these. Errors in foreign code can have arbitrary effects and break invariants of the language. Because objects are an encapsulation boundary, however, from the perspective of the Grace client the foreign module is indistinguishable from a Grace module.

We have written a fairly complete Grace binding to the GTK+ widget

library [69], described in more detail in Section 8.4.1, that demonstrates foreign objects. A Grace program can use this module as follows:

```
import "gtk" as gtk
def window = gtk.window(gtk.GTK_WINDOW_TOPLEVEL)
def button = gtk.button
button.label := "Hello, world!"
button.on "clicked" do { gtk.main_quit }
window.show_all
gtk.main
```

This code creates a window with a "Hello, world!" button that terminates the program, using a Grace transliteration of the underlying GTK+ interfaces. GTK+ is an object-oriented library, and its object features are mapped directly to Grace objects. Here, notwithstanding that the `"gtk"` module is not Grace, to the client code this is an ordinary module import. Because object implementations are always opaque the module object is indistinguishable from one defined in Grace code. Our prototype compiler understands how to find and load a module including these bindings, along with any metadata needed for compilation.

How these objects are constructed may vary. In the case of our GTK+ module, the GTK+ API has been pre-processed to generate wrappers conforming to the runtime's object format. In the similar module wrapping the HTML Document Object Model for the ECMAScript backend, the foreign system's native reflection is used at runtime. In the next section we discuss a speculative use of foreign objects to access external data sources.

Because these foreign objects appear as ordinary Grace objects, all of the gradual typing functionality discussed in Section 5.2.2 will work unchanged. While the actual implementation is unknown, the public interface of the object is subject to the same strictures as any other object. In the case where the interface of the foreign module or other objects returned from its methods is unknown or subject to addition at runtime, the gradual enforcement can be based on the information currently available.

5.5.2 External data

Because the source of an import is a string whose interpretation is left to the implementation, we can give certain strings special interpretations. In particular, we can interpret some strings as references to external data sources like web services, databases, and local metadata, to be reified as foreign objects. The overall effect is similar to F#'s "type providers" [175]. These foreign objects can be implemented either dynamically (as in the GTK bindings) or by code generation, as in F#. Minigrace supports both approaches by providing a general hook in the import system, but much future work remains to build useful bindings to external data sources, determine how they should be integrated into the import system, and answer questions about typing.

The idea behind this feature was uncovered serendipitously as we developed the import facility for objects-as-modules, showing the power of a simple mechanism used consistently. We hope to extend our prototypes to support a wider range of external data sources and investigate dynamically-provided types further in the future.

5.5.3 Resource imports

As described so far, it is possible to write

```
import "path/to/module" as mod
```

to get a Grace object containing the top-level definitions from the given module, bound to the name "mod". Given that this is "just" an object, there is no reason it necessarily has to arise from Grace source code. We have already described the ability to import native code conforming to the appropriate ABI, for example, in Section 5.5.1.

Given this existing mechanism, it would also be useful to access other resources through it. In Grace, everything is an object, so the representation of, say, an image, is naturally an object. Because the implementation of objects is opaque, we cannot know what is in it or where it came from

unless we wrote the object constructor ourselves. We could access these through the import mechanism.

We can again interpret the import path in a new way, as referring to a piece of data in some known file format, which can be presented as an object. Such data might include help text, which would be represented as a string, or a PNG image to be used as an icon, which would be represented as an image object; we call these outside pieces of data *resources*. Resources are not part of the source code directly, but are imported from the outside.

To do so we need to know three things: that what we are importing is a resource, what kind of resource it is, and how to find it. We can resolve the last through the ordinary import lookup (and the package manager described in Section 5.4). The others are more complex.

There are two conventional and widely-deployed[2] methods for determining a file type: by the file name, especially a particular part called an extension, and by "magic number"[3] inspection of the file contents. We considered both of these approaches.

Magic number inspection is fraught and prone to failure, as well as requiring incorporating a database of identifying features into the compiler. We considered this undesirable, particularly when we wanted to work across varying platforms.

Using the file extension ties the interpretation of the string to a particular filename layout, and requires we map these extensions onto file handlers somehow. We found this less objectionable, but worked to eliminate the rough edges where possible.

Our ultimate design draws from the "extension" mechanism. As import strings are (overall) uninterpreted, we must define what we consider an extension to be. We did not wish to eliminate already-working import

[2]We dismiss out-of-band metadata such as Macintosh OSTypes and Apple UTIs for this reason, as well as HTTP's and MIME's Content-Type header, as they are not available on all platforms.

[3]A "magic number" is a specific sequence of bytes known to be at a particular location in a file of a given type. The term originally referred to the first two bytes of a file on Unix systems, but now has the broader meaning given.

statements from continuing to function.

To do so, we instate a syntactic restriction on the import paths of Grace modules: the given import path may not contain a "." character (the Unicode character U+002e FULL STOP) after the final occurrence of "/" (U+002f SOLIDUS). Remember that, when importing a module, we do not include ".grace" in the name; this restriction only prohibits importing from a path implying a source file name like "foo.bar.grace".

If such a character does appear, this import statement is a *resource import*. A resource import still binds the given name to an object identified by the given import path; the difference is that the meaning of the import path may be interpreted by other code.

What code is that? It is determined by the part of the name following the ".": in essence, the file extension determines an import handler to be used. If a module contains the import statement:

```
import "my/tool/logo.png" as logo
```

then the `"png"` import handler is handed the path `"my/tool/logo.png"`, as a string, and returns an object to be bound to logo. This object can be anything the handler likes, but is probably some representation of an image in this case. If instead the module contains

```
import "my/tool/licence.txt" as licence
```

then licence is likely to be a String.

How are these extensions mapped to handlers? A per-program registry is defined, and user code can add entries to it. When using the "gtk" module, for example, it may extend the registry with an entry mapping the "png" extension to something creating a GTK+ image object. Other modules may provide their own definitions. The system may predefine some handlers, like "txt" as a String mapping, but these could be overridden.

What if we assume a Smalltalk-style programming interface, with a closed world, and no "files"? Again, as the interpretation of the path is defined in user code, it can be mapped according to whatever approach makes sense in that system. The extensions are simply to identify an

interpretation in that case.

What advantages does this have? We can access resources, like images, documentation, or sound files through a single consistent interface. It fits entirely within the existing semantics of the language, and also provides a logical location to store relevant resources adjacent to the code that requires them, and even to distribute them in the same way as code through the package manager described in Section 5.4. We extended our package manager to support resource imports, which it treats simply as opaque files to retrieve and place in the correct directory structure for the ordinary import mechanism to handle. The meaning of `import "logo.png" as logo` is likely to be clear to the reader, even if they are unfamiliar with other code. The disadvantage is that there is an added point of complexity in the import system, and that a class of otherwise-valid filenames is excluded from use as a module name.

There are some drawbacks to this design: firstly, we have introduced a global registry, which inherently creates the potential for side effects at a distance and the potential of conflicts. Secondly, typing is an issue; because resource imports are handled at runtime we may not be able to typecheck them statically. Gradual typing addresses many of the typing issues. Finally, we have given an undesirable semantic importance to file extensions; the "morally correct" way to represent file types is probably by MIME type (now Internet Media Type [55]), but doing so is challenging on most platforms, and some file extensions may correspond to multiple types. The latter problem can be mitigated by defining "false" extensions, which do correspond directly to a type and mangle the import path appropriately.

5.6 Comparison with related work

In this section we contrast our design with the module systems of Python, Newspeak, and Go.

5.6.1 Python

Python's module system also constructs objects containing the top-level declarations of a source file and makes them available to other code through an **import** ... **as** ... statement. Python's system is the most similar to our design, although there are several important differences.

In order to support namespaced modules, Python relies on the meta-mutability of its objects: new fields can be added to an existing object at runtime. In the case of the statement:

```
import x.y
```

the module `x` is first imported, with its source code found in x/__init__.py. The object representing this module is bound to x in the local scope; if `x` had already been imported elsewhere, the existing object would be aliased. In either case, `y` is then loaded, from either x/y.py or x/y/__init__.py. The object representing `y` is stored as the value of a new field y on the x object, so that the module can be accessed as x.y, just as it was imported. This addition of a field to an existing object is observable, because the module may have been imported elsewhere and so be aliased.

Grace objects are not meta-mutable in this way, and such modification breaks multiple of our system requirements: in particular, we required that all dependencies be explicitly manifest in the source code of a module. Python's approach permits latent dependencies that are invisible and may be unrealised by the programmer: if their module imports `x`, and another module imports `x.y`, the programmer will be able to use x.y.z without ever importing `x.y` themselves. If the other module ceases to be imported, the programmer's code will break mysteriously. In our system a module is never modified by importing another module.

As well, the creation of new fields causes potential name clashes, violating our namespace rule. If module `x` already contains a field y, its value will be destroyed by importing `x.y` and replaced by the imported module object. The author of `x` does not necessarily know of the existence of `x.y`, as

modules can be installed into other namespaces since they are represented simply by directory structure; there is no way for the author of `x` to avoid these potential clashes with its local definitions. This behaviour can be observed in the following example:

`x/__init__.py`

```
y = 1
print("In loading x, y is {}".format(y))
def test():
  print("Now in x, y is {}".format(y))
```

`x/y.py`

```
print("Now loading y")
```

`test.py`

```
import x.y
x.test()
```

Running `test.py` will output:

```
In loading x, y is 1
Now loading y
Now in x, y is <module 'x.y' from 'x/y.py'>
```

The value of x.y has visibly changed, even inside the module itself. We considered this behaviour highly undesirable, and have experienced real code where both field overwriting and implicit dependencies have caused problems. We avoided both of these issues in our module design by identifying modules through strings instead of base syntactic forms, and ensuring that every module object was entirely separate from any other.

Our modules are treated consistently with the rest of the language, simply treating the file as though it were enclosed in **object** { ... }; in Python, modules are a sui generis construct, and the only way that an object can ever be created without writing its body in a class. While in Grace the same rules apply to all object definitions including modules, in Python

there are slight differences between the two in initialisation and scoping. Because Grace objects include typing and encapsulation, so do modules in our system.

5.6.2 Newspeak

Bracha *et al.* [15] describe modules as classes in Newspeak. In this language a module definition is a top-level class, whose instances are termed "modules". Classes can be nested, and the code in a class can access external state in three ways: lexically, from an outer scope; from an argument provided at instantiation time; or from a superclass.

The most obvious difference is that in our design, modules are objects, rather than classes. There is always at most one instance of any module object in a Grace program; if a programmer wishes to permit multiple instances of some object, they are free to define a class themselves. Because Grace classes are simply syntactic sugar for objects anyway, our system provides the flexibility of using either approach, with the module able to act as a class if desired.

Because Newspeak module definitions are always at the top level, with no surrounding class, they must be given all other modules they will use by one of the other means. Most commonly, the class is parameterised with a "platform" object exposing references to all the module objects the class will use, but it is also possible to inherit access from a superclass. In the case of a "platform" object dependencies are not present in the module source at all; instead, dependencies are known or determined in the outside and injected in. Given only a module's source it is not generally possible to know what other modules the module relies upon.

Passing in dependencies dynamically has some advantages: notably, as modules are parameterised over dependencies, the same module can be used multiple times with different dependencies. Dependencies may even be determined at run time. The downside, however, is that given a piece

of code one cannot track what is happening without knowing where it fits into a particular running system. We considered explicit dependencies to be important, especially for the audience of Grace, and so ensured that they were always manifest in the source code.

Further, we felt that setting up or even passing along such an object is another point of complexity for novice programmers that will seem to be a magic incantation. Through this incantation certain unknown modules come into view, but which they are may change. In contrast, our practice of binding modules using **import** statements makes clear both which modules are being used, and how they are named.

5.6.3 Go

In the Go language [60], a package may comprise several files, all of which declare themselves part of the same package. All of these files will be in a single directory, and are all combined together to present a single flat interface; the ordering of different files in a package is undefined. Our design does not include this multiple-file approach to a module, although we strictly do not prohibit it either, as the interpretation of an import path is up to the implementation. Nonetheless, we believe that file-per-module is the more suitable approach, particularly as Grace wishes to support top-level executable code for both programs and modules; in Go, at most one `main` function can be defined, which will be the entry point to the program, while many `init` functions can execute initialisation on a per-file basis with undefined ordering. Our design executes a module, whether the program's entry point or not, linearly from top to bottom, which we find more straightforward.

A package may be accessed by clients using Go's **import** statement. As in our design, **import** takes a string literal as argument to identify the package to access. The string is formally defined to be opaque, but in practice maps onto a filesystem path in some implementation-defined way,

exactly as in our design.

Associated tools are able to interpret import paths as URLs and use them to fetch and install modules from version control systems at remote locations when required. These tools have special support for some code-hosting sites, as well as explicitly annotating an import path with version-control information, and for finding the source repository by parsing HTML found at a web location. By contrast, our package-management design (Section 5.4) depends only on literal HTTP requests that retrieve the code to be used, combined with manifest dependencies; we consider our approach more suitable to the needs of Grace, as it depends only on access to web space and needs no external tools that instructors or students would need to learn to use in order to publish code. As the source code already includes imports for all dependencies, modules are single files, and the import paths can be transformed directly into HTTPS URLs, there is no need to repeat any information. Nonetheless, we were influenced by Go in permitting a URL interpretation of import paths.

Once imported, the module's public interface is available through a dotted name, but the module itself has no run-time existence. The static appearance of modules is similar in Grace and Go, using dotted qualified-name notation and a similar import statement, but our modules beget real objects: while `math.Sin(x)` is a valid method access from a module in both Go and our design, in ours `math` is a real object and this is a real method request; in Go the expression is a special case in the parsing with static dispatch. Go also permits an unqualified import that brings all definitions into the local scope; we considered this a bad practice in general and do not support it at all. For some uses of such imports the dialect system described in Chapter 6 may suffice.

5.7 Conclusion

Grace requires a module system in order to express non-trivial programs. We have presented a design based on reifying source files as objects, conceptually by embedding the source code directly inside an object literal. Our design leverages existing features of the Grace language to build a semantically consistent model of modules that could be applied in similar languages.

We derived a set of requirements a module system should meet: namespacing, sharing of objects, type independence, controlled export, explicit dependencies, interchanging multiple implementations, separate compilation, and allowing foreign implementations. We showed how modules-as-objects meets each of these requirements. We also showed how careful design of the functioning of our **import** statement permitted a range of functionality within the same semantic model, including room for easy extensions like package management and resource imports.

We will use our module system as a key component of building dialects in the next chapter. Modules-as-objects provides the consistent semantic model for dialects. We also use it extensively in our Minigrace compiler (Chapter 8), which is composed of many modules.

Chapter 6

Dialects[1]

Grace is intended to support a variety of approaches to teaching programming, including objects-early, objects-late, graphics-early, and functional-first curricula. Similarly, it should be possible to teach courses using dynamic types or static types, to start with dynamic typing and then gradually move to static typing, or to do the reverse, without having to change to a different syntax, IDE, and libraries.

To support these different approaches not only must the language contain the material features (gradual typing, first-class functions, object literals) but it must provide support for structuring a program and a course in these ways. For this reason the language has always intended to support *dialects*: language variants supporting different approaches.

This chapter describes how we have designed a coherent system for dialects, permitting not only pedagogical dialects but a variety of domain-specific languages. Our design builds on well-understood features found in many languages, combining them in a novel way to allow both the addition and the removal of features from the language, while permitting interaction between code in different dialects and retaining the fundamentals of the underlying language intact.

Our design is mindful of the intended audience of Grace — namely

[1]This chapter expands on a paper published in ECOOP 2014 [71].

students and instructors — and takes care to limit the burden placed on these users. At the same time, the dialect system is powerful enough to create languages with very different appearance, purpose, and behaviour, to permit experimenting with novel type systems or other restrictions, and to allow languages aimed at domain experts who are not programmers so that they can specify the logic in terms they understand.

The research questions we were forced to confront were how

- to "close the gap" between core functionality and library functionality, so that programmers using facilities provided by a library don't see them as being "second class";
- to devise a dialect mechanism that can be used to restrict the language, as well as augment it; and
- to do both of the above *as simply as possible*.

The contribution of this chapter is to show how a novel combination of a few carefully-selected features — lexical nesting, concise syntax for lambda-expressions, multipart names for methods, optional typing, and pluggable checkers — gives support for quite a wide range of dialects. We illustrate the power of the dialect mechanism by presenting several case studies, including:

- a domain-specific graphical micro-world,
- a dialect for writing dialects,
- a dialect giving an extended static type system.

The next section presents our design for dialects, explaining how dialects build on key features of the Grace programming language. Section 6.2 then validates our contribution by presenting case studies of dialects defined in our system. Section 6.3 discusses alternatives and extensions to our design. Section 6.4 briefly describes our implementation. Section 6.5

contrasts our approach with a range of related work, and Section 6.6 concludes.

6.1 What is a dialect?

Dialects are modules that provide supersets or subsets of the standard Grace language. Dialects can not only make extra definitions available to their users; they can also restrict the language by defining and reporting new kinds of errors, and can change the way in which existing errors are reported. Dialects support the definition of language subsets to aid novice programmers, and of domain-specific languages.

An *extensional* dialect provides new definitions to its clients; a *restrictive* dialect prohibits certain programs that would otherwise have been acceptable. A single dialect may be both extensional and restrictive.

6.1.1 Structure

A module declares the dialect in which it is written with a dialect declaration as its first line:

```
dialect "beginner"
```

Where no dialect is specified, the module is in the standard language. A dialect declaration loads the module identified by the string, just as if it were imported (see Section 5.2.1). Unlike an import statement, however, the dialect declaration does not bind the imported object to a name: instead, the *dialect object* is installed as the lexically-surrounding scope of the module that uses it, as shown in Figure 6.1 on the next page.

Any request inside the client module for a method defined in that outer scope — most likely a receiverless request — will access a method of the dialect. This resolution rule is the same as that used for any other receiverless request, such as one from inside an object literal out to a surrounding

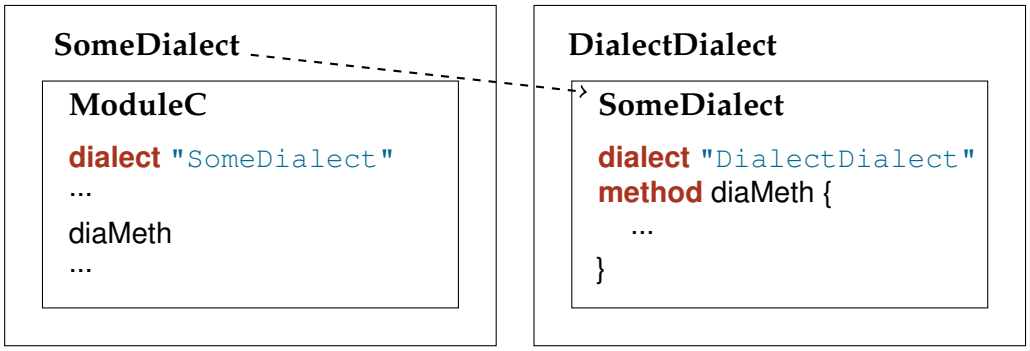


Figure 6.1: Object nesting structure with dialects. Notice that dialect use is not transitive: ModuleC is inside SomeDialect, and SomeDialect is inside DialectDialect, but ModuleC is not in DialectDialect.

object. As a Grace module is an object (Chapter 5), this is in fact exactly what is happening.

In Figure 6.1, ModuleC is written in SomeDialect, indicated by the `dialect "SomeDialect"` line. As a result, the ModuleC scope is placed directly inside SomeDialect. A receiverless request for a method not defined locally, such as that to diaMeth, is dispatched to SomeDialect.

A module that defines a dialect may itself be written in a dialect. This exposes a difference between dialectical nesting and other kinds of lexical nesting: the current dialect is always the *outermost* lexical scope. Dialect nesting is not transitive, which is a deliberate design decision. As special-purpose dialects, particuluarly those defining educational subsets, will often be less powerful than the language as a whole, the dialect itself will likely be written in a dialect (or the unrestricted language) that provides features which should not be exposed to clients.

In Figure 6.1, SomeDialect is itself written in the DialectDialect, and so is treated as lexically inside it. ModuleC is written in SomeDialect, but it is not inside DialectDialect, either directly or transitively. A receiverless request inside ModuleC to a method not found in SomeDialect will not reach DialectDialect, although the SomeDialect object and scope are the

same in both cases.

Standard Prelude

When no dialect is specified, the module is assumed to be written in the full Grace language described in Chapter 3. As in many languages, a standard prelude of built-in definitions is provided by default in Grace, including standard control structures and utility methods. The dialect method provides a coherent explanation for how this standard prelude works: a program with no dialect declaration generates a module object nested inside the standard prelude object, and so in the prelude's dialect.

Because a dialect replaces the default nesting, the author of a dialect can choose whether or not to expose the standard prelude's methods to clients. If they wish to do so, they can inherit all the methods of the standard prelude into their module with:

```
inherits StandardPrelude.methods
```

A dialect author who wishes to restrict or replace the default definitions needs do nothing to leave them out.

6.1.2 Pluggable checkers

As well as providing new definitions, dialects may restrict access to particular features of the language, or offer additional and more specific error and warning messages. The latter are particularly useful in the context of Grace because novice students can benefit from error messages that are tailored to the more restricted things that they are trying to do, compared to more advanced programmers.

Restrictions and new error messages are implemented by the dialect module defining a *checker method*, which is executed when modules written in the dialect are compiled. The checker method is passed the abstract syntax tree of the client module to inspect. The checker cannot modify the tree, but can check any properties the dialect needs, and indicate to the

compiler whether it should proceed or terminate with an error, and what that error should be.

Checkers have the same ability to find and report errors as the compiler itself. They can perform any analysis they require: for example, a dialect may wish to perform a flow analysis to ensure that method parameters are used. In fact, Grace's static type checking is itself implemented in a checker. If an error is found, the checker can report that error to the user, including whatever information the dialect author thinks is relevant, and either carry on to find more errors or stop at that point. Several modules that provide varied degrees of checking can be used within the same overall program, so a student's code can be subjected to strict constraints, while still being able to use a module provided by their instructor written in a more powerful dialect.

While a checker can examine the code of its client module using any technique the programmer wishes, we provide two mechanisms to make dialect-creation easier. The most basic is support for the Visitor pattern [56] on the AST nodes, which we show in use in Section 6.2.4; the other is a dialect to support largely-declarative definitions of checkers, which is presented in Section 6.2.3.

6.1.3 Run-time protocol

A dialect may wish to run code immediately before or after a module using it, perhaps for logging, initialising data structures, or launching a user interface. To enable this, the dialect protocol includes two further methods the dialect can define: atModuleStart and atModuleEnd.

atModuleStart

atModuleStart is requested, if it exists, immediately before the module written in the dialect is executed. The method receives a single argument: a string containing the import path of the client module. In Figure 6.1, the

string `"ModuleC"` would be provided to SomeDialect.

The dialect object is always fully initialised at this point. Other modules may have executed code and may be partially or fully initialised, but none of the body of the client module has run yet (in fact, the module object does not even exist at this point).

atModuleEnd

atModuleEnd is requested immediately after the body of the client module completes. This method also receives a single argument, this time a reference to the client module object itself. The entire body of the client module has executed at this point, as have any modules it imports, and it is fully initialised.

Multiple modules in the same dialect may be imported in such a way that these two methods are interleaved. In this case, it is always guaranteed that modules are processed in "stack" or FILO order: after atModuleStart is given the import path, an equal number of atModuleStart and atModuleEnd requests (possibly zero) have been made to the dialect module before the corresponding atModuleEnd request for the first module.

6.2 Case studies of dialects

To explain and illustrate the power of our design, this section presents case studies of dialects and their implementations. All case studies are fully implemented and included in our source repository.

6.2.1 Logo-like turtle graphics

Our first case study is a simple dialect that supports procedural turtle graphics, like Logo. This dialect is designed to be used by early students to learn geometry and basic control structures with as little overhead as possible — in particular without the syntactic and semantic overhead of a more

```
dialect "logo"
begin
def length = 150
def diagonal = length * 1.414
lineWidth := 2
square(length)
turnRight(45)
lineColor := blue
forward(diagonal)
turnLeft(90)
lineColor := red
forward(diagonal / 2)
turnLeft(90)
forward(diagonal / 2)
turnLeft(90)
lineColor := blue
forward(diagonal)
end
method square(length) {
    repeat 4 times {
        forward(length)
        turnRight(90)
    }
}
```

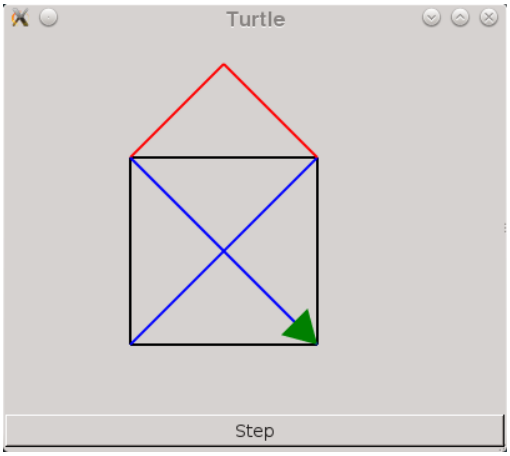


Figure 6.2: A simple program in our Logo-like dialect and its output. This example parallels one on the Xtext web site [43].

object-oriented style. We define a dialect giving access to simple movement primitives and presenting what amounts to a procedural language, demonstrating the basic behaviours of extensional dialects. Figure 6.2 shows a simple program in this dialect, and its output.

This dialect is straightforward to implement, and is shown in Figure 6.3. It defines methods for each instruction to be given to the turtle (turnLeft, forward, etc). The bodies of these methods perform the relevant task, in this case by delegating to an existing turtle graphics library.

More interestingly, the dialect defines variables holding pieces of state,

```
import "turtle" as turtle
import "StandardPrelude" as StandardPrelude
inherits StandardPrelude.new

def red = turtle.red
def green = turtle.green
def blue = turtle.blue
def black = turtle.black

var lineWidth := 1
var lineColor := black

method forward(dist) {
    turtle.move(dist, lineColor, lineWidth)
}
method turnRight(ang) {
    turtle.turnRight(ang)
}
method turnLeft(ang) {
    turtle.turnLeft(ang)
}
method penUp {
    turtle.penUp
}
method penDown {
    turtle.penDown
}
method repeat (n) times (blk) {
  var counter := 1
  while {counter <= n} do {
     blk.apply
     counter := counter + 1
  }
}

method atModuleEnd(mod) {
    turtle.start
}
```

Figure 6.3: The complete Logo-like dialect. The turtle module performs the platform-specific drawing operations.

such as the lineColor variable used in Figure 6.2. These variables are in scope in the client code, and so can be assigned directly: lineColor := red, exactly like any other variable assignment (potentially, introducing the concept to the student). The dialect could alternatively define accessor methods lineColor and lineColor:= to add customised behaviour to accesses. A number of colour constants (red, blue, etc) are also defined in the dialect, available to the client.

Finally, the dialect uses the atModuleEnd method defined as part of the runtime protocol (see Section 6.1.3) to launch the platform-specific visual display when the client code has completed.

Because Grace includes first-class blocks and multi-part method names, we can also implement new control structures as methods that take blocks as arguments. The dialect can provide a Logo-style repeat loop as a method that declares a counter variable and delegates to an ordinary while loop.

6.2.2 Design by contract

Courses taking a formal approach to software engineering may wish to teach programming disciplines such as Design by Contract, using pre- and post-conditions, and loop variants and invariants, as in Eiffel [125]. A dialect can provide these facilities in Grace. Our approach here is reminiscent of Scala, but based on dialects rather than traits [137].

The simplest support is for assertions — for example, asserting that the array used to store keys in a hash table has the same size as the array used to store the hash table's values.

```
assert {hashTable.keyArray.size == hashTable.valueArray.size)}
```

This assert "statement" is defined in a dialect as a method that accepts a Predicate (a parameterless block that returns a Boolean when evaluated). If the value of the predicate is false, the assertion has failed, so we raise an appropriate exception:

```
method assert( condition : Predicate ) {
 if ( ! condition.apply) then { InvariantFailure.raise }
}
```

We can extend this technique to support pre- and post-conditions on methods, inspired by Eiffel's "require", "do", and "ensure "clauses:

```
method setHours ( hours' : Number ) {
  require { (0 <= hours') && (hours' <= 23) }
   do { hours := hours'; hours }
   ensure { result -> (result == hours') && (hours == hours') }
}
```

The identifier `result` in the `ensure` clause refers to the value returned by the method.

This construct can be written straightforwardly, based on Grace's multi-part method names. As in Eiffel, pre- and post- conditions are checked dynamically.

```
method require( precondition : Predicate )
       do ( body : Block )
       ensure ( postcondition : Predicate ) {
   if ( ! precondition.apply )
      then { InvariantFailure.raise "Precondition Failure" }
   var result
   catch { result := body.apply }
      case { _ -> InvariantFailure.raise "Unexpected Exception" }
      finally {
         if ( ! postcondition.apply(result) )
            then { InvariantFailure.raise "Postcondition Failure" }
      }
      return result
}
```

Going still further towards Eiffel, we can add support for specifying and checking loop variants and invariants in programs written in the dialect:

```
loop {
    print(letters[i])
    i := i+1
}
invariant { i <= (letters.size + 1) }
until { i > letters.size }
variant { letters.size − i + 1 }
```

Once again, expressions defining variants and invariants (as well as the code for the loop body) are supplied as blocks, which are evaluated as required by the implementation of the loop()invariant()... method.

6.2.3 Dialect for writing dialects

Programmers writing different dialects will have similar needs. In particular, writing checkers requires inspecting the user's code and determining whether or not it is acceptable; the *form* of this inspection will be the same in many dialects. We have abstracted these repeated tasks into a dialect of their own. Our dialect dialect makes it possible to declare rules to test different parts of the source code and to report errors; these rules are used in the static dialect in Section 6.2.4. The dialect dialect can also maintain state; we demonstrate the use of this for type checking in Sections 6.2.6 and 6.2.7. The dialect dialect hides the details of the checking process and allows programmers to write largely declarative dialect definitions.

Fundamentally, Grace checkers are methods that examine the nodes of the program's abstract syntax tree at compile time. A checker either accepts a node, or raises an exception to report an error. The AST nodes support the Visitor Pattern to assist in this examination. Although quite efficient, this kind of code is too low-level to be written by many instructors, who may nevertheless need to write dialects for use in their teaching.

The dialect for writing dialects simplifies the process by implementing a generic visitor that applies higher-level rules. These higher-level rules are specified by a combination of declarative and imperative code, leveraging

the pattern-matching functionality described in Chapter 4.

The dialect maintains a list of rules specified by the dialect client, stored in the rules field of the **dialect** module. The client declares rules by requesting the rule method.

```
method rule(block) {
  rules.push(block)
}
```

The block argument is expected to implement a partial function (see Section 4.4.8), which the dialect will attempt to match against each node. For example, if the dialect client declares a rule:

```
rule {v : VarDec -> fail "Var declarations are not permitted."}
```

then the rule will be applied to all nodes in the AST; if any are matched by the VarDec pattern, the body of the rule will execute, and it will report an error.

The dialect maintains a list of rules to apply in a module-level object rules. The rule method takes as an argument a single-argument block that accepts an AST node, and adds it to the list of rules.

Rules can be more declarative still. One useful more complex rule is when()error(), which reports a given error when a test is satisfied.

The first argument is again a partial function, now returning a Boolean indicating whether to report an error or not. The second argument is the message to give when the test is true. A dialect could ban the variable name x by declaring a rule:

```
when { v : VarDec -> v.name.value == "x" } error "x is a bad name"
```

This rule combines three parts: a pattern match (or partial function application) applying only to VarDec nodes, a Boolean test of a property of that node, and an error message to report when the other two are satisfied. The dialect author does not need to consider the structure of the AST or manually write logic generating errors; everything happens automatically once they describe what they want. Nonetheless, they can write arbitrarily complicated tests or errors if they wish to do so.

These more advanced rules are defined in terms of the basic rule method:

```
method when(pred : UnaryPredicate) error(msg : String) {
  rule { node ->
    def matches = pred.match(node)
    if (matches.andAlso {matches.result}) then { fail(msg) }
  }
}
```

Here rule is given what is in fact a total function — one with no constraints on its domain — that always runs, and then tries to apply its own partial function condition. When the partial function runs and returns true, the method reports a failure with the provided message.

The dialect dialect defines a single visitor over the AST that runs all the rules over every node:

```
method visitDefDec(node) -> Boolean {
  runRules(node)
}

method visitVarDec(node) -> Boolean {
  runRules(node)
}
// And so on.
```

Matching rules will be executed until either the entire client module has been examined or an immediately fatal error is triggered.

Sometimes it is useful to examine some nodes from a perspective that is different from the way that the AST is defined. For example, parameters appear within method, block, and class definition nodes, but the dialect-writer may wish to treat them all in the same way. To simplify the matching of all parameters, regardless of location in the tree, the dialect constructs special parameter nodes to run the rules against.The dialect also defines a pattern Parameter to match these nodes. This pattern allows the dialect author to write a rule against all parameters wherever they appear, rather than

having to write separate rules to deal with each place in which a parameter may appear. The pattern-matching infrastructure is flexible enough to permit these departures from the basic system simply by defining a custom match method. The static dialect in Section 6.2.4 uses this pattern to ensure that all parameters are annotated with types. Similarly, specialised patterns While and For match while and for loops, common cases that a dialect may want to examine. A dialect author can easily create similar patterns for their own constructs using the aRequestPattern.forName(...) method provided by the dialect dialect.

The pattern-matching approach trades off some efficiency for ease of programming, but efficiency is not a primary goal of Grace. Moreover, we expect most programs, especially in beginner dialects (which are likely to have the most additional checks) to be quite small. A declarative approach allows checkers to be expressed concisely, and to be understood without a deep understanding of the whole of the implementation.

6.2.4 Requiring type annotations

An instructor can require that, for all or part of a course, all student code is fully annotated with types, so that no dynamically-typed code is permitted. The static dialect allows access to all of the ordinary language features, while reporting compile-time errors to students who omit the types on their declarations. If the student's type annotations are wrong, typechecking will catch the error separately. The definition of this dialect is relatively straightforward. We can use a visitor, as shown in Figure 6.4, or the dialect-writing dialect to express it more concisely:

```
dialect "dialect"
inherits StandardPrelude.methods
when { d : Def | Var -> d.decType == UnknownType }
    error "declarations must have a static type"
when { m : Method -> m.returnType == UnknownType }
    error "methods must have a static return type"
when { p : Parameter -> p.decType == UnknownType }
    error "parameters must have a static type"
method checker(code : Code) {
    // The checker method here delegates all the processing
    // to a method provided by the dialect dialect, which will
    // apply the rules defined above.
    check(code)
}
```

The first two rules provide a particular error message to display, specify what kind of node they care about — **var**, **def**, and **method** declarations — and what should trigger the error message. Here, the error appears when the declaration type is Unknown (which is the type of an un-annotated declaration). The last when()error clause matches against the Parameter pattern from the dialect dialect, which was described in Section 6.2.3. The checker method in the static dialect delegates to check from the dialect dialect; check applies all of the declarative rules we have given.

The Visitor implementation shown in Figure 6.4 defines a Visitor object that examines the nodes the dialect needs to check. The object inherits from ast.baseVisitor, which defines default behaviour for every kind of node, so the dialect need only override the methods for nodes it wishes to examine specially. The first two methods examine **var** and **def** declarations, testing whether the declared type is Unknown and raising an exception (to be caught by the dialect infrastructure in the compiler, with the given message and location of the node) if so. The third and fourth address parameters in blocks and **method** declarations, which must now be examined separately.

```
import "ast" as ast
import "StandardPrelude" as StandardPrelude
inherits StandardPrelude.methods
def CheckerFailure = Exception.refine "CheckerFailure"
def staticVisitor = object {
 inherits ast.baseVisitor
 method visitDefDec(v) {
  if (v.decType == UnknownType) then {
   CheckerFailure.raiseWith("no type given to declaration of '{v.name.
    value}'", v.name)
  }
 }
 method visitVarDec(v) {
  if (v.decType == UnknownType) then {
   CheckerFailure.raiseWith("no type given to declaration of '{v.name.
    value}'", v.name)
  }
 }
 method visitMethod(v) {
  for (v.signature) do {s->
   for (s.params) do {p->
    if (p.decType == UnknownType) then {
     CheckerFailure.raiseWith("no type given to declaration of '{p.
    value}'", p)
    }
   }
  }
  if (v.returnType == UnknownType) then {
   CheckerFailure.raiseWith("no return type given to declaration of
    '{v.value.value}'", v.value)
  }
 }
 method visitBlock(v) {
  for (s.params) do {p->
   if (p.decType == UnknownType) then {
    CheckerFailure.raiseWith("no type given to declaration of '{p.
    value}'", p)
   }
  }
 }
}
method checker(values) {
  for (values) do {v->
    v.accept(staticVisitor)
  }
}
```

Figure 6.4: Requiring static types implemented as a Visitor

The checker method loops over the AST it is given and has each node accept the Visitor.

In both cases the dialect is quite short and an instructor teaching Grace should be able to implement at least one of them without difficulty. Other context-free checks — prohibiting mutable variables, enforcing particular method or variable names, requiring object literals over classes, and so on — can be implemented in exactly the same form, as can those requiring only local context, such as enforcing that all parameters are used. Dialects requiring further context can either track it in the appropriate way themselves, or use further features of the dialect dialect described in Section 6.2.6.

6.2.5 Literal blocks

Because the basic control structures of Grace are designed as methods with multi-part names, they admit syntax that may be confusing for those familiar with other languages. In particular, the condition of a while−do loop is written in braces, because it is a block with deferred and repeated execution; it is perfectly valid, however, to use parentheses and suppply an expression evaluating to a block as the first argument instead, as with any other method. A programmer familiar with languages using different syntax may accidentally provide a different argument than they expect:

```
while (x > 0) do { x := x − 1 }
```

The expression x > 0 will be evaluated immediately and a Boolean passed to the method. The programmer will receive a type error, but we could provide a more specific message using a restrictive dialect for new or transitioning programmers.

This dialect ensures that the condition of a while loop is written in braces, as a literal block, and will not permit passing a reference to a block defined elsewhere[2]. It uses the compiler's "suggestions" infrastructure to

[2]The dialect could alternatively check the type of the given expression, and permit those referring to blocks; the principle of the dialect is the same but we elide this complexity here.

```
literal_test.grace[4:7-14]: Syntax error: The
condition of a while loop must be written in {}.
  3: var x := 0
  4: while (x < 10) do {
-----------^^^^^^^^
  5:     print "Counted to {x}."

Did you mean:
  4: while {x < 10} do {
```

Figure 6.5: The output of the literalblock dialect when an error is found.

show a potential corrected code line to the user, which the user interface presents as an actionable suggestion. The complete source of the dialect is in Figure 6.6.

The dialect dialect exposes methods generating the appropriate checker failures, which the compiler will report to the user. In this case, the error is reported as ranging from the first parenthesis to the last, and the user will be prompted as shown in Figure 6.5. A user interface can present this suggestion as an action to be taken, as the web-based IDE does.

6.2.6 Ownership types

Ownership types [26] are a way of enforcing structure on the heap. This case study implements a rudimentary ownership system using the dialect dialect, maintaining state to remember the ownership status of fields.

In this dialect, a field can be annotated **is** owned to enforce that it not be aliased. An owned field can only be initialised or assigned a freshly-created object, and its value cannot be assigned to another field of any object. This is a drastic simplification of real ownership systems, but suffices for the example. In the following program, we would expect an error on the assignment to y, as it aliases the owned field a in a field on the module object:

```
// This dialect enforces that the condition of a while loop must
// be a literal block written inline in the source code, to avoid
// any potential confusion with (). It offers a suggestion to the
// user when they write the condition in parentheses.
dialect "dialect"
import "StandardPrelude" as StandardPrelude
inherits StandardPrelude.methods

// The dialect dialect provides a shortcut While pattern which
// matches while()do requests and destructures the AST node into
// the condition and the body.
rule { req : While(cond, _) ->
    if (cond.kind != "block") then {
        reportWhile(req)
    }
}

method reportWhile(req) {
  // Get a reference to the entire condition 'part' of the request.
  // We will use this to generate the suggestion of replacing the
  // parentheses with braces, if applicable.
  def badPart = req.with[1]
  // Ignore certain degenerate cases where there is no condition, and
  // situations where the condition spanned multiple lines since they
  // are likely to be a different kind of mistake. In all of these
  // cases the source line length of the part's argument list will be
  // reported as zero.
  if (badPart.lineLength > 0) then {
    // We will suggest replacing the () used in the condition with {}.
    // The suggestions system allows modifying the code the user
```

Figure 6.6: Source code of the literalblock dialect.

```
    // wrote to something that they may have meant, and then printing
    // out the suggestion with "Did you mean?".
    def suggestion = createSuggestion
    // These replacements are made right to left, so that
    // offsets in parts accessed later on are still valid.
    suggestion.replaceChar(badPart.linePos + badPart.lineLength)
        with("}")
        onLine(badPart.line)
    suggestion.replaceChar(badPart.linePos)
        with("\{")
        onLine(badPart.line)
    // Report an error to the user, highlighting the part of
    // the code that is incorrect, and including our suggestion.
    fail "The condition of a while loop must be written in \{}"
        from(badPart.linePos)to(badPart.linePos + badPart.lineLength)
        suggest(suggestion)
  }
  // Report an error to the user, highlighting the part of the
  // code that is incorrect.
  fail "The condition of a while loop must be written in \{}."
     from(badPart.linePos) to(badPart.linePos + badPart.lineLength)
}

method checker(code) {
    check(code)
}
```

Figure 6.6: continued.

```
dialect "ownership"
var y
def x = object {
    var a is owned := object {}
    y := a
}
```

The dialect dialect (see Section 6.2.3) provides support for tracking state through an analysis of the program. The rule declarations can optionally return a value representing the "type" of the given node in the tree, and the dialect also supports tracking and accessing different scopes.

To declare that all numeric and string literals will always be considered freshly-created objects, we can write a rule:

```
rule { _ : NumberLiteral | StringLiteral ->
    "fresh"
}
```

This rule means that when the dialect asks for the typeOf a number or string node the string `"fresh"` will be returned. The dialect can choose to return an arbitrary object here; for our purposes, a simple string is adequate.

What about variables? There are two considerations for variables: the declarations, which may be annotated **is** owned, and references to them, which must be connected to the declaration. We will deal with the declarations first. The dialect dialect allows storing state associated with particular variables, so we define that when encountering a **var** or **def** declaration, we store whether it should be considered "owned" or "normal":

```
rule { v : Var | Def ->
    if (isOwned(v)) then {
        scope.variables.at(v.name.value)put("owned")
    } else {
        scope.variables.at(v.name.value)put("normal")
    }
}
```

Next we handle looking up identifiers:

```
rule { ident : Identifier ->
  scope.variables.find(ident.value) butIfMissing { "normal" }
}
```

This rule states that, on encountering an identifier and being asked for its type information, find it in scope and return the associated value, or `"normal"` if it is not found (identifier resolution errors will be handled elsewhere).

Our sample program includes an object literal, which we address with the following rule:

```
rule { obj : ObjectLiteral ->
  scope.enter {
    for (obj.value) do { node ->
      checkTypes(node)
    }
  }
  "fresh"
}
```

Object literals create a new scope for their fields, and instruct the dialect dialect to apply the declared rules against their body **inside that scope**. An object literal is also a freshly-created object, of course, so we return `"fresh"` here as well.

To find our error we will have to examine variable assignment ("bind") statements:

```
rule { bind : Bind ->
  def dest = bind.dest
  def dType = typeOf(dest)
  def value = bind.value
  def vType = typeOf(value)
  if ((dType == "owned") && (vType != "fresh")) then {
    OwnershipError.raiseWith(
      "An owned field can only be assigned a fresh object",
      dest)
  }
  if ((dType == "normal") && (vType == "owned")) then {
    OwnershipError.raiseWith(
      "Only 'normal' and 'fresh' values can be assigned "
        ++ "to normal variables, not '{vType}'",
      value)
  }
}
```

This rule ties together all the information we have put into the system so far. First we find the type of the declaration, dType, which we will find by looking up the identifier, which will in turn find the value stored by our variable declaration rule. We then find the type of the assigned value, vType, by applying the matching rule we defined earlier (numeric or string literal, identifier, object literal).

With both of these pieces of information in hand, an OwnershipError is raised when the ownership types do not match up appropriately. The second check will match our error, and the compiler will report:

```
ownership_test.grace[5:(10)]: OwnershipError: Only
'normal' and 'fresh' values can be assigned to normal
variables, not 'owned'.
  4:     var a is owned := object {}
  5:     y := a
--------------^
  6: }
```

As presented here, this dialect enforces only a very simple ownership system. A more complex system is possible with the same overall structure, by applying the algorithms enforcing the rules of well-known ownership systems; that is out of scope for this project.

6.2.7 Type checking

Because dialects can perform arbitrary checks over the whole of a module, various kinds of check can be moved from the compiler into a dialect, among them typechecking. The basic type system of Grace is structural. The typechecker for this system is to be implemented in a dialect as a static checker; Jones [71] has built a preliminary structural typechecking dialect using the dialect dialect. We present a brief overview of the dialect here; the system was presented in full in the paper at ECOOP 2014 [71].

The Minigrace compiler does not perform any compile-time type checking, instead deferring type checks until runtime. If, however, a module is written in the structural dialect, the dialect will perform structural subtyping checks before the compiler generates code for the module.

The dialect is implemented in the dialect dialect, using its scoping and state tracking. Type rules are written to map expressions to their types and object scopes are retained, including method declarations. Rules both ensure that the type of a node is correct and optionally return the type that node should have.

```
rule { req : Request −>
  match(typeOf(req.in).getMethod(req.name))
    case { _ : NoSuchMethod −> fail "no such method" }
    case { mt : MethodType −>
      for (mt.signature) and(req.with) do { s, w −>
        for (s.params) and(w.args) do { p, a −>
          if (!typeOf(a).isSubtypeOf(p.decType)) then {
            fail "argument does not satify parameter type" }
      }
    }
    mt.returnType  // A request for typeOf(req) will receive this value
  }
}
```

Figure 6.7: An example method request rule from the structural dialect.

An example structural rule is in Figure 6.7. The dialect contains several classes for representing types and uses these for the types of objects, rather than the simple strings used in the ownership dialect in Section 6.2.6.

The dialect provides full structural typechecking for Grace without generics in 1,300 lines of code, including the structural subtyping algorithm and definitions of data structures. There are 18 rules, one for each case that either introduces or may violate a type constraint. This dialect is nontrivial, but could nonetheless be implemented by a programmer knowing both Grace and the type system in question in only a few days (and was).

The typechecking dialect is compatible with other checkers. To complete the implementation of a fully static variant of Grace, the structural and static dialects can be combined:

```
import "static" as static
import "structural" as structural
inherits StandardPrelude.methods
method checker(code : Code) {
    static.checker(code)
    structural.checker(code)
}
```

Simply invoking the checkers from each dialect suffices to enforce the constraints enforced by both. We envisage the structural, static, ownership, and other dialects both restrictive and extensional being composed together in this way.

6.2.8 Relations

Relations, or associations, are representations of the semantic connections between entities in the program's model. UML supports the concept of relations, but Grace does not support them primitively, so relations must usually be put into a more concrete form in a Grace implementation. This has the disadvantage of losing some of the information in the model. A dialect could provide the "objects as associations" [132] design.

```
dialect "object-associations"
def Attends = Relationship<Student, Course>
def Teaches = Relationship<Course, Faculty>
def Prerequisites = ReflexiveRelationship<Course>
// Set up or obtain our data objects
def james = student(...)
...
Attends.add(james, cs102)
...
for (Attends.to(cs102)) do { each -> ... }
```

The dialect allows the programmer to manipulate relationships directly in a natural way, and without obscuring the purpose of the code. The dialect user can write code that is similar to what they might write in a relationship-based language.

Relationship and ReflexiveRelationship here are simply ordinary (generic) methods: they return objects that will track relationships they are told about, and are parameterised by the types involved in the relationship. Relationship objects, like Attends above, maintain a mapping or mappings of objects and expose add, to, from, and other methods to the user.

As a dialect, the dialect can also check code to ensure it uses the exposed objects appropriately. The dialect could even, although we have not implemented this, examine the code and suggest other locations where relationships might be used; a variant similar to the static dialect in Section 6.2.4 could require they be used in place of ordinary fields where appropriate. Where these features are not required, however, and where relationships will be rare, this dialect can also be used as a library; because dialects are simply modules and modules are simply objects, they can be designed with multiple disparate usage patterns in mind.

6.2.9 Finite state machines

We can define a dialect for expressing finite state machines, and for processing and computation on these machines. The dialect allows the machine to be described declaratively, and results in a module object that encapsulates the machine:

```
dialect "fsm"
// Define our states and an action to associate with them
def startState = state { print "Starting" }
def runState = state { print "Running" }
def endState = state { print "Done" }
// Define transitions for various inputs from each state
in(startState) on("A") goto(runState)
in(runState)
  on("A") goto(runState)
  on("B") goto(endState)
method process(symbol : String) {
  // This method delegates processing of state transitions to
  // the transition method, which is defined in the dialect.
  transition(symbol)
}
```

This module performs the computation of a particular finite state machine, and could be imported and used by client code that needs to use that machine. The author of the client code does not need to understand the machine, only its interface (the process method), and the author of the machine does not need to understand the client code. All of the complexity of setting up the machine has been abstracted into the dialect; the goal is for the description of the machine to be intelligible to domain experts (or engineering students!) who do not know Grace. Using a domain-specific language embedded in Grace code allows other code to use the machine directly, without any additional tooling. The user's process method delegates the responsibility for handling state transitions to the dialect-provided transition method, which takes an input symbol (here `"A"` or `"B"`) and performs the appropriate state transition.

6.2.10 GrAPL

Using dialects and Grace's operator methods we can define a dialect reminiscent of APL. Sample code written in the dialect is shown in Figure 6.8.

The grapl dialect defines objects mimicking APL vectors and having many common operators. Programs performing these kind of mathematical operations can be written quite concisely in this dialect, but are likely not to be as readable as those written in more traditional style. Nonetheless, this program has an ordinary Grace parse: ⍋ must be a prefix operator, because it has no receiver before it, while ← must be an infix operator because it is between two terms. The GrAPL versions of these programs require additional parentheses over the original APL versions, because Grace does not include complex operator precedence rules and instead requires the programmer to disambiguate when multiple non-arithmetic operators are used together, simplifying the parse.

While we do not endorse using the dialect system to build languages of this sort, the ability to do so (with a remarkable degree of fidelity to the

```
dialect "grapl"

N ← [1, 2, 3, 4]
print(N) // Prints [1, 2, 3, 4]
print(N + 2) // Prints [3, 4, 5, 6]
print(+/N) // Prints 10

// Standard Lotto example, written exactly as in APL.
print(L[⍋(L ← (n 6 ? 40))])
// Calculate primes up to 20 — note that the / function has its parameters
// reversed here, because of Grace's evaluation order.
print((P ← (n 1 ↓ ⍳20))/~(P∈(P∘.×P)))
```

Figure 6.8: Sample GrAPL code. The "Lotto" example generates six non-repeating numbers drawn from the range $1 - 40$, sorted in ascending order; the prime calculator selects all those numbers in the range $2 - 20$ that are not the product of any two numbers in that range.

original language) shows the flexibility of the dialect system. Conceivably this dialect could be used as a bridge between existing APL code and newer Grace code; because all dialects share the same underlying semantic model, this dialect can interoperate with others.

6.2.11 GPGPU parallelism

Grace-CUDA [68] is a library providing access to NVIDIA's CUDA [134] system for their GPUs, allowing the programmer to run certain data-parallel numeric workloads substantially faster. We describe our implementation of this library in Section 8.4.2; at this point we simply treat it as an external library.

The library provides several methods for accessing the system, in each case taking at least one block of Grace code to execute and some collection to process, and also includes a compiler plugin for Minigrace. The plugin has two roles: it translates the blocks of Grace code to be run on the GPU into CUDA's extended C format and compiles them, and it rewrites the

method requests to pass in additional information required at runtime. In particular, the runtime needs to be passed the values of variables captured by the block explicitly and also needs to know the inferred types they have.

A dialect's checker is permitted only to examine the abstract syntax tree, and not to modify it. A dialect consequently cannot perform the transformation that Grace-CUDA's compiler plugin does. Nonetheless, we can build a dialect providing some of the functionality **without** using a compiler plugin, which illustrates both the power and the limitations of the dialect design.

Through the combination of runtime support and static analysis that dialects can provide, we can create a sublanguage supporting data-parallel computation on suitable graphics processing units, within the Grace language.

We will use the code generation library from the compiler plugin to translate the Grace source into CUDA code. The code generator can turn some explicitly-typed Grace code into equivalent CUDA code, and also infer the types in some circumstances. We will use the library solely from the dialect, without plugging into the compiler itself.

For illustration, our dialect defines its own `for()do` and `for()map` methods which run their bodies on the GPU when possible. When the code cannot be run on the GPU, these methods execute native Grace equivalents in the main thread. The source is shown in Figure 6.9.

Our dialect is written using the `dialect` dialect from Section 6.2.3. We use the checker functionality to run Grace-CUDA's static analysis and CUDA code generation over all of the blocks in the program. Many blocks will be unsuitable, such as those using arbitrary objects or side effects, and these will not be compiled into CUDA. Those that are found suitable will be built in a way that the runtime library knows how to connect back to the source block.

We define our own `for()do` and `for()map` loop methods here, while providing the standard language otherwise. If Grace-CUDA knows how to

```
dialect "dialect"
import "cuda" as cuda
import "cudap" as cudaplugin
import "minigrace/collections" as collections
import "StandardPrelude" as StandardPrelude

inherits StandardPrelude.methods

method for(iterable) do (blk) {
    if (cuda.isFloatArray(iterable)) then {
        if (cuda.existsCompiledSource(blk)) then {
            return cuda.using(iterable)do(blk)
        }
    }
    return StandardPrelude.for(iterable)do(blk)
}
method for(iterable) map (blk) {
    if (cuda.isFloatArray(iterable)) then {
        if (cuda.existsCompiledSource(blk)) then {
            return cuda.over(iterable)map(blk)
        }
    }
    def returnValue = collections.list.new
    for (iterable) do {value->
        returnValue.push(blk.apply(value))
    }
    return returnValue
}

do { b : Block ->
    cudaplugin.tryInferredCompile(b)
}

method checker(ast) {
    check(l)
}
```

Figure 6.9: The source of the CUDA dialect.

translate the provided iterable object onto the GPU, and the static analyser managed to generate CUDA code for the block, we will ask the library to run that code across the given data. If either of these conditions are not met, we run the standard "for" loop with the real Grace block.

Other high-level languages have been compiled to CUDA [34, 63], most notably X10, which has a nearly full implementation. These have usually required either compiler plugins or whole new compiler backends, but have been able to compile a much wider range of programs than this dialect or even Grace-CUDA as a whole can support. With this dialect, we can transform only some of the low-hanging fruit — individual blocks in loops — with a sensible fallback at runtime.

The dialect functionality, combining both static analysis and runtime support at once, allows us to cover many common cases without using a compiler plugin to modify the generated code. The user of this dialect need not notice a difference: the code they write is exactly Grace code, and the semantics are preserved. This application goes further than the goal of our design was to support, but demonstrates the flexibility of this combination. A more complete version of this dialect is included in the Grace-CUDA repository.

6.3 Discussion

We considered three major alternative approaches to dialects: inheritance, delegation, and special-purpose macros. We rejected all of these in favour of the approach described here, each for a different reason.

6.3.1 Inheritance

With an inheritance-based approach, the module using a dialect inherits from the dialect, and dialect methods can be invoked using a receiverless request, since they would be available on **self** in the module scope, and

through **outer** in any nested scopes. The dialect's methods could also be defined as confidential if required.

This approach was inspired by SIMULA, and envisaged in the early descriptions of Grace. As the language developed, several problems with this approach revealed themselves. Most of these problems arise because inheritance in Grace (as in most other languages) is transitive, so dialects implemented via inheritance would also be transitive.

Transitive dialects have some unwelcome behaviours: in particular, they mean that a module that inherits from (i.e., is written in) a dialect will have all of the dialect's methods available on the module object itself. For example, if a dialect were itself defined by a dialect (as in 6.2.3) then all the features of the dialect-defining dialect would also be included in any module that uses that dialect. Constructing a dialect that was also usable as a module would be impossible without contaminating the interface of all the dialect's users with its methods.

When using a dialect-defining dialect the client may wish to redefine methods provided by the outer dialect, such as control structures. This redefinition is not in itself a problem, but inheritance makes it so; any uses of these methods in the outer dialect will now refer to the redefined version, by the ordinary behaviour of self-calls under inheritance. The redefined method may have drastically different semantics or simply be unsuitable for use outside the client dialect (perhaps because it relies on initialisation performed by the client dialect). This opportunity to cause "spooky action at a distance" breakages is undesirable and could lead to errors that even the instructors defining the dialects find difficult to deal with.

As well as these issues of transitivity, we were intrigued by the idea of dialects scoped to smaller lexical ranges, which would not be easily possible in an inheritance-based system. While our design presented here does not include such dialects, we discuss future possibilities for permitting them in Section 6.3.4.

For these reasons we discounted the inheritance approach.

6.3.2 Delegation

We also considered supporting dialects by delegation. In particular, we considered translating a dialect statement into an import statement for the dialect module, along with a set of local (re)declarations of methods, one for each of the public methods of the dialect. Each of these local methods would forward to the corresponding method of the dialect. In this way, encapsulation of the dialect module is preserved; the effect is similar to unqualified imports in other languages. For example, given a dialect module containing:

```
method for(i)do(b) is public { ... }
method helper is confidential { ... }
```

and a module using it, the **dialect** keyword would be translated into:

```
import "someDialect" as secret
method for(a1)do(a2) is confidential {
    secret.for(a1) do(a2)
}
```

Only public dialect methods would get local forwarding methods, so local definitions of the dialect would be hidden. The local forwarding methods would be marked confidential, so that they would not be available to clients of the module. This approach would again make the dialect methods available as requests on **self** in the module scope.

Many of the issues with the inheritance approach do not arise here. The dialect object is used compositionally, but new methods are defined in the client module. The concept of exposing only public methods seemed attractive, but did not allow for a method to be exposed to a client written in the dialect without also exposing that method to all other code. While this exposure might seem unimportant, because dialects can examine code written in them they can enforce properties of how their methods are used; exposing methods to **import** clients may impact safety or use as capabilities.

There were two reasons why we rejected this design. The first is that it added another mechanism — delegation — into the language. Grace

already has three relationships between objects: simple references, inheritance, and lexical nesting: delegation would add a fourth.

The second reason is that the proposed semantics for delegation were very similar to the existing semantics for lexical nesting. Nesting makes outer objects' methods available to the objects nested inside them, but not to those objects' clients; those methods can be involved via implicit requests, or explicitly via **outer** (rather than **self**); self-requests in the outer object go to that object, not back to the original **self**. Given these similarities, it seemed simpler overall to extend nesting to encompass dialects, rather than introduce another separate mechanism.

6.3.3 Macros

The third option was to add macros, an additional language mechanism, allowing a dialect to define their own syntax and semantics from scratch. This is the approach taken in Racket [182], discussed in more detail in Section 6.5 below. Macros provide vastly more power than Grace's dialects: they may reorder or prevent the evaluation of arguments, introduce new bindings not mentioned in the source code, or transform the program in arbitrary ways.

For example, an SQL-style select macro in Racket could share an iteration variable across several expressions:

```
(for n (numbers)
  (where (< n 5))
  (select (* 3 n)))
```

In contrast, an equivalent form in Grace would make the sub-expressions (arguments to where and select clauses) blocks, with the value of the current number being provided as an argument to each block in turn:

```
for (numbers)
  where { n -> n < 5 }
  select { n -> n * 3 }
```

(C#'s lambdas have the same limitations as Grace's blocks, which is why C# has a built-in "macro" that re-writes its select statement into expression using multiple lambdas. [6]).

There are a number of reasons why we chose not to use macros to implement dialects in Grace. The first is that, without macros, dialects cannot introduce new syntactic forms; this means that code written in a dialect remains readable without knowledge of the dialect it is using. Thus, the parse of a Grace program does not depend on dialects, types, or operator definitions: syntactically, there are only method requests. A novice can understand that control passes to a given method on a given receiver, with the arguments written in the source, without needing to understand what that method does or how it does it. A macro-driven approach does not permit that, and the relevant macros must be understood to know what the effect of a piece of code is. We did not like the disconnect between source and semantics that macros give: what the code says, and what it does, become decoupled.

The second reason is that, without macros, Grace code that *implements* a dialect uses essentially the same language features as code that *uses* a dialect. Instructors do not have to learn a powerful new feature (macros) to write dialects, and do not have to understand a new feature to be able to debug code using dialects.

The final reason is that macros are an additional feature that have not (so far) been required in Grace. Because Grace aims to be minimal, and hopefully easy to learn and easy to use, we did not want to add complex and powerful additional features unless we could not find any simpler alternatives.

Mutating dialects

Dialects are able to check, but not modify, the AST of the module they apply to, because mutating the AST can lead to code written in the dialect not being syntactically meaningful without the dialect. This was the problem

we wanted to avoid by leaving out macros. At times, however, it may be worthwhile to allow limited modifications of the user's code, for example, to implement new syntactic sugar, or to make use of or pass on information that is discarded before run-time. A possible future extension of the system could allow these modifications — particularly decorations of the tree — in a structured way.

Such a "mutating dialect" could implement the single transformation to blocks required to support Grace's pattern-matching facilities, which was instead included in the compiler (described in Chapter 4). Similar modifications could be made experimentally using a mutating dialect and later incorporated into the language if they proved useful. Developing a structured way of laying out the transformations that would not lead to the issues that macro systems often face would be key to a workable implementation of this extension.

6.3.4 Local dialects

In the current design, dialects are chosen for the whole of a module. Because dialects rely on lexical scope, an obvious extension is to permit dialects to be applied to smaller "local" lexical scopes, perhaps for the extent of a block, an object constructor, or a class. For example, we could shift into the turtle graphics dialect in the middle of a for loop to draw the bars of a histogram.

```
def histogram = source.getData
for (histogram) do { datum ->
  dialect "turtle" do {
    forward(datum * 10)
    right(90); forward(10); right(90)
    forward(datum * 10)
    left(90); forward(10); left(90)
  }
}
```

We have not pursued this extension for several reasons. Local dialects do not seem to be necessary to support teaching — the primary purpose of Grace dialects. Local lexically scoped dialects may indeed be useful for domain specific languages used to support modelling, such as the relationship and finite state machine dialects described earlier (6.2.8 and 6.2.9), but for pedagogical purposes, students will typically write a single module in a single dialect.

The interaction of dialect scoping and ordinary lexical scoping needs careful thought. In many cases, code in the new dialect may well want to access identifiers from elsewhere in the module, but not from the outer dialect, while in other cases programmers may want to augment the existing dialect on a temporary basis.

Pragmatically, we can generally do without lexical dialects at the cost of extra modules. The above code example could be refactored so that the body of the for loop becomes a method in a separate module that is written in the turtle dialect; the loop would then request that method from the other module.

Nonetheless, there are reasonable use cases for nested dialects, and we took care at least not to preclude them in our design. It is unclear which approach to combining them to take, and requires further study. It may be possible to define an algebra of combining dialects that permits selecting the relevant behaviours for each particular use case, but we do not attempt to do so in this thesis.

6.3.5 Default methods

Some dialects, like the finite state machine dialect described in Section 6.2.9, will have simple methods that are very often defined in the modules using the dialect in order to provide access to a feature of the dialect itself. These methods usually forward to a method defined by the dialect. It might be helpful if a dialect could provide these methods itself and have them

automatically included in the interface of the modules that use it (unless disabled), or to have the end programmer be able to "opt in" to them individually or collectively.

Such methods could be marked by an annotation in the dialect and recognised by the compiler. They could also be included using a mutating-dialect feature as described above, but direct support would be significantly easier to use for most dialect authors.

In the case where the client module has no need to inherit from any other code, default methods can be provided through inheritance. The dialect can provide a class or other suitable inheritance source, and clients that wish to opt in to those methods can do so. Where the client already inherits from something else, however, this route is not possible in a language without multiple inheritance.

6.4 Implementation

The dialect system described here is implemented in Minigrace, our prototype compiler for the Grace language. The Minigrace distribution includes the case study dialects shown in Section 6.2.

We encountered certain implementation issues in integrating dialects into Minigrace, which we will discuss here, including solutions and possible alternatives.

6.4.1 Lexical scoping

Integrating our dialects system into a language implementation requires decoupling lexical lookup at the point of name resolution from true physical nesting. A name in an outer scope is no longer required to have been defined locally (nor predefined), and so both knowing it is present statically and accessing it dynamically require an extension to the language.

Minigrace tracks lexical scoping in a straightforward way, using a stack

of "scope" objects representing each level of nesting and containing details of each name local to that scope. Minigrace conceives of two varieties of scope: an object scope used for both **object** and **class** literals as well as module bodies, which may contain methods; and a local scope used for **method** and block bodies, which contain only local variables. This distinction is because resolution of the two is different: a local variable is always accessed either directly or through a closure environment, while object fields are accessed by method requests on their containing object. An identifier resolution pass of the compiler rewrites all unqualified method requests to incorporate explicit receivers: either **self** or some number of **outer**s uniquely identifying an object surrounding the request site.

To incorporate dialects we added a third variety of scope, a dialect scope. A dialect scope has two additional properties: it is always treated as the outermost scope (that is, identifier resolution never proceeds further); and accesses to the dialect scope are resolved to direct requests with a special identifier dialect as receiver (instead of to a chain of **outer**s as long as the level of lexical nesting). The dialect identifier cannot be accessed from user code because it conflicts with the **dialect** keyword, and so is only used for implicit dialect requests; it behaves like a local variable that cannot be typed by the user and through which method requests are treated as having lexical visibility. The dialect module object is bound to this identifier at run time, and all dialect methods are accessed through it. This special identifier is not strictly necessary, as **outer** at the top level of a module also refers to the dialect, but allows the produced AST and generated code to be simpler and permits dialect requests to be identified easily in other phases of the compiler (including in a dialect's checkers).

6.4.2 Executing checkers statically

Grace does not define a strict compile-time/run-time phase distinction, but Minigrace is broadly speaking a typical compiler with a notion of com-

pilation time producing a program that can be run subsequently. Dialect checkers must execute at this compile time, or at least before any of the main body of the program executes.

There are two obvious ways to perform this compile-time execution: one is to interpret the checker code, and the other is to dynamically load the dialect module and run the checker method in the ordinary way. We chose dynamic loading for Minigrace, as it reduced development effort by reusing behaviour provided by the operating system.

We originally implemented Minigrace on Linux-based systems, and attempted to have it function on other Unix-like platforms as well. On these systems it is possible to load any dynamic library using the `dlopen(3)` function [79] and then access symbols from it by name, while any unresolved symbols (such as standard library elements and the Minigrace ABI functions) can be found in the host executable. Thus we needed only to permit generating dynamic modules (shared objects or dynamic libraries), which are obtained primarily by flags given to the platform linker and underlying C compiler, and to support dynamically loading them later.

We added a `--dynamic-module` flag to the compiler instructing it to use the appropriate system-specific compilation flags to create a dynamic module instead of an executable or static object file. No significant changes were required to code generation to achieve this extension. We extended our mirrors module to include a loadDynamicModule method wrapping `dlopen`, loading the module's initialisation code, executing that code, and returning the resulting module object to the requestor. With these two extensions, executing the checker method requires only loading the dynamic module, requesting the checker method (passing in the AST of the client module), and being prepared to trap both method-not-found errors (when there is no checker) and checker failures.

This approach works for POSIX-compatible systems, but causes problems under some other operating systems, notably Windows. Cygwin [160] provides an environment supporting many tools including GCC in a

broadly Unix-like environment, and it is possible to use Minigrace within this environment. Cygwin uses the Microsoft Portable Executable format [126] for its executables, the native format of Windows, and this format does not permit dynamic resolution of arbitrary symbols in the same way that the Linux and BSD systems do, instead requiring these symbols to be specified in advance. As a result, dialects do not function under Cygwin. We believe it is possible to extend the compiler to allow dynamic modules on Windows, but did not do so as this platform was not our primary focus.

ECMAScript

Minigrace also targets ECMAScript to run in a web browser. To support dialects on this target we followed the same approach as our native code backend. As all modules are in essence dynamic on this platform, represented as ECMAScript functions returning Grace objects, special dynamic modules were not required, but we did implement the identical mirror interface for the platform, simply constructing the string of the function name and retrieving it from the global namespace. No special behaviour is required when generating a module to be used in this way. The code for executing checkers is exactly the same as in the native code target.

6.4.3 Side effects

Because a dialect module is loaded statically to run its checkers, any side effects from the body of the module will occur in the compiler. Sometimes these effects are benign, but on other occasions they will be unsuitable for execution statically (for example, opening a graphical window). A dialect can avoid side-effecting code in its main body by using the atModuleStart and atModuleEnd methods described in Section 6.1.3. This ability is one of the principal reasons we included atModuleStart in the design.

We also encountered issue on the ECMAScript frontend when code in the dialect body assumed it was running in the main page context, such as

by accessing DOM elements. These dialects caused errors when the client module was compiled in a background thread, as our user interface does to prevent lengthy pauses. Again, this processing can be deferred until tun time using atModuleStart and atModuleEnd.

6.4.4 Security concerns

Since dialect code executes statically, when the user may not be intending to execute their program (at least in Minigrace, which has a phase distinction), the dialect has the opportunity to execute code when the user is not expecting such behaviour. For this reason dialects must be trusted slightly more than other code. In the common case, the user will intend to run their program anyway, and so any security concern is no larger than without dialects. Where the program is not to be run immediately, however, or is to be run in a different context than it is to be compiled in, security concerns may require taking care around which dialects are permitted for use. Implementors should ensure that any such system either restricts the set of dialects available to use to those known to be trusted (including any dependencies they might have), or that statically-executed code is sandboxed.

6.5 Comparison with related work

In this section we contrast our approach with those taken in Racket particularly, and Scala, Ruby, Wyvern, and Cedalion.

6.5.1 Racket

Racket combines a Scheme-based language and an accompanying IDE designed for teaching. Racket includes "language levels" by way of advanced macro systems; Racket's language levels inspired Grace's desire for dialects to permit language subsets.

Racket supports variant languages in two ways: through basic Lisp-style macros, and by replacing or augmenting the parser. Both of these make different trade-offs than each other and than our design. Racket languages are strictly more powerful than our dialects, but this power comes at a cost.

The basic language levels of Racket are defined through macros; the user includes both ordinary functions and macro definitions into their program with their language declaration and these together provide the desired behaviours. Macros are strictly more powerful than our dialects; they can change the meaning of parts of the program, reorder, delay, or prevent execution, and introduce new code that was not present in the programmer's source. With this power comes a trade-off common to all macro systems: it is not possible to understand the flow of execution without understanding the macros in use, or making a lucky guess at them. An argument to a function may never be evaluated, may be evaluated multiple times, may be evaluated in a different scope, or may be transformed to mean something entirely different.

In this aspect Racket's behaviour is entirely unlike our design. It is always possible to parse and to follow the control flow in a Grace dialect, without any knowledge of the dialect itself. Our GrAPL dialect (Section 6.2.10) has the same parsing and execution rules as any other Grace program; a reader can follow the chain of method requests without any understanding of APL or the dialect, but at the same time we were forced to deviate semantically from the system we were impersonating because we could not prevent the evaluation of arguments. Our system trades away the flexibility of macros for consistent semantics: what appears to be a method request will always in fact be a method request, and what appears to be an argument to that request will always be evaluated at that point.

Racket also permits further deviations from the basic language by allowing the total replacement of the "reader" which turns input text into S-expressions. By replacing the reader a language can be defined entirely

from scratch, with no syntax in common with Racket itself. The Racket distribution includes an implementation of Algol-60, and a programmer needs only declare `#lang algol60` in order for the rest of the source to be treated as Algol.

Our system does not support this degree of departure from the underlying language; while a dialect may, by the combination of multi-part methods, operators, and pre-defined objects, present a language with a similar feel to another, programs written in that dialect must still conform to the overarching Grace syntax. This limitation is both a blessing and a curse: a programmer who already knows the other language may not be immediately at home, but working within a single consistent syntax allows integrating code from different paradigms and gradually moving from one to another.

Compared with Racket, the author of a Grace dialect does not need to embark upon full-scale metaprogramming (nor do they have the opportunity). To define a dialect without a checker, programmers define the methods, classes, variables, and types they want to have available to users exactly as they would in any other program. To provide dialect checkers, programmers need to understand the visitor pattern, or use the "dialect" dialect to write a largely declarative specification of a visitor (or examine the abstract syntax tree manually), entirely within Grace's standard syntax and semantics.

All Grace dialects have the same semantics as any other Grace program — method requests with arguments passed by value. Grace's parse depends only upon syntax, not on types or other implicit operations, so programmers can always determine the flow of execution from a program's surface syntax. By avoiding macros we avoid code that does not do what it appears to do: arguments are always evaluated before methods are requested, new bindings are never introduced implicitly, and parse or type errors can stem only from what was actually written in the input source code. A macro-based system cannot guarantee any of these points.

6.5.2 Scala

Scala [138, 167] includes several features supporting domain-specific languages. These features build on Scala's static type system (for example, implicit parameters are determined by type). Scala's treatment of syntax and semantics is determined by the static type information it has available.

By contrast, Grace programs have the same semantics with or without type definitions, and Grace's syntax, while flexible, does not admit ambiguities that need to be resolved by static types. Our dialect design is entirely type-independent: types are neither required nor used, and in fact dialects are able to define their own meaning for types.

Scala also includes powerful macro features [47, 18] integrating the compiler and runtime. These have similar power and problem to Racket macros, and contrast with our design in the same way.

6.5.3 Ruby

Domain-specific languages are built in Ruby in two ways: by leveraging the language's open classes, and through dynamic binding. Both of these differ sharply from our approach.

Through open classes, a Ruby programmer can add methods to existing objects and to new instances of existing classes. The programmer can extend the built-in Numeric class to let a user write "3.years.ago" to represent a time. Grace classes are not open in this way; furthermore, Grace is constructed around objects, not classes, so extending a class is a dubious concept to begin with. Nonetheless, it may be useful for a dialect to provide additional functionality on built-in objects like numbers and strings. Our design does not include provision for such extension; integrating some means to do so within the overall structure of Grace would be interesting future work.

The second strategy uses dynamic binding to execute a block of code in the context of an existing object, including access to any methods defined in that object. Different DSLs may be used at different points by evaluating

code in different contexts. As in our design, these do not extend the language syntax itself; as Ruby's syntax is less flexible than Grace's, the DSLs do not have quite the same freedom. Our design does not involve any dynamic metaprogramming; it is always possible to determine statically the exact scope a piece of code will run in, and the binding of an identifier. We believe this makes a program in an arbitrary dialect easier to understand than one using an arbitrary Ruby DSL.

6.5.4 Wyvern

Wyvern [131] supports nested domain-specific languages within Wyvern code [139]. A DSL block is identified by indentation or certain paired quoting characters.

Wyvern's approach is entirely type-directed. Each type can have at most one DSL associated with it, and anywhere that a DSL is used where that type was expected the parser and translator associated with that language will be invoked.

As Grace code is not required to be statically-typed, such a type-directed approach is unsuitable. Because each Wyvern DSL defines its own grammar entirely, the syntax can depart radically from the surrounding language, and not be comprehensible to a reader who is unfamiliar with the DSL in use, or who does not know which type was expected in this context. In our design we sought to avoid such situations by confining dialects to working within the underlying Grace syntax.

Wyvern allows embedding multiple language variants within a single file, which our system does not permit, although we discuss possibilities and associated challenges in Section 6.4.1.

6.5.5 Cedalion

Cedalion [112] is a "language-oriented programming" language: the idea is to define a new domain-specific language for each problem domain

spanned by a program. All Cedalion languages may interoperate within a single program, because they all share the same host language. In this respect they resemble Grace dialects: within the same fundamental semantics, many different variants may coexist simultaneously.

To support many languages at once, however, Cedalion uses a "projectional editor" [190] to edit code. Rather than editing raw program text, as in Grace and most other languages, the programmer instead edits the abstract syntax tree directly. In fact, a Cedalion language defines a display grammar for that syntax tree, rather than a parsing grammar for text or semantics for methods. In effect, all code in all language variants uses exactly the same language, but the programmer looks at some parts of it differently than other parts.

This approach contrasts with ours, where the same surface syntax persists in every dialect, but where the syntax itself is quite flexible. A reader of one Cedalion language has no more benefit in understanding another than an outsider, while an author in the language needs not conform to any other overriding syntax. In both cases, Cedalion takes the opposite position to our design. In particular, we do not believe that mandating a special editor disconnected from the underlying syntax is appropriate for either an educational or a general-purpose language, as it inhibits the exchange and discussion of code in other media, such as textbooks and email.

6.5.6 Haskell

Haskell domain-specific languages use typeclasses and **do** notation to embed themselves in the language. In both cases, static type information determines the semantics of the code and which functions to execute.

A semantics relying on static types is undesirable for a gradually-typed language like Grace. Haskell's available syntax is more constrained than Grace's dialects, and the scope for extension is more constrained by what already exists in the language. A Haskell DSL will have difficulty relying

```
grammar org.xtext.tortoiseshell.TortoiseShell
with org.eclipse.xtext.xbase.Xbase
import "http://www.eclipse.org/xtext/xbase/Xbase"
generate tortoiseShell "http://www.xtext.org/tortoiseshell/
    TortoiseShell"
Program :
 body=Body
 subPrograms+=SubProgram*;
SubProgram:
 'sub' name=ValidID
 (parameters += FullJvmFormalParameter)*
 body=Body;
Body returns XBlockExpression:
 {XBlockExpression}
 'begin'
 (expressions+=XExpressionInsideBlock ';'?)*
 'end';
Executable:
 Program | SubProgram;
```

Figure 6.10: Logo grammar definition for Xtext, adapted from the Xtext web site [42].

on some subset of the functions or operators from a Haskell type class, while Grace dialects may define exactly the methods and operators they need.

6.5.7 Xtext

Xtext [48] is an external domain-specific language system. In Xtext, a language author defines both a grammar for the language and an "inferrer", which is essentially a code generator. In this way syntax and semantics are separated. This approach is common to external DSL systems, so we present a basic example of how such a language works, adapted from one on the Xtext web site [42]. This DSL is for a "turtle graphics" language, like Logo.

```
class TortoiseShellJvmModelInferrer extends AbstractModelInferrer {
  public static val INFERRED_CLASS_NAME = 'MyTortoiseProgram'
  @Inject extension JvmTypesBuilder
  def dispatch void infer(Program program,
                          IJvmDeclaredTypeAcceptor acceptor,
                          boolean isPreIndexingPhase) {
    acceptor.accept(program.toClass(INFERRED_CLASS_NAME)).
      initializeLater [
      superTypes += program.newTypeRef(typeof(Tortoise))
      if(program.body != null)
        members += program.toMethod("main",
          program.newTypeRef(Void::TYPE)) [
          body = program.body
        ]
      for(subProgram: program.subPrograms)
        members += subProgram.toMethod(subProgram.name,
          program.newTypeRef(Void::TYPE)) [
          for(functionParameter: subProgram.parameters)
            parameters += functionParameter.toParameter(
              functionParameter.name, functionParameter.parameterType)
          body = subProgram.body
        ]
    ]
  }
}
```

Figure 6.11: Xtext "inferrer" for Logo, adapted from the Xtext web site [42]. This generates a new class containing code generated from the user's code, extending an existing turtle graphics class.

For Xtext, we first define the grammar as shown in Figure 6.10. This is a very simple grammar, delegating most of the language details to the Xbase language (a basic language definition included with Xtext). The grammar defines the "sub", "begin", and "end" keywords, which bound programs and subprograms. Each program body consists of Xbase statements.

Xtext then requires an "inferrer" to translate the language defined in the grammar to an implementation language, typically Java. The Logo sample "inferrer", shown in Figure 6.11, translates each Logo program to a Java class which subclasses an existing turtle graphics class to gain access to the various drawing methods and fields. Logo commands are translated to self-calls on the class.

Comparing the Xtext Logo to our Grace Logo dialect from in Section 6.2.1, the first clear difference is that in our system we define the dialect's behaviour and syntax together. The dialect is much shorter than the Xtext version, with the same syntax and semantic model as the rest of the language. By piggy-backing on the existing Grace language, and taking advantage of its flexibility, we avoid defining separate grammars and translations: by defining both in one place, the relationship between the two is immediately clear, and the author does not need to understand how to write a grammar or any additional rules.

Of course, in doing so our design trades off overall flexibility to gain brevity and simplicity. Unlike our dialects, Xtext can define a wholly new grammar, defining a language resembling the original Logo as closely as desired. Xtext languages can define entirely new syntax, or mix different syntaxes together. We consider the trade-off in our system worthwhile. A dialect is simpler to implement than a corresponding Xtext language, and offers an obvious progression into the full, unrestricted Grace language.

6.6 Conclusion

We have described how a novel combination of language features — lexically nested objects, syntax for blocks and multipart methods, optional typing, and pluggable checkers — supports dialects. Because our dialects are based on these standard language features, programmers can write dialects much as they write any other Grace program — by defining objects and methods — without having to learn additional macro systems, define lexers, parsers, and semantic rules, or use metaprogramming to modify class definitions on the fly. To illustrate the power of this dialect mechanism, we have presented several case studies of dialects of varying complexity.

Dialects allow multiple sublanguages to be defined within one overarching language, either to restrict or to extend what is available to the programmer. Different parts of the same program may use different dialects, including the dialects themselves, without affecting other code. Dialects are similar to modules, except that they *surround* the code written using the dialect, rather than being included by it. Dialects can both add language features by defining methods, and remove language features, or enhance error reporting, using checkers. Dialects enable Grace to support both multiple teaching languages for novices and domain specific languages for advanced students.

Dialects by lexical scope mean that the same semantic model applies to all programs, with or without a dialect. With careful choice of the underlying language syntax many of the specialised features supported by advanced DSL systems, or tailored teaching languages, may also be achieved in our system, without any additional language features.

Chapter 7

Tiled Grace[1]

Visual programming environments like Scratch [162] present a program as a combination of nested "jigsaw piece" tiles manipulated by drag-and-drop, and have been used successfully with new programmers [53, 17, 124, 115]. These environments present a limited language with a restricted expressive domain, meaning that eventually programmers must move on to a "real" textual programming language and, in many cases, learn to program over again [158, 146]. Tiled Grace is a programming environment for Grace bridging these two worlds: programs may be edited using a drag-and-drop tile interface, but with tiles that map exactly to the concrete text syntax. In Tiled Grace, users can switch to a conventional textual view at any time, and can edit that text before switching back to the tile view, making the correspondence between tiles and source code clear.

This chapter is structured as follows. In the next section we describe Tiled Grace and explain the design choices we made in it. Section 7.2 motivates the existence of Tiled Grace. Section 7.3 describes the additional functionalities we implemented on top of the base system. Section 7.4 describes the user experiment we ran using Tiled Grace , and Section 7.5 the results we obtained. Section 7.6 positions Tiled Grace among related work, while Section 7.7 discusses future work. Section 7.8 concludes.

[1]This chapter expands upon papers published in VISSOFT 2013 [72] and 2014 [73].

7.1 Tiled Grace

Tiled Grace presents an editing environment for Grace programs based on drag-and-drop *tiles*. A tile represents a single syntactic unit in the program, such as a string literal, variable assignment, or method request. For example, tiles for a string "Hello!" and variable "x" are depicted as:

Some tiles, like the string tile above, have text input fields for the user to enter a value.

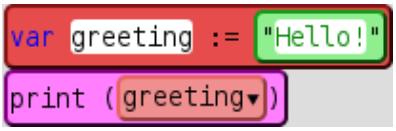
Some tiles have *hole*s in them, where another tile may be placed. For example, a variable assignment tile has two holes: one for the variable to be assigned to, and one for the value to be assigned:

The holes are the empty grey rounded-rectangular areas. Other tiles with holes include operators such as + and ⋆, method requests, and print statements. The user can place a tile inside a hole to build up their program.

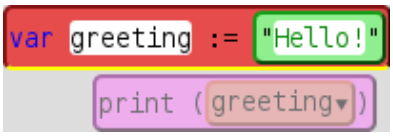
To assign the string "Hello!" to the variable "x", the user combines these three tiles:

To put a tile into a hole, the user can drag the tile they want to use over the hole, which will be highlighted when they are over it, and then drop it there. The hole will expand to fit its new contents.

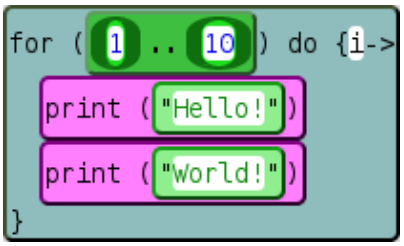
Tiles can be connected together in sequence as well. To create a variable and print its value, a **var** tile and a print tile can be joined together.



The user can join tiles together in this way by dragging so that the top of the tile they want to join on is near to the bottom of the tile they want to join onto, and dropping the tile there. The tile being joined onto will be highlighted:



Some holes can hold multiple tiles, such as the holes in the body of a loop. The first tile can simply be dropped in as for any other hole, and then other tiles can be joined onto the bottom of it. The following code prints "Hello!" and "World!" ten times each in alternation:



A complete program and its output is shown in the Tiled Grace interface in Figure 7.1. The interface is divided into three main areas: a large workspace area on the left, a toolbox of available tiles, and text and graphical output areas on the right.

Tiles may be dropped anywhere in the workspace pane, and the user can construct different sub-programs in different parts of the area. Different categories of tile can be selected from a pop-up menu that appears when using the toolbox. At the bottom of Figure 7.1 the dialect selector, run button, and other interface controls are displayed.

Different kinds of tile are shown in different colours. Closely related concepts, such as variable declaration, reference, and assignment, have similar colouring.[2]

[2]In the present prototype, these colours are simply assigned in sequence around the colour wheel.

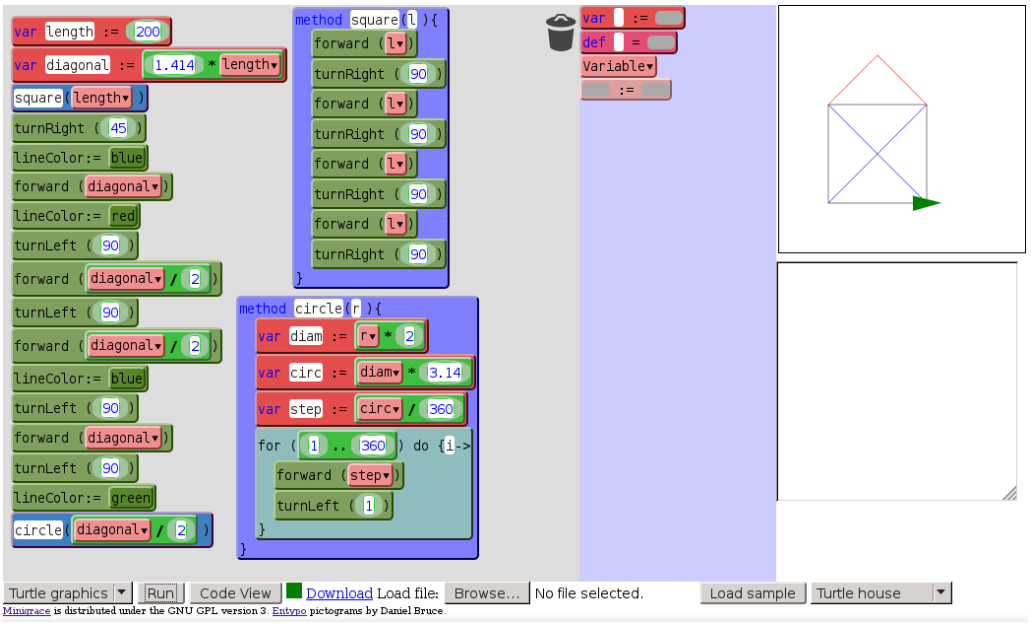


Figure 7.1: Tiled Grace editing a small program in the "turtle graphics" dialect.

The feel of Tiled Grace is similar to Scratch [162], which inspired this work. Tiled Grace differs in that it is backed by a genuine textual language: the tiles themselves correspond to the syntax of the Grace language, in order to support students when they eventually move out of Tiled Grace and begin writing textual programs. Tiled Grace goes a step further still: because the tiled representation maps exactly onto the textual representation the user can switch to a standard syntax-highlighted textual view at any time.

The transition from tiled to textual view is shown through a smooth animation. Each tile and block of code has a continuous visual identity throughout the transition.

First the tiles fade out to blocks of the corresponding textual code, then the blocks glide into place in a linear textual program, and finally the display switches to editable text. The entire transition takes just under two seconds. When the user chooses to switch back to tiles, the same behaviour

occurs in reverse.

Figure 7.2 shows this transition in progress: while editing the same program as shown in Figure 7.1, the user has switched to a textual view. First the tiles fade out to blocks of the corresponding syntax-highlighted textual code, while remaining in the same physical location (frame (b)). The code blocks then glide into place (frame (c)), finishing in a linear textual ordering. Finally, the tiles become fully editable ordinary text, as shown in frame (d). In this way, the relationship between tiles and the corresponding part of the textual program is clearly visible.

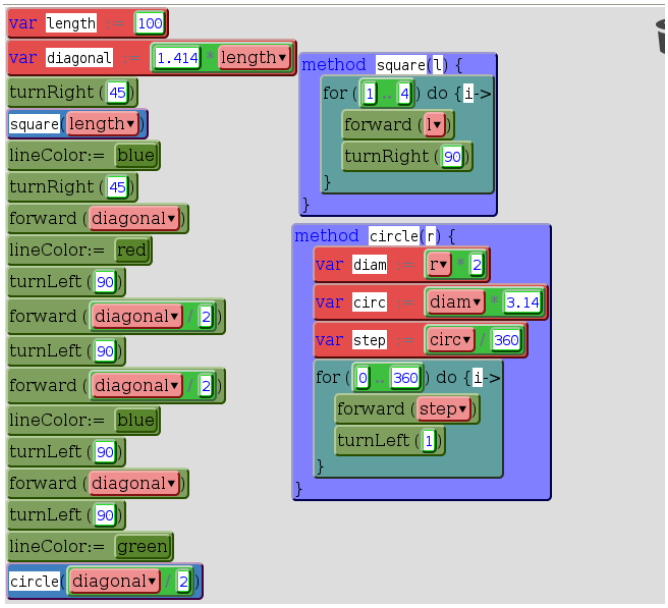
Each separate group of connected tiles is regarded as an independent part of the program. The ordering between them in the textual display is arbitrary, but consistent across the lifetime of the program. The displayed text is editable if the user wishes: they may change the source code, including adding and removing whole lines or blocks, and then transition back to the tiled view.

7.1.1 Implementation

Tiled Grace is built on Minigrace's ECMAScript backend (Chapter 8), with a new front-end interface. Tiled Grace runs in a web browser without installation, and can be accessed at `http://michael.homer.nz/minigrace/tiled/`. Tiled Grace runs in at least recent versions of Firefox, Chrome, and Internet Explorer[3] at the time of writing.

Tiled Grace presents a user interface where tiles can be dragged around, and represents the program as a tree of those tiles (mapping closely onto the Document Object Model [98] tree of web browsers). The structure of this tree differs from the abstract syntax tree of the language in several ways; notably, empty holes are possible in Tiled Grace and can be represented fully in the tree, but they are not part of the concrete syntax. Tiled Grace also presents requests for dialect methods differently than for user methods,

[3]Because of technical limitations, some features are restricted in Internet Explorer.



(a)

(b)

Figure 7.2: Frames of the animated transition from tiled to textual view. Transitioning from textual to tiled view shows the same intermediate states in reverse. The transition from tiles to code, and the movement of code, is smoothly animated. (continues on facing page)

```
var length := 100
var diagonal := 1.414 * length
turnRight (45)
square(length)
lineColor:= blue
turnRight (45)
forward (diagonal)
lineColor:= red
turnLeft (90)
forward (diagonal / 2)
turnLeft (90)
forward (diagonal / 2)
lineColor:= blue
turnLeft (90)
forward (diagonal)
turnLeft (90)
lineColor:= green
circle(diagonal / 2)

method square(l) {
    for (1 .. 4) do {i->
        forward (l)
        turnRight (90)
    }
}
method circle(r) {
    var diam := r * 2
    var circ := diam * 3.14
    var step := circ / 360
    for (0 .. 360) do {i->
        forward (step)
        turnLeft (1)
    }
}
```

(c)

```
1  dialect "logo"
2  var length := 100
3  var diagonal := 1.414 * length
4  turnRight (45)
5  square(length)
6  lineColor:= blue
7  turnRight (45)
8  forward (diagonal)
9  lineColor:= red
10 turnLeft (90)
11 forward (diagonal / 2)
12 turnLeft (90)
13 forward (diagonal / 2)
14 lineColor:= blue
15 turnLeft (90)
16 forward (diagonal)
17 turnLeft (90)
18 lineColor:= green
19 circle(diagonal / 2)
20
21 method circle(r) {
22     var diam := r * 2
23     var circ := diam * 3.14
24     var step := circ / 360
25     for (0 .. 360) do {i ->
26         forward (step)
27         turnLeft (1)
28     }
29 }
30
31 method square(l) {
```

(d)

Figure 7.2: continued.

as specialised tiles are constructed for dialect methods and listed in the toolbox directly.

To execute the code, Tiled Grace generates textual Grace code from the program tree, which is then compiled by Minigrace into ECMAScript, which is then executed.

7.2 Motivation

Why build Tiled Grace when Scratch, Alice, and similar systems already exist? Our design goal for Tiled Grace is to avoid some pitfalls and problems that have been encountered with these existing systems. Pedagogical evaluations of the system are future work, but we performed a user evaluation of the usability of our tool, described in Section 7.4. In this section we describe the issues with other systems that motivated the different design choices we made in Tiled Grace .

One issue that has been encountered in introductory visual languages is that learners do not consider them "real" programming languages [103, 104]. Lewis *et al* found that more students rated a picture of random green-on-black symbols from the film *The Matrix* as "definitely" or "somewhat like" programming than an image of the Lego Mindstorms programming environment (a colourful drag-and-drop system), even though those students had been learning Scratch.

Similarly, Powers, Ecott, and Hirshfield found that students learning Alice and a textual language in the same course frequently felt that Alice was not a "real" language [158]. Students who struggled with the textual-language part of the course felt that what they had been doing in Alice "didn't count" or was "too easy", that textual code was "real programming" and were inclined towards believing that they were not actually capable of programming; this inclination is harmful in itself.

In Tiled Grace we aim to avoid or ameliorate this perception by presenting the textual and visual representation of code coequally, and clearly

the same language on both sides. The textual-tiled combination was our original grounding conception for Tiled Grace .

Another reported problem with moving on from visual to textual languages [158], and moving between languages early in learning in general, is that learners find it difficult to connect analogous concepts in one language to the other. Our animated transition between visual and textual representation aims to demonstrate the exact parallel between the two.

In particular, it is known from both educational psychology in general and computer science education specifically that transitioning between languages early in learning is unhelpful [142]. A course structure predicated on such a transition will likely run into trouble, but introductory tertiary courses in Scratch and Alice move on to "real" languages early, often within the first course, as programs become too complex for such languages. Permitting both views should avoid this transition, so that learners can begin in (Tiled) Grace, move gradually into (textual) Grace, and continue in that full-strength language as long as required.

One issue with language transitions is that they are essentially "one-way" processes: the learner must apply what they know about the earlier language to the later, but movement in the other direction is restricted. Tiled Grace has a deliberately permeable barrier: a user can use the visual language, the textual language, and the visual language again, even within the same program if desired. Allowing movement in both directions necessitates some trade-offs (particularly that the programmer can only switch views when there are no static errors in their program), but we consider it appropriate to the goal of the language.

The initial conception of Tiled Grace came after working with Scratch. We had previously used Scratch as part of outreach programmes from the university and in the early phases of our work on Grace, and were invited to teach Scratch to a class at a local intermediate school (pupils aged 10-12). In all of these cases we noted that users were very (and readily) engaged with the tool, much more so than we had observed with new programmers

using Java or Python. We also knew from experience, however, that as we had come to know the system better we had found the drag-and-drop interface of Scratch increasingly tiresome to use and felt its restrictions more and more, particularly as we had known how to program when we began. We wanted a system that combined the engagement of Scratch with the power of a "real" language and the ability to move on smoothly to more flexible programming styles. With this in mind we conceived the idea of switching between two views of the same code, which was the basis of Tiled Grace. Although Tiled Grace is somewhat weaker both as a textual interface and as a visual interface than special-purpose programming systems, the combination of the two is powerful and we wished to experiment with it.

Another key motivation was our dialect system, which has no real parallel in the other visual language systems. Scratch, Greenfoot, and Alice all expose different degrees of complexity appropriate to different levels of development, but each only exposes a single level. Advanced users of Scratch find the limitations frustrating (as we did), but permitting more flexibility for early learners can hinder them. A key decision in the design of Tiled Grace was that it would support dialects from the ground up, so that learners could move into less restrictive language variants as they went, while staying in the same language and same interface. Again, that integration involves some trade off, but we consider it worthwhile to allow a user to remain within the same fundamental language as long as possible.

7.3 Functionality

On top of the basic functioning of Tiled Grace described in Section 7.1, the tiled view and its duality with the textual representation offer new possibilities for system behaviour. In this section we describe the design and implementation of functionality for handling errors, showing information about definitions, dealing with language variants, and type checking.

7.3.1 Handling errors

The very duality of view Tiled Grace is built around creates new opportunities for error. As well as the common errors of textual editing that are possible in the text view, tiles permit other forms of error that are unlikely to occur in text. The interface between the two forms must prevent errors spreading from one to the other.

While the tiled view prevents most syntax errors, the user may still omit to fill in required components — for example, not specifying a variable name or leaving the hole on one side of an operator empty — or invalidate the program in other ways by moving a reference to a variable outside its scope or filling in an unsuitable value. In each case, the textual representation of the program would be incorrect or misleading. To combat that, the user can only switch views when the program is valid: when there are unfilled holes or other errors when the user attempts to change view, the error sites will be highlighted and the view unchanged. A graphical indicator shows whether the program is currently valid at all times; when the indicator is red the user may hover over it to highlight all existing errors, which are labelled with their cause (for example, an empty hole may have the message "Something needs to go in here"). These error sites are shown by desaturating all of the code area except the error sites, and overlaying an associated error message at the site. An example is shown in Figure 7.3 where the user has hovered over the red square indicating an error, which was green for the unmodified version of the program in Figure 7.1.

In the text view, as in any textual editor, the user is unrestricted in the kinds of error they can produce. The code is continually compiled in the background and errors marked where they occur, with the user able to access a standard compiler error by hovering their mouse pointer over the error marker on the offending line. If the user tries to switch to the tiled view while the program does not compile, they will be presented with the error and asked whether they want to revert to the last-known-good version.

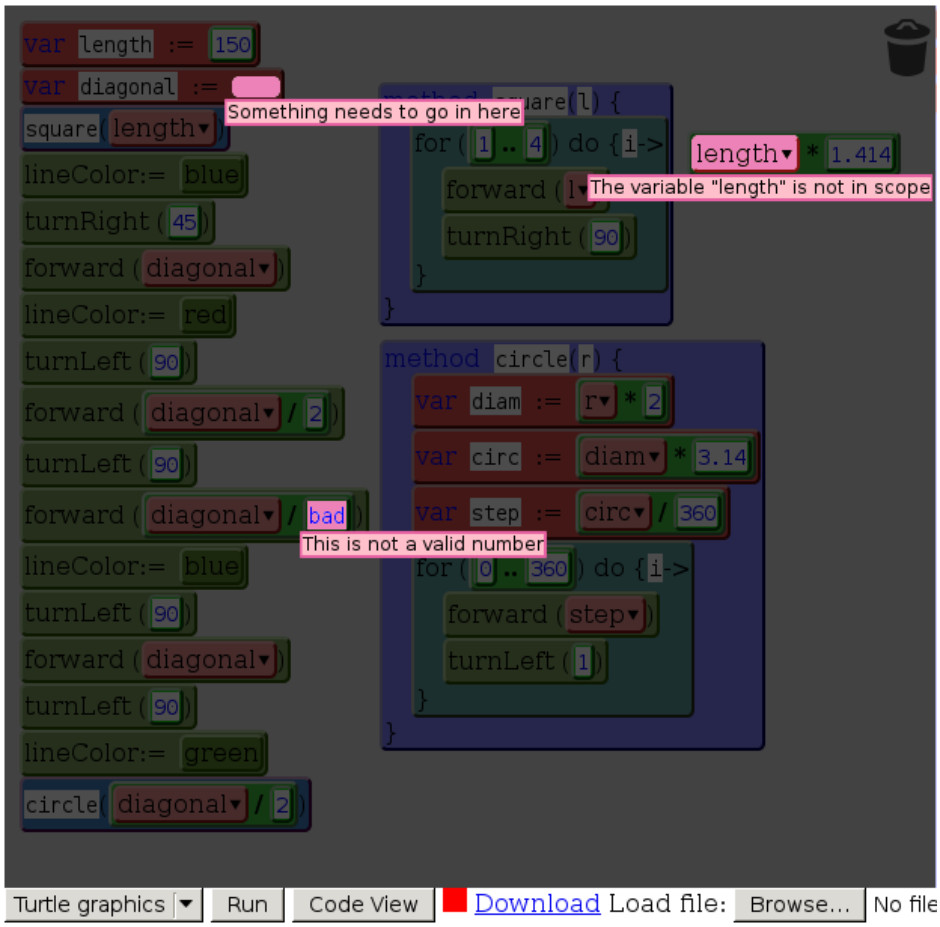


Figure 7.3: Errors displayed in a modified version of the turtle graphics example.

Design alternatives

Alternative indications of error sites are possible; as well as desaturation, we have experimented with overlaid arrows, as well as borders, animations, and combinations of these. We found desaturation to be the clearest indicator of those we tried in experimental implementations, although we intend to investigate combining it with small animations.

By ensuring the program is valid when changing views, errors are not propagated any further than necessary and no additional long-term errors are created by changing views. This was a difficult choice, as some errors would be easier to solve if the user could look at the program in two ways.

While some erroneous code would be difficult to represent on one side or the other — an empty hole would presumably be nothing at all in the textual view, while basic syntax errors in the textual view will not have corresponding tiles — other kinds of error, such as variables used out of scope and type errors, affect both views in much the same way and have straightforward representations in each. Scoping errors in particular are shown very clearly with tiles. We discuss future possibilities in Section 7.7. For the moment we have chosen to ensure that the program is always valid immediately after a transition.

7.3.2 Overlays

As well as visualising the code itself as tiles, Tiled Grace can visualise relationships between parts of the code (see Figure 7.4). When a user hovers their mouse pointer over a variable reference, the code view will be overlaid with a line from that reference to the variable's definition site, as well as to any assignments to the variable in scope. Hovering over a variable declaration produces an overlay that indicates all the uses of that variable in scope. Similarly, hovering over a method definition identifies any requests of that method in the program, while hovering over a request (including a request of a method that came from the dialect) highlights the definition

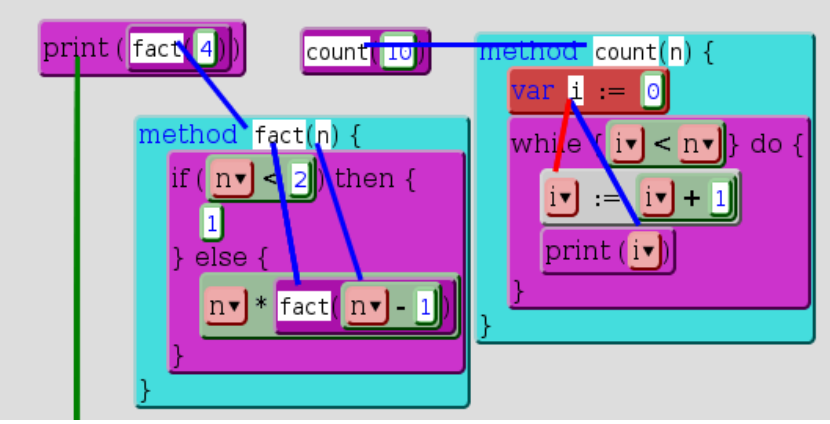


Figure 7.4: Composite image of multiple overlays at once. All blue lines run between a use and the definition of a variable or method. The red line indicates a reassignment of a variable; the green indicates a method from the dialect, pointing at the dialect selector. Only one of these overlays can be shown at one time.

of the method. In this way the programmer can easily read the program in execution order, rather than top-to-bottom, which has been found to be helpful for novices [86]. If applicable, multiple overlays may appear at once. These overlays are similar to those found in spreadsheets [61] to illustrate the dependencies of a formula.

In the textual view the user may hover their mouse pointer over a variable or method to see an overlay showing the definition or use sites. In this view, the overlay is very like the similar overlay in DrRacket [182].

7.3.3 Dialects

Grace dialects can extend the methods available to the programmer (as in the turtle graphics dialect in Figure 7.1) as well as restrict features of the language or create new errors, as described in Chapter 6. Tiled Grace supports both of these components.

When the user selects a dialect to use, Tiled Grace creates tiles for all of the provided methods, based on a description of the dialect. This

description can be automatically generated from the dialect itself, but many dialects will benefit from manual annotations to reflect the intention of the dialect better. A simple example of when this may be desirable is when building control structures: the while()do method takes two blocks as arguments, one as the condition (which is a block because it may be executed more than once), and one as the body of the loop. Although both parameters are blocks, the intention is different. The body is expected to contain many statements, while the condition will likely be a single brief expression. The dialect description can make this intention clear, as well as expressing other limitations.

The default control structures in Tiled Grace show this distinction: the condition of a loop is (while still shown as a block) a one-line, single-expression field, while the body is a multi-line block that accepts many statements. Additional control structures defined in other dialects can have the same (or different) behaviour if they wish.

Our support of dialects is an important generalisation of Blockly's (Section 2.5.2) ability to choose an extended sub-language to use. Because these dialects persist in textual form, and even originate in it, the user retains the ability to use and understand them even outside Tiled Grace itself. Our dialects may also define and report new classes of error, shown in the same way as all other errors.

Defining dialects

Dialects are described for Tiled Grace's purposes by ECMAScript objects giving the set of operator tiles and methods available in the dialect. A method definition can be as simple as a name, but various metadata can also be included. The definition of the while()do method in the standard dialect looks like this:

```
"while()do": {
    name: "while()do",
    parts: [
        {
            name: 'while',
            args: [
                {type: 'Block', returns: 'Boolean', multiline: false,
                    description: "Condition."}
            ],
        },
        {
            name: 'do',
            args: [
                {type: 'Block', returns: 'Any', multiline: true,
                    description: "Something to do."}
            ],
        }
    ],
    category: "Control",
    returns: "Done",
    description: "While a condition is true, do something.",
    multiline: true,
}
```

This method's tile will be located under "Control" in the toolbox and returns Done, the no-result type. Attempting to place this tile in a hole that expects a value of some other type will trigger an error. A tooltip is defined to describe to the user what the tile does, and similar tooltips given for the two argument holes the tile includes.

The argument lists are the most interesting part: as alluded to earlier, control structures may wish to indicate how their holes should be rendered. The condition parameter is defined as a Block returning a Boolean; attempting to place a non-boolean expression in the hole will cause an error. The

body is also a Block, this time returning Any, the top type. The interesting difference is in the multiline field: the condition is not a multiline block, and so is shown within a single line and accepts a single tile, while the body is defined as multiline and so will accept multiple tiles and display vertically.

The full variety of available features in the dialect description for Tiled Grace is explored in the sample dialects included with the implementation.

7.3.4 Type checking

Type checking in a drag-and-drop interface raises additional obstacles versus conventional static type checking. While we can run a standard algorithm over the code and display the results, given the way the user interacts with the system we would prefer to show errors at the time they are made, or even to prevent their occurrence altogether.

We chose to use a variant on our overlay approach to report errors as the user tries to perform the action that would cause an error, while also preventing the user from doing so. Any hole, including both those within built-in tiles and those from dialects, can be annotated with the types it will accept. Any tile can be similarly annotated with the type of the object it represents. As Tiled Grace variable declarations do not include static type annotations, all type annotations are currently built in (either to the tool directly or as part of dialect definitions), but the underlying system would need no change to extend to other types were they added.

For example, a string tile is annotated with the type "String", and both holes in a + tile are annotated as accepting only "Number". When the programmer tries to place one into the other, as in Figure 7.5, the hole is marked in pink and an error message displayed nearby: the user will not be able to drop the tile into the hole. In this way, the type error is prevented from being introduced into the program in the first place, removing the need for a typechecking pass. Nonetheless, some classes of type error could be introduced within textual code and not be caught there, and then make

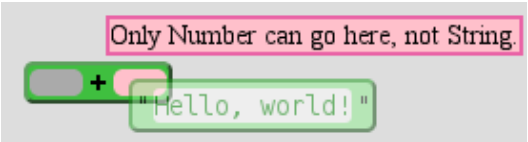


Figure 7.5: The display of a simple type error the user is making, where they try to place a string tile somewhere that only numbers are permitted.

it through the transition to the tiled view. As a result, the error-handling step described in Section 7.3.1 also checks that all holes and their contents are well-typed, and any errors that are found are reported in the same way.

Discussion

Our type-checking system provides some guidance to programmers, and does avoid many kinds of type error. Preventing the introduction of errors, and presenting a simple explanation at the point of the attempt, stops the user digging themselves into a deeper hole before realising that something is wrong. Our system does not prevent the user exploring the system and testing out what can and cannot be done.

We thought it would be helpful to indicate to users the appropriate placement of tiles before they move them. Scratch partially achieves this effect through its "jigsaw puzzle" pieces: holes and tiles of different types have different physical shapes, so a boolean constant or expression will not fit into a numeric expression. While immediately understandable, the approach has flaws, notably that there is a limited range of sensible shapes that can be readily distinguished and consequential limit on the number of types that can be in the system. As well, "multi-type" holes are very difficult: in Scratch it is not possible to have an array of booleans, only of its combined string-number type. These constrictions make this approach problematic to implement in Grace, as it is a language with many types, and several places that can hold variables of any type (for example, variable declarations and equality tests).

We considered colour-coding types, such that our any-type holes would

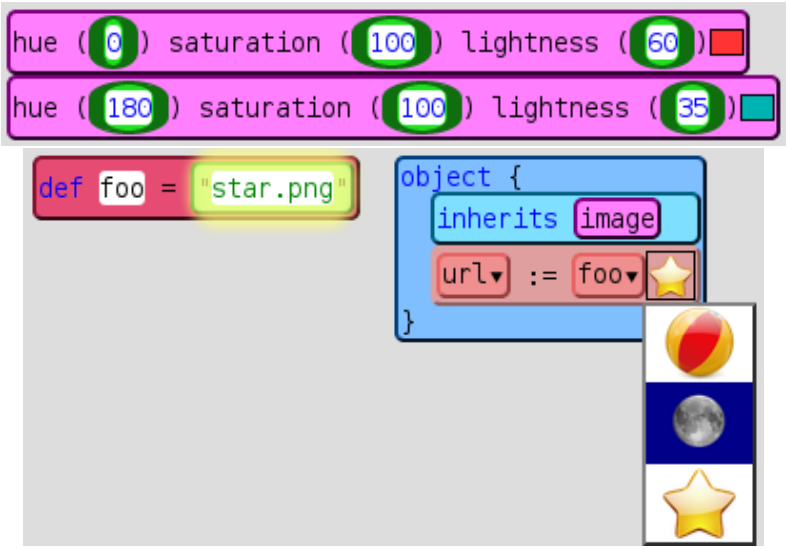


Figure 7.6: Two hints showing a colour selector (top) and an image preview (bottom). The menu allows changing between known images, and will here update the remote **def** foo.

be a neutral colour, while strings, numbers, booleans, other objects, and dialect definitions would have their own colours which could be matched on both tile and hole. The tile colours already approximate such a system, so it was attractive. Tiled Grace allows user-defined objects, however, which have novel types of their own determined by the methods within them. It became clear that colour-coding based on types would not be practical, particularly when considering how subtyping relationships could be represented.

7.3.5 Hints

One advantage of a non-textual display of code, such as our tiled view, is the flexibility to render additional "out-of-band" information within the program display for the benefit of the programmer. In Tiled Grace we call these "hints" and a dialect may define them for its tiles. The dialect we built for graphical programs includes two hints, both showing a graphical

representation of some text the programmer wrote.

The first hint is on a tile for defining colours using the hue-saturation-lightness scale. A small block of colour appears on the tile, updated in real time as the programmer edits the values or definitions leading to them. The second involves images: the dialect provides the ability to construct "image" objects, which render an image at run time. The image used is determined by the name assigned to the `url` field of the object. The hint catches these assignments, shows a preview of the image referred to, and offers a drop-down menu for the user to select from known images. If the user chooses a new image, the code is updated, even if the original definition site is remote from the code at hand. Both of these are depicted in Figure 7.6.

Implementation

By nature, these hints are tied closely to the implementation of Tiled Grace. A hint is defined in ECMAScript within the dialect definition: within the description of a tile a function can be defined which will be run each time source code is regenerated (principally, after any change to the code). This function will be passed a reference to the tile concerned. The function can inspect the code around the tile and display relevant information, as well as accessing Tiled Grace's API functions, such as that to find the definition site of a variable referred to in code. While the dialect implementor must know the internal structure of Tiled Grace and the browser environment in some detail in order to add a hint, the end user receives additional help with little or no effort on their part.

7.4 Experiment

We ran a user experiment trialling Tiled Grace with 34 participants, primarily students enrolled in undergraduate courses in the School of Engineering

and Computer Science at Victoria University of Wellington. The experiment took place in March–April 2014. Participants were asked to use Tiled Grace to write, modify, correct, and describe programs, while we measured their use of different features of the system. Participants also completed questionnaires about themselves and their use of the system. This experiment was approved by the Human Ethics Committee of Victoria University of Wellington. Our ethics application and the approval document is given in Appendix D.

7.4.1 Research questions

Our experimental design was guided by key questions we wished to answer (as well as by practical considerations, particularly timing).

Tiled Grace was motivated by the idea of presenting both tiled and textual views of code coequally. One question was therefore: **Do users find the ability to transition between views useful?**. A particular novelty of our approach is that we transition code visually in such a way that each piece of code has a continuous visual identity throughout, so the user knows which text corresponds to which tile. To that end we asked: **Do users appreciate explicitly animating the conversion to and from text?**

We designed and implemented a novel error reporting system for Tiled Grace that has no analogue in the existing drag-and-drop languages. The functioning of this system was guided by intuition and discussion, but we wished to measure it as well: **Do users find the error reporting of Tiled Grace useful?**

Finally, given that Tiled Grace is intended to be used as an introduction for novice programmers, one of the key issues is simply engagement: a tool that users do not enjoy will not be used. Visual interfaces are generally assumed in the literature to be engaging, and that assumption conformed with our experience, but we needed to measure that for Tiled Grace as well: **What degree of engagement do users exhibit with Tiled Grace?**

7.4.2 Participation

Participants were recruited by announcements in lectures, forum posts, word of mouth, and direct recruitement, and invited to make an appointment to perform the experiment. These are the standard techniques used for experiments in the department. Participants were able to attend in pairs if they wished, with each person performing the experiment simultaneously yet independently. Two enticements to participate were provided: each participant could opt into entry in a random draw to win one of three $50 gift vouchers, and a bowl of assorted confectionery that was available during the experiment and during some in-person recruiting sessions.

7.4.3 Instruments

The experiment was conducted in a room provided by the School of Engineering and Computer Science of Victoria University of Wellington set up for this purpose. The experimental room had two ordinary workstation computers set up, as shown in Figure 7.7. Each machine had an ordinary keyboard, mouse, and screen, and was running Windows 7. All recorded information, including questionnaires, occurred within a web browser. The same browser, Google Chrome 33, was installed on each machine. The experimentor sat at a distance positioned to see both (if applicable) screens and observed participants during the experiment.

7.4.4 Protocol

On arriving at the experimental room each participant was given an information sheet and a consent form. Participants had time to read the information sheet and ask questions before signing the form. Participants under 18 years of age required parental consent to perform the experiment, but no such participants volunteered for the study. The information sheets and consent forms are reproduced in Appendix D.

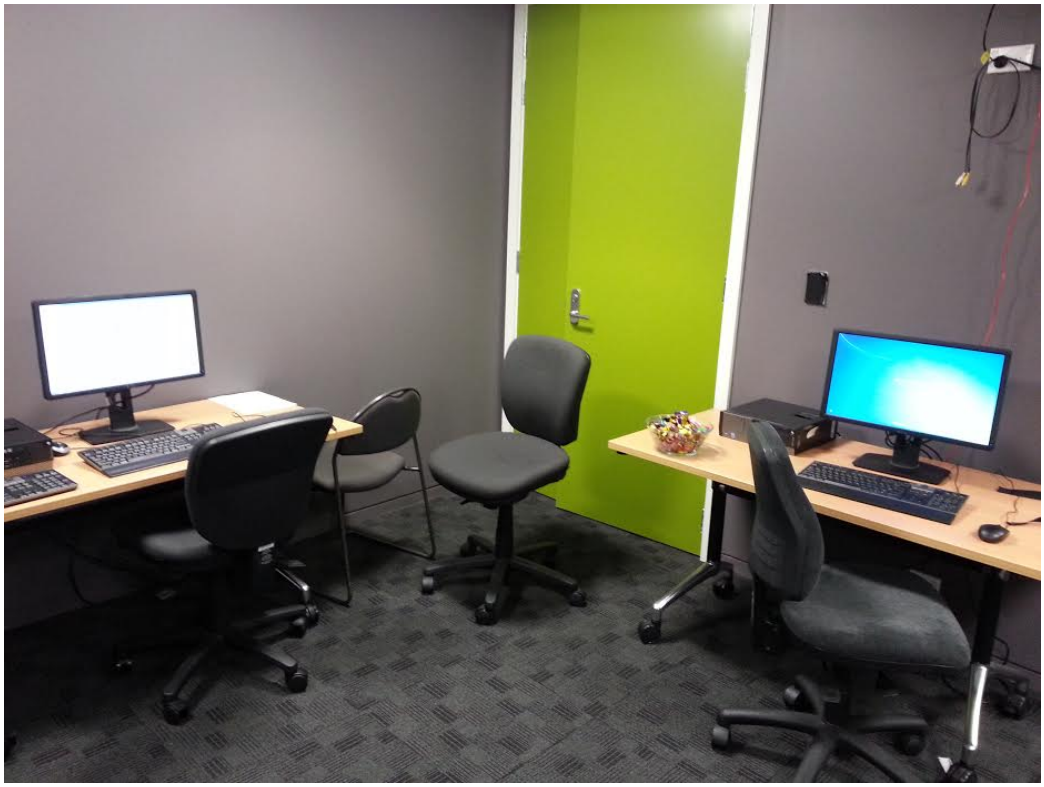


Figure 7.7: A photograph of the room used for experimental trials. The two experimental computers are on the far left and right edges of the picture, with up to one participant on each machine. The experimentor was positioned approximately where the camera is during trials.

After the completion of the consent form each participant was led to a workstation with the initial questionnaire open and was invited to fill it in. The survey responses were recorded electronically. The complete contents of the initial questionnaire is given in Appendix D.

Our experiment involved a tutorial guided by the experimentor and five tasks, all of which involved being presented with a program along with instructions on what to do with it. We selected the tasks with the goal of having users interact with all different parts of the system in mind, while also wishing to have the entire experiment complete within a reasonable time span (about 30 minutes).

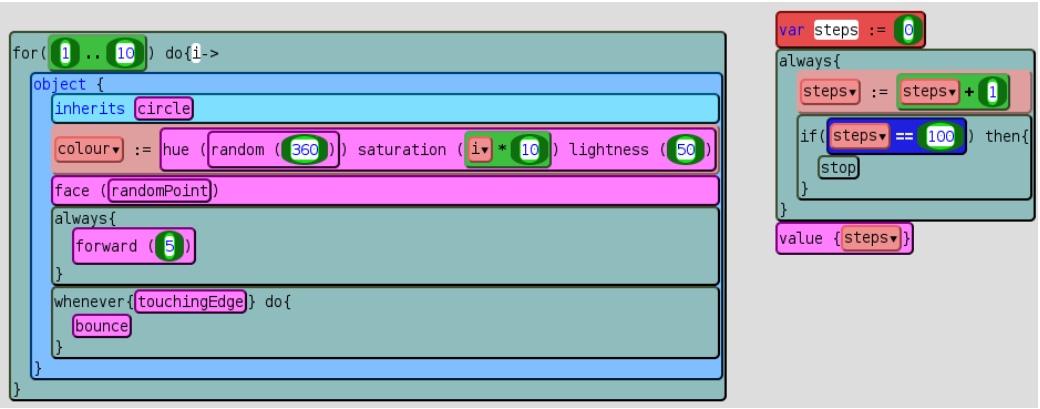


Figure 7.8: The tutorial program used in the experiment.

Tutorial

Following the completion of the initial questionnaire participants were given a scripted tour of the experimental system. Where multiple participants attended at once, both participants received the tour at the same time on a single machine, but were separated on different machines for the experiment itself. In either case participants used a freshly-loaded version of the intrumented interface that had not previously had any interaction for the body of the experiment. The tour used the tutorial program depicted in Figure 7.8, with the extended layout shown in Figure 7.9.

The script for the tour is given in Appendix E. We provided no introduction to the Grace language or to any other features of the system not mentioned in the transcript. If participants had questions at this point, we repeated portions of the speech if relevant, and otherwise declined to answer.

After setting participants up in this way we retired to a distant part of the room and left participants to proceed as they wished. One participant did not move on from the tutorial program after 20 minutes and we intervened to suggest that they do so at that point.

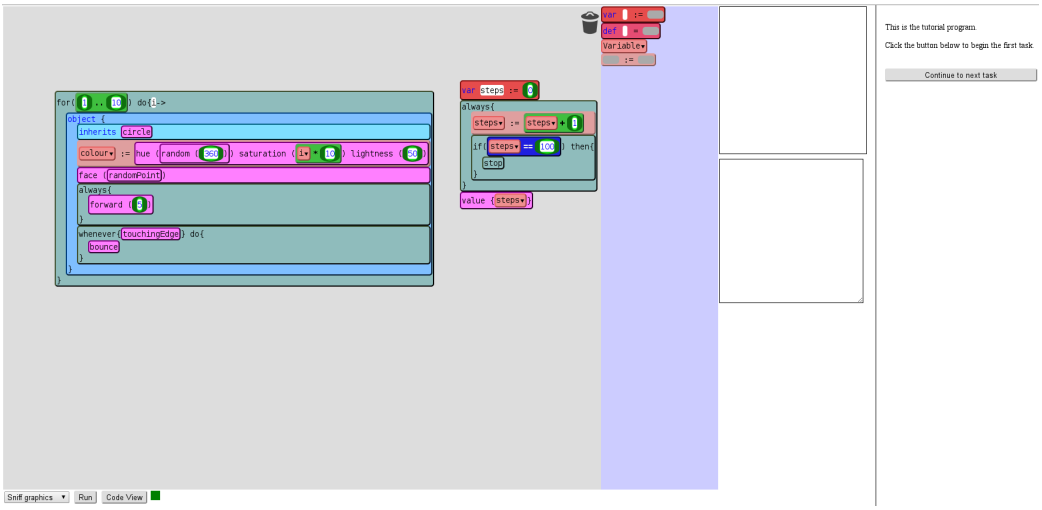


Figure 7.9: The layout of Tiled Grace used for the experiment. An additional column on the right-hand side contains the task descriptions, a button for moving to the next task, and (for some tasks) an input text area.

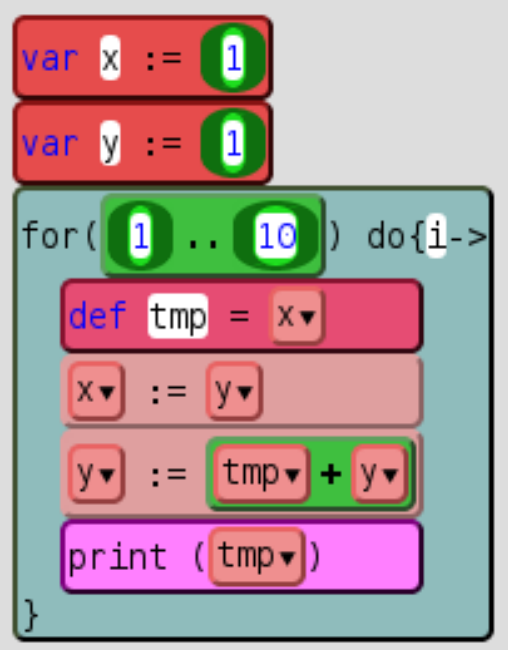
7.4.5 Data collection

While participants used the experimental system their on-screen interaction was recorded by the tool. Every drag, variable selection, text modification, switch of views, or attempt to run the program was noted, and a snapshot of the program was taken after every change. These logs were automatically saved to the server while the participant used the system. No audio or video recording was used in the experiment.

Task 1

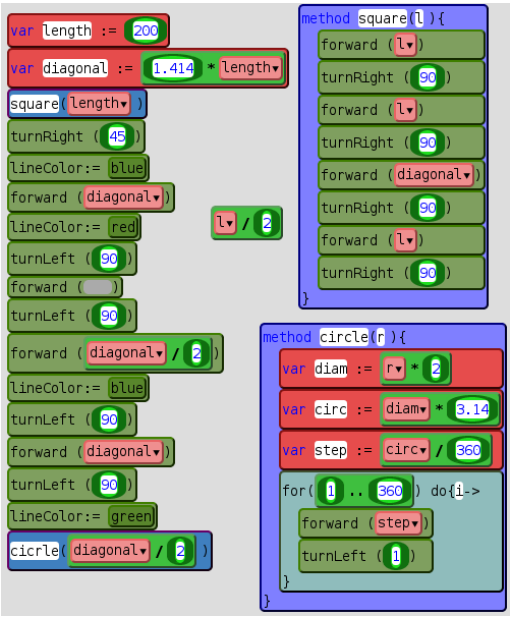
The first real task in the experiment was a small program printing the first ten Fibonacci numbers [157]. Participants were asked to modify the program to print twelve factorial numbers instead. Participants were given five minutes on this task. The initial program and the task description are shown in Figure 7.10.

Our intended solution was:



This program computes the first ten Fibonacci numbers and prints them onto the screen. The first two Fibonacci numbers are both 1, and each number after that is the previous two added together: **1 1 2 3 5 8 ...**. The program will print these out now.
Modify this program so it prints the first twelve factorial numbers instead. The first factorial number is 1, and each number after that is the previous number multiplied by how far through the sequence you are: **1, 1 * 2 = 2, 2 * 3 = 6, 6 * 4 = 24, ...**. Make the program print these instead.

Figure 7.10: The first task in the Tiled Grace experiment.



This program should draw an envelope shape with a circle around it, but it is broken and has some errors at the moment. Fix the program so that it draws an envelope.
The program uses a "turtle graphics" system, where your instructions tell a drawing robot where to move: forward 10, turn left 90 degrees, forward 10 more, and so on, with the robot leaving a trail behind itself.

Figure 7.11: The second task in the Tiled Grace experiment.

- To change the “10” to “12” in the loop bounds.

- To change the “tmp + y” tile to multiplication.

We selected this task as it could be represented by a single linear block of tiles, without involving any additional spatial confusion that might occur from having multiple different blocks. Of all the task programs, this one is the closest to the simple conventional textual programs participants may have encountered in introductory programming courses. We also wished to have a task involving updating variable assignments, but did not wish for that task to be very complicated. We were concerned that the task might be overly mathematical and so included explicit descriptions of the two sequences. This was the simplest task we could find where the solution involved multiple steps.

Task 2

The second task in the experiment used the turtle graphics dialect described in Section 6.2.1. The initial program given contained several errors preventing the program from compiling, and participants were asked to correct the errors as well as making the program draw a circled envelope shape. Participants were given five minutes on this task. The initial program and the task description are shown in Figure 7.11. The errors are shown in Figure 7.12. The intended solution was:

- To swap the erroneous `l` and `diagonal` variable tiles marked in Figure 7.12.
- To move the (now) `diagonal / 2` tile into the empty hole.
- To correct the spelling of `circle` in the method request tile.

This task tests our error-reporting feature. We took an existing demonstration program and introduced errors by moving tiles out of place and changing text. We wished to ensure that all participants used the error reporting in the same context at least once.

Task 3

The third task used the graphical dialect used in the tutorial program. The initial program included a circle which bounced randomly in the window and a face that followed the mouse pointer. Participants were asked to swap the behaviours of these two items: make the face bounce around and the circle jump to the mouse pointer. A second prompt suggested trying to change the colour of the ball, and to make it bigger. Participants were given five minutes on this task. The initial program and the task description are shown in Figure 7.13.

The intended solution for the first part of the task was simply to swap the behavioural content of the two objects: to move the `always` tile from

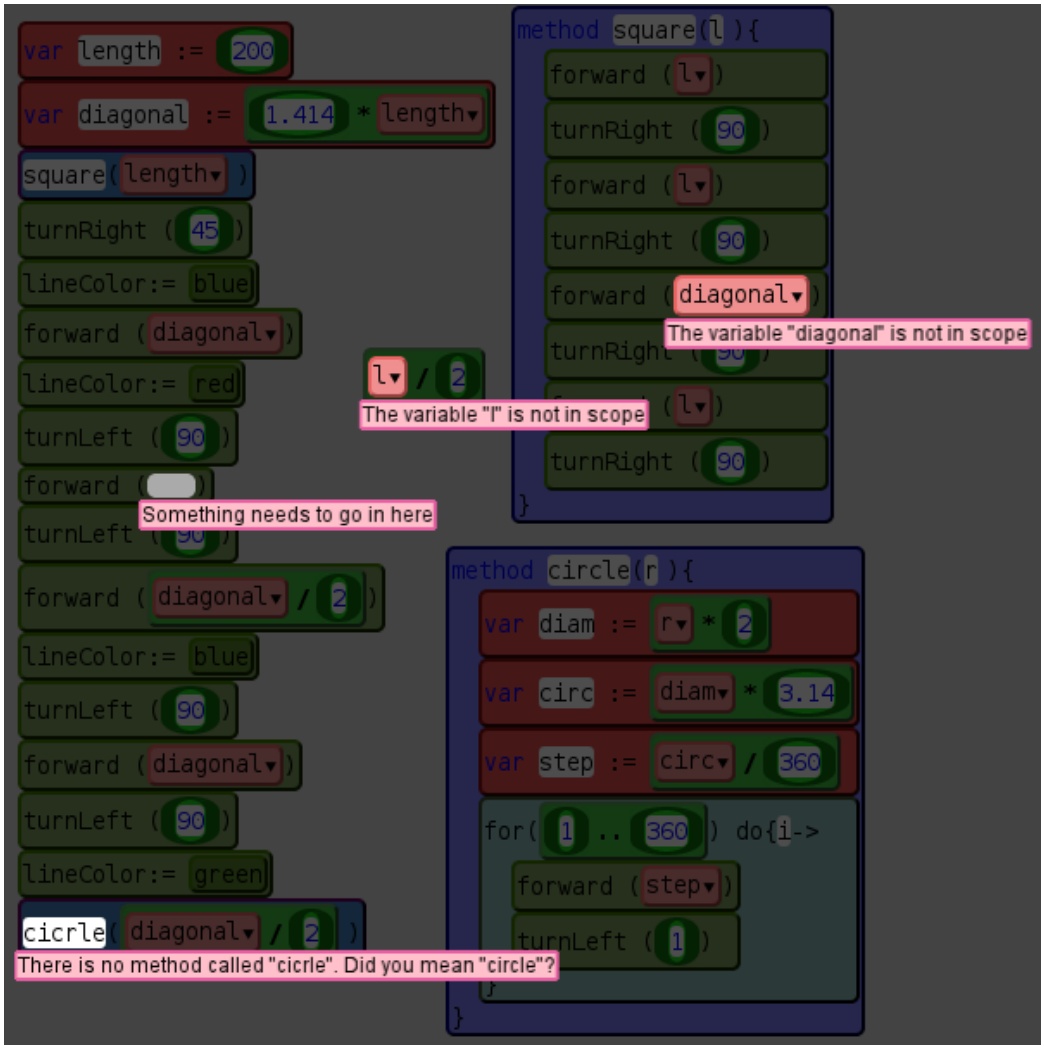
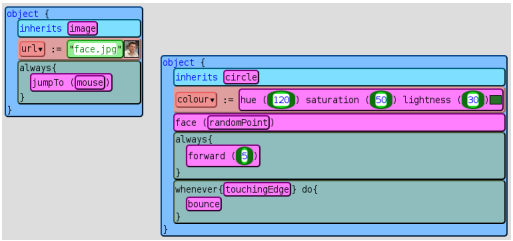


Figure 7.12: The program used in Task 2 of the Tiled Grace experiment, with the included errors highlighted.



This program has a ball bouncing around the screen, and an image that follows your mouse around. Make the ball follow the mouse and the image bounce around instead.
Try to make the ball a different colour, then try to make it bigger.
When you run your program a small red square will appear in the top-right corner, which you can click to stop the program before you run it again.

Figure 7.13: The third task in the Tiled Grace experiment.

the face into the circle, and everything from face(randomPoint) downwards from the circle into the face object. This could be accomplished either with tiles or by copy-and-paste in the text view.

Participants could change the colour of the circle by changing any one or more of the numbers in the hue(120) saturation(50) lightness(30) tile. Changing the radius was intentionally more complicated: participants would need to add a new variable assignment tile, choose the radius variable to assign to from the pop-up list of variables in scope, and set it to a number. Participants could discover the existence of radius from the drop-down list in the existing colour assignment.

We selected this task for two reasons. The solution to the first part is very simple, but offers two paths (moving the tiles and copy-and-paste of text), while also seeking to see how much understanding the user has of the task when they can modify the code. The second part requires users to explore the system further and to make use of variables other than those given to them already. As with all our experimental tasks, this one could be completed either with tiles or using the textual editor, and we were interested to see which one users would pick.

Task 4

The fourth task prompted participants to describe the behaviour of a program, rather than modify it. They were presented a program written in the graphical dialect used in Task 3 and the tutorial. The users were then asked to type a description of its behaviour, without running it. The "Run" button was disabled for this task. The program was initially presented in the text view, but participants could switch to the tiled view and back at any time. The program source is in Figure 7.14.

A reasonable description of the program behaviour would be:

> Creates a purple square and an orange square. Both squares constantly move forward and bounce off the walls when they hit. They start out pointing at right angles to each other.
>
> A circle always moves slowly towards one of the squares. When it touches the square it's following it changes target to follow the other square instead.

Partipants were prompted to move on after four minutes for this task, to allow time to complete filling in their description after the prompt while remaining within a small window.

The purpose of this task was to see which interface users preferred when asked only to comprehend a program, rather than modify it. The program was presented in the textual view first both to remind participants who may have forgotten about view switching and to allow them to show whether they had gained any knowledge of the text from their use of tiles.

Task 5

The fifth task was the end of the experiment. A program was presented, but no task was assigned. The task description said:

> You're done! You can play with the system here. Move on to the final questionnaire when you're ready

```
dialect "sniff"
object {
    inherits rectangle
    width := 50
    colour := "orange"
    whenever {touchingEdge} do {
        bounce
    }
    always {
        forward (2)
    }
}
def orange = above

object {
    inherits circle
    var target := orange
    always {
        face (target)
        forward (1)
    }
    whenever {touching (target)} do {
        if (target == orange) then {
            target := purple
        } else {
            target := orange
        }
    }
}

object {
    inherits rectangle
    width := 50
    turn (90)
    colour := "purple"
    whenever {touchingEdge} do {
        bounce
    }
    always {
        forward (2)
    }
}
def purple = above
```

Figure 7.14: The source code presented in Task 4 of the Tiled Grace experiment.

The program implemented a crude orbital simulator in the graphical dialect, incorporating the Earth orbiting the Sun and the Moon orbiting the Earth. A mars object was defined, but had no behaviour.

The intention of this task was to provide a chance to measure free experimentation by users and to see whether they would continue to use the system by choice; this measure of engagement is similar to Kelleher *et al.*'s use of "sneak time" [92]. The task serves primarily to measure implicit engagement in this way and was not designed with any particular interaction feature in mind. No time limit was placed on this task, although we would suggest participants begin the final questionnaire ten minutes before the next experiment was due to begin.

Final questionnaire

The final questionnaire was administered in the same manner as the initial questionnaire. This questionnaire was longer than the initial questionnaire, and asked participants about their interactions with the system and what they preferred. Three free-text entry fields were provided with prompts for the participant to say what they liked or disliked about the system, and to include any other comments they had. The complete questionnaire is given in Appendix D.

Following completion of the questionnaire participants could leave. A bowl of assorted confectionery was in the room and attendees were invited to take from it at any time. Most did so at the conclusion of their trial.

In the questionnaire we sought to measure what participants found difficult or easy in the experiment, how engaged they were, which features they had used (particularly the ability to switch views), and what they liked or disliked about the system. Questions primarily asked participants for such information directly and gave them a seven-point Likert item to answer on. Some questions sought to gauge participant perceptions of information that we had measurable data on; to that end, we asked about which view of code they used most and how often they switched views.

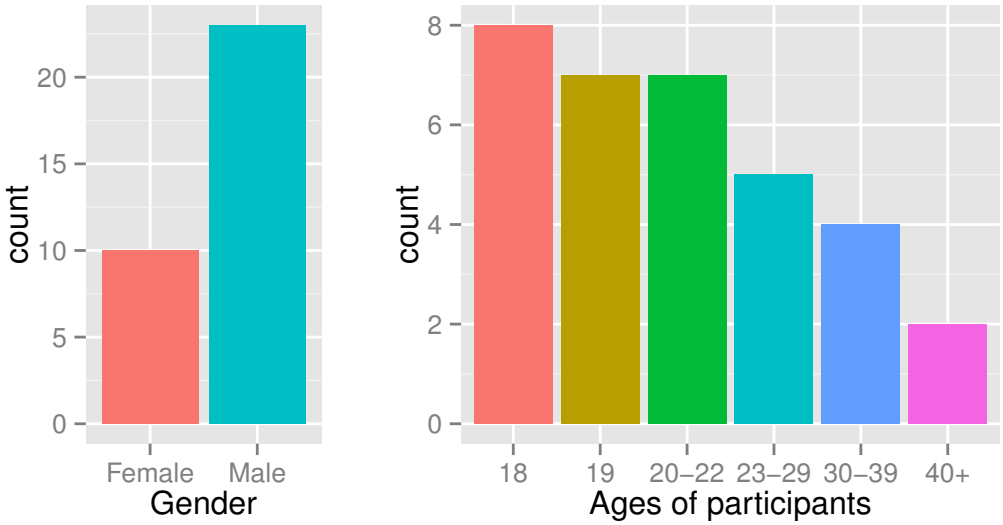


Figure 7.15: Basic sample demographics of our experiment.

7.5 Results

In this section we present the results of our experiment.

In total there were 34 participants in the experiment. Participants were able to withdraw at any time during the experiment and for two weeks afterwards. Participants were not required to provide a reason for withdrawing. One participant chose to withdraw and is excluded from the sample, leaving a final sample size of **33** participants.

Anonymised data and tooling from this experiment is available in the auxiliary data included with the thesis (see Appendix A).

7.5.1 Demographics

Participants were primarily drawn from students in the School of Engineering and Computer Science at Victoria University of Wellington and so represent at best the demographics of the source. 23 (70%) of participants were male while 10 (30%) were female. The median age of participants

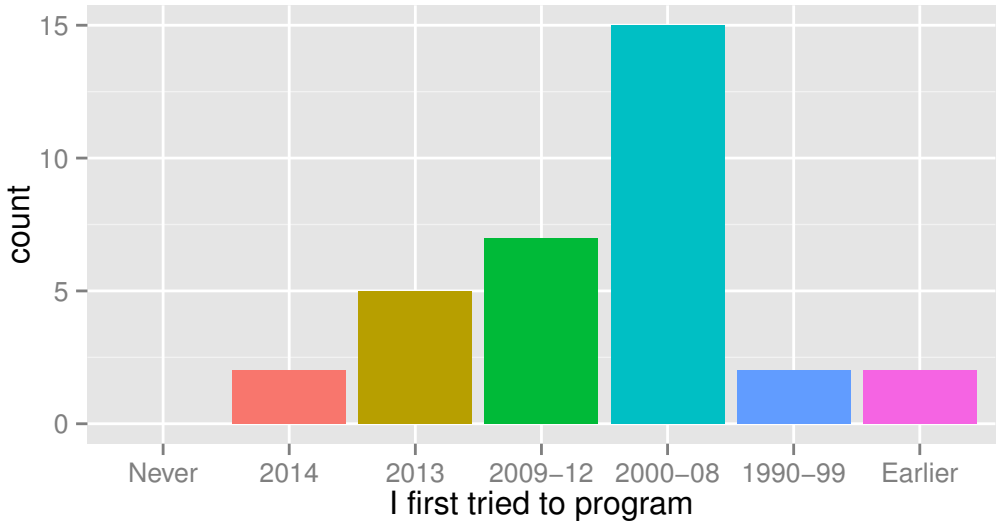


Figure 7.16: Bar chart of when participants first tried to program a computer. The experiment took place in March–April 2014.

was 20 and the most common age was 18. There are progressively fewer participants in older age bands. A full breakdown of these demographics is shown in Figure 7.15.

Both of these fields were free text entry. We case-folded responses for gender, but made no other modifications to that data. One participant stated his age as "40's" (sic). For statistical purposes we assigned this participant an age of 45.

7.5.2 Programming experience

Three questions spoke to a participant's past programming experience.

We asked when they first tried to program a computer. We chose this phrasing to try to level out participants' views of what "counts" as programming. Potential answers were presented on a seven-point Likert item with options "Never", "2014", "2013", "2009-2012", "2000-2008", "1990-1999", and "Earlier". Responses are shown in Figure 7.16. The modal and

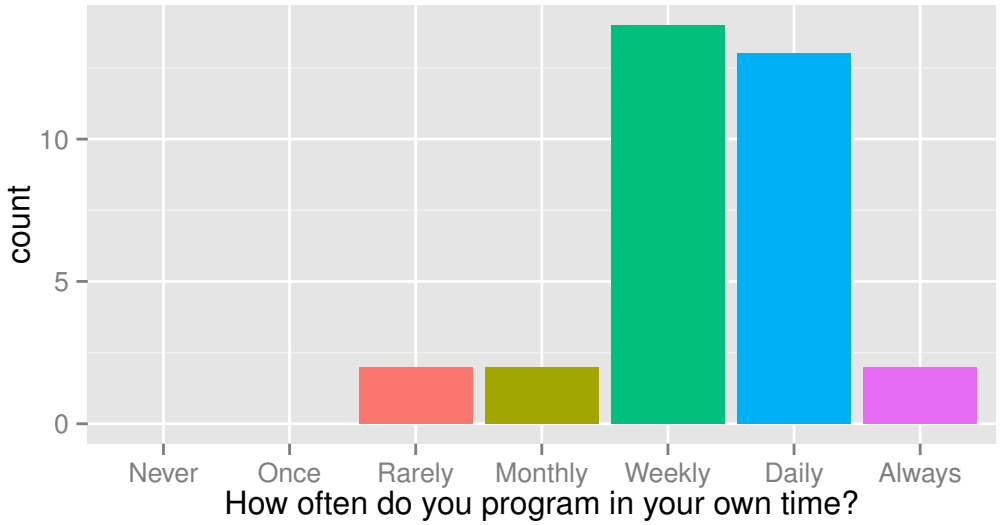


Figure 7.17: Bar chart of how often participants program in their own time.

median response was 2000-2008, with a sharp drop off for earlier dates, to be expected given the age distribution of participants.

We asked how often participants program in their own time. By this phrasing we attempt to exclude classwork. Five participants enquired at this question whether this meant "for fun" and we indicated in the affirmative; it is unclear whether other participants interpreted homework as being done "in their own time". Potential answers were presented in a standard frequency seven-point Likert item with options "Never", "Once", "Rarely", "Monthly", "Weekly", "Daily", and "Always". 14 participants (42%) answered "Weekly" while 13 (39%) answered "Daily". All other answers received no more than 2 responses (6%). Responses are shown in Figure 7.17.

We also presented a list of 72 technologies (mostly programming languages, but also IDEs and other tools) and asked participants to indicate any they had used before. Several participants asked what "used" meant, and we indicated that it meant used in any capacity and degree. While we are interested in certain technologies in particular, which we will look at in

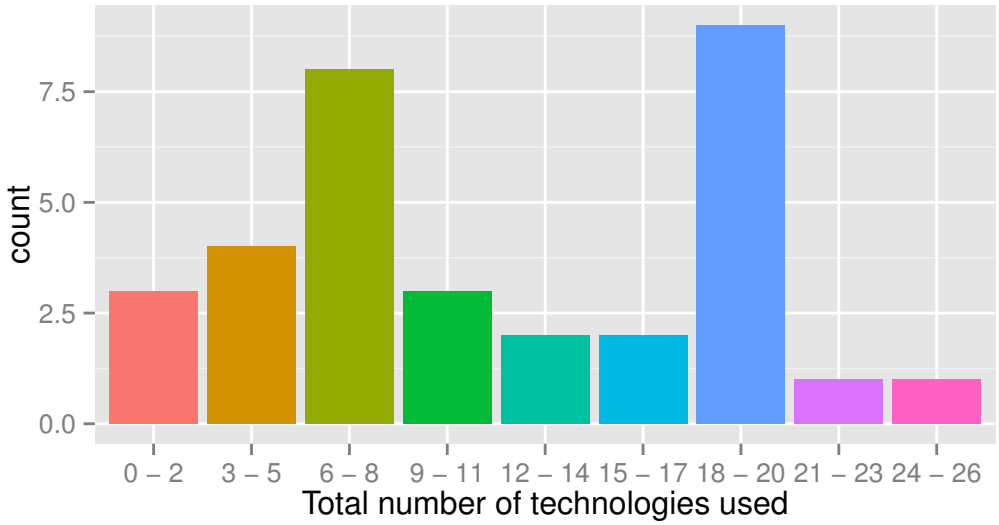


Figure 7.18: Bar chart of how many technologies participants claimed to have used.

the next subsection, here we simply count how many technologies participants claimed to have used, as shown in Figure 7.18. The total number of technologies ranged from 1 to 25. The median was 10.

7.5.3 Technologies used

Figure 7.19 shows which technologies participants claimed to have used. The most popular technologies were Java and Eclipse, both of which are used in undergraduate courses in the school. Both Java and Eclipse had been used by 26 participants (79%). Also particularly popular were Python, with 25 participants (76%), and HTML, with 24 (73%).

One participant had encountered Grace previously. It is unclear to what extent he had used the language, but we do not believe it can have been substantial.

Four participants had previously used Scratch, the system most similar to our drag-and-drop interface, while six had used Alice, another intro-

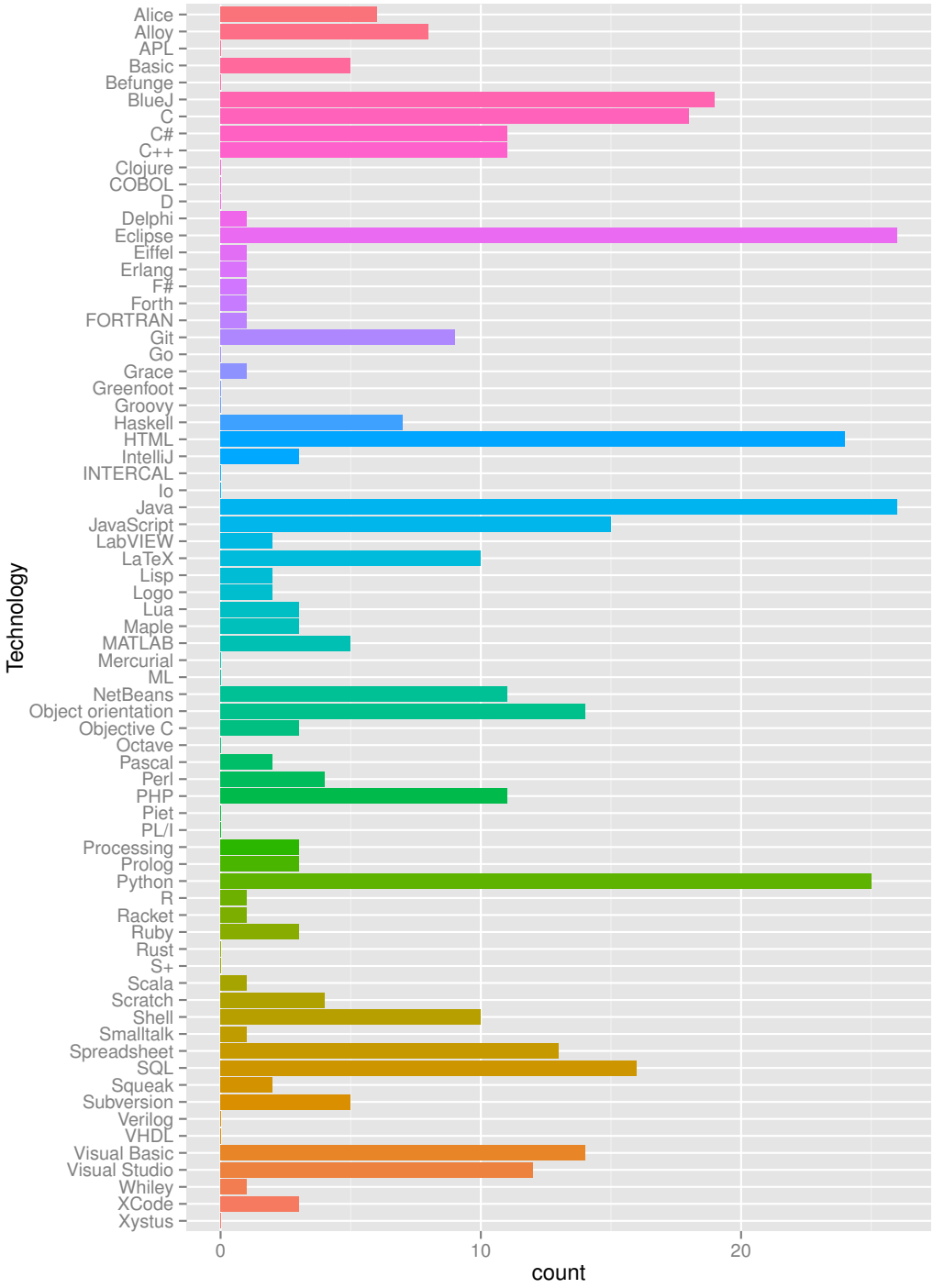


Figure 7.19: Which technologies participants had used in the past.

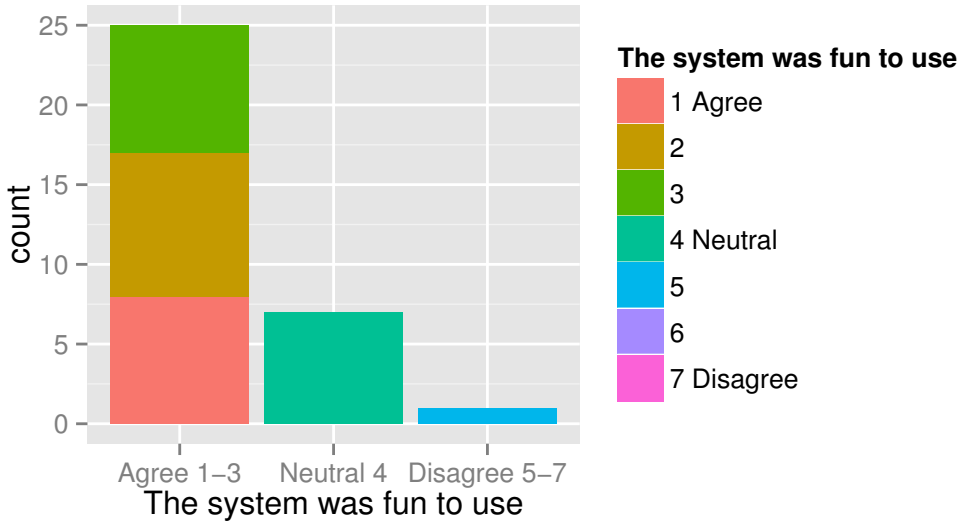


Figure 7.20: Participant responses to the statement "The system was fun to use". All three responses on each of the agreement and disagreement sides are stacked together.

ductory programming language. No participants named Xystus, a false technology we inserted to detect any participants who checked every box indiscriminately.

7.5.4 Engagement

A key measure of this system is user engagement. We attempted to measure engagement in multiple ways. In the simplest, we asked participants in the final questionnaire whether the system was fun to use. Responses were on a seven-point Likert item and shown in Figure 7.20. Responses 1, 4, and 7 were labelled "Agree", "Neutral", and "Disagree".

The most common response was 2, with nine participants (27%), while 1 ("Agree") and 3 were chosen eight times (24%) each. 25 participants in total (76%) chose one of the responses on the Agree side. One participant chose 5, a light disagreement, while seven (21%) were neutral. The median

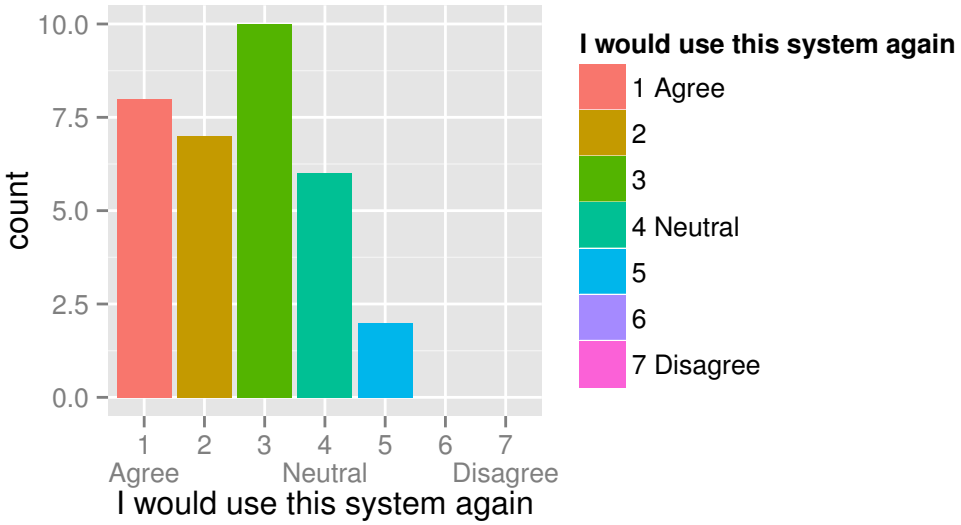


Figure 7.21: Participant responses to the statement "I would use this system again".

response was 2, a medium agreement.

In the final questionnaire we also asked participants whether they would use the system again. Responses were on a seven-point Likert item and shown in Figure 7.21. Responses 1, 4, and 7 were labelled "Agree", "Neutral", and "Disagree".

The most common response was 3, with ten participants (30%), while 1 ("Agree") was chosen eight times (24%) and 2 seven times (21%). Again 25 participants in total (76%) chose one of the responses on the Agree side. 2 participants (6%) chose 5, a light disagreement, while six (18%) were neutral. The median response was 3, a light agreement.

The fifth task of our experiment included a program but no actual task, instead informing participants that they were finished, that they could continue to use the system if they wished, and to move on to the final questionnaire when they were ready. By this we intended to measure implicit engagement: would participants use the system unprompted? Figure 7.22 shows whether participants interacted with the system for

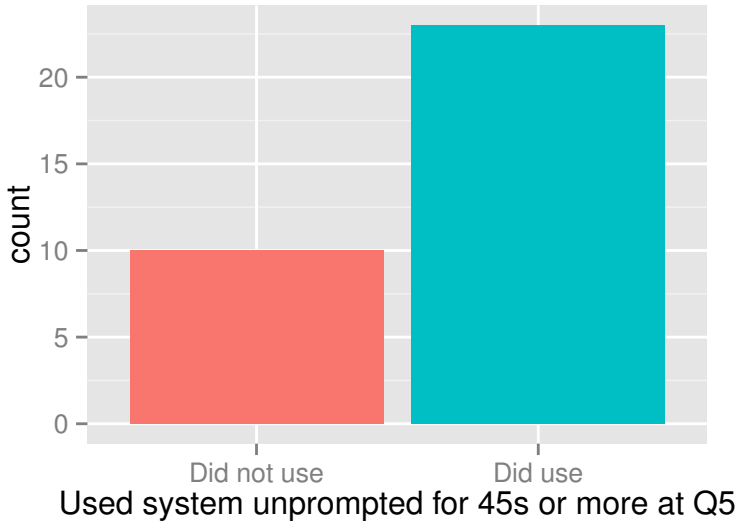


Figure 7.22: Participant engagement with Task 5.

45 seconds or more. We chose this threshold conservatively, allowing 30 seconds for participants to read the task description, look at the program, and potentially run it before moving on to the questionnaire, and adding a 15 second buffer. 23 participants (70%) used the system for at least 45 seconds at this point, while 10 (30%) moved directly on to the questionnaire.

Figure 7.23 shows how long participants spent on Task 5. We measured time from the point a participant arrived at a question until the last event (drag, run, edit, and so on) recorded from them. Some participants returned to the task after completing the final questionnaire; in order not to count the time spent filling in the questionnaire, any gap between adjacent events of more than 60 seconds caused the clock to pause at the first event and resume at the second. In this way we may have undercounted some time that the participant was in fact using the system, but we are reasonably confident that we have not overcounted. The longest time a participant spent on the task was 13 minutes and the minimum was 0 seconds. The median time was 1:43 and the mean 3:10.

We take from these results that participants were reasonably engaged

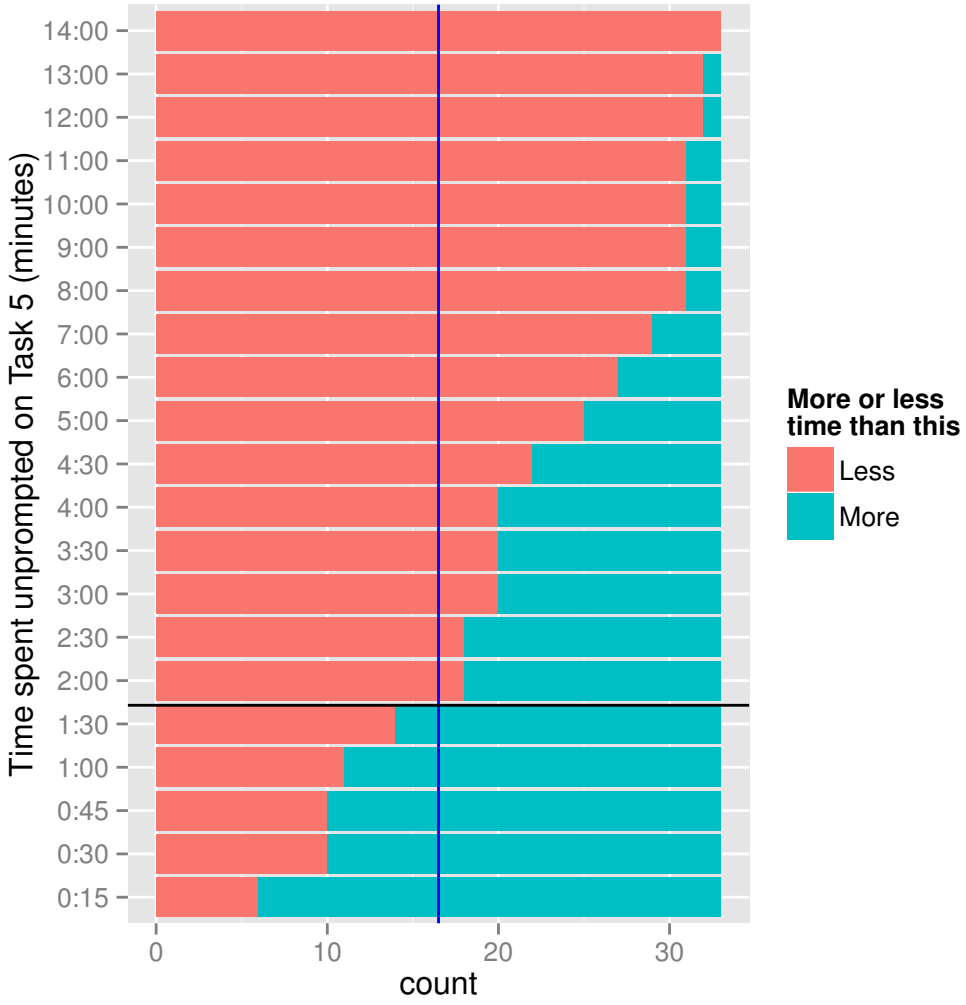


Figure 7.23: Time spent unprompted on Task 5. Note that the y axis is nonlinear in time. The black horizontal line marks the median time; the blue vertical line indicates the median participant.

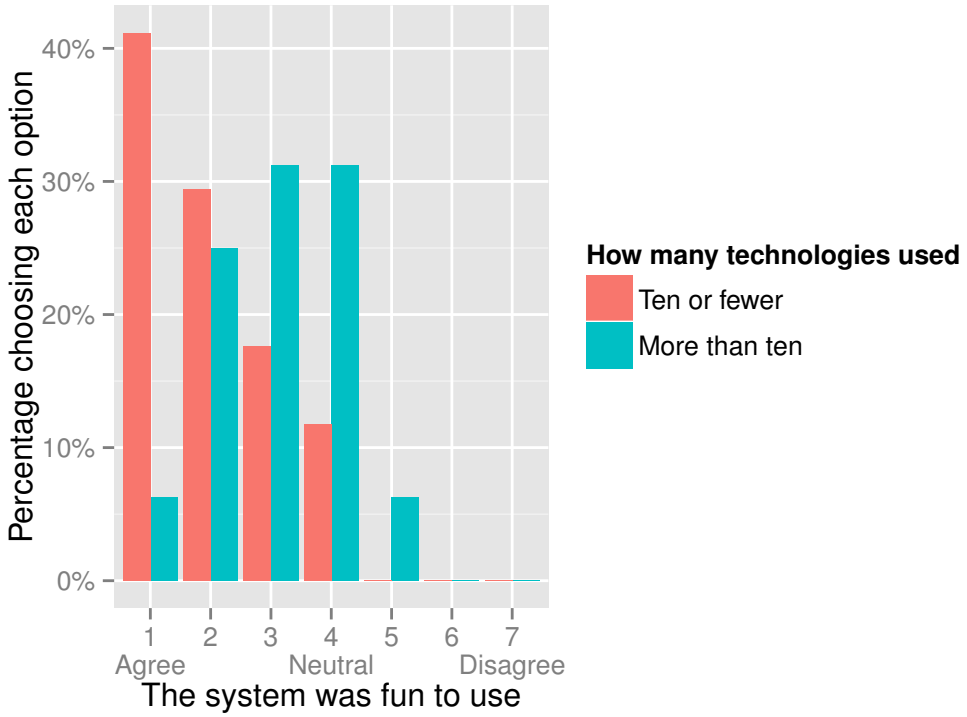


Figure 7.24: Participants' agreement with "The system was fun to use" split by how many technologies they had used.

with the system. Large majorities in every case indicated some degree of engagement, including both when explicitly asked and through revealed preferences.

Not all participants were as enthusiastic, and we note one trend shown in Figure 7.24 in particular. If we recall the list of technologies we asked participants about their use of, we can divide participants into two groups: those who have used more than the median number of technologies (16 participants, or 48%), and those who have not (17 participants, or 52%). We can then examine the proportions in each group giving each response to the statement "The system was fun to use". On doing so we see that participants with less experience are substantially more positive than those with more. Fully 41% of less-experienced participants fully agreed with

the statement, while only 6% of more-experienced participants did so. Similarly, 31% of more-experienced participants were neutral, while only 12% of less-experienced participants were. We considered this trend worth examining more closely.

Figure 7.25 plots the number of technologies used against the participant's view of how fun the system was. The blue line is a linear regression model for the data, showing a trend correlating technologies with disagreement. We can attempt to determine how significant this trend is. Figure 7.26 gives a number of analyses of the model, while Table 7.1 shows the model represents a meaningful trend significant at the 99% level.

Table 7.1

	Dependent variable:
	The.system.was.fun.to.use
TotalUsedTechnologies	0.087***
	(0.027)
Constant	1.498***
	(0.361)
Observations	33
R^2	0.254
Adjusted R^2	0.230
Residual Std. Error	1.032 (df = 31)
F Statistic	10.534*** (df = 1; 31)
Note:	*p<0.1; **p<0.05; ***p<0.01

While the correlation coefficient is moderate, indicating that other factors are in play, it appears that all other things being the same a more experienced user will enjoy the system less. This result is consistent with our and others' experience with Scratch, and not a significant issue for a tool designed for introductory programming. Similar results occur with our other methods of measuring programmer experience, but as these

measures are much coarser the impact is reduced.

7.5.5 Error handling

The tiled interface both prevents some kinds of error from occurring at all and provides the opportunity for entirely new kinds of error. Tiled Grace includes novel error reporting for such code, as described in Section 7.3.1. We asked participants whether finding errors in the code was easy, and also whether fixing them was easy. The results are shown in Figure 7.27. Responses were on a seven-point Likert item with responses 1, 4, and 7 labelled "Agree", "Neutral", and "Disagree".

Most participants agreed that finding errors was easy. The modal answer was 1 ("Agree"), with 13 participants (39%), while 26 in total (79%) gave an answer on the Agree side. The median answer was 2, a moderate agreement. Responses were much more varied on the question of fixing errors, with every response from 1 to 5 being chosen by between five and seven participants. Fixing errors in an unfamiliar system, language, and codebase under time pressure would not generally be expected to be easy, so this result is not surprising.

We also asked participants whether they found the syntax difficult to deal with. While the interpretation of this question was up to the participant, it might indicate what about fixing errors participants found difficult. The results of combining the two questions are shown in Figure 7.28; we see no particular trend between the two.

7.5.6 View switching

Tiled Grace permits switching between tiled and textual views of code at any time. We measured participants' use of this feature and asked them several questions about it.

One particular focus of the tiled interface was the elimination of basic syntax errors like mismatched brackets or using the wrong symbol.

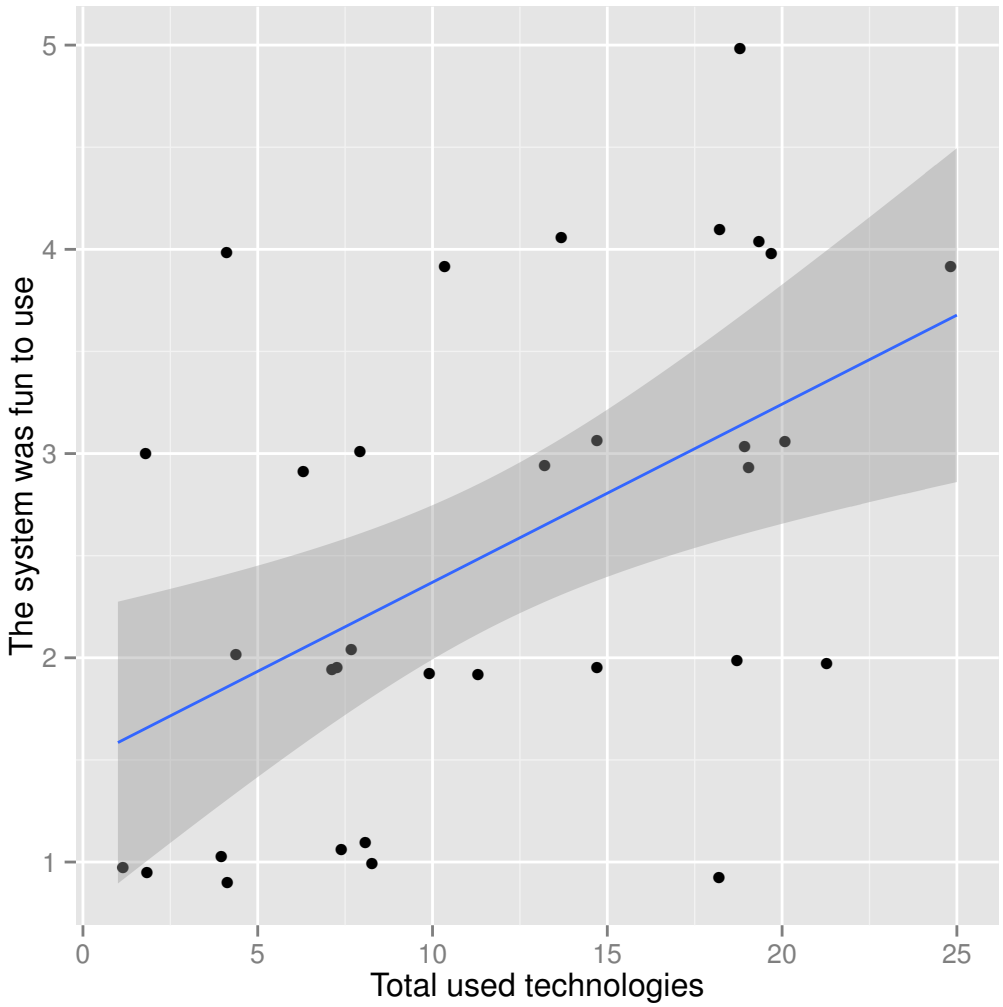


Figure 7.25: Total used technologies versus how fun participants rated the system. The blue line is a linear regression model. Points are jittered slightly on the vertical axis to avoid overlap.

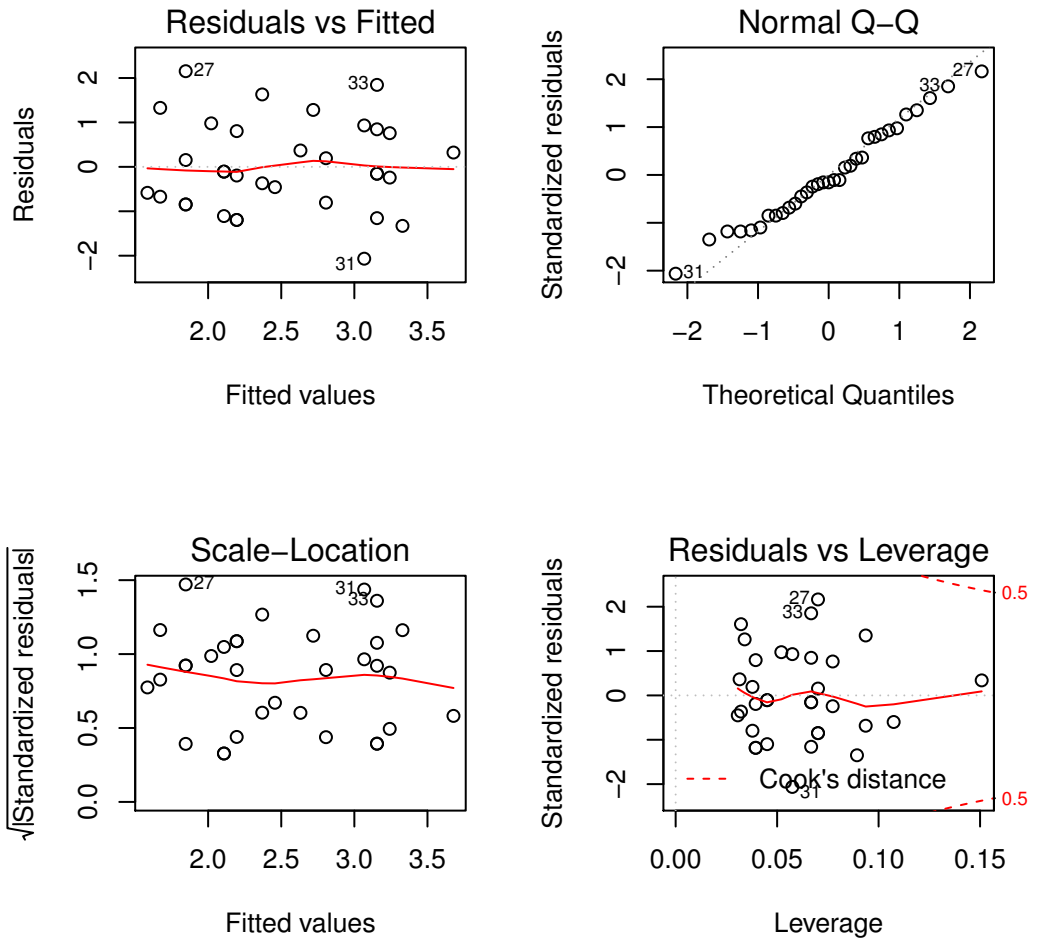


Figure 7.26: Significance plots for the regression model shown in Figure 7.25

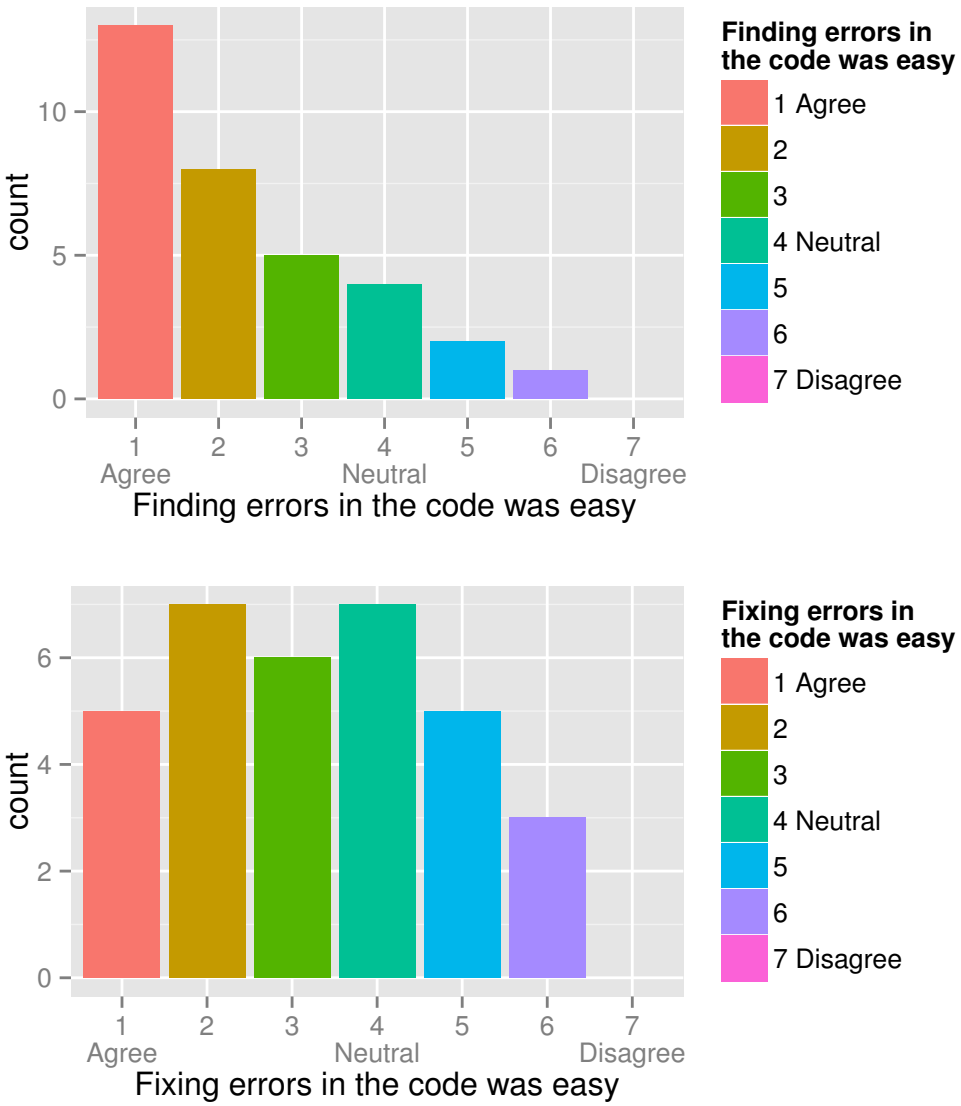


Figure 7.27: Participant agreement that finding and fixing errors was easy.

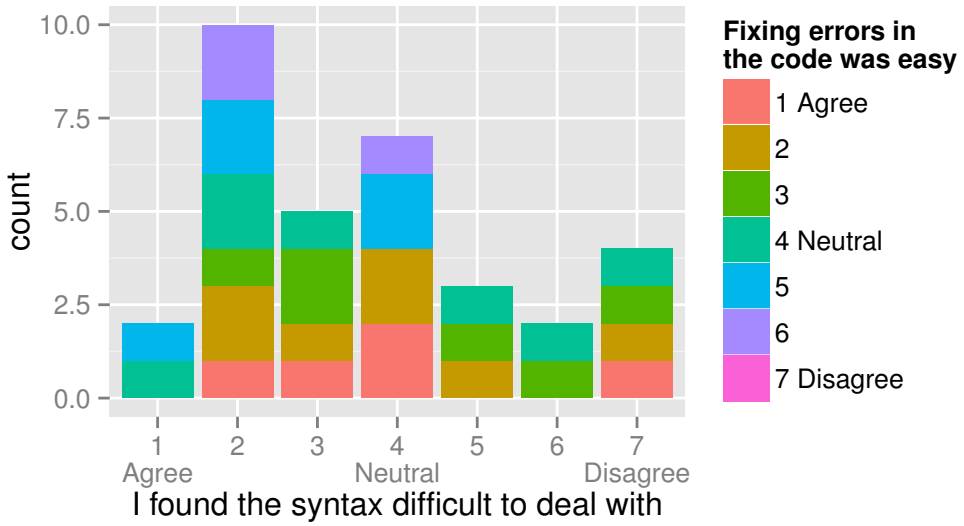


Figure 7.28: Participant agreement that syntax was difficult, broken up by agreement that fixing errors was easy.

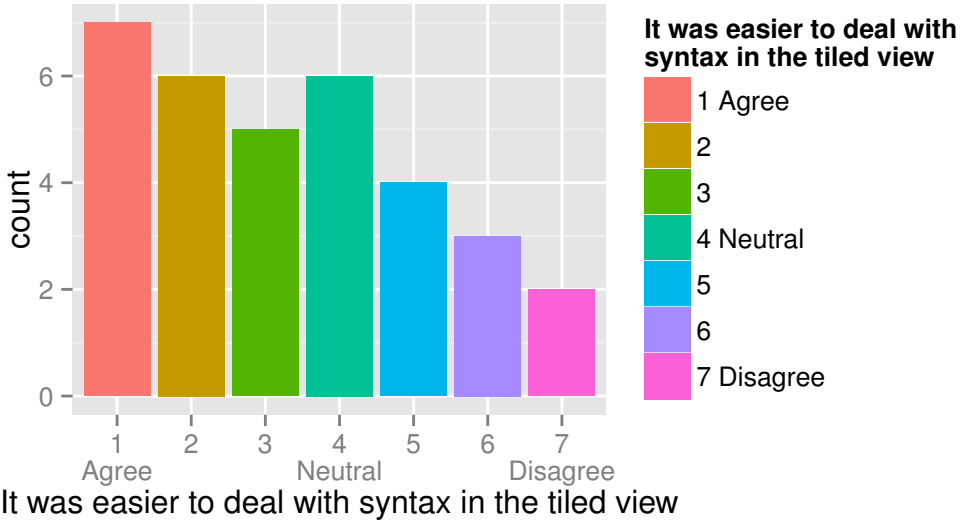


Figure 7.29: Participant agreement that dealing with syntax was easier in the tiled view.

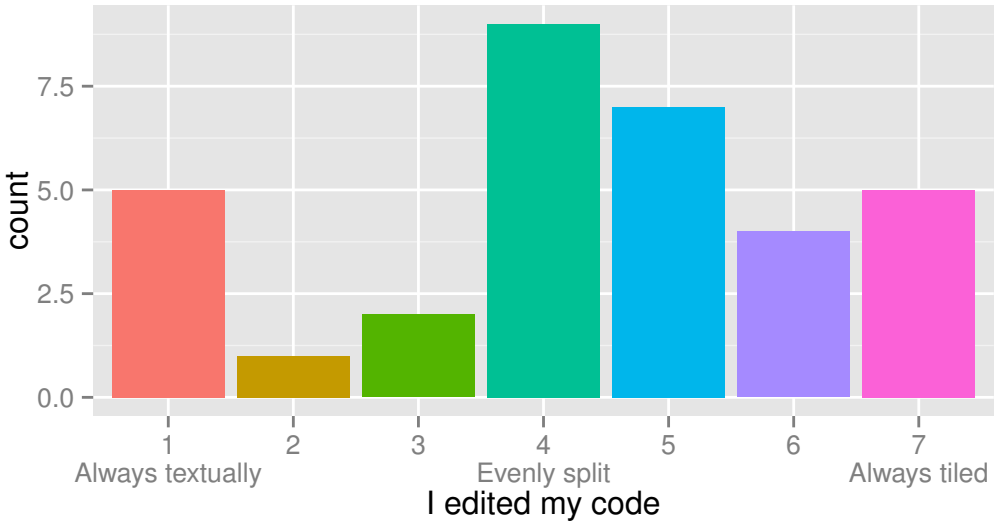


Figure 7.30: How participants felt they had edited their code.

We asked participants whether they found the syntax of the unfamiliar language easier to deal with in the tiled view, with the results shown in Figure 7.29. Most participants (18, or 55%) chose an answer on the Agree side and answers were steadily less common moving towards Disagree. The exception was a spike in "Neutral". This spike may represent participants who felt unqualified to answer this question as they had principally used one view or the other.

We asked participants how they had edited their code. Most participants felt that they had used the tiled view a substantial portion of the time, with 16 (48%) indicating they used it most of the time and a further nine (27%) indicating they had split their time evenly. Five participants (15%) said they used the textual view exclusively. These results are shown in Figure 7.30. Responses were on a seven-point Likert item with 1, 4, and 7 labelled "Always textually", "Evenly split", and "Always tiled".

We can check whether participants' recollections of their usage are accurate or not. The version of Tiled Grace used in the experiment was instrumented to record all interaction and we can compute the proportion

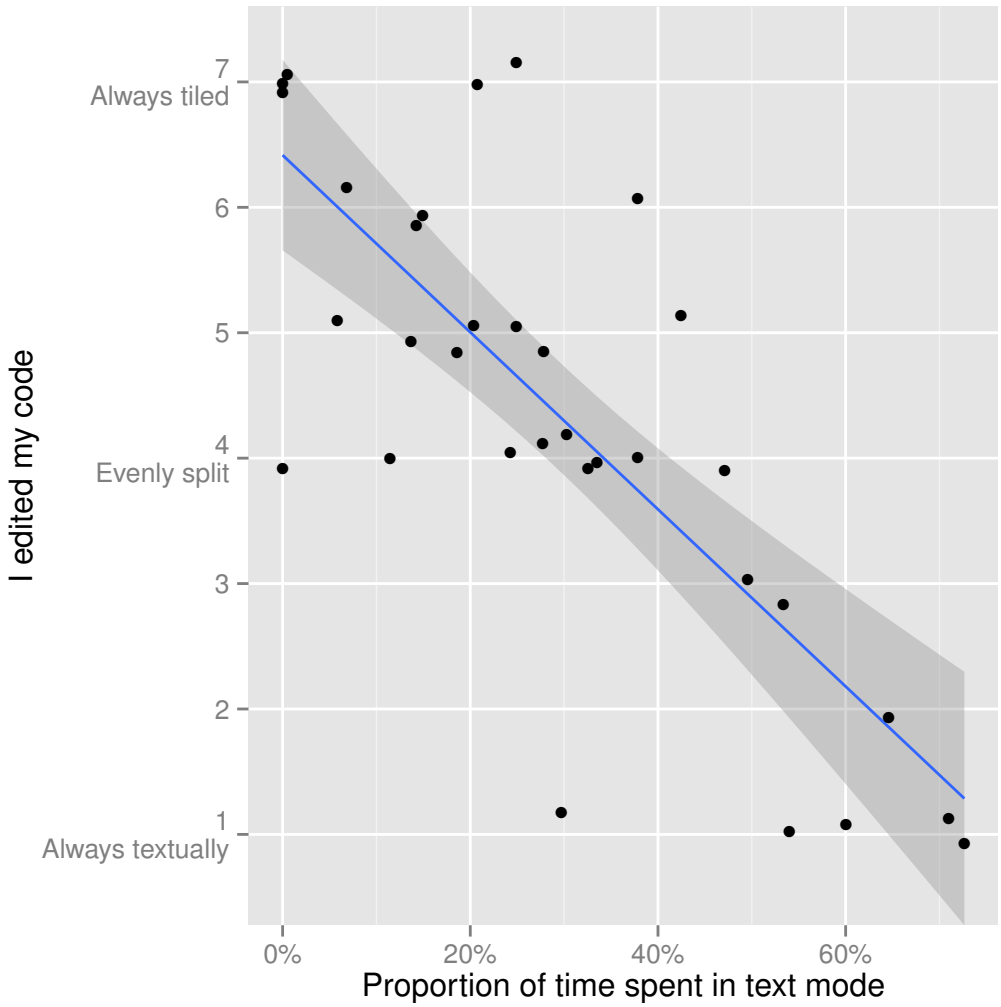


Figure 7.31: How participants felt they had edited their code versus how they actually did so. This includes time spent on Task 2, which could not be switched to the text view until the basic errors had been resolved.

	Statistic	Task 1	Task 2	Task 3	Task 4	Task 5	Total
Proportion of time in text view	Min.	0%	0%	0%	3%	0%	1%
	1Q	0%	0%	0%	65%	0%	24%
	Med.	17%	53%	8%	100%	0%	33%
	3Q	50%	84%	32%	100%	17%	52%
	Max.	83%	94%	76%	100%	83%	78%
Number of switches of view	Min.	0	0	0	0	0	0
	1Q	0	0	0	0	0	4
	Med.	1	1	1	0	0	6
	3Q	3	3	3	1	2	11
	Max.	8	8	8	5	14	22

Table 7.2: Summary and distribution statistics for the usage of different views per Task and overall.

of time each participant spent in each view. These proportions are plotted against responses to the previous question in Figure 7.31. The general trend evident in the figure is that participants were reasonably accurate, although no participant actually spent more than 78% of their time in text view. There are some notable outliers, however, particularly one participant who indicated they were evenly split between tiled and text views but in fact never used the text view at all. Nonetheless, participants appear to be quite accurate in their recollection of this fact overall.

We counted how many times participants switched views during the experiment. The median number of switches is six, the first quartile is four, and the third quartile is eleven. These are shown on a boxplot and histogram in Figure 7.32. Participants varied quite significantly in their use of the view-switching feature, using it between zero and 22 times.

We can also look at switches and time spent in text view on a per-task basis. Summary and distribution statistics of time spent in each mode and view switches broken down by task are in Table 7.2. Most participants used the tiled view the majority of the time, but most also used the text view a nontrivial amount of the time on each of the first three tasks. We plot these measurements against each other in Figure 7.33. This plot shows a

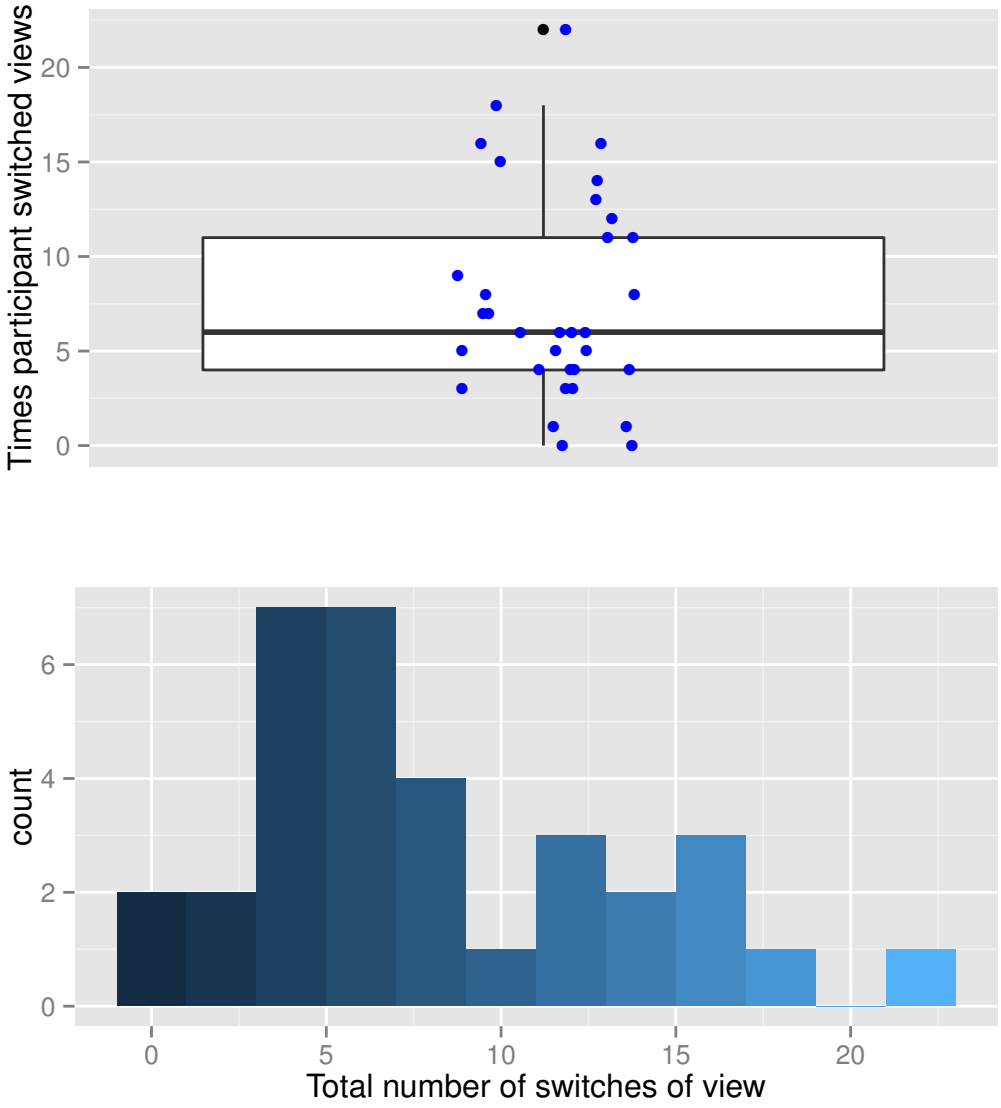


Figure 7.32: How often participants switched views

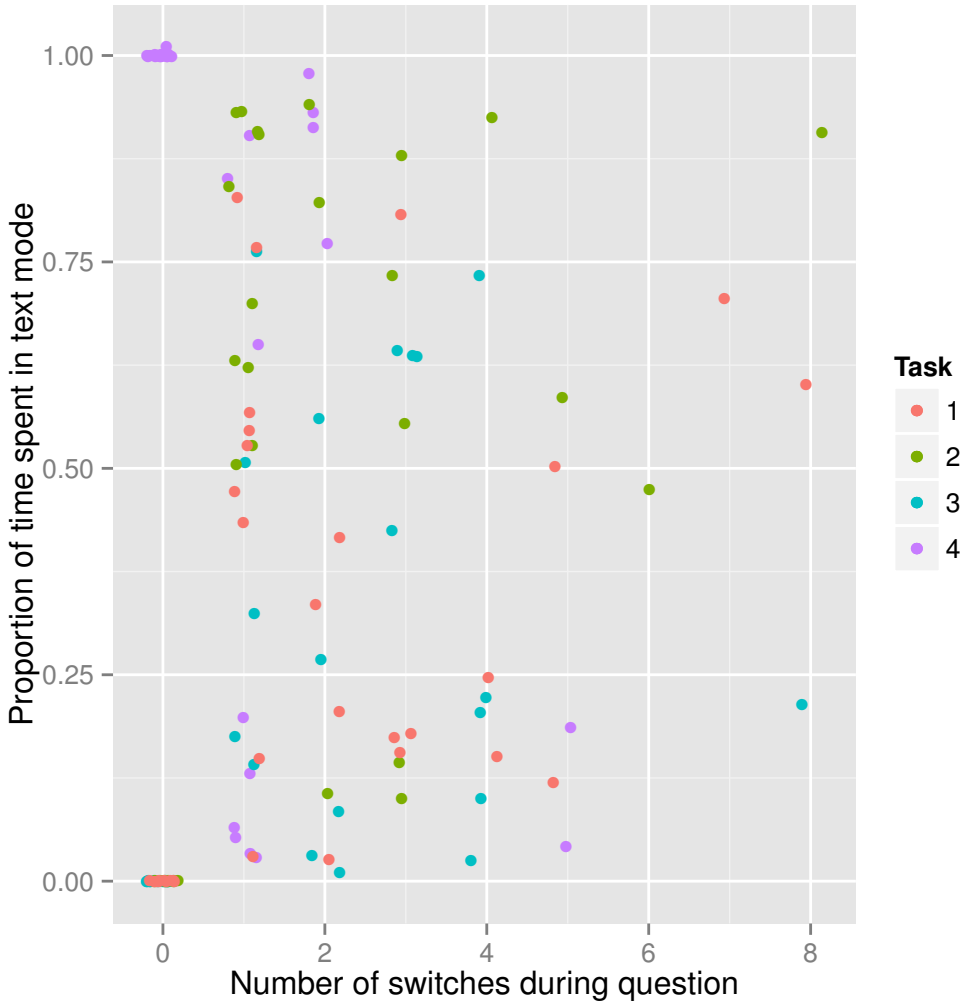


Figure 7.33: How often participants switched views versus the proportion of time they spent in text mode for each task.

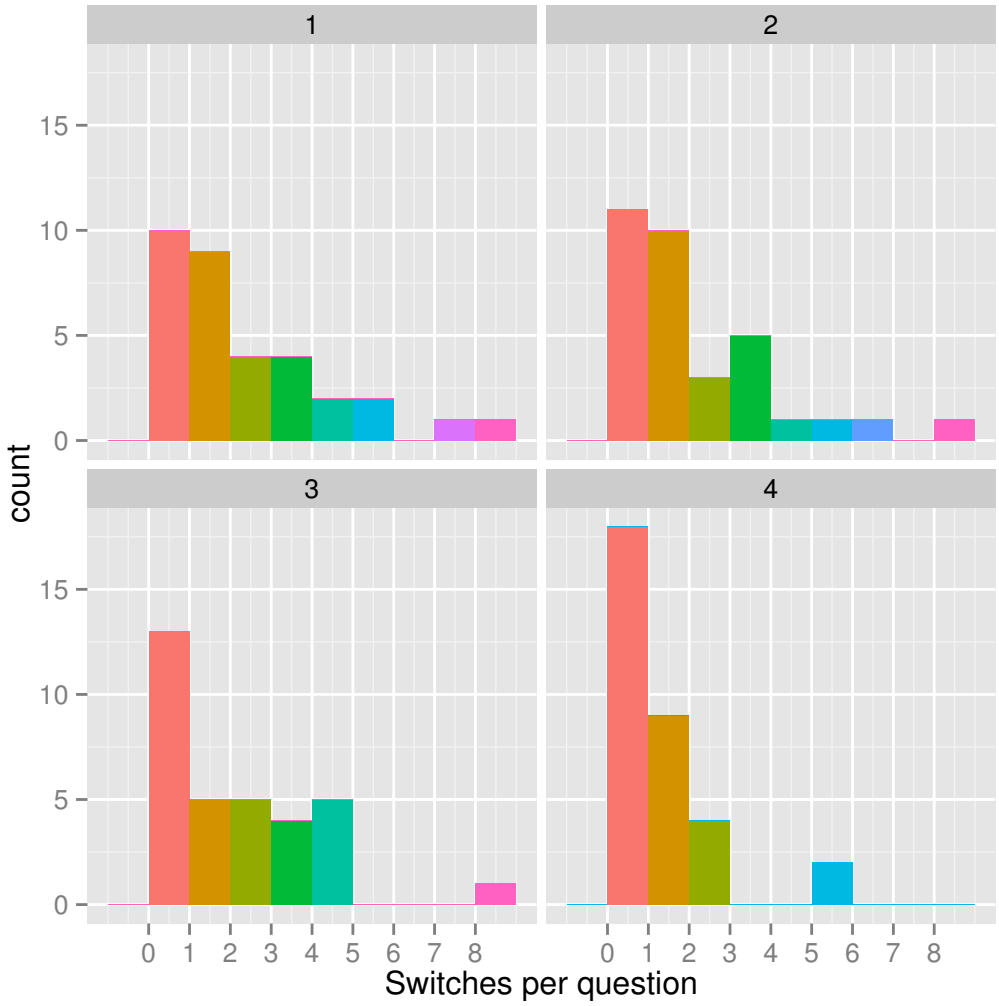


Figure 7.34: How many switches of view participants made for each task. Each task is plotted separately.

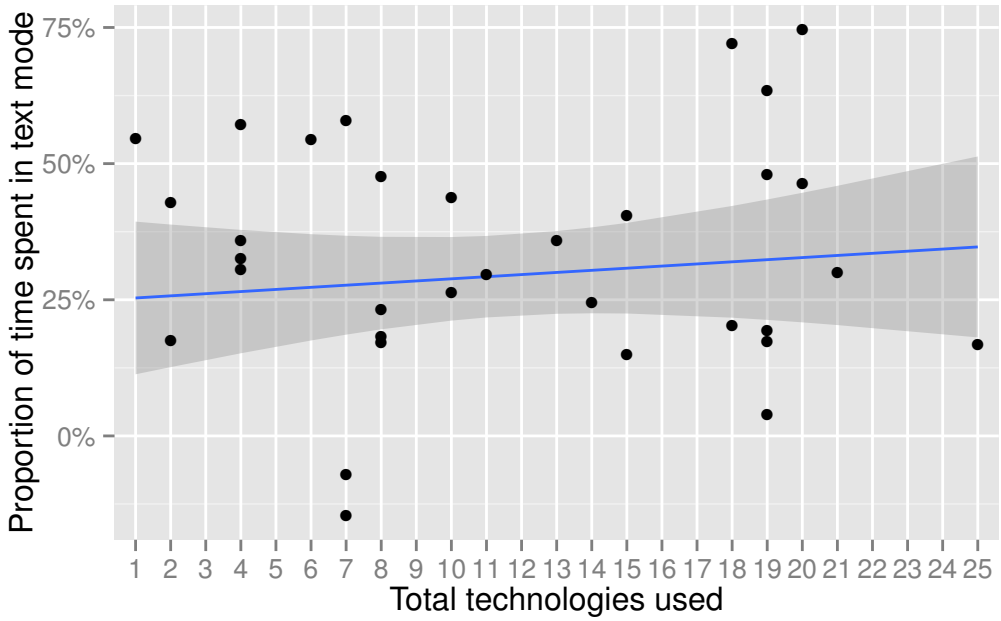


Figure 7.35: Use of text view versus by number of technologies used.

spread of both switches and time spent in text view across all questions. Similarly, we can look at how many switches were made on each task, shown in Figure 7.34. These are reasonably consistent across tasks, with one notable surprising exception: the majority of participants (18, or 55%) did not switch out of the text view for Task 4, which asked participants to describe a program first presented in the text view. This was the only task for which the majority of participants did not switch views at all. It may be that participants simply did not think to switch views; an alternative possibility is that they find text more useful for comprehension, but the tiled view more helpful for editing code. We will examine these possibilities more closely when analysing the freeform text responses from participants.

We wondered whether more-experienced participants would use the text view more than less-experienced participants, particularly in light of the results we found regarding engagement earlier. Figure 7.35 plots total technologies used against time spent in text view. We do not see any

significant trend in this data, suggesting that experience is not a strong predictor of use.

7.5.7 Freeform responses

We gave three prompts to participants at the end of the questionnaire, with text areas for their responses. We asked:

- What did you like about this system?
- What did you dislike about this system?
- If you have any other comments to make, please write them here.

These questions were taken directly from model questionnaires available in the department.

Participants could write arbitrary text in response to these questions, or nothing at all. Most participants elected to fill in the first two fields, and many wrote comments in the third. We coded participants' responses to the like and dislike questions and show the distribution in Figures 7.36 and 7.37.

Figure 7.36 shows the distribution of coded responses to "What did you like about this system?". Participants could mention multiple topics and be coded for each of them. Any mention of the relevant terms or synonyms was coded into that category. The figure shows all topics that were mentioned more than once. The meaning and raw count of the codes are:

- **Appearance** (3): some participants described the system as "looking nice" and similar terms.
- **Colour** (4): participants said they liked the colours and colour-coding.
- **Errors** (11): participants mentioned error reporting in the tiled view.

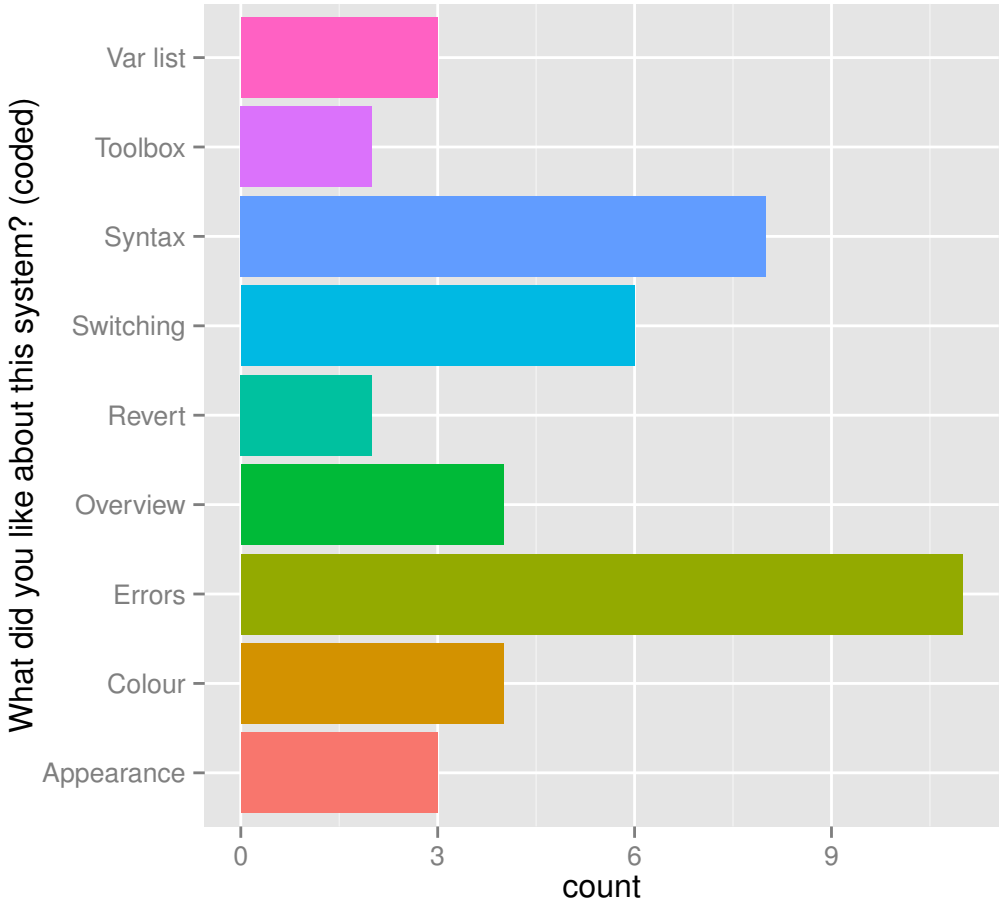


Figure 7.36: Coded participant responses to "What did you like about this system?" Any point with more than one mention is included.

- **Overview** (4): participants noted that they felt the tiled view gave a good overview of the code.
- **Revert** (2): participants liked the offer to revert textual code to the last-known-good version when there was an error and they tried to switch views.
- **Switching** (6): participants said they liked the ability to switch between the views.
- **Syntax** (8): participants noted the tiled view as helping to deal with syntax.
- **Toolbox** (2): participants identified the toolbox of available tiles as helpful.
- **Var list** (3): participants mentioned the list of variables in scope given when clicking the arrow on a variable tile.

The most common response was that participants liked the error reporting described in Section 7.3.1 and found it helpful. Sentences coded in this category were:

- Clearly shows where the problems are when you click run. The red highlighting is good
- It could identify the error and it also gave specific detail about the error
- Error finding and handling looks great, is very easy and straightforward.
- It points out the error pretty quickly
- Easy to find errors and like being able to move the tiles.

- it is extremely easy to check your syntax when using the tiled view as well being extremely useful for bug testing as the program would accurately highlight each incorrect variable.
- Makes obvious errors even more obvious.
- the thing i likes about the system was it made it very easy to find and fix errors.
- Also the flashing errors were among the biggest help as it reduced time in searching for the errors.
- Can easily find out where the errors are.
- The way of showing errors was amazingly helpful.

The most interesting response for this experiment was whether participants liked switching views, which six participants identified explicitly. Comments included:

- Text and tile view switching was great.
- the tiled view is useful in that it allows you to do quite a lot with just a mouse, provides a nice graphical display of the textual. switching between textual and tiled is useful.
- I liked that the tiled view was in some ways similar to the code view.
- I think it could be useful if you have 'scroll blindness' to change your view.

All quoted responses are unedited.

"Scroll blindness", an inability to notice details of code through having been looking at it for too long, was an aspect we had not considered that might support integrating similar functionality into editors for more

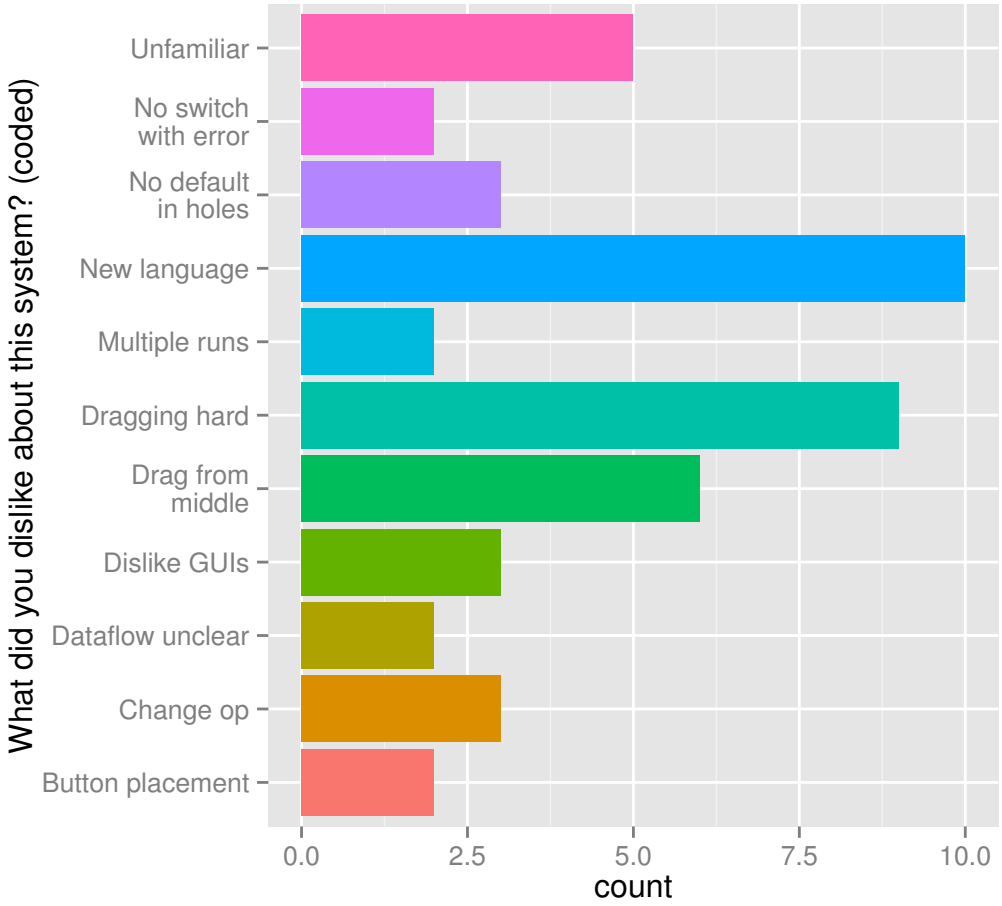


Figure 7.37: Coded participant responses to "What did you dislike about this system?" Any point with more than one mention is included.

advanced programmers as well. Further experimentation will be required to investigate this possibility.

Figure 7.37 shows the distribution of coded responses to "What did you dislike about this system?". Again, participants could mention multiple topics and be coded for each of them, and any mention of the relevant terms or synonyms was coded into that category. The figure shows all topics that were mentioned more than once. The meaning and raw count of the codes are:

- **Button placement** (2): participants said they did not like the placement of user interface buttons or found the placement unnatural.
- **Change op** (3): participants said they found changing the operator on an operator tile overly difficult.
- **Dataflow unclear** (2): participants were uncertain what the flow of data or execution was in the tiled view (where multiple blocks had no particular ordering).
- **Dislike GUIs** (3): participants said they disliked GUIs or graphical interfaces in general.
- **Drag from middle** (6): participants mentioned annoyance when they tried to drag a single tile out of the middle of a stack of tiles.
- **Dragging hard** (9): participants said they found the drag-and-drop too sensitive, or that tiles did not go where they wanted.
- **Multiple runs** (2): participants ran graphical dialect samples again without terminating the previous run, causing flickering.
- **New language** (2): participants disliked aspects relating to using a new language, or wished to have received further instruction on it before the experiment.

- **No default in holes** (3): participants wanted there to be a default value in holes inside newly-created tiles. Participants mentioned both numbers and variables as suitable defaults.
- **No switch with error** (2): participants disliked being unable to switch views while there was an error in the code.
- **Unfamiliar** (5): participants identified the interface (not the language) as unfamiliar to them.

The most common dislike was that the drag-and-drop was too sensitive or insensitive or did not do what participants wanted. Responses in this category were:

- Drag n drop was interesting but I wouldn't like to do it regularly.
- Also when selecting segments of code in the tiled version there feels like there is a frustrating lack of precision to it.
- difficulty in moving things in the tiled view
- Moving blocks around was difficult/frustrating
- The tiled version was a bit fiddly to bring out a variable or something. [participant included in "other comments": Once I found out you could remove one thing by dragging the one below it and then dragging the one away, then joining the two parts again. I guess I might have missed that in the explaination.]
- have to be very precise with the mouse movement
- Moving the tiles around seems pretty confusing sometimes, when I tried to move a number into a tile, it just went above it instead of inside the tile.
- The interface was not well explained, it was hard to understand how to drag things out of a block. Once figured out it was far easier.

- Dragging new objects such as variables into sections was not easy, sometimes it was not clear on what to change in order for a new variable x to be. This required me to go into the code view and do it manually.

All quoted responses are unedited.

A common note here that we also observed during the experiment was that some participants found it difficult to drag a tile into a hole. The system required the mouse pointer to be inside the borders of the hole to consider the drag to be over the hole, and not just a portion of the dragged tile; the hole would be highlighted (yellow) when the pointer was over it and a tile was being dragged. We considered this standard behaviour for drag-and-drop and did not give it any significant design thought before the experiment. Our preliminary trials did not show this issue.

We have confirmed subsequently that the default behaviour of the standard interface widgets on Windows, Mac OS X, KDE, and GNOME conforms to this expectation; nevertheless, multiple participants had repeated difficulty with this action. It may be that this convention is in fact unintuitive and users need to learn it separately for each tool they use. Past human-computer interaction research [84, 59, 113] has found that point-and-click interfaces may involve fewer errors and be faster than drag-and-drop, although recent research with children [5] has shown that they may both expect and prefer drag-and-drop interfaces. Further work may be required; it may be interesting to adapt this tool to a purely point-and-click interface and perform the experiment again. We examined the user interaction data we collected in more detail to try to discover any trends in the data.

We analysed the actions participants took during the experiment to count these "mis-drag" events. We defined a misdrag as a drag and drop onto the background followed immediately by picking up the same tile in a subsequent drag event, without interacting with the system in any other way in between. We defined a "hole misdrag" as a misdrag where the tile was eventually placed into a hole. We defined an "unrealised misdrag" as

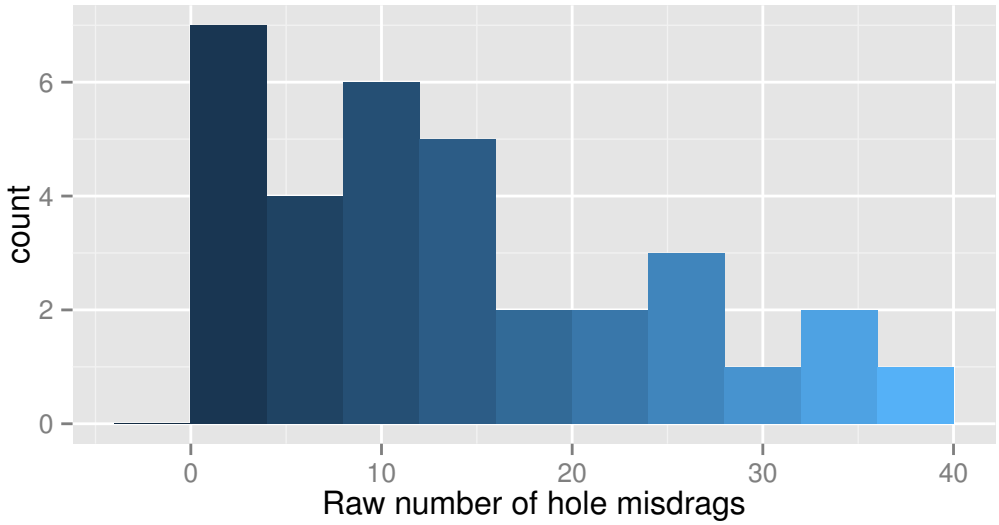


Figure 7.38: Distribution of the raw counts of hole misdrags.

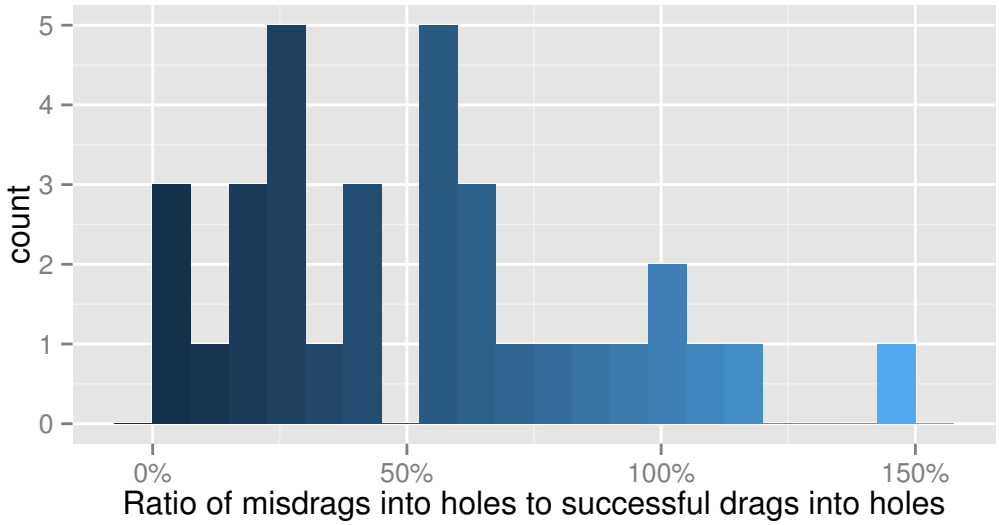


Figure 7.39: Distribution of the ratios of hole misdrags to non-misdrags.

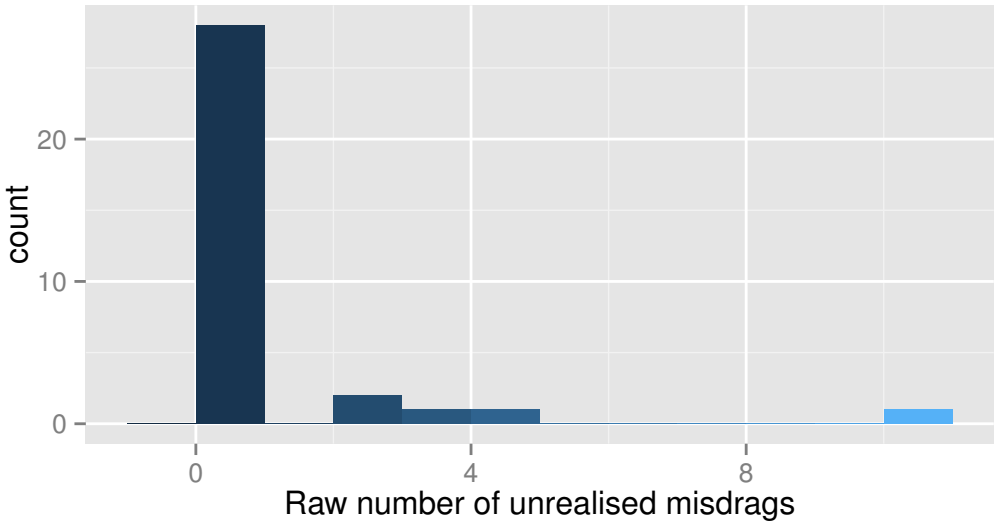


Figure 7.40: Distribution of unrealised misdrags.

one where the participant either tried to run the program or viewed the error overlay immediately after the misdrag, and so could be assumed not to have realised that the tile was not where they wanted. We also counted the total number of drags and the number of drops into holes that were not part of misdrags.

These numbers are by necessity approximate: participants' finger may have slipped, causing them to release a drag early, or they may have dropped the tile in a way that we did not count as a misdrag. It is likely that these counts overstate the true rate of misdrags; in our observation of participants most had no particular difficulty with dragging into holes or otherwise, and dropped elsewhere intentionally. With our metrics, the median number of hole misdrags is 10, which from observation we consider somewhat too high, but we believe that the overall metrics are a reasonable approximation of the phenomenon in relative terms. Figure 7.38 shows the distribution of hole misdrags. The minimum was 1, the first quartile 6, the third quartile 21, and the maximum 36.

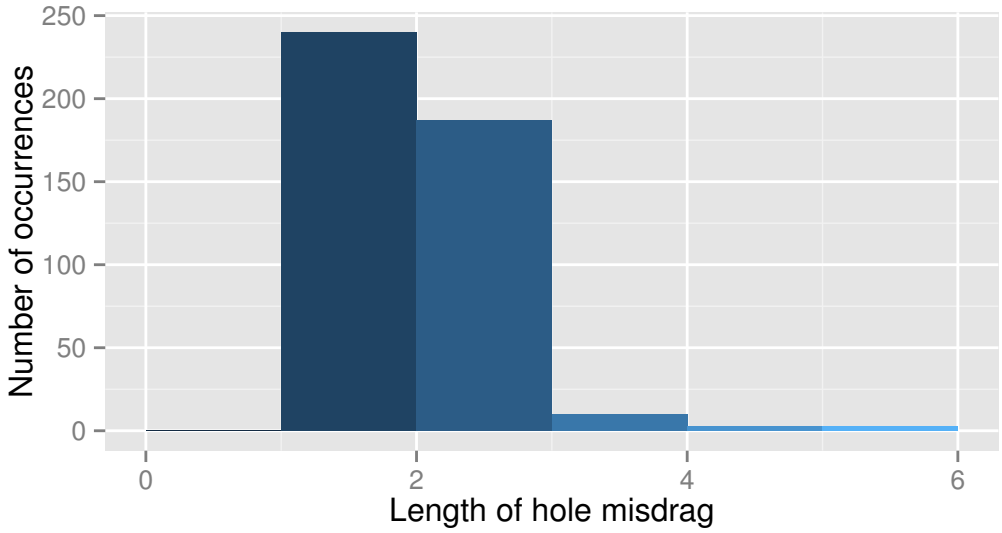
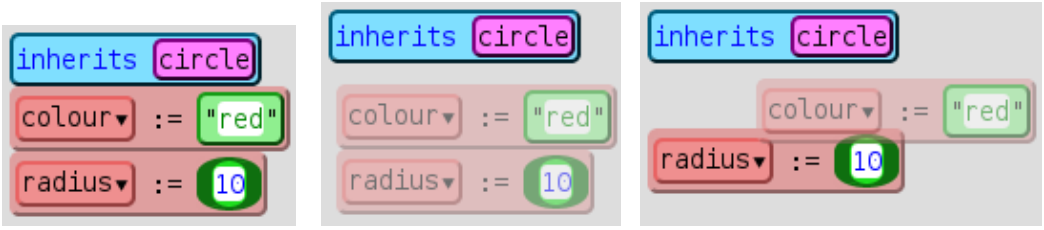


Figure 7.41: Distribution of the lengths of hole misdrags.

Figure 7.38 shows a long tail of participants having significant difficulty. This tail is illustrated more strongly in Figure 7.39 which shows the ratio of hole misdrags to drags into holes that are not misdrags. Five participants had more hole misdrags than successful hole drags, indicating substantial problems. If we look only at unrealised misdrags (Figure 7.40) we see that the substantial majority of participants (85%) had no such misdrags, while again about 15% had difficulty. One participant had ten unrealised misdrags, while four others had between two and four.

Figure 7.41 shows the number of misdrags before successfully placing a tile into a hole. The overwhelming majority of hole misdrags were resolved within two drags, with over half consisting of only a single misplaced drop. Again, a small number of participants took up to five steps to resolve the problem.

We asked participants how often they used a computer with a keyboard and mouse in an attempt to be able to control for past experience with these kinds of user interfaces. One participant indicated “Weekly” and all others either “Daily” or “Always”, so there was no ready correlation here. The



(a) A stack of three tiles. (b) Dragging two tiles off the bottom of a stack of three. (c) Dragging a single tile from the middle of a stack of three.

Figure 7.42: Screenshots of various operations with a stack of three tiles.

fact that most participants had at least 10 hole misdrags, and that some participants had debilitating difficulty, suggests that drag-and-drop may be a problematic paradigm for programming. We discuss this issue further in Section 7.7.

Six participants explicitly noted difficulty dragging tiles from the middle of a stack. The issue identified is that, given a stack of three tiles as shown in Figure 7.42a, dragging the middle (colour := `"red"`) tile will take the bottom (radius := 10) tile with it, as shown in Figure 7.42b (the "Solitaire" behaviour). Participants instead wished to remove only the colour := `"red"` tile to move elsewhere.

Our system does support this "drag from the middle" behaviour, as it was a frustration we similarly had in Scratch, but it is moderated by use of the Shift key: a shift-drag will move and remove only the immediately-selected tile, and not any attached tiles below, as shown in Figure 7.42c. We did not introduce this function to participants during the introductory tour of the system. The default behaviour is that a drag on a tile separates it from any neighbour above, but makes a new block of tiles including all those attached below the tile.

We believe this behaviour is generally sensible, as it ensures consistent behaviour in the system: if dragging never took the tiles below it would not be possible to move stacks of tiles at all, as can presently be done simply

by dragging the topmost tile of the stack. One participant discovered the shift-drag behaviour unprompted during the experiment and made use of it. While we continue to believe this default is correct, it is possible we should have introduced the shift-drag operation in the tutorial to mitigate these complaints. We preferred to leave out any topics that were not central to the tasks at hand to avoid overburdening participants with information, which may have been an error in this case.

7.5.8 Threats to validity

This study has the attendant threats to internal, external, and construct validity accompanying its size, sampling, and population. The overall results are suggestive of future research avenues, but not indicative more broadly.

In particular, we note the following threats:

- Our sample is drawn primarily from volunteer undergraduate students in the School of Engineering and Computer Science at Victoria University of Wellington, and may be both unrepresentative and likely to respond to the study more favourably than others.

- Most participants had meaningful past programming experience, and so are not the true novices that Grace aims to support. The experimental design did not seek to measure or impart learning, but relied on that past experience.

- The coding of freeform responses was performed solely by the experimentor.

- Both the pre- and post-questionnaires were completed in the experimental room with the experimentor present, although not watching their responses. Participants may have been influenced by this situation to rate their experiences more favourably than otherwise.

- When participants came in pairs they may have been influenced to hurry if they observed the other participant completing before them, which may affect their performance or responses.
- The timing limitations on tasks we imposed to make the experiment practical to run may have resulted in participants feeling pressured for time and making mistakes they would not otherwise have made.
- Many results rely on participants' self-reporting, which may be inaccurate, inconsistent, or false, in structured or unstructured ways.

7.5.9 Summary

Our experiment showed that participants generally (76%) enjoyed using our system and that other measures of engagement were high, supporting the use of similar features in development tools. We also found that enjoyment was lower for more experienced users, suggesting that Tiled Grace and similar interfaces may be most appropriate within the novice-user market that Grace aims for. Participants rated the system as less fun in direct correlation to the number of technologies they reported having used, but there was no meaningful trend in use of the different views.

The error reporting in the tiled view, described in Section 7.3.1, was very well received and participants found the overlay drawing attention to error sites helpful. 79% agreed that finding errors in the code was easy, and freeform feedback included remarks that the interface aided speed finding errors. This overlay does not entirely require a tiled view of code, and it would be interesting to investigate a similar interface for other editors.

The ability to switch views was well-received, though participants varied widely in their use of it. Several participants commented that they found having two equivalent views of their code to be helpful in itself, even if they did most of their editing in one view or the other.

We showed in the experiment that participants found having a more conventional textual view of code available to be helpful, even if they liked

to edit the graphical version, and that they liked to have the graphical version available for an "overview" even when they were editing textually. Some also commented that they found the direct equivalence between the two views of the code to be helpful.

Participants also noted features afforded by the tiled view, such as colour coding, a toolbox of available methods, lists of variables in scope, and direct indicators of the definition or usage sites of variables and methods to be helpful. Several of these features could be incorporated into conventional editors.

7.6 Comparison with related work

7.6.1 Scratch

Scratch [162] is a wholly visual drag-and-drop programming environment with jigsaw puzzle–style pieces, aimed at novices and children. Scratch is purely visual: there is no textual representation of Scratch code at all, and some tiles in the system take advantage of layout tricks that could have no corresponding textual version. A Scratch program is able to talk exactly about the graphical microworld the system presents, and no more, so eventually a student must move on and use a "real" language when their programs become more complicated.

Tiled Grace avoids this immediate need by allowing arbitrarily complex programs and always providing an equivalent (and coequal) textual representation for a program. A student may gradually use the textual editor more and more until they are confident in moving to a more standard environment, or even continue to use Tiled Grace indefinitely without expressive penalty.

In our experiment we found that participants appreciated having a conventional textual view available, even when they preferred to edit graphically. We believe from these results that including a bijective textual

representation of code is helpful in visual editors and that Scratch and others should consider incorporating such a representation.

A number of aspects participants in our experiment disliked were common across most tile-based editors including Scratch, notably finding dragging to be a chore, which conformed with our own experience in Scratch. We noted in Section 7.5.7 that some human-computer interaction research has found drag-and-drop to be a suboptimal interface paradigm, and that point-and-click may be superior. We discuss this issue further in relation to both Scratch and Tiled Grace in Section 7.7.

Scratch includes one notable feature that our system does not: when a Scratch program is running, each tile is highlighted in turn as it is executed. The idea behind this highlighting is to make the flow of control clear, particularly the fact that multiple threads of control flow are executing simultaneously in a Scratch program. Our system does not include such highlighting; primarily, this omission is a technical limitation of the ECMAScript environment and the generated ECMAScript code from Minigrace. Because Minigrace generated ECMAScript code, and browsers execute ECMAScript in a single-threaded and blocking fashion, we could not provide any visual update from a program until it completed. Alternative code generation techniques allow solving this problem, but we did not implement these in Minigrace.

Meerbaum-Salant, Armoni, and Ben-Ari [123] found that Scratch programmers may learn "bad habits" from tile-based programming: notably, a very fine-grained and bottom-up style, counter to what is generally seen as good practice. The authors observed that students would begin to build a program by dragging tiles they thought they might use out onto the workspace, and then try to put them together, and would often have unnecessarily many partial solutions to a task joined together awkwardly. In designing our tool we hoped to ameliorate these problems with the parallel textual view, and with the ability to move forward in the same language. Our experiment showed that most participants did use and appreciate both

views, but a pedagogical study is required to determine whether these "bad habits" persist with our approach.

7.6.2 Blockly

Blockly [10] is very similar in ethos to Scratch. Blockly runs in a web browser and incorporates language variants (what we call *extensional dialects*), but in mimicking Scratch also has no editable textual format. The same limitations apply to Blockly and Scratch.

Our experiment found that a bijective textual representation of source code was helpful to users. Blockly supports exporting code to a number of languages, but these exports are not bijective: Blockly code exported to Python behaves similarly to the original Blockly code, but cannot be modified and reintroduced into Blockly, and there is no explicit indication of which parts of the visual representation correspond to which parts of the textual representation. Tiled Grace makes this connection clear through animation, and participants indicated that they liked and understood the correspondence. We believe that making the connection between the two formats explicit is important for participants transitioning from visual to textual programming.

7.6.3 Alice

Alice [31] is a 3D microworld language manipulated by drag-and-drop. Alice uses drag-and-drop both for putting 3D models into the microworld and for editing logic; there is no interaction with concrete textual syntax. Our system does not include a persistent microworld and does not permit manipulating the worlds it does present (through dialects) other than programmatically. Alice programs can only interact with this microworld and cannot express tasks outside of it.

Event handlers on Alice's in-world objects are put in place through drag-and-drop in a similar way to our tiled view, but there is no other

means of modifying them. One notable difference in the way the drag-and-drop logic behaves compared to ours is that Alice code is forced to omit even temporary syntax errors: when placing an "if-then" into the code, the programmer cannot move on to any other task before they fill in the condition. Leaving an empty space for any length of time is not permitted. We consider such a prohibition to be a reasonable option, but note that it obstructs other idioms. In particular, one way of programming with both Tiled Grace and Scratch is to drag multiple tiles from the toolbox onto the workspace when knowing that they will be needed and then assembling them once all are available, avoiding back-and-forth trips to the toolbox. We are unsure which approach is best, but a future experiment could use both.

Powers, Ecott, and Hirshfield experimented with transitioning from Alice to Java (with BlueJ) in an introductory programming course [158]. They observed that many students

> were intimidated by the textual language and syntax, and seemed to have a difficult time seeing how the Java code and the Alice code related

even when working with exactly corresponding Alice and Java code. The authors identify this problem as a potential issue for visual programming languages for novices in general. Our system aims to ease this transition to conventional syntax by explicitly showing how tiled and textual code relate. In addition, Tiled Grace was explicitly designed with a permeable barrier in mind: a user is not forced to move entirely into the textual world at once, but can move back and forth and acclimatise gradually.

7.6.4 Droplet

Droplet [40] is a combined visual-textual editor that treats block-oriented visual display as "syntax highlighting" for text, and can transition between

the two displays. Droplet postdates our work, and has both limitations and extensions relative to Tiled Grace. Droplet uses the same kind of jigsaw-puzzle pieces as Scratch, with the same rules about compatible shapes, and so the same limitations around type compatibility that we encountered. Droplet allows laying out a tiled program only as an exact match to the textual code, but potentially supports multiple entirely different languages, a more general range than the Grace dialects supported in our system, but with the attendant complexities discussed in relation to dialects in general in Chapter 6.

7.6.5 Calico Jigsaw

Calico Jigsaw [9] is a drag-and-drop visual language for the multi-language Calico development environment. Execution of Jigsaw code is based on translation to Python, and Jigsaw programs can be exported to Python code, while some Python programs can be imported into Jigsaw code; the Jigsaw syntax is not itself similar to Python's, however. Unlike Tiled Grace, Jigsaw's code export is to a complete textual program and does not provide a direct user-visible mapping to and from the corresponding tiles. The transition is intended to be one-way and one-time.

Our experiment found that users appreciated the direct connection between equivalent pieces of code given by our visual transition and the ability to switch to a different view of the code and back within a single editing session. Calico and Jigsaw do not attempt to provide such a transition or ability; instead, the authors intend to permit translation to multiple textual languages for comparison (although no others are supported at present), rather than having users remain in a single language as Tiled Grace aims to enable.

7.7 Future work

While Tiled Grace includes simple type checking to prevent common errors, we would also like, if possible, to signal these errors and others in advance by some feature of the tiles themselves. Scratch and Blockly use a "jigsaw puzzle" approach, where only tiles that "fit" can be placed in any given position, but this is not complete; some tiles may be the correct shape but still not allowed (in Blockly) or not sensible (in Scratch) in a particular location. We plan to investigate variations of shape, colour, and other attributes to indicate these restrictions in advance of a user trying to perform the task in the program.

The graphical design of the tool would benefit from further consideration. The current colouring of tiles is essentially arbitrary, while the overlays are functional but may sometimes obscure important areas of the program. The colours used in the interface are not ideal for conveying semantic meaning and should only be used in addition to other indicators. We intend to create a more consistent design and investigate variations to the overlay displays such as transparency and alternative pathfinding.

At present our view-switching system only allows programs with no current errors to be switched to the other view. In part this is for technical and representational reasons: some erroneous code has no clear representation in one view or the other. Some errors, however, could be seen on both sides of the divide, and users may benefit from being able to look at them in two different ways. In future we may allow at least some classes of error to pass through the barrier between the two views, but establishing which errors are suitable, both technically and in terms of not creating additional confusion for the user, is design work remaining to be performed.

In Section 7.4 we outlined experimental results that suggest modifications to the system and new avenues of investigation. In particular, we noted that some participants had substantial difficulty with drag-and-drop, and many noted some degree of difficulty, suggesting that drag-and-drop

may not be the most suitable interface paradigm for programming. After the experiment we examined the human-computer interaction literature and we discussed in Section 7.5.7 that some human-computer interaction research has suggested that drag-and-drop is a problematic interaction mechanism in general, and that a point-and-click arrangement is less error-prone. Further research is required to determine the impact of this issue in relation to visual programming; in particular, given the target markets of Scratch and, to a lesser extent, Grace, and the recent work of Barendregt [5] on children's interaction with various interfaces, more structured classroom-style experimentation using child participants may be in order.

Another interesting piece of future work is the interaction of this sort of system with touch-screen devices; drag-and-drop is naturally a common paradigm on such devices. TouchDevelop was designed for use on touch-screen devices but primarily uses a point-and-click interface, and it would be interesting to see whether our results on the difficulty some users had with drag-and-drop programming hold on such devices.

Our experiment showed that users did appreciate and find useful the combination of tiled and textual views. Grace is an object-oriented language and code in the experiment was primarily either OO or procedural, with our focus on use in the area of introductory programming where Grace and Scratch sit. It would be interesting to explore similar interfaces in other paradigms or areas where users may have syntactic difficulty, such as mathematics, logic, formal methods, and type theory.

7.8 Conclusion

Tiled Grace is a graphical editing environment for Grace, inspired by visual program editors such as Scratch. Tiled Grace visualises code as nested "tiles" that can be manipulated by drag-and-drop, eliminating many syntax errors. Tiled Grace's tiles always correspond exactly to Grace's textual syntax, so that users become familiar with the textual syntax while dragging and

dropping tiles. The user can switch between the tiled and textual view, with the program editable in both forms. Tiled Grace can also visualise relationships between definitions and uses of variables and methods.

We conducted an experiment to measure user engagement with Tiled Grace, and how people would use the tiled view, view-switching and error-reporting provided by the tool. We found that engagement was high, although lower in more experienced users, and that in particular the error reporting we designed for the tiled view was very well received and used. Participants varied dramatically in their use of the tiled and textual views, but many found the ability to switch between them helpful, suggesting that it is beneficial in itself.

Chapter 8

Implementation

This chapter discusses Minigrace, our prototype compiler for Grace, and the accompanying external libraries we wrote; it provides context, historical information, and reflections only, including implementation details. The chapter will be structured as follows: the next section provides an overview of Minigrace, then Section 8.2 presents the architecture and implementation of the compiler. Section 8.3 gives and reflects upon the development history of Minigrace at a high level, chronologically from the earliest experiment onwards, with a particular focus on self-hosting a compiler for a developing language. Section 8.4 describes the external libraries we built to work with Minigrace. Finally, Section 8.5 notes the contributions of other developers to Minigrace.

8.1 Overview

Minigrace is a compiler for Grace, supporting most of the specified language. Minigrace is written in Grace itself and has been developed in a self-hosting fashion from a very early stage. The compiler targets both native code (via C) and ECMAScript (JavaScript) and is able to compile itself to either target.

Minigrace comprises 14KLOC of Grace, 6KLOC of C, and 2.6KLOC of

ECMAScript. The revision history includes over 2,000 commits, primarily ours but over 200 by other authors, which contributions are described in Section 8.5.

The source code of Minigrace is freely available from `https://github.com/mwh/minigrace`. Minigrace is distributed under the GNU GPL version 3 [54] or any later version. We developed Minigrace on Linux-based systems, but it has been known to work on other POSIX-compatible systems including NetBSD and Mac OS X. Build and installation instructions are included with the source.

The Minigrace executable by default generates C code from the given Grace code, compiles and links that code, and executes the resulting program. To generate ECMAScript code instead (saved to *`modulename`*`.js`), the user can give the `--target ecmascript` option. To refrain from executing the program, they can give the `--make` option.

Because Minigrace can generate ECMAScript from Grace code, and because Minigrace is itself written in Grace, the compiler can be compiled into ECMAScript and executed in a web browser. We discuss this backend further in Section 8.2.5. The web interface is available at `http://michael.homer.nz/minigrace/js/`, and supports exactly the same code that the native backend does, although some platform-specific libraries exist on both target platforms.

The complete revision history of Minigrace is included in the auxiliary data included with the thesis (see Appendix A).

8.1.1 Extensions and limitations

Minigrace supports most of the specified Grace language described in Chapter 3, but has some limitations and extensions.

- Minigrace includes syntax for list collection literals written [1, 2, 3].
- Minigrace supports "elseif" clauses in conditional requests. The conditions of these clauses are written in braces because they are not

necessarily evaluated. (Earlier versions of Minigrace used parentheses instead, which was semantically incorrect.)

- Minigrace permits an extension flag to set the default visibility of object fields for an entire module, using either `-XDefaultVisibility=public` on the command line or `#pragma DefaultVisibility=public` at the top of the module. The Minigrace source code uses this feature to cope with changes to visibility rules made during development.
- Code interpolated into strings cannot contain a "}" character. The reasons for this limitation are given in Section 8.2.1.
- Numeric literals in non-decimal bases may only be integral.
- Inheritance across module boundaries works only for top-level classes and methods.
- Run-time type checks are rudimentary and use only method names, where present at all.
- The `if()then()else` method cannot be overridden, and no method name can have a first part of "`if`".

These extensions exist primarily for historical reasons touched on in Section 8.3, while the limitations are unimplemented functionality.

8.2 Architecture

Minigrace follows a conventional compiler architecture: first program text is lexed into tokens, then tokens are parsed into a(n abstract) syntax tree, additional processing such as identifier resolution occurs on that tree, and finally executable code for the target platform is generated. This section gives an overview of each phase of the compiler and any interesting aspects of them.

8.2.1 Lexer

Minigrace reads source code into a string and then iterates over the characters to build up tokens. Each character is examined to determine what kind of token it may be a part of, given the existing state of the lexer, accumulated until the end of the token, and then stored in sequence annotated with the kind of token, the corresponding source text, and the location in the input. This phase is known as lexing or tokenisation.

The lexer examines each character using to determine its Unicode character class, and maintains a `mode` recording which kind of token is in progress. When a character that cannot appear as part of the current mode occurs, either the current token ends and a new token begins or a lexical error is reported. Some modes, notably the mode for a string literal, accept all characters until the closing quote. Escape characters are replaced with their final values during lexing, so all subsequent phases see the final representation of strings.

The lexer, as the first piece of self-hosting code written, has a large amount of legacy detritus. Early versions of the system supported only `if()then`, with no `else`, and the lexer included code with the form

```
var done := false
if (condition ...) then {
  ...
  done := true
}
if (done.not) then { ... }
```

for a long time. The lexer was also by far the slowest phase of the compiler, primarily because it examined every character with multiple method calls each. To ameliorate this issue to some extent we implemented special patterns in platform-specific native code within the Unicode module (Section 8.3.3), that could test the character against multiple classes or characters at once.

Limitations

The Minigrace lexer is incorrect with respect to interpolated strings. When interpolations were added to the language we incorporated support for simple interpolations: when a "{" character appears inside a string literal, we immediately end the string token and insert an opening parenthesis token before it, and then insert a "++" (string concatenation operator) token and another opening parenthesis. The body of the interpolation is then tokenised as usual until a "}" character appears, whereupon the interpolation ends and tokens for ") ++ "" are inserted. This approach works for many interpolations, and in particular for the common cases where only a variable name is interpolated, but fails for more complex code.

An interpolation ends at the first "}" character seen, meaning that the interpolated code cannot contain that character anywhere without causing bad tokenisation; it is not possible to interpolate an if-then-else, for example. This limitation has a significant architectural component: in order to know when a "}" ends the interpolation, and when it is a part of the interpolated code, the lexing and parsing phases must be interleaved to some extent. The early Minigrace architecture in the initial Parrot-based versions did not support this sort of interleaving, and adding that behaviour would require a substantial rewrite of both phases, if not a major architectural change.

8.2.2 Parser

The Minigrace parser is a reasonably standard recursive-descent parser, with additional global state dealing with Grace's indentation rules. The parser is the largest module in Minigrace, primarily because of the large volume of code for reporting syntax errors.

Abstract syntax tree

The parser returns a list of nodes defined in the `ast` module. The AST in fact sits somewhere between a concrete parse tree and a true abstract syntax tree.

The available nodes are defined as classes in the `ast.grace` file; each object contains at least a kind field, indicating which node it is, and a value field containing the primary contents of the node (which varies by node). All also support an accept(visitor : ASTVisitor) method, implementing the Visitor pattern; a map(block)before(beforeBlock)after(afterBlock) method, implementing a tree map with the given setup/teardown code; a pretty(depth : Number) method returning a textual representation of the AST starting from that node; and a toGrace method returning Grace source code corresponding to that node. Some nodes have other fields or methods specific to their role.

8.2.3 Identifier resolution

The identifier resolution pass turns all bare identifiers into fully-qualified expressions: x may become **self**.x, **outer**.x, **outer**.**outer**.x (and so on), or may remain as x if it refers to a local variable. In this way the code generation passes need only deal with method requests having explicit receivers and with local variables. To aid these later passes further, dialect methods are resolved to a special receiver dialect that the code generators can treat differently than other receivers.

Identifier resolution rebuilds the syntax tree, replacing identifier nodes with fully-qualified expressions, and replacing their parent nodes with new nodes containing the replacement, up to the top of the tree. This entirely-new syntax tree is what subsequent passes receive, and the old tree from the parser is discarded. The map(block)before(beforeBlock)after(afterBlock) method mentioned in Section 8.2.2 performs the rebuilding, with a block of code to modify each node and blocks to set up and tear down

scoping rules.

In order to determine where an identifier should be resolved this phase needs to track the nesting of scopes. For code within the module being compiled this tracking is simply a stack of maps from method and variables names to metadata about them. The metadata includes what kind of binding this name is (**def**, **var**, **method**, etc.), as well as information necessary for inheritance.

For identifiers obtained from other modules the task is more complicated. Outside code becomes involved in two ways: by inheriting from an object defined in another module, and from the dialect. Both are handled by loading in compilation metadata cached earlier, which is inserted into the correct parts of the scope hierarchy.

When Minigrace compiles a module it generates an additional file `modulename.gct`. These `.gct` files list the methods and classes defined in the module, and the methods available in any object that can be inherited. All dialect methods are added to the outermost scope at the beginning of the module. Any imported module has its import name bound to all of the imported metadata, so that inheriting from a method inside that module is resolved correctly. When inheriting, either from a method in another module or not, all methods of the inherited object are inserted into the local object's scope.

The `.gct` file includes further metadata about the module itself, and in particular the list of imported modules and the path used when compiling the module are both necessary for importing. In the ECMAScript backend, accesses of a `.gct` file are transparently redirected to a local cache of the contents in a ECMAScript object called `gctCache`, as ECMAScript does not provide filesystem access in web browsers.

The identifier resolution pass also performs another task: finding any matching blocks (Chapter 4) — that is, blocks with a single parameter — and replacing the node with one remembering the pattern separately, and possibly now including multiple parameters. This task occurs during

identifier resolution as this pass is already rebuilding the entire syntax tree; adding an additional check and replacing a node has a very low additional cost at this point. Any new parameters bound must be known before examining code inside the block so that scoping errors can be reported correctly, so matching blocks are replaced before they are first examined for identifier resolution.

8.2.4 Code generation: C

Code generation to C is primarily syntax-directed: each node generates some C code representing itself, and somewhere asks any child nodes to generate code for themselves. All of this generation occurs in the `genc.grace` module, driven by the compilenode method, which delegates behaviour to various compileX methods for different nodes.

Some points, notably object and method definitions, require additional passes to arrange storage space or scoping details in advance, or to arrange circular references correctly. Other than these exceptions, most nodes cause between two and ten lines of C code to be generated for themselves and simply request the same for their children.

All compiled modules include a function representing the module entry point called Object module_*modulename*_init(). This function returns a pointer to the module object, and contains all top-level code of the module. When a module is compiled as the program entry point a standard C **int** main(**int** argc, **char** **argv) entry point function is generated as well. This function sets up internal structures that the runtime requires and meta-information about the compilation such as module search paths, and executes the module's init function. To permit both executing and importing the same module without recompiling in between, this function is declared with **#pragma** weak main so that only the correct version exists in the final executable. **#pragma** weak main gives the main function weak linkage, meaning that only the first definition of main found during linking

will be included in the executable. This pragma directive is non-standard and limits the portability of the generated C code.

When importing a module there are two cases: the module to be imported is an ordinary (static) module, or it is a dynamic module like the Unicode module we describe in Section 8.3.3. In either case a weak symbol Object module_*modulename* is declared, which will hold the module object once imported; this symbol is used as a guard to ensure that each module is imported at most once. In the case of a static module, an import simply resolves to calling the module's init function; a dynamic module must be dlopen(3)ed [79] and the init function symbol retrieved. In both cases, the result of calling the function is saved into the global symbol, and the body of the C code implementing the module is identical. Whether a module is to be imported statically or dynamically is determined at compile time by which of a `.gso` (dynamic), `.gcn`, or `.grace` (both static) file is found first.

The generated C code is aided by a library `gracelib.c`, which is statically linked into every program. The library includes helper functions for setting up objects and methods, structure declarations, the garbage collector, and implementations of built-in objects like numbers and strings. The most important part of this library is the callmethod function, which implements method lookup and execution. Each object has a pointer to its "class", which contains an array of method structures; calling a method requires finding the method of the correct name, obtaining the function pointer, and calling that function with the given arguments. A function implementing a method accepts five parameters: the receiver, the number of parts in the request for this method, a pointer to an array of the number of arguments in each part, the arguments given, and a set of flags indicating certain special behaviours. Both library methods and generated code use this same format for their method functions, as do native-code libraries.

Local variables in generated C code are declared as pointers to locations in reified stack frames. Each scope has its own stack frame object that

contains an array of slots for every variable that will be used in that scope; at the top of the scope's C code local C-level variables are declared aliasing those slots. The reified stack frames are necessary to support Grace's closure semantics: a block or object may continue to access a local variable in a surrounding scope after the scope has terminated, and multiple objects may access the same variable. Frames are garbage collected.

Local variables in a scope are aliased to slots in a stack frame at the beginning of the scope:

```
Object *var_x = &(stackframe->slots[3]);
```

Variables in surrounding scopes are obtained from the closure environment passed to the method:

```
Object *var_x = getfromclosure(env, 1);
```

In this way, generated code inside the body of the scope need not know where a variable came from: every variable is accessed the same way, with `*var_x` to dereference the pointer. A pointer to the closure environment is stored inside the object the method belongs to at the time it is created (solely to preserve the scope from garbage collection while the closure is alive). The environment holds a pointer to the surrounding stack frame as well as an array of pointers to slots in that frame (or surrounding frames) belonging to variables used in the inner scope.

8.2.5 Code generation: ECMAScript

Code generation for ECMAScript follows the same basic approach as for C: each node generates some ECMAScript for itself and has code for its children generated as well. Because ECMAScript has closures, first-class functions, objects, and string-keyed maps built in, the implementation and generated code is simpler.

Although ECMAScript includes objects, Minigrace uses native objects only for storage and identity. Method lookup in Grace has different semantics than in ECMAScript and some extra behaviours are possible, like

super calls; as a result, the implementation of an object includes a mapping of method names to functions, of field names to data, and of the object to its inherited part-object. A `gracelib.js` file contains the implementations of built-in objects again, as well as the helper functions used for code generation.

All method calls are directed through a callmethod function, just as in C. The function looks up the appropriate function to invoke and executes it in the context of the receiver object. All functions accept at least one argument, an array of the number of arguments in each part of a request, and can accept others for its actual arguments. Javascript has full support for variadic arguments and indirectly invoking any function, so this representation is much simpler than in C.

Modules are again represented by functions named after the module, but the object is pre-allocated by the import mechanism and the function invoked in its context. Because ECMAScript programs are running interpreted in the web browser all imports are transacted dynamically simply by calling the function if the module has not already been imported. This behaviour is different than when executing the compiler as native code: if an import path does not correspond to an existing module there will be a static error when running natively, but when running the compiler in ECMAScript no error will occur, and the compiled program will give an error at run time.

Because ECMAScript supports closures natively, local variables in Grace code are always translated directly into local variables in the generated ECMAScript code. Any blocks or methods lexically inside a scope will be inside the ECMAScript scope as well, and so be able to access the variables directly. To avoid name conflicts all variable names are prefixed with var_.

8.3 History

This section gives a chronological history of Minigrace, and some reflections on self-hosting a compiler for a new language.

8.3.1 Parrot to LLVM

We began developing a Grace compiler in April 2011. We were interested in leveraging existing tooling if possible, in particular the Parrot Compiler Toolkit [145] or LLVM [110, 97]. We chose to begin with Parrot for our first prototype in order to use its built-in support for defining grammars and accompanying code generation rules. We intended to implement a minimal subset of the language suitable for (potentially) building a further compiler, and termed this minimal implementation "minigrace".

We were rapidly able to implement a small but expressive subset of Grace. We began implementing a lexer and parser for Grace in Grace, and extended our Parrot language as we encountered new needs. After a few weeks we had produced a parser in Grace capable of parsing itself correctly, and needed to make a choice whether to continue with Parrot or not. We chose to move on to LLVM at this point as we had had some difficulties integrating the Parrot virtual machine's object model with Grace's, and because we were at that time considering optimisation work as a focus for this thesis. We wrote further Grace code that generated textual LLVM bitcode from the syntax tree our parser produced. "Bitcode" is what LLVM calls its intermediate representation; we generated textual bitcode, rather than using the preferred API, because from Parrot we could only work with text input and output. Approximately a month after the first steps with our compiler we had produced a self-hosted compiler targetting native code by way of LLVM.

The implemented language at this point was very small: variable, object, and method definitions, with special-cased loops but no real lexical scoping. Identifiers could consist only of lower-case letters, and there was no access

to files: only reading standard input and writing to standard output was supported. Some of these restrictions continued longer than intended, and code from this era still exists in current versions of Minigrace. We made no attempt to optimise either time or space usage of the compiler at this point, and it was both slow and memory intensive.

8.3.2 Self-hosting with LLVM

The compiler at this point consisted of a single Grace file (around 1850 lines) and a library of helper functions and structures written in C, which could be compiled with LLVM and linked with the generated code. The C library was around 600 lines when the compiler reached a self-hosting state.

The compiler had very heavy memory usage because we had made no attempt to optimise anything; in particular, every string literal in the program created a separate object every time that line executed, with these objects persisting indefinitely, and similarly for numbers. Our first task with this early version was to reduce memory usage, which was already impractical; we managed to reduce usage by 95% when compiling the compiler itself through interning strings, numbers, and method structures.

Because Minigrace was self-hosting we required a "known-good" version of the compiler to use for building new versions. At first this version was the Parrot-based implementation, and then a native-code version generated from the first self-hosting version. Maintaining these versions was sometimes a challenge, particularly when introducing breaking changes to the language. The first versions of the compiler had no conception of string escape sequences: a backslash in a string literal meant a backslash, while quotation marks needed to be alternated to produce a quoted quote. The Grace language required escape sequences, and we would find them very useful for writing a compiler, but introducing support for them would break our existing code that used backslashes for their literal presence. To

introduce the new feature required a careful dance: first we needed to add support for a backslash inserting the following character (but not itself) literally, increment the known-good version, double all existing backslashes in the compiler, and increment the known-good version again immediately after. This case was the first of many where we would encounter difficulty self-hosting a developing language.

To aid in future updates to this known-good version we added infrastructure to build them automatically: our Makefile would specify the version we required as a version-control hash, and depend on an executable of that name being available. A script would check out a given old version of the compiler and generate those executables on demand. This approach was less fragile than maintaining these versions manually, as we had been doing to date, but also caused us other problems in future.

A further issue we had from self-hosting in this way was that we would at times introduce an error we did not detect: the compiler would build correctly using the known-good version, and the version so generated would itself compile the compiler, but this final version would have incorrect behaviour because of a non-fatal bug in an earlier version. To combat this issue we introduced a three-layer compilation: when we built the compiler we would first build the compiler using the known-good version, then with the version we had just built, and then again with the most recent version built. With no bugs, the last two versions would be identical, but code generation bugs could cause misbehaviour with any fewer number of passes. This approach made building the compiler drastically slower, but improved correctness.

Another difficulty we faced during this stage of development was the language itself: because the language definition was changing, at times the language the compiler implemented would be obsoleted overnight. Integrating the changes was sometimes a challenge, particularly when they altered the semantics of existing code that we had included in the compiler. This situation was an especial problem because of our multi-phase build:

code we wrote in the compiler must function on both the older known-good version and the version described by the compiler itself. At times we could make code work on both versions simply by supporting both for a time, but for semantic changes to existing concepts we could not. To that end we introduced a further hack: a method runonnew()else that ran one block of code on a "new" version, and another on an "old" version, determined by the version control hash contained in the executable. We would then only need to support both syntactic forms of any change for a time, until we had flushed the old version out of use.

8.3.3 Unicode

The Grace language specification has always included the phrase "Grace programs are written in Unicode", without further detail. We interpreted this statement to mean that identifiers could use non-ASCII Unicode characters; enquiries as to what exactly "written in Unicode" meant in concrete terms did not lead anywhere. We elected to support program source encoded in UTF-8 and to allow Unicode letters and numbers as part of identifiers.

To support Unicode identifiers our compiler needed to know whether each character it looked at was a letter, number, or something else. There are existing libraries providing advanced Unicode support, notably ICU [77]. We chose not to rely on any external libraries for mandatory features of Minigrace, however, because the majority of our existing user base would have significant difficulty following instructions to install outside dependencies.

Instead, we implemented a simple module wrapping the important details from the `UnicodeData.txt` file provided by the Unicode Consortium first in a C header and then in a C module implementing the linker-level interface of a Grace module with methods. Our module supported only the necessary details for our purposes, namely querying the character class of a codepoint, but could be shipped directly with the source

code of Minigrace.

Because the Unicode Character Database is quite large, linking our Unicode module into every program was time-consuming (particularly with LLVM, whose linker is very slow). We extended the compiler to support dynamically-linked modules, so that the Unicode module could be built once with the compiler and then dynamically loaded without recompilation or relinking until changed. This module and experience influenced our thoughts on module system design, which we addressed in Chapter 5.

8.3.4 Generating ECMAScript

While waiting for a specification update we began experimenting with ECMAScript[1] code generation, and were able to write a module as close to a direct port of the existing LLVM code generator, as well as the library code.

Because Minigrace is written in Grace, producing a compiler that ran in a web browser required only writing a compiler backend that would generate ECMAScript. We could then run the existing compiler through this backend and produce an in-browser compiler. Reaching this point took approximately a week's work: we consider that a good rate for porting to an entirely new platform, and an advantage for self-hosting the compiler. Many subsequent changes to the compiler, such as those to syntax and identifier resolution, required no platform-specific modifications at all, while for standard library changes and new structures or semantics we now had to maintain two different backends. We did not find that an overly

[1]ECMAScript [44] is the standardised version of the language commonly called JavaScript; the name JavaScript refers to Mozilla's (formerly Netscape's) proprietary version that includes additional extensions. Minigrace targets ECMAScript version 3, the common subset of Mozilla's JavaScript, Microsoft's JScript, and Adobe's ActionScript, and we use the name ECMAScript exclusively to be clear that we are not using any JavaScript-specific extensions. We also do not use any new ECMAScript 5 [45] features, but target the common language [45, Annex D & E] between the two versions.

onerous burden, particularly as ECMAScript was quite easy to add new features for because it had built-in objects and first-class functions.

8.3.5 Generating C

From the early stages of our LLVM code generation backend we had had a small C library providing helper functions. Our intention had been to move the majority this code into Grace eventually. Over time this library had grown substantially, however, and we were increasingly writing larger helper functions to avoid complexities of LLVM.

We had found textual LLVM bitcode to be difficult to work with, in particular when debugging code generation, and would often write a small C helper for what we would early on have generated bitcode for. We decided to implement an all-C backend that avoided LLVM entirely; we chose C at this point in order to be able to reuse the existing library code, which was approximately 2100 lines at this point.

The C backend reached parity with the LLVM code generator in October 2011, and for a time we maintained both in parallel. Within a month we switched the default backend to C, while continuing to update LLVM, but ultimately permitted LLVM to atrophy until its removal several months later. We found generated C code substantially easier to debug, in particular because we could insert code inline to correct or log state at the points we thought could be problems. As we were no longer planning to explore optimisation questions, we had no particular need for LLVM by this point.

8.3.6 Garbage collection

Memory usage had been an issue in Minigrace from the beginning, but we had been able to paper over the problems by reducing our allocations and interning objects. As the compiler grew, however, the memory used in compiling itself grew as well, and we were reaching the point of running out of memory on our development machine. We had earlier tried to

incorporate the Boehm conservative garbage collector, but found it not to work well out of the box with LLVM. At this point we simply wrote a basic mark-and-sweep collector of our own; while not very efficient, the collector reduced memory usage substantially and permitted the compiler to continue growing.

The collector maintains a list of all currently-existing objects, updated on every object allocation, a second list of GC roots, and a "shadow stack" of pointers to local references to objects. Garbage collection can be triggered at any attempted object allocation, and first clears a "visited" flag on every object before proceeding through the GC roots and instructing them to mark themselves and all objects reachable from them recursively. Finally, the collector examines all the live objects and frees any that were not reachable from any root. This approach is essentially the standard mark-and-sweep technique.

Writing the garbage collector exposed a number of existing bugs in our library and code generation, mostly involving buffer overruns. We were able to improve the quality of the compiler as a whole accordingly.

8.4 External libraries

We wrote some additional libraries to be used with Minigrace, but not part of the compiler itself. In this section we briefly describe those libraries and their interesting aspects.

8.4.1 Grace-GTK

Grace-GTK [69] wraps the GTK+ widget system in Grace modules. Grace-GTK consists of a Python script to parse the header files of GTK+ and its associated libraries GDK and Cairo, and then generate C code wrapping all GTK+ objects in Grace objects and GTK+ methods in C code conforming to the ABI for Grace methods. These modules can then be imported exactly

as any other dynamic module. A complete, albeit simple, program is:

```
import "gtk" as gtk

def window = gtk.window(gtk.GTK_WINDOW_TOPLEVEL)
window.title := "Hi!"

def button = gtk.button
button.label := "Hello, world!"

button.on "clicked" do { gtk.main_quit }
window.add(button)
window.on "destroy" do { gtk.main_quit }

window.show_all

gtk.main
```

This program displays a window with a single button saying "Hello, world!" and terminates when the button is clicked. To the programmer this code looks like any other Grace code, with field accesses and method requests on objects exactly as usual. The names and overall structure match the underlying object-oriented design of GTK+.

Grace-GTK supports a large but eclectic subset of the complete GTK+ interface. Methods and classes are generated when the Python script understands how to coerce a type in the header file into a Grace representation and back; if any argument or return type is not understood, the method is omitted. The library is primarily a proof of concept of our foreign objects design (Section 5.5.1), and its implementation was discussed further in that section. Grace-GTK is available from https://github.com/mwh/grace-gtk.

8.4.2 Grace-CUDA

Grace-CUDA [68] connects Grace code to NVIDIA's CUDA general-purpose graphics programming unit library [134]. The library consists primarily of a compiler plugin to Minigrace that rewrites part of the source code to access a runtime CUDA library, and code for generating NVIDIA's restricted C subset that runs on its graphics cards. Graphics cards execute the same code on many processors simultaneously in lockstep, greatly speeding up data-parallel computations.

With the plugin loaded any requests for methods like cuda.over(...)map, which accepts a block of code to run on the GPU, are intercepted. The loop body is translated to CUDA's C variant and the method request rewritten to pass along an identifier for that generated code to run time. When the program runs, the Grace-CUDA run-time library will load the provided data onto the GPU and execute the code. The loop body appears as essentially standard Grace code to the programmer.

A simple program to multiply some numbers together would be:

```
import "cuda" as cuda
def size = 1000000
print "Starting population..."
var x := cuda.floatArray(size)
var y := cuda.floatArray(size)
var z := cuda.floatArray(size)
for (0..(size−1)) do {i−>
    x.at(i)put(i)
    y.at(i)put(i)
    z.at(i)put(i)
}
print "Starting CUDA..."
def res = cuda.over(x, y, z) map {a, b, c−>
    a * b * c
}
print "1000^3: {res.at 1000}"
```

```
print "93397^3: {res.at 93397}"
```

This program multiplies the elements of three arrays elementwise. A cuda.floatArray is an array of numbers backed by floats, which CUDA prefers to use. More complicated programs are possible with Grace-CUDA using other methods and special type annotations. Samples for matrix multiplication and mass computation of exponentiations and quotients are included in the distribution. The method design reflects in part the underlying CUDA structures for executing code on and passing data to the graphics card, meshed with Grace's existing philosophy of control structures as methods dictating the use of blocks. Specialised data types are required because CUDA arrays have different semantics to collections of Grace numbers.

We described a dialect using part of the Grace-CUDA library without the plugin in Section 6.2.11. Grace-CUDA is available from https://github.com/mwh/grace-cuda.

8.5 Outside contributions

Although we were the primary author of Minigrace, other authors have contributed fixes or components of the software. In this section we outline those contributions.

Daniel Gibbs worked on error messages as part of his honours project, and implemented many new syntax error reports. He also implemented the "suggestions" infrastructure for showing users what they may have meant to write.

Timothy Jones implemented a Java backend to the compiler (since defunct) as a summer research assistant and contributed many bug fixes and work on annotations as a PhD student.

Jan Larres implemented pretty-printing the syntax tree into Grace and an interactive interpreter and read-eval-print loop as a research assistant. He also implemented the visitor pattern available in AST nodes.

Scott Weston implemented better path searching for imported modules and configurable install paths for the compiler as a summer research assistant.

Miscellaneous bug fixes and platform extensions were submitted by Alex Sandilands, Andrew P. Black, Charlie Paucard, and Bart Jacobs.

Chapter 9

Conclusions

In this thesis we investigated extensions to the Grace language allowing programmers to express their intentions in a manner suitable to the task at hand and their understanding of it. Our extensions not only satisfy the goals of Grace but are general in purpose and application such that they could be incorporated in other object-oriented languages. They are designed to support both experienced and novice programmers within the same language and using the same features, giving each what is required without obstruction from the other. We presented case studies and a user experiment to validate our designs.

The contributions of this thesis are:

- An object-oriented pattern-matching design with syntax and semantics integrated with the language.
- A system of modules constructed with objects.
- The design of a system of language extensions and restrictions through dialects.
- A novel interface integrating visual and textual editing of the same code.

Pattern matching. We have shown that an object-oriented design for pattern matching can be fully integrated into a language with minimal changes to the language. Both built-in and user-defined patterns can follow the same object framework and semantics, and through a system of combinators many complex patterns can be built up without the user needing to understand the implementation of patterns at all. We fully implemented such a system into a working compiler and justified our design against alternative approaches. We used our patterns in practice as part of implementing dialects.

Modules as Objects. We presented an object-based module system, building the desirable attributes of modules without introducing significant new concepts into the language. Our module system provides namespacing, sharing, controlled export, and explicit dependencies, while permitting multiple and foreign implementations transparently. We showed how a flexible construction of module import, in combination with modules as objects, allows accessing both ordinary code and external resources through the same interface, and a simple package management system with the bare minimum of overhead for instructors or library publishers. We used our modules in practice as part of implementing dialects.

Dialects. We gave the design of our system of dialects, which permit variant languages to be embedded within a base language. Dialects build on our module system, and through the module system many dialects may coexist within a single overarching program. A dialect can both extend the language, providing new constructs or definitions to the user, and restrict the language available, preventing the user from using a feature or providing different errors or feedback when they do. Dialects are defined entirely within the base language, without any macros or meta-functionality, meaning an author does not need to learn any other languages or tools. We demonstrated the power of this design through a series of case studies, ranging from a graphical microworld through to typechecking and engineering domain-specific languages, along with a dialect for building other

dialects and one presenting a radically different language model, all within the same language.

Tiled Grace. We showed a novel interface for editing programs combining both visual and textual representations of a program. This interface allows a user to look at their code in two different ways, with the connection between representations made clear through animation, and permits modifying the code in either view. The interface is fully integrated with our dialect system, allowing programs in multiple dialects without further modification. We gave the results of a user experiment we performed where 33 participants used our tool, showing that they found it engaging, valued the feedback for errors available in the tiled interface, and appreciated the ability to access their code in two different ways. We also found that a proportion of participants had substantial trouble using drag-and-drop for programming, suggesting that the drag-and-drop paradigm may not be ideal for this use, although the large majority of participants had no such difficulty.

Finally, we described Minigrace, our compiler for Grace. Minigrace is written in Grace itself and fully self-hosted, supporting almost all of the Grace language, and able to target both native code and ECMAScript. We implemented all of our language features in Minigrace and reflected on our experience in doing so, as well as in developing a self-hosting compiler for a language under design. Minigrace, as well as associated tools and libraries we built to use with it, is fully open-source code and publicly available.

9.1 Future work

There is more work that would be interesting to follow up on from the results of this thesis. This section will briefly explore interesting follow-on studies by topic, and direct the reader to the future-work sections within each chapter.

Pattern Matching. Our pattern-matching system permits examining objects with a concise syntax, but there are existing specialised matching syntaxes, such as regular expressions and parsing expression grammars. Future work could explore how to integrate these systems within the framework we presented. We discuss this work in more detail in Section 4.7.6. Another extension would be to permit patterns to be used in place of types, giving arbitrarily powerful checking; we implemented this extension in Minigrace, but further exploration of the implications and limitations, and the best way to design patterns for this use, remains to be done. We discuss this generalisation in Section 4.6.

Modules as Objects. Accessing external data sources through the module import system should be possible, but we implemented no significant data sources in this way. Future work can explore how such sources should be exposed and the correct way to integrate gradual type checking. We discuss these and other extensions in Section 5.5.

Dialects. Our dialect system operates at the module level, but at times it would be useful to use a different dialect for only part of a module. In doing so, questions arise around access to other scopes: does code in an inner dialect have access to the outer dialect's methods? The outer scope's variables? The correct answers to these questions are not obvious, and may vary in different circumstances; future work should explore how best to integrate localised dialects into the system. At times it might be useful for dialects to make minor syntactic extensions, or decorations of the syntax tree, which our system avoids. We discuss these items of future work in more detail in Section 6.3.

Tiled Grace. Our system does not permit users to change between views while there is a static error in their program. While some errors in programs have no sensible representation in one of the tiled or textual views, others could reasonably cross the barrier and allow the user to look at their code two different ways, and future work could establish which errors should be permitted and how to represent them. Our system does not tell users where

they may place a tile before they pick it up, as Scratch does, but it would be useful to indicate to the user what they may do as early as possible.

Our experiment in Sections 7.4 and 7.5 suggested that drag-and-drop may not be the most helpful interface paradigm for all users, and we suggested that repeating the experiment with a point-and-click interface could be instructive. In a similar vein, touch-screen devices are another visual paradigm where drag-and-drop is common, and results may differ on such a device. We also noted other factors that bear further investigation, particularly our finding that enjoyment of the system was inversely correlated to past programming experience. We also believe that other languages would see similar benefits from a bijective visual interface and that these interfaces should be considered. We noted in particular potential application in other paradigms such as type theory and formal methods. We discuss all of these issues in Section 7.7.

Appendix A

Auxiliary data

This thesis is accompanied by additional auxiliary data for the benefit of future researchers. We will briefly describe each of the included elements:

Minigrace The source code of Minigrace, our Grace compiler, its complete revision history as a Git repository, and pregenerated tarballs suitable for bootstrapping.

Grace web IDE Our web interface to Minigrace's ECMAScript backend.

Tiled Grace The source code of Tiled Grace, our combined graphical-textual interface described in Chapter 7, and its complete revision history as a Git repository.

Experimental data Anonymised data from the Tiled Grace experiment described in Chapter 7, including survey responses, instrumentation data, analysis scripts, survey systems, and the instrumented interface used in the experiment.

Grace-GTK The source code of the Minigrace bindings to the GTK+ widget library described in Section 8.4.1.

Grace-CUDA The source code of the Minigrace plugin and runtime library for GPU programming described in Section 8.4.2.

The data can be obtained:

- on the CD accompanying the deposit copy of the thesis;
- in ResearchArchive–Te Puna Rangahau, the public-facing repository for research outputs from Victoria University of Wellington, New Zealand, accessible online at http://researcharchive.vuw.ac.nz/, alongside the thesis itself;
- from the author's personal web site at http://michael.homer.nz/phd.

Appendix B

Extended examples

Some pieces of example code in this thesis were truncated to focus on essential elements. In this appendix we present extended versions of some of these incorporating additional code that would be required in real use.

B.1 Scala matching

This example incorporates the first matching example from Section 2.2.1 verbatim, with surrounding code creating a viable typing environment.

```
class StringOrIntOrTriple[T]
object StringOrIntOrTriple {
  implicit object StringWitness extends StringOrIntOrTriple[String]
  implicit object IntWitness extends StringOrIntOrTriple[Int]
  implicit object TripleWitness extends StringOrIntOrTriple[Tuple3[Int,Int,Int]]
}

object Bar {
  def test[T : StringOrIntOrTriple](x : T) =
    x match {
      case 1 => "one"
      case "two" => 2
      case y : Int => "scala.Int"
      case (a, b, c) => "A triple: " + a + ", " + b + ", " + c
    }
}

object test {
  def main(args : Array[String]) {
    println(Bar.test(1))
    println(Bar.test("two"))
    println(Bar.test(3))
    println(Bar.test((1,2,3)))
  }
}
```

B.2 F♯ matching

This example incorporates the F♯ matching example from Section 2.2.4 verbatim, with surrounding code creating a runnable program.

```
let hd(l : _ list) = l.Head
let tl(l : _ list) = l.Tail
let nonempty(l : _ list) = not l.IsEmpty

let (|Cons|Nil|) l =
    if nonempty l then Cons(hd l, tl l)
    else Nil

let rec length l =
    match l with
        |Cons(x, xs) ->
            1 + length xs
        |Nil ->
            0

[<EntryPoint>]
let main argv =
    let mylist = [ 1 ; 2 ; 3 ; 4 ]
    printfn "1234 list: %i" (length mylist)
    printfn "argv list: %i" (length (List.ofArray argv))
    System.Console.ReadLine() |> ignore
    0
```

This program outputs the length of the fixed list `1;2;3;4` and the number of command-line arguments given to the program.

Appendix C

Package Manager Example

We have implemented our package management system into a tool called gracepm, which is included in the Minigrace distribution. An early prototype of the concept was built by a summer research assistant in 2013-2014; the model was ours and we implemented gracepm independently. The package manager supports two major modes: install, which retrieves and installs a particular module and its dependencies, and satisfy, which ensures that all dependencies of a given Grace source file are met. A worked example in more detail is given in Appendix C. We will explain these definitions through an example.

Suppose a user has been told to install a particular module for a course. They will be given a path to give to the package manager, such as ecs.victoria.ac.nz/~mwh/test. They will run:

```
gracepm install ecs.victoria.ac.nz/~mwh/test
```

gracepm will fetch https://ecs.victoria.ac.nz/~mwh/test.grace and store it in ~/.local/lib/grace/modules/ecs.victoria.ac.nz/~mwh/test.grace. This location is exactly where an import of this path will look for the file. The package manager will inspect the file it has retrieved and also install any of its dependencies recursively.

After installing the library above, given the module:

```
import "ecs.victoria.ac.nz/~mwh/test" as t
t.greet "reader"
```

the import will succeed, and the program will successfully greet the reader.

What if the user already has a file (perhaps a template), and simply wishes to make it work? If the module above is called greet.grace, they can run:

```
gracepm satisfy greet.grace
```

and the package manager will find the import, install it and its dependencies, and return.

From the end user's perspective, they have simply applied an opaque string and the module they wanted worked from then on.

Appendix D

Tiled Grace Experiment

This appendix contains the consent form, information sheets, questionnaires, human ethics application, and human ethics approval for our experiment on Tiled Grace described in Chapter 7.

TE WHARE WĀNANGA O TE ŪPOKO O TE IKA A MĀUI VICTORIA UNIVERSITY OF WELLINGTON

Participant Consent Form

Principal Investigator	**Investigator**	**Investigator**
Michael Homer	Prof. James Noble	Dr. David Pearce
PhD Student	Professor of Computer Science	Senior Lecturer
mwh@ecs.vuw.ac.nz	kjx@ecs.vuw.ac.nz	djp@ecs.vuw.ac.nz
CO 254	CO 234	CO 231

School of Engineering and Computer Science
Victoria University of Wellington

Please tick and sign

I have read the information sheet supplied and the researchers have satisfactorily answered any questions I may have had.

I consent to taking part in this study and understand that I have the right to withdraw from this experiment within two weeks of data collection by emailing the researchers accordingly.

I consent to the researchers using mouse movement, selection, and keyboard input data and answers from my participation in a non-identifiable way in a PhD thesis and related conference / journal papers and corresponding anonymised public dataset.

☐ I would like to be emailed a soft copy of the report.

☐ I would like to be entered in the random prize draw.

You can provide an email address that will be used to contact you if you win one of the prizes, and to send you a soft copy of the report if requested. If you do not wish to participate in either of these you do not need to provide an address.

Email address: .....

Signed: .....

Signed (parent or guardian if under 18): .....

Date : .....

TE WHARE WĀNANGA O TE ŪPOKO O TE IKA A MĀUI VICTORIA UNIVERSITY OF WELLINGTON

Participant Information Sheet

Principal Investigator	**Investigator**	**Investigator**
Michael Homer	Prof. James Noble	Dr. David Pearce
PhD Student	Professor of Computer Science	Senior Lecturer
mwh@ecs.vuw.ac.nz	kjx@ecs.vuw.ac.nz	djp@ecs.vuw.ac.nz
CO 254	CO 234	CO 231

School of Engineering and Computer Science
Victoria University of Wellington

This experiment is being undertaken towards Michael Homer's PhD in the Software Engineering group at Victoria University of Wellington. The Victoria University Human Ethics Committee has granted ethics approval for this experiment. The research looks at editing source code using a combination of tiled drag-and-drop editing and conventional textual editing. Participants will be asked to edit a number of programs and to write small programs completing a particular task.

Any person above 14 years of age is free to participate in the study, but any participants below 18 years of age will require parental consent. Participants will be asked to fill out a pre- and post-test questionnaire as well as carrying out the experimental tasks. Participants are able to refuse to answer any given question, and are able to withdraw from the study without question within a fortnight of the data collection.

While the experiment will be recording the user interaction, only the on-screen interaction will be recorded. There will be no video or audio recording of the participants. Participants will be observed and the information gathered will be assessed for accuracy, but your completed data will only be reported in aggregate form. The final dataset will also be made available in an anonymised form for other researchers. The entire process should take approximately thirty to forty minutes.

Voucher prizes will be awarded to randomly-selected participants after data collection is complete. Participating in the draw is optional, and requires providing an email address. The provided address will be used only for the purposes of this experiment and record of it will be destroyed afterwards.

If you have any questions or would like to receive further information about the project, please contact us via the details supplied above.

Thank you for your participation in the study.

Initial Questionnaire

Principal Investigator	**Investigator**	**Investigator**
Michael Homer	Prof. James Noble	Dr. David Pearce
PhD Student	Professor of Computer Science	Senior Lecturer
mwh@ecs.vuw.ac.nz	kjx@ecs.vuw.ac.nz	djp@ecs.vuw.ac.nz
CO 254	CO 234	CO 231

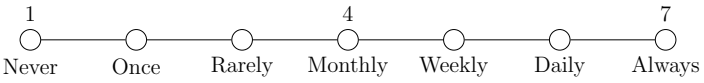
School of Engineering and Computer Science
Victoria University of Wellington

Age: .....

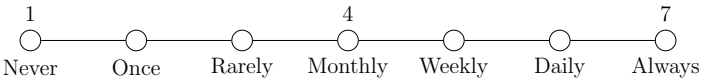
Gender: .....

Please tick the appropriate circle:

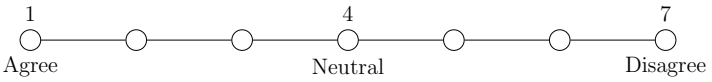
How often do you use a computer with a keyboard and mouse:



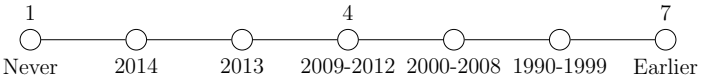
How often do you program in your own time:



I enjoy programming:



I first tried to program:



TE WHARE WĀNANGA O TE ŪPOKO O TE IKA A MĀUI VICTORIA UNIVERSITY OF WELLINGTON

Circle any of the following you have used before:

Go	Visual Basic	Spreadsheet
Whiley	COBOL	Forth
Alice	Scratch	Scala
Alloy	BlueJ	Octave
Clojure	C	Lua
S+	INTERCAL	LabVIEW
Racket	Git	Eclipse
Grace	Shell	XCode
Object orientation	Greenfoot	Mercurial
SQL	Prolog	Io
HTML	Logo	Basic
LaTeX	Delphi	Python
C♯	D	Java
Pascal	R	Haskell
Erlang	Befunge	Ruby
Piet	ML	HTML
Eiffel	Maple	APL
Perl	PL/I	MATLAB
IntelliJ	Subversion	Squeak
Rust	FORTRAN	Lisp
Visual Studio	NetBeans	Groovy
C++	Verilog	JavaScript
F♯	Objective-C	Processing
VHDL	Smalltalk	PHP

TE WHARE WĀNANGA O TE ŪPOKO O TE IKA A MĀUI VICTORIA UNIVERSITY OF WELLINGTON

Final Questionnaire

Principal Investigator	**Investigator**	**Investigator**
Michael Homer	Prof. James Noble	Dr. David Pearce
PhD Student	Professor of Computer Science	Senior Lecturer
mwh@ecs.vuw.ac.nz	kjx@ecs.vuw.ac.nz	djp@ecs.vuw.ac.nz
CO 254	CO 234	CO 231

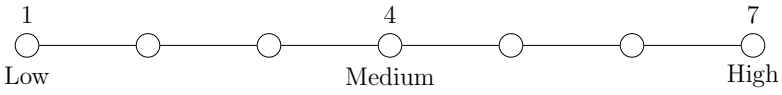
School of Engineering and Computer Science
Victoria University of Wellington

For each pair circle which of the two options was the biggest problem:

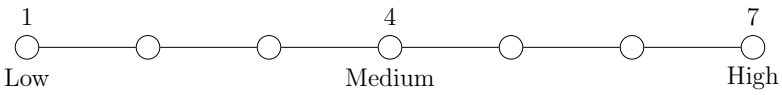
- Effort / Getting it right
- Time / Effort
- Getting it right / Frustration
- Physical Demand / Getting it right
- Frustration / Effort
- Getting it right / Time
- Mental Demand / Physical Demand
- Frustration / Mental Demand
- Time / Frustration
- Physical demand / Frustration
- Physical Demand / Time
- Time / Mental Demand
- Getting it right / Mental Demand
- Mental Demand / Effort
- Effort / Physical demand

Please tick the appropriate circle:

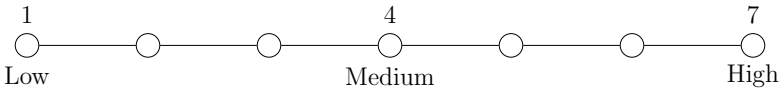
How mentally demanding was the task:



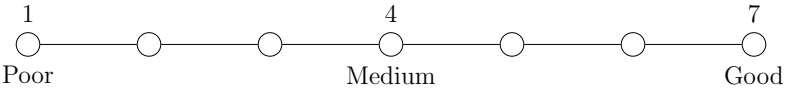
How physically demanding was the task:



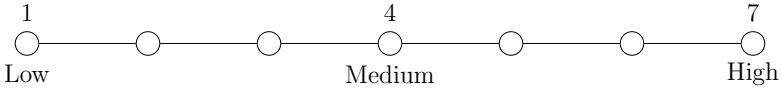
Rate the time stress:



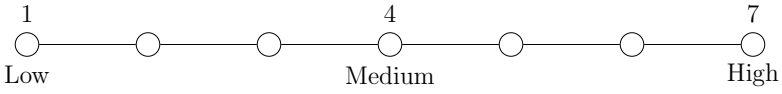
How would you rate your performance:



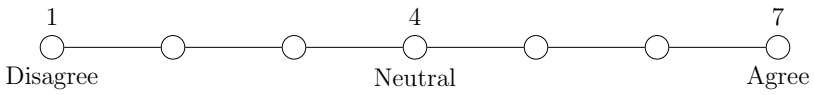
How much total effort did this require:



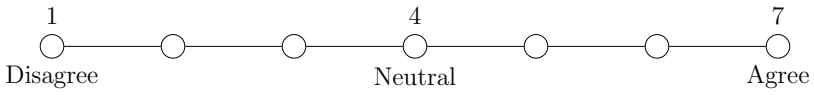
How frustrating did you find the task:



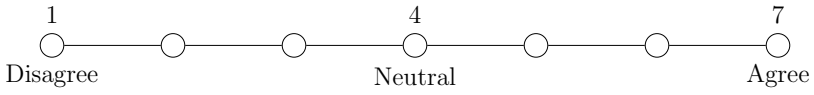
The system was fun to use:



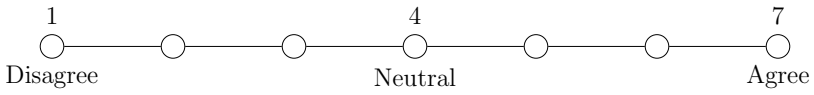
The system was novel to use:



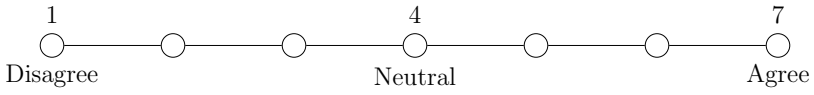
I enjoy programming:



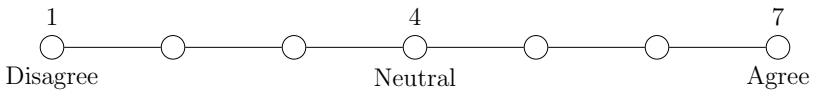
I preferred the tiled view of my code:



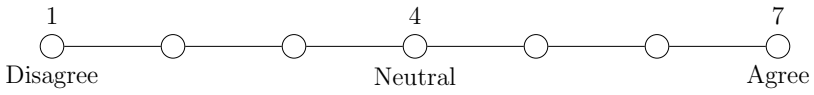
I changed between the tiled and textual views of my code a lot:



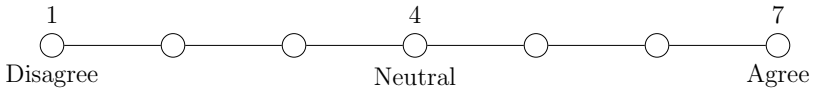
I found the syntax difficult to deal with:



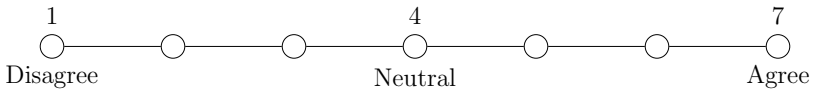
The system did what I wanted:



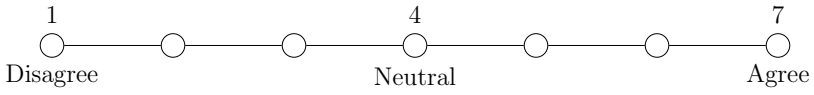
I preferred the textual view of my code:



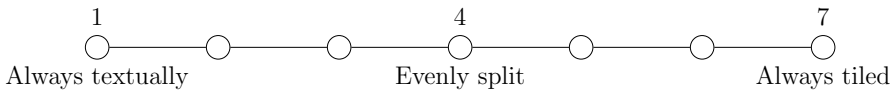
Fixing errors in the code was easy:



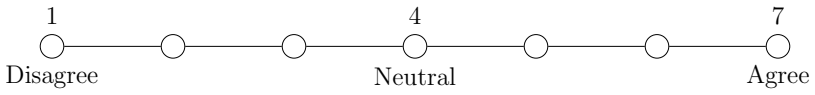
Finding errors in the code was easy:



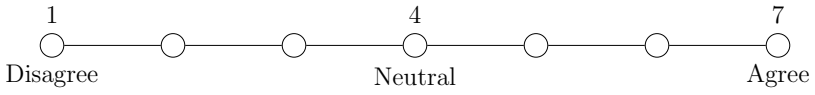
I edited my code:



It was easier to deal with syntax in the tiled view:



I would use this system again:



What did you like about this system?

What did you dislike about this system?

If you have any other comments you wish to make, please write them here:

ResearchMaster

Human Ethics Application

Application ID :	0000020573
Application Title :	Evaluating a drag-and-drop interface for Grace
Date of Submission :	24/01/2014
Primary Investigator :	Michael Homer
Other Investigators :	Prof Dale Carnegie
	Dr David Pearce
	Prof James Noble

Research Form

Information

Christmas closedown: The end-of-year deadline for applications to the Human Ethics Committee and subcommittees (apart from the School of Information Management Sub-committee) is 30 November 2013. Applications that are received before 1 December will be reviewed and applicants notified before the Christmas break. Applications received after this will not be reviewed until 3 February 2014.

Welcome to the Human Ethics Application Form

The following advice will assist you in completing this process:

Help contact

For information about Human Ethics, go to the Human Ethics web page.

For help, please email the Ethics Administrator.

Policy

You must read the Human Ethics Policy before beginning your application. The Policy includes a link to a sample consent form, information sheet, and transcribing confidentiality form which may be useful (see last page).

Health research may require HDEC approval. To find out if your research requires this, read the HDEC Guidelines or contact the chairperson of your committee for clarification.

Student research

If you are a student, check with your supervisor before filling in this form. You may need to complete School requirements before applying for ethical approval.

Student applications will be automatically forwarded to supervisor(s) and then Head of School/Delegate for approval when the form is submitted. Once the Head of School has approved it, the form will be automatically forwarded for committee review.

Technical

This online system works best on Internet Explorer and Safari. It may not work on your iPad or tablet.

A guide to using this online form, which includes a workflow showing how the approval process works, can be downloaded here.

If your application involves other researchers, you can use the Comments function of this form to communicate about the application with each other. Click on the Application Comments or Page Comments icon on the top right of the screen to view and add comments. Comments left on the form once it is submitted will be visible to your Head of School and committee reviewers, so **remember to delete any private comments before submitting the form**.

Process

All applications will be automatically forwarded to the Head of School for review when the application is submitted. Once the Head of School has approved it, the form will be automatically forwarded for committee review.

You will normally receive an outcome of the review of your application within three weeks, unless you apply during an advertised close-down period (for instance, applications may not be reviewed in December and January). NO part of the research requiring ethical approval may commence prior to approval being given.

To apply for an amendment or extension to an approved application, open the approved form and click on Apply for amendment/extension. You will then be able to complete the Amendment/Extension page and resubmit the form.

Application Details

1. Ethics category code*

   Human

   Clearance Purpose code

   Research Only

2. Application ID

   0000020573

3. Please select 'Human Ethics Committee', 'Education Faculty Ethics Committee', or 'Pipitea Ethics Committee' below (online application is not yet available for other committees)

Human Ethics Committee

4. Title of project*

Evaluating a drag-and-drop interface for Grace

5. School or research centre*

Engineering and Computer Science

6. Please list all personnel involved in this project. Ensure that all are listed with the correct role. **If you are a student, do not add your supervisor here: you will be asked to add this information on the next page.**

Please ensure that only one person is listed as Principal Investigator.

To add a person, search for their Victoria ID if known, otherwise *either* their first *or* last name (whichever is the most unusual). Click on the magnifying glass to search for results.

Press the **green tick** at the bottom right corner to save the person record.

Add anybody who is involved in this project as:

- Associate Investigator
- Other Researcher
- PhD Student
- Masters Student
- Research Assistant

Click on the help button if you are having difficulty adding people to the list.*

1	Given Name	David
	Surname	Pearce
	Full Name	Dr David Pearce
	AOU	SECS
	Position	Associate Investigator
	Primary?	No
2	Given Name	Michael
	Surname	Homer
	Full Name	Michael Homer
	AOU	SECS
	Position	Principal Investigator
	Primary?	Yes

7. Are any of the researchers from outside Victoria?*

- ○ Yes
- ◉ No

8. Is the principal investigator a student?*

- ◉ Yes
- ○ No

Next time you save this form or move to a new page, a Student Research page will appear after this one. Please complete the two questions on the Student Research page.

Student Research

7a. What is your course code (e.g. ANTH 690)?*

COMP690

7b. Please add your primary supervisor (the supervisor who should review this application).

If your supervisor is also the Head of School or the school ethics officer, you will need to discuss with your School who should approve this application as Head of School or delegate. The supervisor and Head of School or delegate **must not be the same person.**

To add your supervisor, search for their Victoria ID if known, otherwise *either* their first *or* last name (whichever is the most unusual).

Press the **green tick** at the bottom right corner to save the person record. *

1	Given Name	Robert
	Surname	Noble
	Full Name	Prof James Noble

	AOU	SECS
	Position	Supervisor

If your supervisor is also the Head of School, you will need to assign a different person to the Head of School or Delegate role on the Signoff page.

7c. What is your email address? (this is needed in case the committee needs to contact you about this application)*

mwh@ecs.vuw.ac.nz

Note that system-generated emails (eg approval notifications) will not necessarily come to this address. System-generated emails will come to the email address stored for you in Student Records. To change the record in Student Records, log into My Victoria, and click on Student Records. You will be able to update your email address from there.

Project Details

9. Describe the objectives of the project*

The objectives are to see whether users can use the new drag-and-drop interface we created, to see if and how they make use of the ability to transition between tiled and textual display of their source code, and to measure how much they enjoyed using the system.

10. Describe the benefits and scientific value of the project*

We will find out how users interact with drag-and-drop displays of source code and whether such displays, in combination with the ability to transition back and forth with textual display, are useful.

11. Describe the method of data collection. Note that later in this form, in the Documents section, you will need to upload any relevant documentation such as interview schedule, survey, questionnaires, focus group rules, observation protocols etc. Delays are likely if the interview questions are missing from the Documents section. *

We will present each participant with the included the pre- and post-questionnaires and a web browser running our Tiled Grace tool. The tool will record screen contents and user interactions. There will be no video or audio recording, but we will observe and take notes. They will be asked to produce a number of small programs using the tool, some beginning from partially-completed programs, and to describe programs and what they have done, after a period of being shown the mechanics of the system.

12. Does your research have more than one phase?*

- ○ Yes
- ◉ No

Key Dates

If approved, this application will cover this research project from the date of approval

13. Proposed end date for data collection*

30/06/2014

14. Proposed end date for research project as a whole*

31/12/2014

Proposed source of funding and other ethical considerations

15. Indicate any sources of funding, including self-funding (self-funding means that you are paying for research costs such as travel, postage etc. from your own funds) (tick all that apply)

- ☐ Internally funded
- ☑ Externally funded
- ☐ Self-funded

15a. Describe the source of funding*

Royal Society Marsden Fund Grant of James Noble

15b. Indicate any ethical issues or conflicts of interest that may arise because of sources of funding e.g. restrictions on publication of results*

None.

16. Is any professional code of ethics to be followed?*

- ◉ Yes
- ○ No

16a. Name the professional code(s) of ethics *

Association for Computing Machinery Code of Ethics

17. Is ethical approval required from any other body?*

- ○ Yes
- ◉ No

18 Depending on the characteristics of your participants or location of the research, you may need to arrange permission from another body or group before proceeding. If this is the case, explain and describe how you are addressing this*

I will use computer laboratory resources from the School of Engineering and Computer Science and will schedule access through their procedures and booking system.

Treaty of Waitangi

19. How does your research conform to the University's Treaty of Waitangi Statute? (you can access the statute from Victoria's Treaty of Waitangi page)*

This research conforms to the Treaty of Waitangi Statute and does not raise any relevant issues under it.

Information about participants

20. How many participants will be involved in your research? If you are using records (e.g. historical), please estimate the number of records*

20-40

21. What are the characteristics of the people you will be recruiting?*

Engineering or computer science students, in particular those taking COMP102/COMP112/ENGR101

22. Are you specifically recruiting any of the following groups?

- Māori
- Pasifika
- Children/youth
- Students
- People who are offenders and/or victims of crime
- People with disabilities
- People in residential care
- People who are refugees

Please indicate below.*

- ◉ Yes
- ○ No

22a. Which of these groups will you be recruiting? (tick all that apply)*

- ☐ Māori
- ☐ Pasifika
- ☐ Children/youth
- ☑ Students
- ☐ People who are offenders and/or victims of crime
- ☐ People with disabilities
- ☐ People in residential care
- ☐ People who are refugees

23. Have you undertaken any consultation with the groups from which you will be recruiting?*

No.

24. Provide details of consultation you have undertaken or are planning*

None.

25. Outline the method(s) of recruitment you will use for participants in your study*

Speaking at lectures of relevant courses, advertising posters in laboratories, online forums and lists associated with ECS courses, and word of mouth.

26. Will your participants receive any gifts/koha in return for participating?*

- ◉ Yes
- ○ No

26a. Describe the gifts/koha and the rationale*

There will be 2-3 prizes of gift vouchers amounting to no more than $50 per person to random participants, and there may be confectionery for participants.

27. Will your participants receive any other assistance (for instance, meals, transport, release time or reimbursements)?*

- ○ Yes

◉ No

28. Will your participants experience any special hazard/risk including deception and/or inconvenience as a result of the research?*
○ Yes
◉ No

28a. Give details and indicate how you will manage this*
I am not exposing participants to any special hazard or risk.

29. Is any other party likely to experience any special hazard/risk including breach of privacy or release of commercially sensitive information?*
○ Yes
◉ No

30. Do you have any professional, personal, or financial relationship with prospective research participants?*
○ Yes
◉ No

31. What opportunity will participants have to review the information they provide? (tick all that apply)*
☐ They will be given a transcript of their interview
☐ They will be given a summary of their interview
☑ Other
☐ They will not have an opportunity to review the information they provide

31a. Explain how participants will be able to review the information they provide*
They can review the questionnaires before submitting them.

Informed consent

32. Will participation be anonymous? **'Anonymous' means that the identity of the research participant is not known to anyone involved in the research, including researchers themselves.** It is not possible for the researchers to identify whether the person took part in the research, or to subsequently identify people who took part (e.g., by recognising them in different settings by their appearance, or being able to identify them retrospectively by their appearance, or because of the distinctiveness of the information they were asked to provide).*
○ Yes
◉ No

33. Will contributions of participants be confidential? Confidential means that those involved in the research are able to identify the participants but will not reveal their identity to anyone outside the research team. Researchers will also take reasonable precautions to ensure that participants¿ identities cannot be linked to their responses in the future.*
◉ Yes
○ No

33a. How will confidentiality be maintained in terms of access to the research data? (tick all that apply)*
☐ Access to the research will be restricted to the investigator
☑ Access to the research will be restricted to the investigator and their supervisor (student research
☐ Focus groups will have confidentiality ground rules
☐ Transcribers will sign confidentiality forms
☐ Other

33b. How will confidentiality be maintained in terms of reporting of the data? (tick all that apply)*
☑ Pseudonyms will be used
☐ Participants will be named only in a list of interviewees
☑ Data will be aggregated and so not reported at an individual level
☐ Participants will be referred to by role or association with an organisation rather than by name
☐ Names will be confidential, but other identifying characteristics may be published with consent
☐ Other

34. How will informed consent be obtained? (tick all that apply to all phases of the research you are describing in this application)*
☐ Informed consent will be implied through voluntary participation (anonymous research only)
☑ Informed consent will be obtained through a signed consent form
☐ Informed consent will be obtained by some other method

Access, storage, use, and disposal of data

35. What procedures will be in place for the storage of, access to and disposal of data, both during and at the conclusion of the research? (tick all that apply)*
☑ All written material will be kept in a locked file; access restricted to investigator(s)
☑ All electronic information will be password-protected; access restricted to the investigator(s)

- [x] All questionnaires, interview notes and similar materials will be destroyed
- [ ] Any audio or video recording will be returned to participants and/or electronically wiped
- [x] Other procedures

35a. Describe the procedures for the storage of and access to the data*

Original electronic data will be stored with access restricted to the investigators. We will later remove all personally identifiable information and allow other scientists to access this anonymised version of the data to comply with publishing requirements of computer science.

35b. Will the data be destroyed at the conclusion of the research?*

- ○ Yes
- ◉ No

35c. How many years after the conclusion of the research will the materials be destroyed?*

999.00

36. If data and material are not to be destroyed, indicate why and the procedures envisaged for ongoing storage and security

The physical copies that have personally identifiable tags will be destroyed at the proposed date of completion of the project as a whole. Only anonymised electronic copies will remain.

Dissemination

37 How will you provide feedback to participants?*

Participants will have the option to provide an email address to receive the results of the research after completion.

38. How will results be reported and published? Indicate which of the following are appropriate. The proposed form of publications should be indicated on the information sheet and/or consent form*

- [x] Publication in academic or professional journals
- [x] Dissemination at academic or professional conferences
- [x] Availability of the research paper or thesis in the University Library and Institutional Repository
- [x] Other

38a. Describe how the results will be disseminated*

An anonymised version of the data set will be made public.

39. Is it likely that this research will generate commercialisable intellectual property? (check the help text for more information about IP)*

- ○ Yes
- ◉ No

Documents

40. Please upload any documents relating to this application. Ensure that your files are small enough to upload easily, and in formats which reviewers can easily download and review*

Description	Reference	Soft copy	Hard copy
Participant information sheet(s)	informationSheet.pdf	✓	
Participant consent form(s)	consentForm.pdf	✓	
Questionnaire or survey	pretest.pdf	✓	
Post-questionnaire	posttest.pdf	✓	

Getting feedback on your application

You can seek feedback on your draft application, for instance from a mentor or a school Ethics representative **before** submitting it for review.

There are two ways of doing this:

1. Emailing your application to someone

You can email your application and any associated documents to another person at Victoria. To do this:

1. Click on the Action tab (on the left of the screen)
2. Click on Email application
3. Search for the person using **either** their first name **or** their last name (whichever is the most unusual)
4. Select the documents to include from the Document list (eg the Application PDF)
5. Click on Send or Zip and send

If you wish to send your application to someone outside Victoria, one option is emailing the application to yourself and then forwarding it.

2. Assigning a peer reviewer

You can add someone to the form as 'peer reviewer'. This means that they will be able to access your form by logging onto ResearchMaster. They will also be able to comment on your form online. **If you are a student, don't add your supervisor to the form as a peer reviewer - to get supervisor feedback, submit the form. Your supervisor may then make comments on it and ask you to review it further before it goes to the committee for review.** To do this:

1. Click on the Review tab on the left of the screen
2. Click on 'Peer reviewers'
3. Search for the person using their person code if known, or **either** their first name **or** their last name (whichever is the most unusual)
4. Click on the person's name
5. You may then also want to send the peer reviewer a notification, by clicking on Notify Peer Reviewer on the Actions tab

Signoff

41. Use this section to record signoff by all other researchers involved in this project (except for the principal investigator. Principal investigators do not need to sign off).

    Principal Investigators may sign off on the behalf of researchers external to Victoria University who may be unable to access this site. In these cases, please upload evidence of the researchers' signoff (eg, a scanned email) on the Documents page.

    To sign off:

    1. Click on the pencil icon on the far right of the line with your name on it
    2. Click on I Accept
    3. Add the date
    4. Click on the green tick icon on the bottom of the signoff window
    5. Go to the Actions tab and click on 'Notify lead researcher that signoff is complete'

1	Full Name	Dr David Pearce
	Position	Associate Investigator
	Declaration Signed?	Yes
	Signoff Date	14/01/2014

42. Please add the Head of School (or delegate. e.g. school ethics officer) who should approve this application. This will be your own Head of School, or the person in your School responsible for approving Ethics applications. The form will be forwarded to this person automatically once it is submitted. **Please check with your School administration team if you are unclear who should be assigned this role. Adding the wrong person could lead to delays in processing your application.**

    If you are a student, the Head of School or delegate must not be the same person as your supervisor.

    Once you've searched for your Head of School/delegate, click on the green tick to add them, and then also save the application before submitting.

    Heads of School or delegates should process this form by clicking on the Actions tab and either approve it, or return it to the researcher for further changes. *

1	Given Name	Dale
	Surname	Carnegie
	Full Name	Prof Dale Carnegie
	AOU	SECS
	Position	Head of School (or delegate)

Please ensure that you **save your application before submitting it**. Once you have saved your application, to submit it, click on 'Actions' on the left hand side of the screen and then 'Submit for review'.

If you are a student, your application will go to your supervisor and then Head of School for approval once you submit it. If you are a staff member, your application will go straight to the Head of School for approval once you submit it.

If you have any feedback about this online form, please email it to ethicsadmin@vuw.ac.nz

Amendment or extension request (available only for approved applications)

43. Are you applying for an extension, an amendment, or both?*

    ◯ Extension
    ◯ Amendment
    ◯ Both an extension and an amendment

    This question is not answered.

Please check that you have answered all mandatory questions and have saved the application before submitting your form. To submit your form, click on the Action tab and then click on Submit for review

TE WHARE WĀNANGA O TE ŪPOKO O TE IKA A MĀUI
VICTORIA
UNIVERSITY OF WELLINGTON

Phone 0-4-463 5676
Fax 0-4-463 5209
Email Allison.kirkman@vuw.ac.nz

MEMORANDUM

TO	Michael Homer
COPY TO	David Pearce James Noble
FROM	Dr Allison Kirkman, Convener, Human Ethics Committee
DATE	1 March 2014
PAGES	1
SUBJECT	**Ethics Approval: 20573** Evaluating a drag-and-drop interface for Grace

Thank you for your application for ethical approval, which has now been considered by the Standing Committee of the Human Ethics Committee.

Your application has been approved from the above date and this approval continues until 31 December 2014. If your data collection is not completed by this date you should apply to the Human Ethics Committee for an extension to this approval.

Best wishes with the research.

Allison Kirkman
Human Ethics Committee

Appendix E

Tiled Grace Tour Script

Speech	**Action**
This is a system for editing programs graphically. There are blocks of tiles you can drag around.	We would drag one block of tiles around the page at this point.
You can drag them into things, and you can drag them out of things.	Here we would drag one group into the body of the object literal in the other group.
Over here you can choose from different sets of tile you might want to use.	We indicated the toolbox and changed between groups here
Suppose we want to make five plus two. Plus. Five. Two.	We constructed the expression `5 + 2` out of tiles.
Five divided by two.	We changed the expression to `5 / 2` to indicate how to change operators.
To delete something, drag it over here; it goes red, you let go, it's gone.	We deleted the `5 / 2` we had made.

Let's run this program. It's going to make us an artwork.	We ran the program.
Another thing we can do here is switch to a textual view of our code at any time.	We switched to the textual view.
This is real editable text; you can do anything you'd do with text here.	We would select text with the mouse at random here
Let's make the program move a little faster — ten units each time.	We changed the distance given to `forward` each step here.
And run.	We ran the program
And it's faster. Now we can switch back,	We switched back to the tiled view.
and you see that our 10 has come across with us too.	
Through the experiment you'll be working with different variants of the language. The effect of that will be that different tiles become available from the sidebar. This program is in a graphical dialect with shapes and movement.	We indicated the groups in the toolbox again.
There's this stop button in the corner to stop the programs running when you need to.	We indicated the stop button in the corner.

Now let's suppose there's an error in my program. Let's change this 10 to say "blah".	We changed the 10 we had inserted into the forward() tile to "blah". We chose this error to introduce as it is not one deliberately included as a part of any experimental task.
Now if I try to run it, I can't, and it flashes at me; if I try to switch views, I can't, and it flashes at me.	We clicked the "Run" and "Code View" buttons.
If I hover over this red square	We hovered the mouse pointer over the error indicator.
it fades out everything in the program except what it thinks is wrong, and it tries to tell us what the problem is. Here it says "'blah' is not a valid number.", which is fair enough, so let's change that back	We reverted "blah" to "10".
— the square's gone green, and we can run again, and it works.	
What about if we had an error in the text view?	We switched views to the text view.
Let's make the same change here,	Here we changed the same 10 to blah.

and we get a red square again, and a little marker in the margin. If we hover over that marker	We hovered the mouse pointer over the error indicator in the text margin.
it'll tell us what the error is on this line, just a compiler error — "unknown variable or method name 'blah'". If we try to run now,	We clicked the "Run" button.
we can't, and if we try to switch views	We clicked the "Code View" button.
we also can't, but it'll give us this popup. Here's the error we just saw, and here's an offer to reset our code back to the last version that didn't have an error in it, and that we can switch views with. Let's do that,	We chose "OK" in the DOM confirm() dialogue box.
and it's given us our 10 back, and we can switch views again. So you can change between tiles and text at any time unless there's an error in the program at the moment.	We switched back to tiled view.
Through the experiment you'll be doing different tasks. Each of them will give you a program over here	We indicated the program source area on the left.

and a task up here	We indicated the task description text area in the top-right.
The tasks will be things like "change this program to do something different", "fix this problem the program has", "describe this program". When you're done with a task, click this button	We indicated the "Continue to next task" button.
to move on to the next one. It'll also prompt you to move on after a few minutes — don't worry about that, it's just so we can fit everything into a 30-minute window, just move on when it says. Now I'll get you into the experiment proper.	We switched the participant or participants to a different browser tab displaying the original tutorial program. This tab contained the instrumentation necessary to record interaction data.
This is the same program we were just looking at. You can get used to the system here, and then move on to the first task with this button when you're ready. The final questionnaire is in the next tab to the right.	We left participants at this point and retired to our observation point.

Bibliography

[1] Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steele Jr., and Sam Tobin-Hochstadt. The Fortress language specification version 1.0β. Technical report, Sun Microsystems, Inc., March 2007. Cited on page 120.

[2] William F. Atchison, Samuel D. Conte, John W. Hamblen, Thomas E. Hull, Thomas A. Keenan, William B. Kehl, Edward J. McCluskey, Silvio O. Navarro, Werner C. Rheinboldt, Earl J. Schweppe, William Viavant, and David M. Young, Jr. Curriculum 68: Recommendations for academic programs in computer science: A report of the ACM curriculum committee on computer science. *Commun. ACM*, 11(3):151–197, March 1968. Cited on page 11.

[3] Lennart Augustsson, Howard Mansell, and Ganesh Sittampalam. Paradise: A two-stage DSL embedded in Haskell. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, ICFP '08, pages 225–228, New York, NY, USA, 2008. ACM. Cited on page 40.

[4] Daniel Ballinger, Robert Biddle, and James Noble. Spreadsheet visualisation to improve end-user understanding. In *Proceedings of the Asia-Pacific Symposium on Information Visualisation - Volume 24*, APVis '03, pages 99–109, Darlinghurst, Australia, Australia, 2003. Australian Computer Society, Inc. Cited on page 54.

[5] Wolmet Barendregt and Mathilde M. Bekker. Children may expect drag-and-drop instead of point-and-click. In *CHI '11 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '11, pages 1297–1302, New York, NY, USA, 2011. ACM. Cited on pages 266 and 279.

[6] Gavin M. Bierman, Erik Meijer, and Mads Torgersen. Lost in translation: Formalizing proposed extensions to C#. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, pages 479–498, New York, NY, USA, 2007. ACM. Cited on page 185.

[7] Andrew P. Black, Kim B. Bruce, Michael Homer, and James Noble. Grace: The absence of (inessential) difficulty. In *Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! '12, pages 85–98, New York, NY, USA, 2012. ACM. Cited on page 57.

[8] Andrew P. Black, Kim B. Bruce, Michael Homer, James Noble, Amy Ruskin, and Richard Yannow. Seeking grace: A new object-oriented language for novices. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, SIGCSE '13, pages 129–134, New York, NY, USA, 2013. ACM. Cited on page 57.

[9] Douglas Blank, Jennifer S. Kay, James B. Marshall, Keith O'Hara, and Mark Russo. Calico: A multi-programming-language, multi-context framework designed for computer science education. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education*, SIGCSE '12, pages 63–68, New York, NY, USA, 2012. ACM. Cited on pages 50 and 277.

[10] Blockly Project. Blockly web site. `https://code.google.com/p/blockly/`. Cited on pages 48 and 275.

[11] Bard Bloom, John Field, Nathaniel Nystrom, Johan Östlund, Gregor Richards, Rok Strniša, Jan Vitek, and Tobias Wrigstad. Thorn: Robust, concurrent, extensible scripting on the JVM. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 117–136, New York, NY, USA, 2009. ACM. Cited on page 21.

[12] Bard Bloom and Martin J. Hirzel. Robust scripting via patterns. *SIGPLAN Notices*, 48(2):29–40, October 2012. Cited on page 21.

[13] Matthias Blume, Umut A. Acar, and Wonseok Chae. Extensible programming with first-class cases. In *Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming*, ICFP '06, pages 239–250, New York, NY, USA, 2006. ACM. Cited on page 111.

[14] Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczales, and David A. Moon. Common LISP Object System specification X3J13 Document 88-002R. *ACM SIGPLAN Notices*, 23(SI):1–142, September 1988. Cited on page 120.

[15] Gilad Bracha, Peter von der Ahé, Vassili Bykov, Yaron Kashai, William Maddox, and Eliot Miranda. Modules as objects in Newspeak. In *Proceedings of the 24th European Conference on Object-oriented Programming*, ECOOP'10, pages 405–428, Berlin, Heidelberg, 2010. Springer-Verlag. Cited on pages 28, 131, and 144.

[16] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/XT 0.16: Components for transformation systems. In *Proceedings of the 2006 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, PEPM '06, pages 95–99, New York, NY, USA, 2006. ACM. Cited on page 44.

[17] Quinn Burke and Yasmin B. Kafai. The writers' workshop for youth programmers: Digital storytelling with Scratch in middle school classrooms. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education*, SIGCSE '12, pages 433–438, New York, NY, USA, 2012. ACM. Cited on pages 13 and 203.

[18] Eugene Burmako, Martin Odersky, Christopher Vogt, Stefan Zeiger, and Adriaan Moors. Scala macros. `http://scalamacros.org`, April 2012. Cited on pages 38 and 195.

[19] Margaret Burnett, Andrei Sheretov, Bing Ren, and Gregg Rothermel. Testing homogeneous spreadsheet grids with the "what you see is what you test" methodology. *IEEE Transactions on Software Engineering*, 28(6):576–594, June 2002. Cited on page 54.

[20] Calico Project. Calico web site. `http://calicoproject.org/Calico`. Cited on page 50.

[21] Rachel Cardell-Oliver. How can software metrics help novice programmers? In *Proceedings of the Thirteenth Australasian Computing Education Conference - Volume 114*, ACE '11, pages 55–62, Darlinghurst, Australia, Australia, 2011. Australian Computer Society, Inc. Cited on page 9.

[22] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson. Modula-3 language definition. *SIGPLAN Notices*, 27(8):15–42, August 1992. Cited on page 32.

[23] Janet Carter, Dennis Bouvier, Rachel Cardell-Oliver, Margaret Hamilton, Stanislav Kurkovsky, Stefanie Markham, O. William McClung, Roger McDermott, Charles Riedesel, Jian Shi, and Su White. Motivating all our students? In *Proceedings of the 16th Annual Conference Reports on Innovation and Technology in Computer Science Education -*

Working Group Reports, ITiCSE-WGR '11, pages 1–18, New York, NY, USA, 2011. ACM. Cited on page 5.

[24] Michael E. Caspersen and Jens Bennedsen. Instructional design of a programming course: A learning theoretic approach. In *Proceedings of the Third International Workshop on Computing Education Research*, ICER '07, pages 111–122, New York, NY, USA, 2007. ACM. Cited on page 10.

[25] Craig Chambers. The Cecil language, specification and rationale. Technical report, Department of Computer Science and Engineering, University of Washington, 1993. Cited on page 120.

[26] David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '98, pages 48–64, New York, NY, USA, 1998. ACM. Cited on page 167.

[27] Douglas H. Clements and Dominic F. Gullo. Effects of computer programming on young children's cognition. *Journal of Educational Psychology*, 76(6), 1984. Cited on page 8.

[28] Damian Conway. Leading::Zeros Perl module. `http://search.cpan.org/~dconway/Leading-Zeros-0.0.2/`. Cited on page 45.

[29] Damian Conway. Lingua::Romana::Perligata. `http://search.cpan.org/~dconway/Lingua-Romana-Perligata/`. Cited on page 45.

[30] Damian Conway. Smart::Comments perl module. `http://search.cpan.org/~dconway/Smart-Comments-1.000005/`. Cited on page 46.

[31] Stephen Cooper, Wanda Dann, and Randy Pausch. Teaching objects-first in introductory computer science. In *ACM SIGCSE Bulletin*, volume 35, 2003. Cited on pages 51 and 275.

[32] J.R. Cordy, C.D. Halpern, and E. Promislow. TXL: a rapid prototyping system for programming language dialects. In *Computer Languages, 1988. Proceedings., International Conference on*, pages 280–285, Oct 1988. Cited on page 44.

[33] Ryan Culpepper, Sam Tobin-Hochstadt, and Matthew Flatt. Advanced macrology and the implementation of Typed Scheme. In *ICFP workshop on Scheme and Functional Programming*, 2007. Cited on page 35.

[34] Dave Cunningham, Rajesh Bordawekar, and Vijay Saraswat. GPU programming in a high level language: Compiling X10 to CUDA. In *SIGPLAN X10 Workshop*, pages 8:1–8:10, New York, NY, USA, 2011. ACM. Cited on page 181.

[35] J Dalbey and M. C. Linn. The demands and requirements of computer programming: a literature review. *Journal of Educational Computing Research*, 1(3), 1985. Cited on pages 7 and 8.

[36] Wanda P. Dann, Stephen Cooper, and Randy Pausch. *Learning to Program with Alice*. Pearson Prentice Hall, 2011. Cited on page 13.

[37] J. Dedecker, T. Van Cutsem, S. Mostinckx, T. D'Hondt, and W. De Meuter. Ambient-oriented programming in AmbientTalk. In *Proceedings of the 20th European Conference on Object-Oriented Programming*, ECOOP'06, pages 230–254, 2006. Cited on page 29.

[38] Victor R. Delclos, Joan Littlefield, and John D. Bransford. Teaching thinking through Logo: The importance of method. *Roeper Review*, 7(3), 1985. Cited on page 8.

[39] Véronique Donzeau-Gouge, Gilles Kahn, Bernard Lang, Bertrand Melese, and Elham Morcos. Outline of a tool for document manipulation. In *IFIP Congress*, pages 615–620, 1983. Cited on page 41.

[40] Droplet authors. Droplet source repository. https://github.com/dabbler0/droplet. Cited on pages 54 and 276.

[41] Jennifer L. Dyck and Richard E. Mayer. Teaching for transfer of computer program comprehension skill. *Journal of Educational Psychology*, 81(1), 1989. Cited on pages 7 and 8.

[42] Eclipse Software Foundation. Little tortoise Xtext language. http://www.eclipse.org/Xtext/7languagesDoc.html#tortoise, last accessed June 4, 2014. Cited on pages 198 and 199.

[43] Eclipse Software Foundation. Xtext web site. http://www.eclipse.org/Xtext/index.html, 2013. Cited on pages 43 and 156.

[44] Standard ECMA-262 ECMAScript Language Specification, 3rd Edition. Technical report, Ecma International, 1999. Cited on page 296.

[45] ECMA-262, 5th Edition. Technical report, Ecma International, 2009. Cited on page 296.

[46] Burak Emir, Martin Odersky, and John Williams. Matching objects with patterns. In *Proceedings of the 21st European Conference on Object-Oriented Programming*, ECOOP'07, pages 273–298, Berlin, Heidelberg, 2007. Springer-Verlag. Cited on pages 16, 17, 74, and 104.

[47] EPFL. Environment, universes, and mirrors – Scala documentation. http://docs.scala-lang.org/overviews/reflection/environment-universes-mirrors.html, 2013. Cited on pages 38 and 195.

[48] Moritz Eysholdt and Heiko Behrens. Xtext: Implement your language faster than the quick and dirty way. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, SPLASH '10, pages 307–309, New York, NY, USA, 2010. ACM. Cited on pages 43 and 198.

[49] Mattias Felleisen, R. B. Findler, M. Flatt, and S. Krishnamurthi. The DrScheme project: An overview. *SIGPLAN Notices*, 33(6):17–23, June 1998. Cited on pages 12 and 13.

[50] Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen. DrScheme: a programming environment for Scheme. *J. Funct. Program.*, 12(2):159–182, 2002. Cited on page 35.

[51] Bryan Ford. Parsing expression grammars: A recognition-based syntactic foundation. *SIGPLAN Notices*, 39(1):111–122, January 2004. Cited on page 22.

[52] Martin Fowler. *Domain Specific Languages*. AW, 2011. Cited on page 38.

[53] Diana Franklin, Phillip Conrad, Bryce Boe, Katy Nilsen, Charlotte Hill, Michelle Len, Greg Dreschler, Gerardo Aldana, Paulo Almeida-Tanaka, Brynn Kiefer, Chelsea Laird, Felicia Lopez, Christine Pham, Jessica Suarez, and Robert Waite. Assessment of computer science learning in a Scratch-based outreach program. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, SIGCSE '13, pages 371–376, New York, NY, USA, 2013. ACM. Cited on pages 13 and 203.

[54] Free Software Foundation. The GNU General Public License v3.0. `https://gnu.org/licenses/gpl`, accessed May 15 2014. Cited on page 282.

[55] N. Freed, J. Klensin, and T. Hansen. Media Type Specifications and Registration Procedures. RFC 6838 (Best Current Practice), January 2013. Cited on page 141.

[56] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, 1994. Cited on pages 87 and 154.

[57] Sandy Garner, Patricia Haden, and Anthony Robins. My program is correct but it doesn't run: A preliminary investigation of novice programmers' problems. In *Proceedings of the 7th Australasian Conference on Computing Education - Volume 42*, ACE '05, pages 173–180, Darlinghurst, Australia, Australia, 2005. Australian Computer Society, Inc. Cited on page 9.

[58] Felix Geller, Robert Hirschfeld, and Gilad Bracha. Pattern matching for an object-oriented and dynamically typed programming language. Technical Report 36, Hasso-Plattner-Instituts für Sofwaresystemtechnik an der Universität Potsdam, 2010. Cited on pages 16, 17, 19, 74, 104, and 108.

[59] Douglas J. Gillan, Kritina Holden, Susan Adam, Marianne Rudisill, and Laura Magee. How does Fitts' law fit pointing and dragging? In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '90, pages 227–234, New York, NY, USA, 1990. ACM. Cited on page 266.

[60] Go Project. Go language website. `http://golang.org/`, 2013. Cited on pages 27, 30, and 145.

[61] Valentina Grigoreanu, Margaret Burnett, Susan Wiedenbeck, Jill Cao, Kyle Rector, and Irwin Kwan. End-user debugging strategies: A sensemaking perspective. *ACM Transactions on Computer-Human Interaction*, 19(1):5:1–5:28, May 2012. Cited on pages 54 and 216.

[62] A N Habermann and D Notkin. Gandalf: Software development environments. *IEEE Transactions on Software Engineering*, 12(12):1117–1127, December 1986. Cited on page 41.

[63] Tianyi David Han and Tarek S. Abdelrahman. hiCUDA: a high-level directive-based language for GPU programming. GPGPU-2, pages 52–61, New York, NY, USA, 2009. ACM. Cited on page 181.

[64] Karl Hasselström and Jon Åslund. The Shakespeare programming language. http://shakespearelang.sourceforge.net, August 2001. Cited on page 45.

[65] Philip Hazel. *Perl-compatible regular expressions*. The University of Cambridge, 2012. pcre.org. Cited on page 114.

[66] Martin Hirzel, Nathaniel Nystrom, Bard Bloom, and Jan Vitek. Matchete: Paths through the pattern matching jungle. In *PADL*, 2008. Cited on page 21.

[67] Richard C. Holt and David B. Wortman. A sequence of structured subsets of PL/I. *SIGCSE Bulletin*, 6(1):129–132, January 1974. Cited on pages 12 and 35.

[68] Michael Homer. Grace-CUDA source repository. https://github.com/mwh/grace-cuda, 2013. Cited on pages 178 and 300.

[69] Michael Homer. Grace-GTK source repository. https://github.com/mwh/grace-gtk, 2013. Cited on pages 137 and 298.

[70] Michael Homer, Kim B. Bruce, James Noble, and Andrew P. Black. Modules as gradually-typed objects. In *Proceedings of the 7th Workshop on Dynamic Languages and Applications*, DYLA '13, pages 1:1–1:8, New York, NY, USA, 2013. ACM. Cited on pages 2 and 123.

[71] Michael Homer, Timothy Jones, James Noble, Kim B. Bruce, and Andrew P. Black. Graceful dialects. In Richard Jones, editor, *ECOOP*

2014 — Object-Oriented Programming, volume 8586 of *Lecture Notes in Computer Science*, pages 131–156. Springer Berlin Heidelberg, 2014. Cited on pages 2, 149, and 173.

[72] Michael Homer and James Noble. A tile-based editor for a textual programming language. In *Proceedings of IEEE Working Conference on Software Visualization*, VISSOFT'13, pages 1–4, Sept 2013. Cited on pages 2 and 203.

[73] Michael Homer and James Noble. Combining tiled and textual views of code. In *Proceedings of IEEE Working Conference on Software Visualization*, VISSOFT'14, Sept 2014. Cited on pages 2 and 203.

[74] Michael Homer, James Noble, Kim B. Bruce, Andrew P. Black, and David J. Pearce. Patterns as objects in Grace. In *Proceedings of the 8th Symposium on Dynamic Languages*, DLS '12, pages 17–28, New York, NY, USA, 2012. ACM. Cited on pages 2 and 73.

[75] R Nigel Horspool, Judith Bishop, Arjmand Samuel, Nikolai Tillmann, Michał Moskal, Jonathan de Halleux, and Manuel Fähndrich. *TouchDevelop: Programming on the Go*. Microsoft Research, 2013. Cited on page 52.

[76] Andrew Hunt and David Thomas. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley, 1999. Cited on page 135.

[77] IBM Globalization Center of Competency. ICU – international components for Unicode. `http://site.icu-project.org/`. Cited on page 295.

[78] Kazuhiro Ichikawa and Shigeru Chiba. Composable user-defined operators that can express user-defined literals. In *Proceedings of the 13th International Conference on Modularity*, MODULARITY '14, pages 13–24, New York, NY, USA, 2014. ACM. Cited on page 43.

[79] IEEE and The Open Group. dlopen. In POSIX.1-2008 [80]. URL: http://pubs.opengroup.org/onlinepubs/9699919799/functions/dlopen.html, 2013. Cited on pages 190 and 289.

[80] IEEE and The Open Group. *POSIX.1-2008; The Open Group Base Specifications, Issue 7; IEEE Std 1003.1, 2013 Edition*. IEEE, 2013. Cited on page 354.

[81] Roberto Ierusalimschy. A text pattern-matching tool based on parsing expression grammars. *Softw. Pract. Exper.*, 39(3). Cited on pages 22 and 115.

[82] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. The evolution of Lua. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, HOPL III, pages 2–1–2–26, New York, NY, USA, 2007. ACM. Cited on page 34.

[83] Takeo Igarashi, Jock D. Mackinlay, Bay-Wei Chang, and Polle T. Zellweger. Fluid visualization of spreadsheet structures. In *Proceedings of the IEEE Symposium on Visual Languages*, VL '98, pages 118–, Washington, DC, USA, 1998. IEEE Computer Society. Cited on page 54.

[84] Kori M. Inkpen. Drag-and-drop versus point-and-click mouse interaction styles for children. *ACM Transactions on Computer-Human Interaction*, 8(1):1–33, March 2001. Cited on page 266.

[85] Barry Jay. *Pattern Calculus: Computing with Functions and Structures*. Springer-Verlag, 2009. Cited on pages 15 and 23.

[86] R. A. Jeffries. Comparison of debugging behavior of novice and expert programmers. In *AERA Annual Meeting*, 1982. Cited on page 216.

[87] Kathleen Jensen and Niklaus Wirth. *Pascal User Manual and Report*. Springer, 1975. Cited on page 12.

[88] Association for Computing Machinery (ACM) Joint Task Force on Computing Curricula and IEEE Computer Society. *Computer Science Curricula 2013: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science*. ACM, New York, NY, USA, 2013. 999133. Cited on pages 11 and 73.

[89] Mark P. Jones. Experience report: playing the DSL card. In *Proceedings of the 13th ACM SIGPLAN international conference on Functional programming (ICFP)*, ICFP '08, pages 87–90, New York, NY, USA, 2008. ACM. Cited on page 40.

[90] Seonghoon Kang and Sukyoung Ryu. Formal specification of a JavaScript module system. *SIGPLAN Notices*, 47(10):621–638, October 2012. Cited on page 30.

[91] Lennart C.L. Kats and Eelco Visser. The Spoofax language workbench: Rules for declarative specification of languages and ides. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 444–463, New York, NY, USA, 2010. ACM. Cited on page 45.

[92] Caitlin Kelleher, Randy Pausch, and Sara Kiesler. Storytelling Alice motivates middle school girls to learn computer programming. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '07, pages 1455–1464, New York, NY, USA, 2007. ACM. Cited on page 235.

[93] John G. Kemeny and Thomas E. Kurtz. *A Manual for BASIC, the elementary language designed for use with the Dartmouth Time Sharing System*. Available from `http://bitsavers.informatik.uni-stuttgart.de/pdf/dartmouth/BASIC_Oct64.pdf`. Cited on page 11.

[94] Brian W. Kernighan. Why Pascal is not my favourite programming language. Computing Science TR 100, AT&T Bell Laboratories, 1981. Available from http://cm.bell-labs.com/cm/cs/cstr/100.ps.gz. Cited on page 12.

[95] Michael Kölling. The Greenfoot programming environment. *ACM Transactions on Computer Education*, 10(4):14:1–14:21, November 2010. Cited on page 52.

[96] B. B. Kristensen, Ole L. Madsen, B. Möller-Pedersen, and K. Nygaard. Syntax-directed program modularization. In P. Degano and E. Sandewall, editors, *Integrated Interactive Computing Systems*, pages 207–219. North-Holland, Amsterdam, 1983. Cited on page 31.

[97] Chris Lattner. LLVM: An infrastructure for multi-stage optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, 2002. Cited on page 292.

[98] Arnaud Le Hors, Philippe Le Hégaret, Lauren Wood, Gavin Nicol, Jonathan Robie, Mike Champion, and Steve Byrne. Document Object Model (DOM) level 3 core specification. http://www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407/, April 2004. Cited on page 207.

[99] LEGO Group. Lego Mindstorms web site. http://mindstorms.lego.com/. Cited on page 49.

[100] D. Leijen and E. Meijer. Parsec: Direct style monadic parser combinators for the real world. Technical Report UU-CS-2001-27, Universiteit Utrecht, 2001. Cited on page 114.

[101] U. Leron. Quasi-Piagetian learning in Logo. *Journal of Computers in Mathematics and Science Teaching*, 4, 1984. Cited on page 8.

[102] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. The OCaml system release 3.12: Documentation and user's manual. http://caml.inria.fr/pub/docs/manual-ocaml/index.html, July 2011. Cited on pages 16, 21, and 113.

[103] Colleen Lewis, Sarah Esper, Victor Bhattacharyya, Noelle Fa-Kaji, Neftali Dominguez, and Arielle Schlesinger. Children's perceptions of what counts as a programming language. *J. Comput. Sci. Coll.*, 29(4):123–133, April 2014. Cited on page 210.

[104] Colleen M. Lewis. How programming environment shapes perception, learning and goals: Logo vs. Scratch. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education*, SIGCSE '10, pages 346–350, New York, NY, USA, 2010. ACM. Cited on page 210.

[105] Sheng Liang. *Java Native Interface: Programmer's Guide and Reference*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1999. Cited on page 34.

[106] Barbara Liskov, Alan Snyder, Russell Atkinson, and Craig Schaffert. Abstraction mechanisms in CLU. *Comm. ACM*, 20(8):564–576, August 1977. Cited on page 31.

[107] Raymond Lister. Computing education research: Geek genes and bimodal grades. *ACM Inroads*, 1(3):16–17, September 2011. Cited on page 6.

[108] Raymond Lister and Ilona Box. A citation analysis of the SIGCSE 2007 proceedings. In *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '08, pages 476–480, New York, NY, USA, 2008. ACM. Cited on page 9.

[109] David C. Littman, Jeannine Pinto, Stanley Letovsky, and Elliot Soloway. Mental models and software maintenance. In *Papers Pre-*

sented at the First Workshop on Empirical Studies of Programmers, pages 80–98, Norwood, NJ, USA, 1986. Ablex Publishing Corp. Cited on page 124.

[110] LLVM website. http://www.llvm.org/. Cited on page 292.

[111] Robert Lockhart. Codemancer home page. http://codemancergame.com/. Cited on page 49.

[112] David H. Lorenz and Boaz Rosenan. Cedalion: A language for language oriented programming. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 733–752, New York, NY, USA, 2011. ACM. Cited on pages 35, 41, and 196.

[113] I. Scott MacKenzie, Abigail Sellen, and William A. S. Buxton. A comparison of input devices in element pointing and dragging tasks. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '91, pages 161–166, New York, NY, USA, 1991. ACM. Cited on page 266.

[114] David MacQueen. Modules for Standard ML. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, LFP '84, pages 198–207. ACM, 1984. Cited on page 31.

[115] John H. Maloney, Kylie Peppler, Yasmin Kafai, Mitchel Resnick, and Natalie Rusk. Programming by choice: Urban youth learning programming with Scratch. *SIGCSE Bulletin*, 40(1):367–371, March 2008. Cited on pages 13 and 203.

[116] Shane Markstrum, Daniel Marino, Matthew Esquivel, Todd D. Millstein, Chris Andreae, and James Noble. JavaCOP: Declarative pluggable types for Java. *ACM Transactions on Programming Languages and Systems*, 32(2), 2010. Cited on page 46.

[117] Simon Marlow. Haskell 2010 language report, 2010. Available at http://www.haskell.org/onlinereport/haskell2010/. Cited on pages 16, 23, and 34.

[118] Paul Marquess. Source filters. *The Perl Journal*, Issue #11, 1988. Cited on page 45.

[119] Eugene McArdle, Jason Holdsworth, and Ickjai Lee. Assessing the usability of students object-oriented language with first-year IT students: A case study. In *Proceedings of the 25th Australian Computer-Human Interaction Conference: Augmentation, Application, Innovation, Collaboration*, OzCHI '13, pages 181–188, New York, NY, USA, 2013. ACM. Cited on page 14.

[120] Eugene McArdle, Jason Holdsworth, and Sui Man Lui. Usability evaluation of SOLA: An object-oriented programming environment for children. In *World Conference on E-Learning in Corporate, Government, Healthcare, and Higher Education*, 2009. Cited on page 14.

[121] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part I. *Commun. ACM*, 3(4):184–195, April 1960. Cited on page 12.

[122] James McKinna. Why dependent types matter. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06, pages 1–1, New York, NY, USA, 2006. ACM. Cited on page 103.

[123] Orni Meerbaum-Salant, Michal Armoni, and Mordechai Ben-Ari. Habits of programming in scratch. In *Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education*, ITiCSE '11, pages 168–172, New York, NY, USA, 2011. ACM. Cited on pages 13 and 274.

[124] Orni Meerbaum-Salant, Michal Armoni, and Mordechai (Moti) Ben-Ari. Learning computer science concepts with Scratch. In *Proceedings of the Sixth International Workshop on Computing Education Research*, ICER '10, pages 69–76, New York, NY, USA, 2010. ACM. Cited on pages 13 and 203.

[125] Bertrand Meyer. *Eiffel: The Language*. Prentice Hall, 1992. Cited on page 158.

[126] Microsoft. Microsoft PE and COFF specification. Technical report, Microsoft, 2013. Available from http://msdn.microsoft.com/en-us/windows/hardware/gg463119.aspx, accessed April 9, 2014. Cited on page 191.

[127] Eliot Miranda. Newspeak foreign function interface user guide. http://wiki.squeak.org/squeak/uploads/6100/Alien%20FFI.pdf, 2009. Cited on page 34.

[128] Hisham Muhammad, Fabio Mascarenhas, and Roberto Ierusalimschy. LuaRocks - a declarative and extensible package management system for Lua. In André Rauber Du Bois and Phil Trinder, editors, *Programming Languages*, volume 8129 of *Lecture Notes in Computer Science*, pages 16–30. Springer Berlin Heidelberg, 2013. Cited on page 132.

[129] Laurie Murphy and Lynda Thomas. Dangers of a fixed mindset: Implications of self-theories research for computer science education. *SIGCSE Bulletin*, 40(3):271–275, June 2008. Cited on page 10.

[130] Linda MvIver and Damian Conway. GRAIL: A zeroth programming language. In *International Conference on Computers in Education*, 1999. Cited on page 14.

[131] Ligia Nistor, Darya Kurilova, Stephanie Balzer, Benjamin Chung, Alex Potanin, and Jonathan Aldrich. Wyvern: A simple, typed,

and pure object-oriented language. In *Proceedings of the 5th Workshop on MechAnisms for SPEcialization, Generalization and inHerItance*, MASPEGHI '13, pages 9–16, New York, NY, USA, 2013. ACM. Cited on pages 42 and 196.

[132] James Noble and John Grundy. Explicit relationships in object oriented development. In *TOOLS 18*. Prentice Hall, 1995. Cited on page 175.

[133] Ulf Norell. Dependently typed programming in Agda. In Pieter W. M. Koopman, Rinus Plasmeijer, and S. Doaitse Swierstra, editors, *Advanced Functional Programming*, volume 5832 of *Lecture Notes in Computer Science*, pages 230–266. Springer, 2008. Cited on page 103.

[134] NVIDIA Corporation. CUDA parallel programming and computing platform web site. `http://www.nvidia.com/object/cuda_home_new.html`, 2013. Cited on pages 178 and 300.

[135] Martin Odersky. In defense of pattern matching. Accessed Aug 2012. `http://www.artima.com/weblogs/viewpost.jsp?thread=166742`, June 2006. Cited on page 104.

[136] Martin Odersky. Pattern matching wrap-up. Accessed Aug 2012. `http://www.artima.com/weblogs/viewpost.jsp?thread=168839`, July 2006. Cited on page 104.

[137] Martin Odersky. Scala contracts. In *Runtime Verification*, 2010. Cited on page 158.

[138] Martin Odersky. The Scala language specification. Technical report, Programming Methods Laboratory, EPFL, 2011. Cited on pages 32, 37, and 195.

[139] Cyrus Omar, Benjamin Chung, Darya Kurilova, Alex Potanin, and Jonathan Aldrich. Type-directed, whitespace-delimited parsing for

embedded dsls. In *Proceedings of the First Workshop on the Globalization of Domain Specific Languages*, GlobalDSL '13, pages 8–11, New York, NY, USA, 2013. ACM. Cited on pages 42 and 196.

[140] Cyrus Omar, Darya Kurilova, Ligia Nistor, Benjamin Chung, Alex Potanin, and Jonathan Aldrich. Safely composable type-specific languages. In *Proceedings of the 28th European Conference on Object-oriented Programming*, ECOOP'14, 2014. To appear. Cited on page 42.

[141] Cyrus Omar, YoungSeok Yoon, Thomas D. LaToza, and Brad A. Myers. Active code completion. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 859–869, Piscataway, NJ, USA, 2012. IEEE Press. Cited on page 54.

[142] D. B. Palumbo. Programming language/problem-solving research: a review of relevant issues. *Review of Educational Research*, 60(1):65–89, 1990. Cited on pages 7 and 211.

[143] Seymour Papert. *Mindstorms: children, computers, and powerful ideas*. Basic Books, Inc., 1980. Cited on pages 8, 13, and 49.

[144] Matthew M. Papi, Mahmood Ali, Telmo Luis Correa, Jr., Jeff H. Perkins, and Michael D. Ernst. Practical pluggable types for Java. In *ISSTA*, 2008. Cited on page 46.

[145] Parrot Foundation. Parrot virtual machine. `http://parrot.org/`. Cited on page 292.

[146] Dale Parsons and Patricia Haden. Programming osmosis: Knowledge transfer from imperative to visual programming environments, 2007. Cited on page 203.

[147] David J. Pearce and Lindsay Groves. Whiley: A platform for research in software verification. In Martin Erwig, Richard F. Paige, and Eric Wyk, editors, *Software Language Engineering*, volume 8225 of *Lecture*

Notes in Computer Science, pages 238–248. Springer International Publishing, 2013. Cited on page 103.

[148] 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. Cited on page 5.

[149] 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. Cited on page 7.

[150] D. N. Perkins and Gavriel Salomon. Teaching for transfer. *Educational Leadership*, 22(32), 1988. Cited on pages 6 and 7.

[151] Perl Foundation. The Perl programming language. `http://www.perl.org/`, 2013. Cited on pages 34 and 45.

[152] Simon Peyton Jones. View patterns: lightweight views for Haskell. `http://hackage.haskell.org/trac/ghc/wiki/ViewPatterns`, 2007. Cited on page 25.

[153] Jean Piaget. *Play, dreams and imitation in childhod*. Cited on page 8.

[154] Jean Piaget. *Science of education and the psychology of the child*. Cited on page 8.

[155] Jean Piaget. *The child's conception of the world*. Rowman & Littlefield, 1951. Cited on page 8.

[156] Jean Piaget. *Six psychological studies*. Vintage, 1968. Cited on page 8.

[157] Leonardo Pisano. *Liber Abaci*. 1202. Cited on pages 24 and 227.

[158] Kris Powers, Stacey Ecott, and Leanne M. Hirshfield. Through the looking glass: Teaching CS0 with Alice. *SIGCSE Bulletin*, 39(1):213–217, March 2007. Cited on pages 203, 210, 211, and 276.

[159] Python Software Foundation. Python website. `http://python.org/`, 2013. Cited on pages 27, 34, and 126.

[160] Red Hat. Cygwin. `http://www.cygwin.com/`. Cited on page 190.

[161] Claus Reinke. Introduce lambda-match (explicit match failure and fall-through). Haskell-Prime Ticket #144, `http://hackage.haskell.org/trac/haskell-prime/ticket/114`, October 2006. Cited on page 24.

[162] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and Yasmin Kafai. Scratch: programming for all. *Communications of the ACM*, 52(11):60–67, November 2009. Cited on pages 13, 47, 203, 206, and 273.

[163] Adam Richard and Ondrej Lhotak. OOMatch: Pattern matching as dispatch in java. In *Companion to the 22nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion*, OOPSLA '07, pages 771–772, New York, NY, USA, 2007. ACM. Cited on page 21.

[164] Anthony Robins. Transfer in cognition. *Connection Science*, 8(2):185–204, 1996. Cited on page 6.

[165] Anthony Robins. Learning edge momentum: a new account of outcomes in CS1. *Computer Science Education*, 20(1):37–71, 2010. Cited on pages 6 and 9.

[166] Anthony Robins, Janet Rountree, and Nathan Rountree. Learning and teaching programming: A review and discussion. *Computer Science Education*, 13:137–172, 2003. Cited on page 5.

[167] Tiark Rompf and Martin Odersky. Lightweight modular staging: A pragmatic approach to runtime code generation and compiled dsls. In *Proceedings of the Ninth International Conference on Generative Programming and Component Engineering*, GPCE '10, pages 127–136, New York, NY, USA, 2010. ACM. Cited on pages 37 and 195.

[168] Ruby. Ruby language. `http://www.ruby-lang.org/`. Cited on page 34.

[169] RubyGems. Rubygems. `http://www.rubygems.org/`. The `gem` tool is bundled with Ruby. Cited on page 132.

[170] Sukyoung Ryu, Changhee Park, and Guy L. Steele Jr. Adding pattern matching to existing object-oriented languages. In *FOOL*, 2010. Cited on page 21.

[171] Michael G. Schwern. Lingua::tlhInganHol::yIghun. `http://search.cpan.org/dist/Lingua-tlhInganHol-yIghun/`. Cited on page 45.

[172] Sisyphus. Inline::Python – write Perl subroutines in C. `http://search.cpan.org/dist/Inline-Python/`. Cited on page 45.

[173] Andreas Stefik and Susanna Siebert. An empirical investigation into programming language syntax. *ACM Transactions on Computing Education*, 13(4):19:1–19:40, November 2013. Cited on page 14.

[174] Rok Strniša, Peter Sewell, and Matthew Parkinson. The Java module system: core design and semantic definition. *SIGPLAN Notices*, 42(10):499–514, October 2007. Cited on page 32.

[175] Don Syme, Keith Battocchi, Kenji Takeda, Donna Malayeri, Jomo Fisher, Jack Hu, Tao Liu, Brian McNamara, Daniel Quirk, Matteo Taveggia, Wonseok Chae, Uladzimir Matsveyeu, and Tomas Petricek. Strongly-typed language support for internet-scale information sources. Technical Report MSR-TR-2012-101, Microsoft Research, September 2012. Cited on pages 34 and 138.

[176] Don Syme, Gregory Neverov, and James Margetson. Extensible pattern matching via a lightweight language extension. In *ICFP*, 2007. Cited on pages 16 and 25.

[177] Clemens A. Szyperski. Import is not inheritance — why we need both: Modules and classes. In OleLehrmann Madsen, editor, *ECOOP '92 European Conference on Object-Oriented Programming*, volume 615 of *Lecture Notes in Computer Science*, pages 19–32. Springer Berlin Heidelberg, 1992. Cited on pages 33 and 124.

[178] `http://racket-lang.org`. Racket documentation - ALGOL-60. `http://docs.racket-lang.org/algol60/index.html`, 2013. Cited on page 36.

[179] The Self-Appointed Master Librarians (OOK!) of the CPAN. The Comprehensive Perl Archive Network. `http://www.cpan.org/`. The `cpan` tool is bundled with Perl. See also `man 1 cpan`. Cited on page 132.

[180] TIOBE Software. Tiobe index. `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`, accessed May 2014. Cited on page 12.

[181] Sam Tobin-Hochstadt. Extensible Pattern Matching in an Extensible Language. `http://arxiv.org/abs/1106.2578v1` [cs.PL]. Cited on pages 15, 26, and 112.

[182] Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. Languages as libraries. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 132–141, New York, NY, USA, 2011. ACM. Cited on pages 13, 32, 35, 54, 184, and 216.

[183] Laurence Tratt. Domain specific language implementation via compile-time meta-programming. *ACM Transactions on Programming Languages and Systems*, 30(6):31:1–31:40, October 2008. Cited on page 41.

[184] Mark Tullsen. First-class patterns. In *PADL*, number 1753 in LNCS, 2000. Cited on page 24.

[185] David Ungar. Annotating objects for transport to other worlds. *SIGPLAN Notices*, 30(10):73–87, October 1995. Cited on page 29.

[186] David Ungar and Randall B. Smith. Self: The power of simplicity. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, OOPSLA '87, pages 227–242. ACM, 1987. Cited on page 29.

[187] David W. Valentine. CS educational research: A meta-analysis of SIGCSE technical symposium proceedings. *SIGCSE Bulletin*, 36(1):255–259, March 2004. Cited on page 9.

[188] Tom van Cutsem. AmbientTalk documentation: Modular programming. `http://soft.vub.ac.be/amop/at/tutorial/modular`. Cited on page 29.

[189] Joost Visser. Matching objects without language extension. *Journal of Object Technology*, 5(8), 2006. `http://www.jot.fm/issues/issue200611/article2`. Cited on page 22.

[190] Markus Voelter. Embedded software development with projectional language workbenches. In *Proceedings of the 13th international conference on Model driven engineering languages and systems: Part II*, MODELS'10, pages 32–46, Berlin, Heidelberg, 2010. Springer-Verlag. Cited on pages 41 and 197.

[191] P. Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '87, pages 307–313, New York, NY, USA, 1987. ACM. Cited on page 24.

[192] Alessandro Warth and Ian Piumarta. OMeta: An object-oriented language for pattern matching. In *Proceedings of the 2007 Symposium on Dynamic Languages*, DLS '07, pages 11–19, New York, NY, USA, 2007. ACM. Cited on pages 16, 22, and 115.

[193] Neil Watkiss and Stefan Seifert. Inline::C – write Perl subroutines in C. `http://search.cpan.org/dist/Inline/`. Cited on page 45.

[194] Allen Wirfs-Brock and Brian Wilkerson. A overview of Modular Smalltalk. volume 23, pages 123–134, New York, NY, USA, January 1988. ACM. Cited on page 32.

[195] N. Wirth. The programming language PASCAL. *Acta Informatica*, 1(1), 1971. Cited on page 12.

[196] Niklaus Wirth. *Programming in Modula-2*. Springer Verlag, 1985. isbn 0-387-15078-1. Cited on page 128.

[197] William A. Wulf, Ralph L. London, and Mary Shaw. An introduction to the construction and verification of Alphard programs. *IEEE Transactions on Software Engineering*, SE-2(4):253–265, 1976. Cited on page 31.