Educational Gems: An Exploration and Evaluation of A Visual Functional Programming Environment

by

Adam Robert Parkin
B.Sc., University of Victoria, 2005

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

Educational Gems: An Exploration and Evaluation of A Visual Functional Programming Environment

by

Adam Robert Parkin
B.Sc., University of Victoria, 2005

Supervisory Committee

Dr. Y. Coady, Co-Supervisor
(Department of Computer Science)

Dr. G. Tzanetakis, Co-Supervisor
(Department of Computer Science)

Dr. M. Zastre, Departmental Member
(Department of Computer Science)

Supervisory Committee

Dr. Y. Coady, Co-Supervisor
(Department of Computer Science)

Dr. G. Tzanetakis, Co-Supervisor
(Department of Computer Science)

Dr. M. Zastre, Departmental Member
(Department of Computer Science)

ABSTRACT

This thesis examines the pedagogical value of a particular visual programming environment (VPE) called the Gem Cutter which is based upon the functional programming paradigm. The contribution of this thesis is two-fold: it provides a qualitative evaluation via the Cognitive Dimensions Framework developed by Green to explore the usefulness of the Gem Cutter environment from a pedagogical viewpoint, and secondly provides a framework called the Word Game Framework designed in the Gem Cutter which can be used to create exercises for sudents learning to program. The Word Game Framework allows students to create interactive turn-based "word-games" and provides an engaging environment for students to explore interesting and useful functional programming concepts such as recursion, higher order functions, type inference, and list processing.

Contents

List of Tables

List of Figures

List of Programs

ACKNOWLEDGEMENTS

it was a long road...
Derek Church [4]

Having been a graduate student for far too long, I have many thanks for many people. To enumerate them all would produce a document of thesis-like length, so apologies in advance for anyone I miss from this list.

Firstly, my SUPERvisors: Dr. G. Tzanetakis saw a 3rd year undergraduate student (who was a foolishly shameful advocate of the C++ programming language) in his programming languages class and introduced him to the beautiful world of functional programming. Dr. Y. Coady saw a geeky, shy 4th year student in her concurrency class and somehow convinced him to undertake graduate studies. Both of them showed a seemingly limitless amount of patience, support, and encouragement throughout my studies and I thank them for this.

I also have to thank the far too many brilliant people at Business Objects (past and present) for both creating a wonderful programming environment, as well as for providing much inspiration along the way. In particular, Luke Evans provided much of the early inspiration behind this thesis and I thank him for his input and insight. Additionally, Kelly Booth and Davor Cubranic were an integral part of the project which formed much of the basis of this thesis, and I thank them for their input and support. Additionally, funding for that project was provided by both Business Objects and the MITACS Accelerate program, so I am very much thankful for the financial support provided by them.

As well, one can not write any kind of acknowledgment without thanking family and friends. I am fortunate to have many friends to thank, but Chris Ware in particular stands out as a colleague who was a "sounding board" on numerous occasions throughout my graduate studies. In terms of family, I am truly grateful for the fact that both my Mother and my Father are still a part of my life, and have the opportunity to see me see this through (see Dad, I will finally finish school!). And of course special thanks go to my wife Susan, who provided many hours, days, months, and years of support, encouragement, and kicks to the behind throughout my time at UVic. Those who have been married while undertaking graduate studies will understand what I mean when I say that I never would have been able to complete my thesis without her and I am truly fortunate to have her in my life.

Last, but not least: *42*.

DEDICATION

To Angel, wherever you may be...

Chapter 1

Introduction

The language used is no more than a vehicle whereby the main objective is to be achieved. It is hard for students to make this separation. While they grapple with the idiosyncrasies of whatever language they are being taught, it is very difficult to think about higher level abstract concepts.

Tony Jenkins [5]

Learning computer programming is hard. So hard that oftentimes Computer Science students new to computer programming struggle very early on in their studies, become discouraged, and move on to other disciplines before reaching the true "science" of Computer Science [6]. Most programming environments used in academic institutions are based upon "real-world" software development tools, designed for software development experts who are already well-versed and trained in the fundamental ideas of computer programming. New students thus face an uphill battle: they both have to learn and understand the abstract principles, theories, and practices that instructors try to convey in early Computer Science courses, while at the same time needing to learn obscure syntactic details that are largely independent and irrelevant with respect to the learning objectives of the course.

Additionally, independent of the language, the tools and environments play a role as well. In a traditional compiler-based textual programming environment when students make a mistake in their program that mistake is not discovered until they finally try to compile their code. And when they try to compile their code they are oftentimes met with cryptic error messages which discourage them at a very early stage [5, 7, 8].

The implication of these difficulties are revealed in the declining enrollment num-

bers we have seen in recent years in Computer Science [9, 10, 11, 12]. As such, there is considerable effort being undertaken to try and alleviate some of the common hurdles that students encounter in early CS1/CS2 style courses so as to avoid the high attrition rates seen in these courses.

One approach has been to try and remove (or diminish) the impact from issues of syntax. In this vein, many visual programming environments (VPE's) such as Alice[13] and Scratch[14] have been produced as part of an effort to try to help alleviate some of the common syntactic errors that can cause difficulties for students new to programming. The hope is that this will thereby allow the key concepts instructors are trying to convey to be more readily apparent. However most of the previously designed VPE's (and all of the VPE's that are in widespread use) are based upon the imperative and/or object-orientated programming paradigms; very few are based upon the functional programming paradigm. This is peculiar, as the importance of the functional paradigm is increasing as time progresses, and it has been hypothesized that one of the major difficulties that students face in early CS1-style classes is learning the subtleties of the object-orientated paradigm [9, 15, 16]. Furthermore, there have been efforts such as the "Teach Scheme, Reach Java" initiative [17, 18, 19] that indicate that perhaps the jump from a functional programming background to a object-orientated one is less of an intellectual "leap" than the other way around [20].

Additionally it has been suggested that one possible way to combat the declining enrollment in Computer Science is to make Computer Science courses more "fun" and engaging [21]. One possible way for this to be done is to incorporate computer games into curricula, and this has been done in various circumstances as a way of encouraging student involvement, motivation, and enjoyment [22, 23, 24, 25, 26].

An open question remains however, would the combination of these two approaches (the use of a visual programming environment to reduce the discouraging effects of syntax errors, and the use of games to increase motivation and engagement) be the ideal way to introduce computer programming to students new to the field? Put another way, do games plus visual programming environments equal students who are better able to succeed as programmers in later courses?

To answer this question, one would need a visual programming environment that is feature complete enough to be adequate for use in a learning environment, and to develop exercises and assignments that make use of that environment. As a result of these observations and as an initial start towards answering the overall question of whether or not games plus VPE's produce students better equipped to progress

onward in Computer Science, this thesis has been written to explore one VPE based upon the functional paradigm, the Gem Cutter, from a pedagogical perspective with the purpose of identifying any potential shortcomings for using it in a learning environment. Additionally, a framework called the Word Game Framework was developed using the Gem Cutter for developing game-related assignments to aid educators who attempt to leverage student interest in games as a way to engage and motivate students in early programming classes. As a proof-of-concept of the Gem Cutter as a teaching tool and the usefulness the Word Game Framework, some exercises targeting some common learning objectives in introductory programming courses are also given.

Thus this thesis represents a significant first-step toward answering the question of whether or not the motivating power of games combined with the removal of syntax errors in a VPE can greatly enhance the learning experience for students learning to program by evaluating a particular environment, as well as the construction of a framework in the environment which equips educators with the ability to easily use games to meet learning objectives.

1.1 Thesis Overview

This thesis is divided into six chapters:

Chapter 1 Introduction motivates the problem of learning to program, and gives an overview of the thesis.

Chapter 2 Background Research explores and summarizes much of the related work to this thesis. In particular, past and current methods of instruction are summarized, as well the examination and evaluation of some newer approaches such as the incorporation of games into computer science courses as well as the use of visual programming environments. Lastly an outline of the overview of the Cognitive Dimensions Framework by Green is given, as well as a summary of how the framework has been applied to other programming environments.

Chapter 3 The Quark Framework gives an overview of the Quark Framework, and the two components that underly it – the CAL language, and the Gem Cutter visual programming environment. A general discussion of some of the strengths and weaknesses of the environment is also provided.

Chapter 4 The Word Game Framework describes this framework developed using the Gem Cutter for allowing the creation of game-related assignments. An

overview of the framework is provided, along with a discussion of the implementation of it. Some potential uses of the framework are explored, as well as some full, concrete, exercises that make use of the framework.

Chapter 5 Evaluation of the Gem Cutter Environment evaluates the Gem Cutter environment by use of the Cognitive Dimensions Framework developed by Green.

Chapter 6 Conclusion contains a summary of the thesis and proposes some related future work.

Chapter 2

Related Work

2.1 Practices of Teaching Introductory Programming

So, if I look into my foggy crystal ball at the future of computing science education, I overwhelmingly see the depressing picture of "Business as usual".

Edsger W. Dijkstra [27]

In this section we explore how computer programming has been, currently is, and will be taught at various post-secondary institutions. A thorough survey of the literature can be found in [28].

2.1.1 Curricula

We're not a vocational school. If someone wants to get a high-paying job, I would hope that there are easier ways to do it than working through a formal computer science curriculum.

Philip Greenspun [29]

Introductory programming courses should always be considered within the context of the curriculum in which they lie. Traditionally, the introductory programming course was one of the first (or was the first) course students undertaking a degree in Computer Science would enroll, as many future Computer Science courses assume basic programming experience as a basic skill.

In North America, many introductory computing courses correspond to the guidelines outlined in the Computing Curricula recommendations by the ACM and IEEE

Table 2.1: The 14 Areas Which Comprise the CC2001 Body Of Knowledge For Computer Science

Discrete Structures (DS)
Programming Fundamentals (PF)
Algorithms and Complexity (AL)
Architecture and Organization (AR)
Operating Systems (OS)
Net-Centric Computing (NC)
Programming Languages (PL)
Human-Computer Interaction (HC)
Graphics and Visual Computing (GV)
Intelligent Systems (IS)
Information Management (IM)
Social and Professional Issues (SP)
Software Engineering (SE)
Computational Science and Numerical Methods (CN)

Computer Society Joint Task Force. The latest full curriculum recommendation came in 2001 and is known as CC2001 [30]. In 2008 an interim revision from the task force was submitted [31]. CC2001 identified 14 areas which comprised the body of knowledge for computer science at the undergraduate level. These are listed in Table 2.1.

Each of these areas are subdivided into units, and each unit consists of a number of topics. Of the various units, some are given additional *weight* in the recommendation by being designated as *core* units, which are intended as being fundamental to **any** student of computer science irregardless of area of emphasis. All units which are not designated as core units are referred to as *elective* units. While the core units comprise a fundamental set of topics for computer scientists, by themselves the core units (and the topics within) do not comprise a *complete* set of material for a computer scientist – they need to be supplemented with other elective units from other areas depending on the area of emphasis, the needs and goals of the education institution, etc.

For many years, this task force proposed a two-course introductory programming sequence commonly referred to as the CS1/CS2 sequence. Most educational institutions in North America have followed this pattern, though the contents of the courses have evolved over time. Roughly speaking, traditionally CS1 tends to focus on introductory programming concepts (control flow and conditional statements, variables, iteration, etc), and CS2 tends to focus more on the introduction of data structures (trees, queues, etc). In CC2001, the importance of the object-orientated paradigm

was recognized, and it was suggested that object-orientated concepts become a central part of the introductory course sequence. Partly as a result of this addition, CC2001 also suggested a third course to follow CS2 (typically called CS3) to ensure "students are able to master these fundamental concepts before moving on"[30]. Adoption of the three course model has been mixed, with many institutions retaining the two course sequence.

2.1.2 Pedagogy

> *If I could have one wish for education, it would be the systematic ordering of our basic knowledge in such a way that what is known and true can be acted on, while what is superstition, fad, and myth can be recognized as such and used only when there is nothing else to support us in our frustration and despair.*
>
> Benjamin S. Bloom [32]

Whereas the curriculum outlines the topics that shall be introduced in a computer science curricula, an equally important (perhaps more important) aspect of the teaching of how to program is that of pedagogy. The difference between curricula and pedagogy is well described by Pears, et al, as "While the curriculum defines what is to be taught, pedagogy deals with the manner in which teaching and learning are managed in order to facilitate desired learning outcomes"[28]. More roughly speaking, the curriculum is the "what" and pedagogy is the "how".

In the relatively short history of the discipline, there have been a number of approaches to how to introduce students to the world of computer programming. One of the most hotly debated issues is that of the paradigm to use early on. Traditionally, the Computing Curricula recommendations by the ACM and IEEE Computer Society Joint Task Force suggested a "programming-first"[1] approach. This style emphasized a procedural programming style early on, and many institutions still follow this model today. However, it has been argued that this approach furthers the misconception that "computer science equals programming", as it puts such a heavy emphasis on programming perhaps at the expense of topics (particularly theoretical topics) more central to the discipline [30]. Additionally, it has also been argued that a programming-first approach unfairly favours students with prior programming backgrounds over students with little or no background. In addition, students who

[1] Sometimes this approach is referred to as "procedural-first", sometimes "imperative-first".

have learned some programming skills on their own before the introductory programming class tend to not have the opportunity to have poor habits corrected in the programming-first approach, as the emphasis of the approach is on simple syntactic details rather than design and algorithmic problem solving [30].

As a result of many of these criticisms and potential shortcomings, as well as the increased prevalence of the paradigm, CC2001 suggested the incorporation of object-orientated principles into their introductory programming sequence. This "objects-first" approach has been the source of much debate in academia since 2001 when the suggestion was made in the CC2001 guidelines, and in particular during a well documented e-mail discussion on the SIGCSE mailing list in 2004 [33, 34, 35, 36]. Some feel that the subtleties, complexities, and level of abstraction required of the object-orientated paradigm represent too great an intellectual leap for introductory programming students [35, 36]. Some believe that the CS1/CS2 curricula is already "too full" as it is, without the additional complexities of the OO paradigm, and that introducing OO into the CS1/CS2 sequence will cause traditional programming constructs to be pushed by the wayside [35]. Some argue that the OO paradigm is an extension of the procedural, and thus procedural programming needs to be fully understood before objects can be discussed [36]. Some of the reasons for introducing objects early include the belief that since the OO paradigm is the most dominant in the world of computing, and will continue to be so for the foreseeable future, many believe that as such it is the most important for students to learn, and thus should be the first that they learn [33]. Additionally, another one of the common difficulties associated with learning the OO paradigm is that of "paradigm shift", and as such it has been argued that if we begin with objects early, then we avoid the paradigm shift problem [35, 36].

In parallel to the programming-first to objects-first evolutionary approach to pedagogy, some have argued for the incorporation of functional programming into early programming courses, and propose a "functional-first" approach. Much of the roots of this approach come from the text "Structure and Interpretation of Computer Programs"[37], which was a highly influential text during the 1980's [38]. SICP, as it is commonly referred, made use of the functional programming language Scheme, and many of the exercises in the text discussed and relied on concepts from the functional paradigm (the lambda calculus, higher order functions, etc). A well-known criticism of SICP was written by Wadler in 1987 which criticized the use of Scheme as the language, but argued in favour of retaining the functional focus of the text

[39]. There has also been research done which would indicate that perhaps beginning in the object-orientated paradigm is a less desirable choice than beginning in the functional paradigm [20, 38]. This has significant repercussions in terms of language choice, as (for example) Java is a heavily object-orientated language, and as such it has been argued is not the ideal choice for a first year course [20, 40]. The TeachScheme/ReachJava project shows that "functional first" does not imply that object-orientation must be avoided in the first year of study [17, 18, 19]. Concerns with "functional first" include the notion that sometimes concepts that are more difficult to express in the functional paradigm are sometimes given diminished importance (I/O being the most prominent example), concerns about introducing recursion early, and the notion that functional programming is not "mainstream" (thus raising the "this is not really used out in the real world" criticism)[41].

In any case, all of these approaches (programming-first, objects-first, and functional-first) share a common characteristic – they are all based upon the notion that we should structure our approach to teaching introductory programming around a particular programming paradigm. Some however feel that this is inappropriate, that rather than focusing on a particular programming style, we should be focused on more general concepts and ideas irregardless of the paradigm. Most notably, there has been a move toward a "design-first" instructional style for the introductory course. Traditionally introductory programming courses are taught by example. The instructor introduces a new syntactic construct from the language being used, shows a number of examples using this construct, then exercises or assignments are given where students have to take the given code from the instructor and modify it to a new problem. Some people feel that this approach emphasizes the language more than design, and given that the specific language in use is not the primary goal of the course, it has been argued that this creates courses whereby students walk away feeling as though programming is all about learning syntax, and not about designing creative solutions to interesting problems. The consequence of this approach is that it makes the teaching of syntax explicit and the teaching of design implicit, potentially causing courses to create students who can take existing code in Java or C++ (whichever language is used) and modify it to a new problem, but cannot design a solution to a problem from scratch. Felleisen et al. outlines a course whereby design is made much more explicit to students, and has reported success in their approach [38]. What is interesting about this approach and others which claim to put general issues such as design as the focus, is that they are almost exclusively rooted in the functional paradigm.

2.1.3 Language Choice

The limits of my language mean the limits of my world.
Ludwig Wittgenstein [42]

The choice of programming language for the introductory course is a critical one, though perhaps an overstated one. Ideally the introductory programming course should be a course which teaches systematic thinking and problem solving, not "how to write Java". Thus, while by necessity the introductory programming courses must make use of a programming language and environment, the primary goal of the introductory course is to teach programming in the abstract sense rather than specific syntax.

In the history of the field, computer science has seen a variety of languages employed at one time or another as being "the teaching language of choice". Almost as varied as the number of languages employed is the number of reasons why one language has been chosen over another. Sometimes a language is chosen based upon faculty preference, sometimes because of industry prevalence. Some languages are designed with learning in mind and as such have a certain appeal to educators. Sometimes support materials in the form of programming environments or documentation will sway a department from using one language to using another. Sometimes the paradigm chosen plays a prominent role[2].

Today C, C++, and Java are the most commonly used languages in both industry and education [28]. Part of the appeal of these languages include the fact that they tend to be the ones most heavily used in industry. There is additionally a sense of a movement away from C and C++ to Java as time is progressing [16, 43]. Various reasons have been proposed for this movement, the most common seems to be that it is a reflection of the increasing adoption of Java in industry [16]. There has been considerable work done in collecting and producing resources to aid educators who are using Java in their first year courses, one of the most prominent being the ACM Java Task Force [44].

There has been considerable debate however as to whether or not Java is the most appropriate choice of language for introductory programming courses [43]. It has been shown that certain background characteristics are highly correlated with success in learning introductory programming courses for which Java is the language

[2]It would seem odd to try and teach object orientated programming in Haskell for example, as that language is based upon a different paradigm

Table 2.2: Summary of Problems in Teaching Java[1]

High Level Issues		
H1	Scale	(remains a concern)
H2	Instability	(remains a concern)
H3	Speed of Execution	(improving over time)
H4	Lack of good textbooks and environments	(improving over time)
Language Issues		
L1	Static methods, including `main`	(remains a concern)
L2	Exceptions	(remains a concern)
L3	Poor separation of interface and implementation	(partly addressed by tools)
L4	Wrapper classes	(added in Java 5.0)
L5	Lack of parameterized types	(added in Java 5.0)
L6	Lack of enumerated types	(added in Java 5.0)
L7	Inability to code preconditions	(added in JDK 1.4)
L8	Lack of an iterator syntax	(added in Java 5.0)
L9	Low-level concerns	(disposition varies)
API Issues		
A1	Lack of a simple input mechanism	(remains a concern)
A2	Conceptual difficulty of the graphics model	(remains a concern)
A3	GUI Components inappropriate for beginners	(remains a concern)
A4	Inadequate support for event-driven code	(remains a concern)

of choice. The most prominent of these characteristics is prior familiarity with the object-orientated paradigm [16], which is not entirely surprising given how the language is heavily based upon that paradigm. However it is troublesome from an educational point of view in that one of the reasons Java is used in introductory programming courses is that it has been argued that it is a suitable choice for an "objects-first" approach due to its object-orientated nature. Furthermore, it has been recognized that most of the programming tools which have been adopted are primarily aimed at the Java programming language. What is not clear is whether or not this is due to a relatively widespread acceptance of the Java language, or if it is a reflection of the need for additional learning support with that language [28].

Additionally there are language-specific concerns that have been raised with Java. The ACM Java Task Force identified a Taxonomy of Problems in Teaching Java, some of which have been somewhat addressed as the language has evolved, and some of which remain problems to this day [1]. The difficulties are summarized in Table 2.2.

As the result of the difficulties associated with the use of C/C++ and Java in the introductory programming course, many institutions are now switching from C/C++ and/or Java to other languages. The most common of these is the Python programming language [45]. Python has the advantage of being a scripted, interpreted language, thus students can "try-out" small expressions or snippets of code without having to get a complete source file free from errors. It also has a relatively "clean" and simple syntax compared to the relatively verbose Java. The use of whitespace for program structure encourages good indentation habits early as well. With all these benefits it is not surprising to see some institutions beginning to adopt Python as the language for their introductory programming courses. Perhaps the most notable adoption of Python happened at MIT, whose Computer Science department had long been a strong proponent of the use of Scheme in introductory programming courses due to the adoption of [37] as the introductory text [46]. Other examples of using Python in CS1/CS2 style courses and the possible benefits and drawbacks of doing so can be found in [47, 48, 49, 50]. While there have been successes with Python in the classroom, not even the strongest supporters of the language claim that it is the "perfect choice".

2.2 Visual Programming Environments

Like Moses, I get to see the promised land, but not set foot in it.
But the vision is clear.
Randy Pausch [51]

One of the major difficulties associated with learning to program is the issue of syntax errors, and as a result of this, there has been a great deal of work done attempting to minimize or reduce this hurdle [8]. One of the major categories of this work is the world of the visual programming environment (VPE)[3]. A VPE is a programming environment where the user interacts with graphical on-screen objects rather than the traditional textual method of writing text-based code. Another definition comes from Kelleher & Pausch:

> ...[VPE's] use graphical or physical objects to represent elements of a program such as commands, control structures, or variables. These objects

[3]Some authors use the term "visual programming language" rather than "visual programming environment"

> can be moved around and combined in different ways to form programs. Novice programmers need only to recognize the names of commands and the syntax of the statements is encoded in the shapes of the objects, preventing them from creating syntactically incorrect statements [8].

2.2.1 Criticisms of Visual Programming Environments

There are two common criticisms of VPE's that have somewhat slowed their adoption: the scaling-up problem, and concerns about transfer of training. The scaling-up problem refers to the programmer's ability to apply VPE's in larger programs. That is, questions have been raised about whether or not the visual representation used in VPE's ultimately are too unwieldy to produce programs of moderate complexity [52]. Transfer of training concerns whether or not skills and knowledge acquired in a VPE can be transferred to a traditional text-based environment in the future. Some support for the claim that a visual environment can provide positive "transfer of training" to a textual environment can be found in [7].

2.2.2 Examples of Visual Programming Environments

There have been a number of VPE's proposed, we shall focus on a few environments which have proven to be of great influence particularly in regards to the educational side of Computer Science. A more thorough discussion of other VPE's can be found in [53] and [8].

An early VPE system that has proven to be quite influential is Prograph, a visual object-orientated dataflow language. Objects in Prograph are modeled as on-screen hexagons, with one half of the hexagon representing the data attributes of the object, and the other half the methods. By double-clicking on a part of the object a user can explore further the various methods and attributes of the object. Methods are contained in "frames", which are represented as directed graphs. Nodes in the graph are icons representing various constructs (objects, methods, conditionals, etc) and are connected together. Flow of data proceeds from the top of the digraph, passing through the various instructions, down to the bottom of the digraph, and outward (if the method has output). One of the major influences of Prograph was this "box and wire" representation, as it was one of the earlier systems to make use of it, and many VPE's have also made use of this metaphor. As well, Prograph has become

something of a benchmark system in evaluating VPE's[4]. It would appear however, that active development on Prograph ceased in the early 1990's.

Another VPE that is somewhat similar to Prograph is LabVIEW[54], a visual environment used for designing electronic circuits. That is, it uses "the metaphor of an electric circuit as a programming model"[53]. It also has a dataflow-style, whereby components are connected together, and data "flows" through the network of components. It also, like Prograph, makes use of the "box-and-wire" representation. It is a proprietary product of the National Instruments corporation, and also has been used as a "model" language when evaluating VPE's.

Neither of the above two environments are based upon the functional paradigm however. One such environment that is heavily based upon the functional paradigm is the VFPE[5], developed by Kelso [53]. The VFPE shares a common computational model with existing functional languages (such as Haskell). One of the central points of Kelso's work is the argument that functional languages have a unique singular visual representation – the syntax tree. As such, the VFPE is in many ways just a visual representation of that tree. A screenshot of the VFPE can be seen in Figure 2.1. The main interface to the VFPE consists of a main expression window (seen in Figure 2.1a) which contains the program being constructed. Components are selected from the menus on the right half of this interface, and dropped into placeholders in the main expression. Functions can be defined, and the definition of sub functions are done in a separate window called a binding frame (as seen in Figure 2.1b). Note that the layout of components is done by the environment, it is not "freeform" as users cannot control where components are laid out in the main expression window.

2.3 Games and Computer Science Curricula

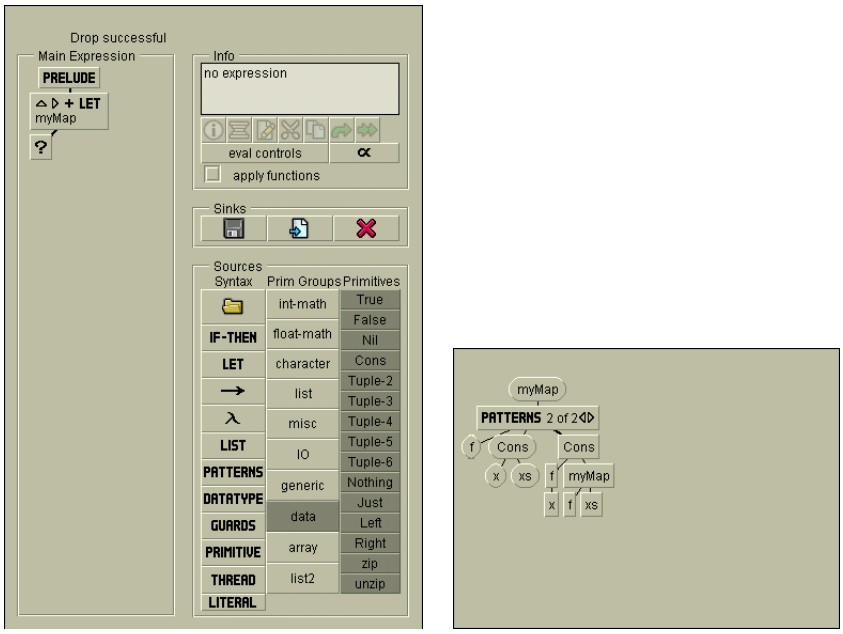
College students today have been called the "Nintendo generation" or the "MTV generation." Their perception of technology and media has been profoundly influenced by these sources. ...these are the kinds of media that Nintendo generation students want to produce when learning computer science.

Mark Guzdial and Elliot Soloway [55]

In recent years there has been a notable decline in enrollment in Computer Science

[4]Green uses Prograph and LabVIEW as the two model systems which he applies his Cognitive Dimensions Framework to

[5]Short for "Visual Functional Programming Environment"



(a) The Main VFPE Window (b) A Binding Frame in VFPE

Figure 2.1: The Visual Functional Programming Environment (VFPE)

courses [9, 10, 11, 12]. As a result of this decline, many institutions are seeking out interesting and new ways to "entice" students to enroll in Computer Science courses and to rekindle interest in the discipline. One such method that has been proposed is to incorporate computer games into Computer Science curricula. Cliburn et al. provides a summary of how games have been used in this fashion in [56].

The rationale behind this move is that since video games are an exciting and compelling application of computer programming, that perhaps that interest in games can be leveraged by instructors in their introductory programming courses and beyond [22, 26, 57]. Furthermore, video games are a multi-billion dollar industry [58], and a common motivation for students in choosing a discipline are the employment opportunities a degree in the field would entail. Since electronic entertainment is so prosperous, having curricula that targets students specifically for this field can be a compelling factor in one choosing a degree in Computer Science.

Some institutions have even taken this move a step further and are now offering gaming themes and concentrations in their degrees. Recently, the University of Victoria added a "Graphics and Gaming" option to their Computer Science major program. Other attempts at this are explored in [59, 60, 61]. Most institutions that incorporate games into their courses or degrees have reported an increase in student enjoyment. It has been shown that students prefer assignments based around games than "traditional" or "story-telling" assignments [62].

While there have been successes reported surrounding the use of games, it is very much a controversial choice, as there are a number of concerns that have been raised in regards to gaming-centered curricula. There is strong evidence to indicate that while gaming may spark initial interest in computing, it does not follow that this initial interest will translate into increases in students undertaking Computer Science majors. A survey of 1,872 students conducted at a "highly selective public technical university" found that while 43% of students indicated that games influenced their interest in computing, only 6.9% realized that interest by becoming Computer Scientists [63]. Furthermore, while student interest generally seems to increase with assignments making use of games, few institutions have reported any noticible improvement in student performance. Cliburn reported findings of a study he performed in the introductory programming course at Hanover College and found that students had a higher overall average score (95.1% on average) on traditional assignments than on game-based assignments (89.1% on average)[56]. Interestingly, he also found that in spite of this, most students (78.9%) still preferred game-based assignments over

"traditional" assignments, perhaps a further testament to the motivating power of games as assignments.

Concerns have also been raised surrounding issues of gender and race. One such concern suggests that games appeal more to male students than female, and that incorporation of games may alienate females from the discipline. Specifically, there has been evidence to show that females tend to prefer games which are cooperative in nature rather than competitive, and that the latter can deter interest of women in games [64]. This would indicate that educators must be careful about how they incorporate games into courses, and to design course materials with this concern in mind [65]. Other works have adopted "story-telling" rather than games as being the vehicle of motivation, and have specifically explored this avenue with middle-school girls with great success [66].

Chapter 3

The Quark Framework

The OpenQuark framework is a framework developed originally at the Business Objects Research Group. It embodies a non-strict (lazy), strongly-typed functional language and runtime for the Java platform. In this chapter we will outline some of the motivation and historical context behind the creation of the OpenQuark framework, as well as give an overview of the two main components of Quark (CAL, and the Gem Cutter), readers more interested in the details of Quark, CAL, and the Gem Cutter are referred to [67, 68].

3.1 History of OpenQuark

Development on the OpenQuark project began in the late 1990's with most development work occurring between 2000 and 2007 at the research group of Business Objects.

Business Objects as a company specialises in business intelligence (BI) software which commonly needs to manipulate data and metadata in well-defined ways. As a result, it was desirable to have a framework which would allow data behavior to be able to be "expressed in a formal manner as components that specific applications can load and compose together into the actual data processes they require"[69].

Thus a primary development goal of the OpenQuark framework was that it would allow data behavior to be declaratively expressed and composable. These assembled declarative models would then be validated statically and then could be executed by the underlying framework which would be able to intermix with Java-based client programs which would handle the responsibility for UI, communications protocols,

interfacing to databases, security, legacy applications, etc. Thus it would allow a sort of functional language based meta-programming.

The framework itself was developed to support a functional-based language called CAL, which featured strong syntactic similarities and features of the Haskell programming language such as modules, type definitions, type class definitions as well as various type-checking operations such as unification and pattern matching [67]. More detail about CAL is given in Section 3.2. The other primary component of the OpenQuark framework is the Gem Cutter visual programming environment. The Gem Cutter VPE itself is a Java Swing application written using the Quark Frameworks Java API, and was developed with the initial motivation being a proof-of-concept that this meta-programming was in fact possible. It was identified as having various other benefits not initially envisioned such as:

- being very effective for exploring and testing the functionality of CAL modules
- prototyping
- a "functional laboratory to explore new ideas with the considerable help of compiler feedback (e.g. type inference) and its interactive, assistive features."[67]
- a tool that can help people new to the functional paradigm to visualize and learn some of the characteristics of the paradigm

A more thorough examination of the Gem Cutter is given in Section 3.3.

The OpenQuark framework was released under a BSD-style open source license in 2006, and now consists of:

- the framework itself (i.e. - all the libraries and API's needed to embed lazy functional semantics into a Java program)
- the CAL language and an interactive interpreter for it (known as ICE), giving the ability to run standalone CAL programs
- the CAL Eclipse plug-in allowing one to write CAL code in Eclipse
- documentation
- the Gem Cutter VPE

3.2 CAL

CAL is a non-strict (lazy), strongly-typed functional language, which targets the Java runtime. That is, CAL source is compiled down to Java bytecode, and is it is thus possible to intermix components developed in CAL with components developed in Java. In this sense it is very similar to the Scala programming language [70]. It is even possible to call Java methods directly within CAL [71], however, obviously if one imports a functionally impure (ie - not referentially transparent) Java method, this compromises the purity of CAL as a whole (though in a very controlled way).

Syntactically, CAL is very similar (and heavily inspired by) the Haskell programming language. Like Haskell it makes full use of a Hindley-Milner type inference scheme, and implements essentially all the non-syntactic sugar features of Haskell 98 [67, 72]. An example of some CAL source can be seen in Program 1 which shows the implementation of the functional programming staple function commonly called `map()` which given a function and a list of items, applies that function to each item in the list, returning the transformed list as a result.

Program 1 The CAL Source Version of the map Gem

```
map :: (a -> b) -> [a] -> [b];
public map mapFunction !list =
    case list of
    []     -> [];
    listHead : listTail ->
        mapFunction listHead : map mapFunction listTail;
    ;
```

3.3 The Gem Cutter

Full many a gem of purest ray serene
The dark unfathom'd caves of ocean bear:
Full many a flower is born to blush unseen,
And waste its sweetness on the desert air.
Thomas Gray [73]

In this section we shall introduce the Gem Cutter, and its interface, focusing on concepts that are directly related to exploring the environment from a pedagogical

standpoint. A full detailed technical description of the Gem Cutter can be found in [67], the "official" manual for the environment can be found at [68], and a simple tutorial on using the Gem Cutter designed for students in a third year software engineering course at the University of Victoria can be found at [74].

The Gem Cutter is a visual interface to CAL code. The interface allows one to create CAL functions called "Gems", which are directly translated into CAL source code, which is then in turn compiled and executed[1]. Thus, the Gem Cutter provides an environment in which one can explore the capabilities of CAL *without writing a single textual line of code*. This direct relationship between the visual representation in Gem Cutter and the textual in CAL, and the teaching benefits that arise from this is one of the primary motivations behind this thesis.

The Gem Cutter starts up in a particular CAL workspace, where a workspace is a collection of CAL modules which form the initial environment for the Gem Cutter. This start up window can be found in Figure 3.1. The main interface of the Gem Cutter is made up of three components: the Scope Window which provides a tree-like overview of the gem currently being constructed, the Gem Browser which provides a list of all imported CAL modules for the current workspace, and the tabletop which is the area one composes (or "cuts") a new gem.

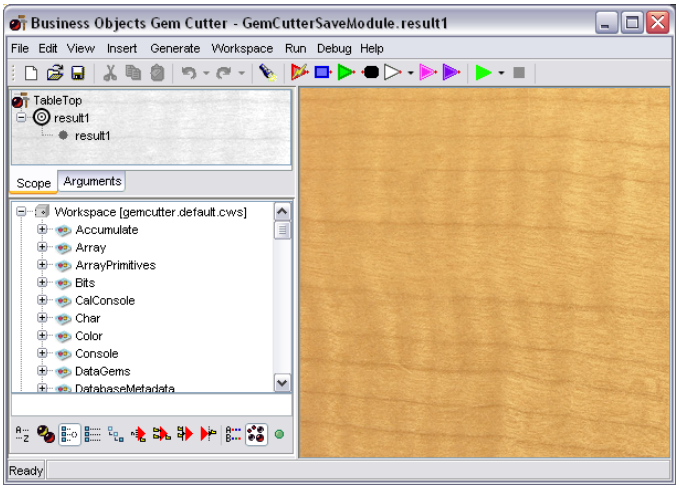


Figure 3.1: The Main Gem Cutter Interface

Interaction with the environment takes place by dragging and dropping various gems onto the tabletop, and connecting them together. Each gem has a single output

[1]It is worth mentioning that the "gems" in Gem Cutter have no direct relation to Ruby Gems, or the RubyGems project

(on the right side of it), and an arbitrary number of inputs (depending on the gem in question) located on the left side. Once a new gem has been constructed it can be saved, and then reused in new gem designs like any other predefined gem[2].

We can also re-open a previously designed gem by right-clicking on the gem in the Gem Browser, and selecting "Open Gem Design". Related to this feature, and an interesting consequence of the fact that Gems are just visual representations of CAL code, is that one can from within Gem Cutter view the corresponding equivalent CAL source code. To do this, one right-clicks on the gem in the Gem Browser and selects "Search For Definition", upon which the interface will find the generated CAL source for the gem in question. This is different from many other VPE's for which there is no direct equivalent textual form. Note however, that while one can view the CAL source for any gem, the converse is not necessarily true. Since many of the gems that are imported were written in CAL and not Gem Cutter, not all gems that are available in Gem Cutter have a gem design.

Like the VFPE described in Section 2.2, the Gem Cutter essentially is a visual representation of the syntax tree of the program under construction. Unlike the VFPE however, it uses a left to right rather than top-down orientation. As well, the Gem Cutter allows one complete control over how gems are laid out on the tabletop. This differs from the VFPE in which layout of components was done automatically by the environment.

3.3.1 Useful Applications of the Gem Cutter

As a state-of-the-art visual programming environment based upon the functional paradigm, the Gem Cutter has a great deal to offer as a pedagogical tool in classroom environments. In this section we outline some of the useful aspects of the environment.

Type Inference Permeates The Interface Type inference can be confusing to new and old programmers alike. It is difficult for new programmers because it is such an abstract concept. It is difficult for older programmers well-versed in traditional statically-typed languages (like C++ or Java) to "let go" of the tendency to want to explicitly state a particular data type for an identifier[3]. The visual programming

[2]Saved gems are by default stored in the GemCutterSaveModule module

[3]This is not without good reason, oftentimes programmers well versed in statically typed languages will think that a lack of programmer specified typing is equivalent to dynamic typing, and thus there is the fear of having type errors creep into their programs

environment offered by Gem Cutter addresses both of these concerns: it allows for new programmers to interact with gems and see how the types of identifiers are inferred. Furthermore, because type checking is enforced whenever connections between gems are made, it "feels" as though one is specifying types by making connections between gems, thereby alleviating concerns about the use of type inference.

Constrained Environment As with most visual programming environments, the Gem Cutter represents a relatively constrained environment compared to more traditional text-based programming environments. Syntax errors are extremely difficult to produce[4], as the structure of code is inherent and implied by the construction of gem graphs. There is no need for statement terminators, there is never any ambiguity from dangling else/if-else clauses, nor is there every any confusion about which operation has greater precedence than another. Furthermore, strict type-checking is enforced upon every connection, thus type errors are essentially removed. As a result, students can focus more on problem solving and structuring solutions rather than fighting with minor syntactic issues.

Promotes Key Ideas In Programming Oftentimes instructors in first year programming classes have difficulty conveying some key concepts such as composition, encapsulation, and modular thinking to students. Typically exercises in early programming classes tend to be small enough that the merits of modularity are not apparent, but rather seem like an artificial burden. In Gem Cutter, each gem is categorized into different modules, making the problem of finding a particular gem far easier than would be the case if there was just a "flat list" of gems. Thus the merits of modularity seem immediately apparent to students. As well, in creating gems one often needs to *compose* predefined existing gems together, and think of them in terms of opaque, "black-box" computational units (enforcing the idea of encapsulation).

Easy to Find Functions With Particular Type Signatures It is not uncommon for someone designing a learning exercise in the functional paradigm to reach a point where they require a function that has a particular type signature. The Gem Cutter provides two main mechanisms for finding functions of a particular type: *Intellicut* and *The Gem Browser*.

[4]Unless one starts making use of code gems

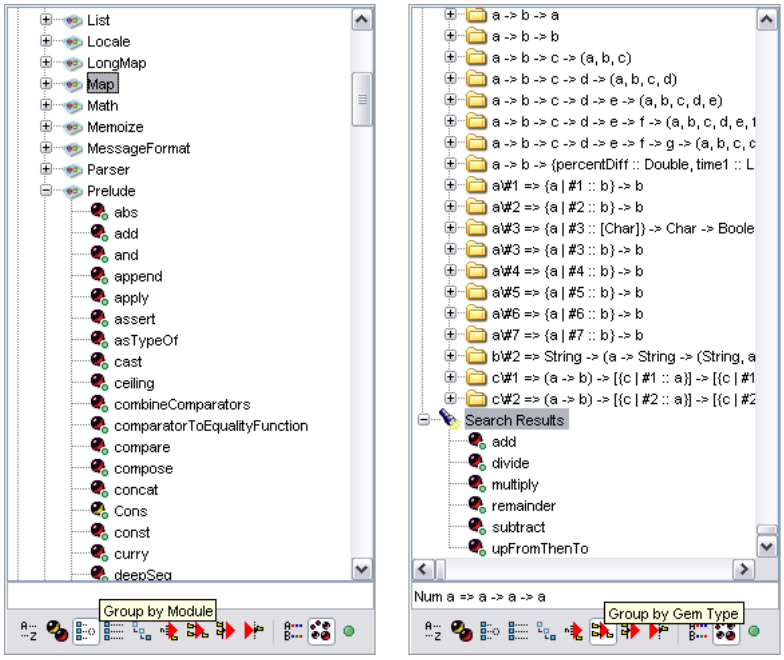
With Intellicut (seen in Figure 3.2), when one hovers the mouse over a particular connection on a gem a small pop up window will appear listing possible type-safe gems which can be linked to this connector. Furthermore, there are three views in this pop up, as one can select to see all valid gems, only the "best" gems, or a "likely" subset of gems. This feature can oftentimes allow one to "accidentally" discover that perfect function to be used with many higher order functions such as `map()`. From a learner's perspective, Intellicut provides a support mechanism which can help "guide" students to correct solutions without sacrificing learning objectives. Intellicut is in many ways similar to syntax-directed editing, or code completion, where the goal is to simplify the development process by alleviating some of the cognitive load that programming entails.

(a) Best 204 Gems for Map.find (b) All 1,113 Gems for Map.find

Figure 3.2: Using Intellicut to find gems

With the Gem Browser, one can choose a variety of different groupings of gems. The default is to group gems by Module, which is oftentimes the best, but it also allows one to group gems by type signature. Both of these are seen in Figure 3.3, where on the left the default "by module" grouping is shown, and on the right the "by gem type" arrangement is shown. There are additional browsing arrangements in addition to "by module" and "by type" including: by arity (number of arguments), by input type (essentially the same as "by type" but with return types removed), or by output type (essentially the same as "by type" but with input types removed). Furthermore, for each arrangement, one can do a context-sensitive search to find exactly what one is looking for. This can be seen in Figure 3.3b where the results of

a search for all gems which are of type `Num a => a -> a -> a` or "all gems which take two numeric types and return a numeric type" can be seen.



(a) Grouping by Module (b) Grouping by Gem Type

Figure 3.3: Various gem groupings in the Gem Browser

Supports and Encourages Incremental Development At any point while developing a gem, one can right-click on that gem and select "Run Gem" to test out the partially constructed gem. This is ideal for students, as at any point while they are working on a solution to a problem they can periodically test out parts of their design as needed to ensure that their gems work as intended. That is, one does not have to create the entire solution before one can test out sub-parts of the solution. Put another way, it encourages new programmers to "test out" small subsections of code, thereby making debugging easier and less time-consuming.

3.3.2 Shortcomings of The Gem Cutter as a Learning Tool

As mentioned, the Gem Cutter is a sophisticated tool that can be effectively used in learning environments. However, given that it was primarily designed as a tool for software developers it is not surprising that there are some aspects of the Gem Cutter

which could be improved to make it more suitable for use in an educational learning environment.

Single-User View The Gem Cutter generally has a single-user view of the world, which makes it difficult to effectively deploy in multi-user environments. For example, in a Unix/Linux environment one typically wants to install an application in one central location, and have user-specific data saved into a user's `/home/username/` directory. However, the current design of the Gem Cutter does not allow for this, the location of all user-generated data is stored within Gem Cutter's `/bin/` directory, thus one has to install the entire Quark framework separately for each individual user. Furthermore, this structure of how user data is stored makes it difficult to move Gem Designs from one machine to another essentially requiring users to make a copy of the GemCutterSaveModule.cal file within the `/bin/cal/debug/CAL/` directory, along with all files in the `/bin/cal/debug/designs/-Gem-Cutter-Save-Module/` directory. This is problematic for students who would like to bring work created in school labs to their home machines and vice-versa.

Inability to Create New Data Types There is currently no facility within the Gem Cutter to create new basic data types. For example, when using the Word Game Framework, one may want to create a `GameState` type which is made up of a series of primitive data types (much like a `struct` in C). However, as it stands now, the only way to do this is to create the data and type constructors for the new type in a separate CAL module, and modify the configuration files of the Gem Cutter to make use of this new module. This significantly impacts the ability of instructors to make use of Gem Cutter to illustrate key ideas in functional type theory[5] as if one restricts themselves to only using the Gem Cutter, they can only supply pre-defined data types for students, rather than have students construct their own.

Difficult to Specify Types The Gem Cutter (like CAL) makes use of a very sophisticated type inference algorithm to infer the types of identifiers as needed. This is useful, as it provides an excellent forum for illustrating the very interesting and useful programming language concept of type inference. However, as is the case with any system which makes use of a type inference algorithm, it can sometimes be desirable to manually specify a specific type for an identifier. This is certainly

[5]Such as type and data constructors, parameterized types, etc.

possible with CAL, for example if we want to say that an identifier "x" is a 2-tuple consisting of an Int and a list of Strings we would write `x :: (Int, [String])`. However, from within Gem Cutter it can be difficult to enforce a particular type on a particular identifier. Currently the easiest way to do this is to use a code gem and write an expression like the previously mentioned "x" declaration. However, it would be preferable (and more within the visual programming paradigm) to perhaps right-click on a gem's connection and use the value editor seen with value gems to construct an explicit type. This would be preferred from a teaching perspective in that much of the point of using a visual programming environment is to remove the possibility of syntax errors, whereas code gems are essentially windows into the world of text-based code writing, and thus allow for the possibility of syntax errors once again.[6]

Inability to Open Multiple Gem Designs Currently the Gem Cutter can only work with a single gem design at at time. This is somewhat awkward, as oftentimes (just like in the text-based paradigm) one will be working on a gem, and then want to modify or examine the implementation of another gem. However, to do this one has to save the first (incomplete) gem, open up the other gem, then re-open the original gem. As well, there are times when a not-yet-complete gem cannot be saved, so this requires arbitrary changes to the gem at hand to be able to save it before opening another gem design.

Dependencies Between Gems A rather significant shortcoming of the Gem Cutter is in regards to type dependencies between gems. For example, if we create a gem called `foo` with type signature `String -> Int`, we can then make use of `foo` within another gem called `bar`. If we then change the type signature of `foo` (say for example we decide we need Double precision numbers as the return type) we will in turn break the design of `bar`. The Gem Cutter will produce an error message similar to that seen in Figure 3.4. Furthermore, and more seriously, we will likely cause problems within the Gem Cutter environment as a whole in that afterwords the GemCutterSaveModule module (where users save their gems) will no longer be accessible until one manually edits the underlying generated CAL code in GemCutterSaveModule.cal to remove or repair the definition of `bar` to be compatible with the new type signature

[6]This is mitigated in code gems in that the code in a code gem is compiled immediately as one writes it, so at least syntax errors will be discovered very early on

of `foo`. This might be reasonable for more "seasoned" computer science students, but it is probable that first year students will feel as though they damaged the environment and will in the future be more timid in attempting gem construction for fear of making the same (seemingly serious) mistake.

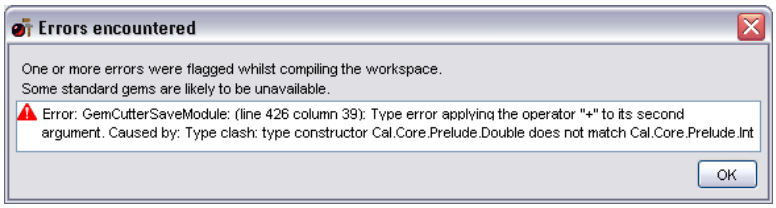


Figure 3.4: The error message displayed by Gem Cutter when a type dependency is broken

Furthermore, it is not only return types that cause this "breakage" to occur. For example, if we have the same `foo()` gem as before, and we decide we need a second argument to `foo()`, we cannot simply add a second argument as if we do then the type of `bar()` will change as well.

To give an idea of how serious this problem is, note that the type dependency relationship between gems is transitive. That is, if `a()` calls `b()` which calls `c()` which calls `d()`, then a change to `d()`'s type signature will break gems `a()`, `b()`, and `c()`. This means that refactoring existing gems is extremely difficult as it means you have to first modify each gem to remove any dependencies between them, and then recreate the dependencies afterwords. From an educational standpoint this is nearly a show-stopper, as students need to have the freedom to make design mistakes, and then go back and correct their mistakes, which is something the Gem Cutter currently does not easily allow, instead essentially requiring one to restart from scratch when a design mistake is found.

Difficult to Create New Modules The Gem Cutter makes use of a special module called `GemCutterSaveModule` and by default places all user-created gems into that module. One can change this, and use a different *already existing* module to save gems into, but there is no facility within Gem Cutter for creating a new module from scratch. This would be desirable from a pedagogical perspective as it would allow one to better teach the concept of modularity in software design. One can, however, create new modules by creating new CAL code modules in a text editor, and then add them to the current workspace.

Inability to Easily Remove Saved Gems Oftentimes when approaching a task for a first time a learner will make mistakes and want to start over from scratch. In textual programming environments, this is easy – just start with a new blank text file. However, there is no mechanism within Gem Cutter for removing previously saved gems. As it stands, the only way to remove saved gems is to manually edit the GemCutterSaveModule.cal file, removing the definition of the gem from that file, and then restarting the Gem Cutter. For seasoned computer scientists this is relatively trivial (once you know where to look), but for new students of the discipline who are not yet conditioned or experienced with editing configuration files by hand, this can be a daunting task. In either case it can be error prone, as if a removed gem is used in another gem's design, one can "corrupt" the workspace environment as seen in the "Dependencies Between Gems" section. It would be preferred to have a more interactive mechanism, where one could right-click on a gem, pick "Remove Gem", and then the Gem Cutter could search all gem definitions to see if removing the gem should be permitted (i.e. whether or not the gem to remove is used elsewhere). Given that facilities for determining if gems are referenced inside other gems already exist within the Gem Cutter, it would appear that this should be a relatively small task to implement this additional functionality.

Using Non-Standard Modules A default, "vanilla" installation of the Quark framework will provide a Gem Cutter environment with most of the standard CAL modules available for use. However, many of the modules which are available as part of the CAL language are not imported into the default Gem Cutter environment, instead appearing as "greyed-out" listings in the Gem Browser. If one wants to use these modules within Gem Cutter they to either *a*) create a new workspace and select all the modules they want within that workspace[7] or *b*) edit the GemCutterSaveModule.cal file by hand to import the module.

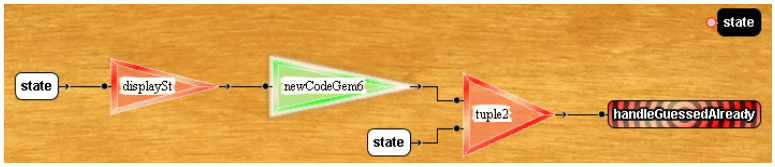
Unfriendly Error Messages Particularly in regards to error messages, the Gem Cutter can produce alert windows that are not particularly friendly to users. As an example, the author of this document once tried to modify the implementation of a previously created gem. The original gem had the CAL source seen in Program 2, and the gem layout seen in Figure 3.5a:

It was modified to remove the code gem which appended two strings together to

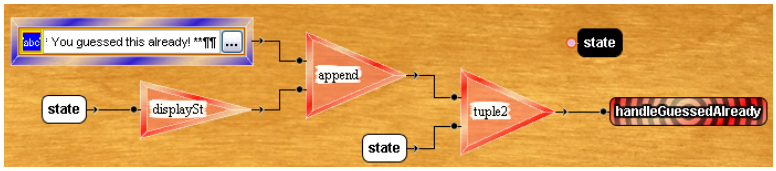
[7]This can be done via the "Workspace" menu within Gem Cutter

Program 2 The CAL Source Version of the handleGuessedAlready Gem

```
public handleGuessedAlready state =
    Cal.Core.Prelude.tuple2
        (
            (\s -> "\n\n ** You guessed this already! **\n\n" ++ s)
                (GemCutterSaveModule.displaySt state)
        )
        state
    ;
```



(a) The Initial Implementation of the handleGuessedAlready Gem



(b) The Second Implementation of the handleGuessedAlready Gem

Figure 3.5: Two Versions of the handleGuessedAlready gem

```
An error condition was flagged whilst compiling the test.
The captured error text was:
Error: GemCutterSaveModule: The declared type of the function
cdlInternal_runTarget is not compatible with its inferred type
(a\#1, a\#2, a\#3) => {a | #1 :: Cal.Core.Prelude.String, #2 ::
Cal.Core.Prelude.Int, #3 :: [Cal.Core.Prelude.Char])} ->
(Cal.Core.Prelude.String, {a | #1 :: Cal.Core.Prelude.String,
#2 :: Cal.Core.Prelude.Int, #3 :: [Cal.Core.Prelude.Char]}).
Caused by: record type unification failed at field #2. Attempt
to instantiate a record variable from the declared type.
```

Figure 3.6: A Lengthy Type Error Message Produced by the Gem Cutter

instead explicitly use the `append()` gem in the Prelude module. The new layout can be seen in Figure 3.5b. However, after doing this and then running the new gem layout the Gem Cutter once produced the error message seen in Figure 3.6. This was problematic from the perspective of new programmers for a few reasons:

- It uses technical and cryptic language (what is meant by "instantiate a record variable from the declared type"?)
- The record type notation, along with the explicit type specifiers make it difficult to pull out key information
- There are "magic" names in the message (for example, cdlInternal_runTarget is the name given to the compiled gem when one tries to run a partial gem layout)
- The entire error message appeared in a single line which meant that the dialog window that contained the error message was 2,419 pixels wide, which would not fit on even a very high resolution screen.
- Most importantly, there should not have been an error, as visually, one was simply connecting the output of two gems into the `tuple2()` gem, which should accept two items of *any* type. After placing a code gem which explicitly specified a type for the "state" parameter[8] the error message ceased to appear (even after the code gem was removed).

No Visual Facility for Lambda Expressions As it stands the only way to create an unnamed function or lambda expression is to use a code gem.[9] While not a problem from a practical point of view, it is difficult to convey to learners the theoretical background of functional programming[10] without discussing lambda expressions.

Poor Documentation Being a relatively newly open-sourced project[11], there is not yet a great deal of user-friendly documentation available. Currently the only real documentation on the Gem Cutter consists of [67] and [68]. The former is a

[8] It was a 3-tuple of type `(String, Int, [Char])`

[9] In fact, a code gem really is a lambda expression, as while they can be named the name is purely for documentation purposes

[10] The Lambda Calculus

[11] The OpenQuark framework was released under a BSD-style license on January 25, 2007 [75]

conference-style technical paper, and the latter is more of an overview of features with little detail on how to resolve problems within the environment. There is however, a community dedicated to providing assistance via the CAL Language Discussion Google Group[12]. This document is the first work we have found which addresses the Gem Cutter from an educational usage standpoint.

Confusion Between Records and Tuples In CAL, tuples are treated as records with the first field named `#1`, the 2nd `#2`, etc. This seems natural, however, it can lead to some confusion in terms of type specifiers. For example, a record consisting of an Int in field `#1`, and a String in `#2` would be of type:

```
(a\#1, a\#2) => {a |#1 :: Int, #2 :: String}
```

However, this looks quite different than `(Int, String)`, even though the two types are equivalent. This can be a source of confusion for students not well-versed in various compound type schemes. An example of this is seen in Figure 3.7 where we have a `tuple2()` gem and a `field1()` gem. The output of `tuple2()` is completely type equivalent to the input of `field1()`, even though the type identifiers look quite different (particularly to the untrained eye).

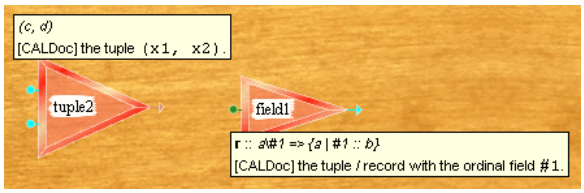


Figure 3.7: The tuple2 and field1 gems

[12]Found at: `http://groups.google.com/group/cal_language`

Chapter 4

The Word Game Framework

There are at least two kinds of games. One could be called finite, the other infinite. A finite game is played for the purpose of winning, an infinite game for the purpose of continuing the play.

James P. Carse [76]

In this section we outline the implementation of a "word game" game engine designed using the Gem Cutter. This engine provides a means for instructors to design a variety of exercises of varying difficulties and to be able to tailor the exercises to their specific needs. From the perspective of students, the game engine represents an interesting and engaging framework within which they can explore various functional programming concepts such as higher order functions, recursion, and list processing. A number of sample exercises using the framework are given in Appendix A.

The inspiration for the framework came from [24] where a similar framework written in Haskell was outlined. However, ours is an extension to this work in that *a*) it exists in both a visual programming environment (The Gem Cutter) as well as a text-based form (in CAL), and *b*) it is more general and flexible allowing for a greater variety of games to be expressed, without sacrificing pedagogical value.

In Section 4.1 we define what we mean by a word game, in Section 4.2 we outline the main API to the framework, in Section B.1 we outline some other implementation details, in Section 4.3 we outline an sample game implementation (the "Hangman" word game) using the framework, in Section 4.4 we outline some possible other approaches to implementing games using the framework, and finally in Section 4.5 we outline some of the pedagogical applications of the framework.

4.1 Word Games

For our purposes a word game is a turn-based, text-based game in which the player will make "moves" that will transition the game from one current state to a (same or different) new state. A game's state may include such details as the number of "lives" a player has remaining, or what previously made moves a player has made. For a more concrete example, a given state of the popular word game Hangman may include details such as the word being guessed, how many guesses have been made, and what those guesses were. For the popular Sudoku puzzle game, the state may include details such as the layout of the board (ie - which squares are blank and which are filled in), as well as the solution to the puzzle. At the very least, a given game state should provide enough information that one can deduce all possible currently valid moves and whether or not the game is over (that is, whether or not the player has won or lost).

The name "word game" is perhaps misleading, as any turn-based game for which a string-based representation can be produced is possible. Other examples of possibilities would include games like chess, checkers, othello/reversi, etc. So long as the current "state" of the game can be modeled, and the game itself is played by transitioning from state to state (or turn to turn), the word game framework can be used.

Of course, while general, there are certainly limitations to where the framework can be used. As mentioned, everything in the game is text-based, thus it is not particularly well-suited to a graphics programming course. As well, all interactions in the game take the form of textual commands the player types, there is no facilities for other user interfaces to be used. And lastly, any games that make use of the framework must be turn-based, thus "real-time" games would not be well-suited to the framework.

In any case, all the possible examples given follow the same set of mechanics: prompt the user to make a move, analyze that move, and change the state based upon it. This is the general underlying computational pattern throughout all games that the Word Game Framework implements. The way in which this is interesting from a functional programming perspective is that the framework is parameterized by a few *functions* which are passed as arguments to the framework. That is, it inherently makes use of, and provides an interesting exercise in, the development and use of higher order functions.

4.2 The Word Game Framework Interface - The Game Gem

For our implementation, we have designed a main *game*() function which game designers interact with. By supplying all the parameters to *game*(), game designers create a complete interactive game which can be played via the Gem Cutter interface.

4.2.1 Parameters to the Game Gem

Our implementation of the *game*() function takes six arguments: a state changing function, an initial state, a prompt string, a title string, a predicate for testing if the end of game has occurred, and a routine for converting a state into a string-based representation. Or more formally (using CAL type notation):

```
game::
    stateChangingFunction ::  a -> String -> (String, a)
    initialState ::  a
    userPrompt ::  String
    title ::  String
    gameOverTest ::  (String, a) -> Boolean
    stateToString ::  a -> String
    returns (String, a)
```

Note that the type variable "a" represents our "state" type. Thus, states are completely arbitrary in our implementation, they can be as simple or as complex as one wishes or needs. No part of the Word Game Framework makes any kind of assumptions about the structure of game states. The return value of *game*() is a tuple consisting of the last message displayed to the player, and the final game state.

Each parameter of the *game*() function is outlined in detail below:

- `stateChangingFunction ::  State -> String -> (String, State)` - a function which encapsulates the state changing rules of the specific game. One can think of this function as being the underlying logic of the game as it is responsible for enforcing the rules of the game at hand. Given a current state and a string which represents the input from the player, this function produces a

string response (to be displayed to the player) and a new state (which may or may not be different from the previous state).

- `initialState :: State` - the initial state the game starts in.
- `userPrompt :: String` - the string to use as a prompt to the player (for example, "Enter your choice:")
- `title :: String` - the string to use as the title of the game. This is displayed to the player once when the game begins.
- `gameOverTest :: (String, State) -> Boolean` - a predicate function which given the last displayed message to the player and a current game state, returns true if the game is now over (for example if the player has won).
- `stateToString :: State -> String` - a function which given a game state, returns a string representation of that state to be displayed to the player.

A screenshot of the implementation of *game*() is shown in Figure 4.1.

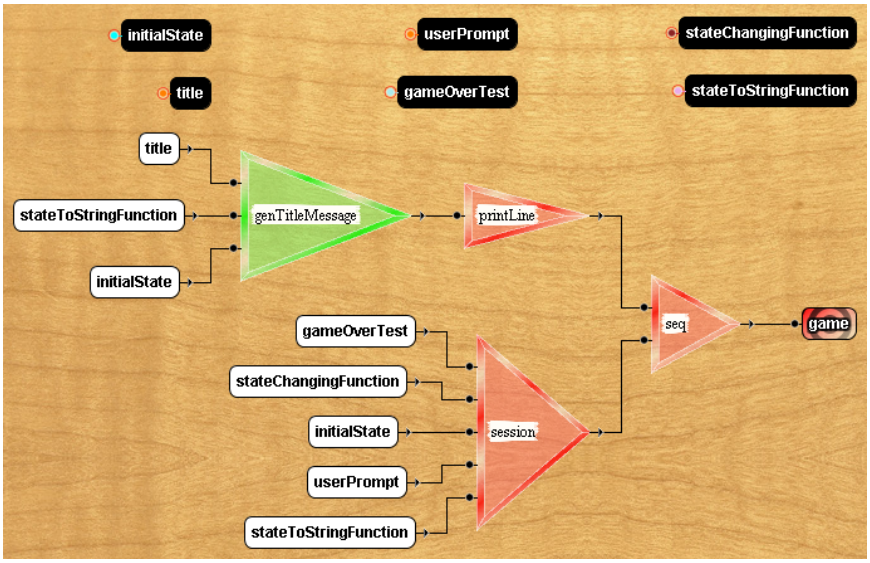


Figure 4.1: Implementation of the game() gem

4.3 Example Game - Hangman

As a concrete example of the use of the Word Game Framework, we now will outline an implementation of the popular Hangman game which uses the Word Game Framework as the game engine.

4.3.1 Hangman

Hangman is a game where players are given a hidden word, and a set number of guesses. On each turn of the game, a player can make a single guess as to a letter they think appears in the word. If they are correct, all occurrences of that letter in the word are revealed to the player. If they are incorrect, then the number of remaining guesses is decreased by 1 (there is no penalty for guessing a previously-guessed letter). The game ends when the player has successfully guessed all letters in the word (in which case they win), or when they have no more remaining guesses (in which case they lose).

4.3.2 Implementation

For our implementation of the game, each state of the game consists of a 3-tuple of the word to be guessed, the number of guesses remaining, and a list of previously guessed letters. Or in CAL type notation a game state is of type `(String, Int, [Char])`. A screenshot showing the implementation of the main `hangman()` routine is shown in Figure 4.2.

Our state changing function was named `makeAGuess`, following the pattern outlined in [24].

Our initial state consists of a word selected at random from a supplied text file, the user-supplied number of guesses to allow before the game is over, and the empty list (since no guesses had been previously made). Or in CAL: `initState = (getRandomWord wordFile, numLives, [])`

We used two string literals in the form of value gems for the prompt and title.

Our game over test was defined locally and is seen in Figure 4.2. We define the end of game to occur when the number of guesses remaining falls below 1 (our state changing function `makeAGuess` sets the number of lives to be 0 when the player successfully guesses the word).

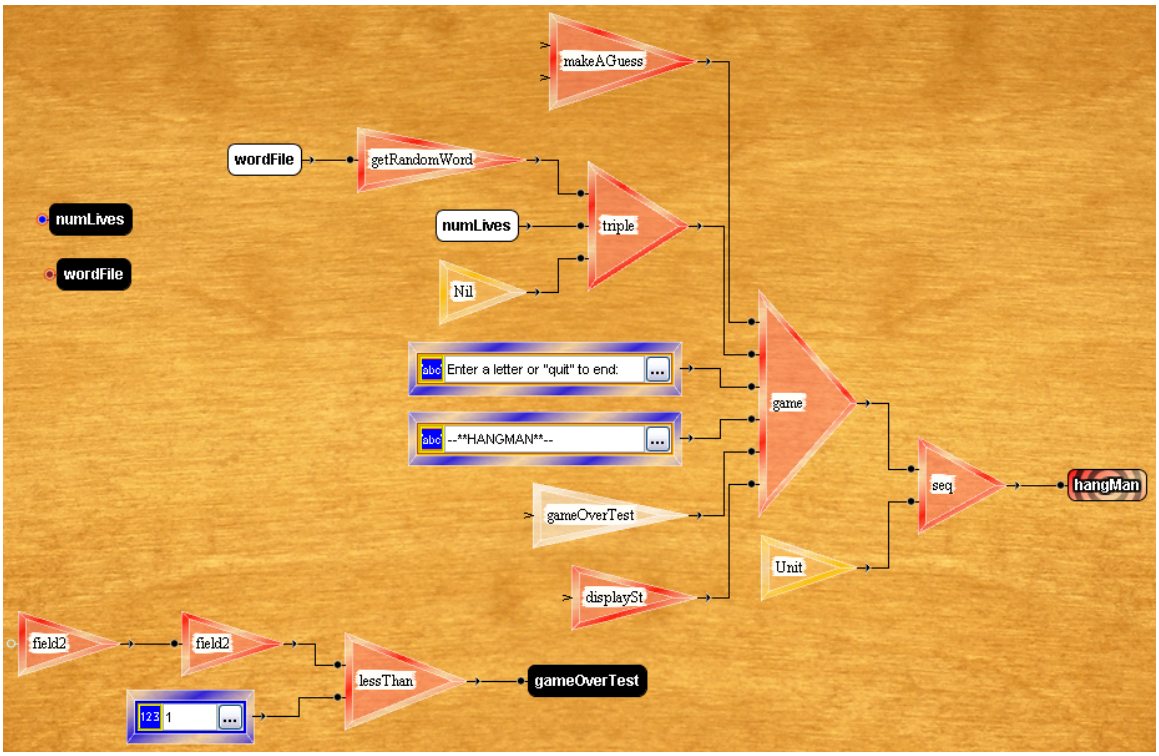


Figure 4.2: Implementation of the hangman() gem

Our state to string function was named `displaySt`, again following the pattern outlined in [24]. Given a state, it produces a string which contains the target word with unguessed letters blanked out (the implementation of which is an excellent exercise in using `map()` and `filter()`), the number of remaining guesses, and the letters not yet guessed.

4.3.3 Hypothetical Session

Below we illustrate a simple gameplay session where the chosen word was "file" and the player had 10 guesses to find the word.

```
--**HANGMAN**--
("quit" to quit)


     - - - -
```

```
Characters not guessed yet: abcdefghijklmnopqrstuvwxyz
Lives left: 10

Enter a letter or "quit" to end:
e

     _ _ _ e

Characters not guessed yet: abcd fghijklmnopqrstuvwxyz
Lives left: 10

Enter a letter or "quit" to end:
k


 ** Character not in word **


     _ _ _ e

Characters not guessed yet: abcd fghij lmnopqrstuvwxyz
Lives left: 9

Enter a letter or "quit" to end:
m


 ** Character not in word **


     _ _ _ e

Characters not guessed yet: abcd fghij l nopqrstuvwxyz
Lives left: 8
```

```
Enter a letter or "quit" to end:
s


 ** Character not in word **


     _ _ _ e

Characters not guessed yet: abcd fghij l nopqr tuvwxyz
Lives left: 7

Enter a letter or "quit" to end:
l

     _ _ l e

Characters not guessed yet: abcd fghij   nopqr tuvwxyz
Lives left: 7

Enter a letter or "quit" to end:
o


 ** Character not in word **


     _ _ l e

Characters not guessed yet: abcd fghij   n pqr tuvwxyz
Lives left: 6

Enter a letter or "quit" to end:
i
```

```
     _ i l e

Characters not guessed yet: abcd fgh j   n pqr tuvwxyz
Lives left: 6

Enter a letter or "quit" to end:
f

     f i l e

Characters not guessed yet: abcd  gh j   n pqr tuvwxyz
Lives left: 6
 ** Well done, you got it! **

 Good Bye!
```

4.4 Other Potential Games

As mentioned the framework is general, and could be applied to a number of games. In this section, we will sketch an outline of how some games could possibly be implemented using the framework.

4.4.1 Sudoku

The Sudoku puzzle game has become increasingly popular in recent years [77]. The "typical" sudoku puzzle consists of a 9x9 grid in which one has to place the digits 1 through 9 in such a way that each row, column, and 3x3 subgrid contain each digit exactly once.

From the perspective of our Word Game Framework, there are a number of ways a Sudoku grid could be modeled. Probably the simplest approach would be as a list of lists of cells, with each inner list representing a row of the Sudoku grid. The drawback to this approach however is that looking up a particular column or sub-grid would not be particularly intuitive.

Another possibility (since the size of the puzzle does not change while the game is being played) would be to make use of CAL's Array data type, model the 9x9 grid

as a 81 element array, and provide a function to convert a (row, col) index into the grid into a single-digit index into the array. This would make accessing the grid in arbitrary ways more natural, as it would simply involve a `filter()` over the array based upon the row or column index supplied. The biggest upside to this approach would be that finding a particular element of the list would be extremely fast ($O(1)$ time). However, converting a (row, col) into an index for non-symmetric sized puzzles would be less straightforward (although certainly doable).

Perhaps the most robust way would be to use an associative array, mapping (row, col) pairs as keys to cell values. Finding the set of cells which are "free" or unassigned would be a simple filter over the map. CAL provides the `Map` module as an API to associative arrays which could be used for this purpose. This approach would be extremely general, allowing for puzzles of any size and shape. A slight downside is that accessing elements of the Map would be slightly slower than the Array version. The Map.find function runs in $O(log\, n)$ time where n is the number of elements in the map, thus generating a given row would mean $O(m \cdot log\, n)$ time, where m is the length of the row/column/subgrid (i.e. the number of digits allowed in the puzzle).

In any case, each of these approaches would provide an excellent starting point for discussions about data structures, and the trade-offs involved in using one particular data structure versus another.

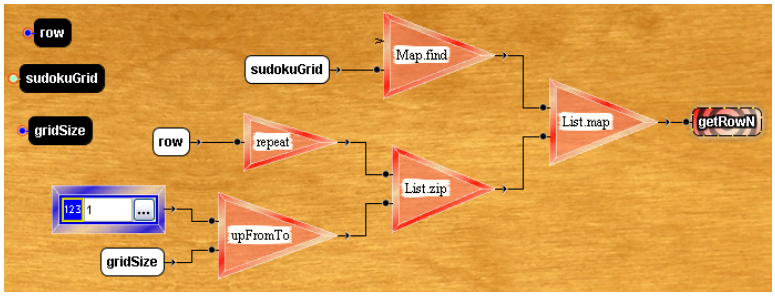


Figure 4.3: Implementation of a getRow gem for a Sudoku game

In terms of the data type of each cell, one could simply use Ints, with a special designated value to indicate that the cell is currently empty (perhaps 0). Alternatively, one could be more sophisticated and model the pattern outlined in [78] and construct a new data type for each element which is of type `DataCell a = Known a | Unknown a | Impossible`. That is, a cell can be a known (fixed) value, an unknown cell (parameterized with the value 0 to indicate empty, or non-zero to indicate

a guessed value), or Impossible to indicate that no value can legally be stored at that location (which would indicate that a guess to an unknown cell is incorrect). To verify a puzzle would require one to implement routines for accessing a single row, column, or sub-grid of the board, each of which would be excellent exercises in the use of common functional programming constructs such as `fold()`, `map()`, and/or `filter()`. Figure 4.3 shows an implementation of a function for getting the row of a Sudoku puzzle modeled using the Map data structure. Note that this implementation makes use of `map()` and `zip()`. Additionally, the use of the `repeat()` gem takes advantage of the lazy evaluation semantics of the CAL language (and provides an opportunity for educators to discuss eager vs lazy evaluation in programming languages).

Thus, each state in our Sudoko game would simply consist of the data structure which models the board. Our initial state would be a given puzzle with all fixed values being of "Known" type, and each blank cell initialized to "Unknown 0".

Our state changing function would take a given board layout and string indicating a (row, col) index into the board and a value to place, and then try to assign that value to the board. Again, there is a great deal of flexibility here, one could create this routine for students but have the state-changing function allow assignments which are invalid (i.e. ones which lead to cells of type "Impossible"), then give as an exercise the task of modifying the state changing function so that only valid assignments are allowed.

Our string to state function would take a given board layout and produce a string-representation of the board. One possible output would be:

```
-------------------------------
| 5  3     |    7  8 | 9  1  2 |
| 6  7  2  | 1  9  5 | 3  4  8 |
| 1  9  8  | 3  4    |    6  7 |
-------------------------------
| 8  5     |    6  1 |       3 |
| 4  2     | 8     3 |       1 |
| 7  1     |    2    |       6 |
-------------------------------
|    6     |         | 2  8    |
|          | 4  1  9 |       5 |
|          |    8    |    7  9 |
```

```
------------------------------
```

Producing each row of the above layout would be a natural application of the `map()` function, and if using the `DataCell` data type, would also be a very nice exercise in learning about parameterized types.

4.5 Possible Pedagogical Uses

The Word Game Framework as outlined is quite general, and there are a number of potential ways it could be used. Some possibilities include:

- One could provide all of the required parameters to *game*(), but leave out a few smaller parts of the state transition function as exercises. Depending on how the state transition function was structured each small portion could be targeted to a particular learning objective (perhaps learning about list processing, or higher order functions). This allows one to target extremely narrow and precise learning objectives.

- One could give the framework as is, give the students a specification of what structure the game states will be and what the rules of the game are, and leave the implementation of the state changing function, the end of game test, and the state to string parameters as the task to complete, thus giving a wide range of freedom for students to apply previously-learned concepts in a more general setting.

- One could supply everything except the end of game test, and ask students to implement that function. Since the end of game test is a function which is passed to *game*(), this is a relatively simple task which naturally illustrates the use of higher order functions.

- One could supply some of the auxiliary routines needed for the state-changing logic, leaving it up to students to connect the pieces together in the correct way. This encourages students to think of the supplied parts as being opaque, enforcing the concepts of encapsulation and function composition.

In all cases, because of the strict type checking semantics of the Quark framework, students will be able to be particularly confident that when their project works, that

it is correct in the general case.[1]

[1]This however does not remove the need for appropriate testing, but merely means that testing will be less time-consuming than in less strongly typed languages (such as C)

Chapter 5

Evaluation of The Gem Cutter Environment

...a scientist must also be absolutely like a child. If he sees a thing, he must say that he sees it, whether it was what he thought he was going to see or not. See first, think later, then test. But always see first. Otherwise you will only see what you were expecting.

Douglas Adams [79]

5.1 The Cognitive Dimensions Framework

Every notation highlights some kinds of information at the expense of obscuring other kinds.

Thomas R. G. Green and Marian Petre [3]

One of the challenges faced in evaluating a programming environment is that programming is such a *subjective* task. Different programmers have different preferences, and value certain features over others. While one programmer may find one aspect of the environment very useful, another may find that aspect counterintuitive. Programmers are famous for being extremely opinionated about the tools and environments they use, developing almost dogmatic devotion to one approach at the expense of others. It can be difficult to find a formal method to use to discuss the merits of one language design over another that can be agreed upon.

When computer scientists and software engineers try to evaluate the usability of a system, they turn to human computer interaction (HCI) principles to do so.

However, as pointed out in [3], this is problematic for the evaluation of programming environments as typically HCI tends to focus on interactive situations rather than notational design. That is, HCI tends to focus more on "micro-tasks" and the finished product, rather than the processes and activities which produce that task. As an example, in evaluating a programming environment it is less important to know that the "compile" option is easy to find than it is to know that the environment does not allow for new abstractions to be created. Moreover, programmers are not HCI experts, and vice-versa. Thus while the vocabulary of HCI may be second nature to those in the field, it is less so for the average computer programmer or software engineer. Similarly, it is not uncommon for programmers to use terminology that is unfamiliar to those who are not software developers. For our purposes, we shall use the "cognitive dimensions framework" outlined by Green in [3] for evaluating differing visual programming environments.

In this section we will outline the cognitive dimensions framework, giving an overview of the various dimensions of the framework, as well as recap some of the ways in which it has been applied to other environments.

5.1.1 Outline of the Cognitive Dimensions Framework

The Cognitive Dimensions Framework consists of 13 different "dimensions" to easily evaluate and critically examine a programming environment. The purpose of the framework is to provide a set of dimensions which capture much of the psychology and HCI of programming, and give a vocabulary which VPL designers and users can use to examine different visual programming environments. It is less of a framework for stating that "language X is better than language Y", but more for statements like "language X can have better Y by changing Z" (where Y is one dimension of the framework, and Z is some aspect of the VPE that impacts Y). Furthermore, the language used in the framework is more accessible to the average software developer, not requiring one to be an HCI expert to use it.

One very important aspect of the framework is that (not surprisingly) different dimensions can represent trade-offs. In fact, this is intentional — there are always trade-offs in designing any system of moderate complexity, and until there is a vocabulary for discussing those trade-offs it is difficult to reason about how one can improve the initial design. The cognitive dimensions framework is intended as being able to fill this void, allowing designers to coherently and critically examine and *converse* about

their designs so that they may find the best compromise given the objective(s) of the system being designed. As Green himself puts it "Like other forms of engineering, design is a matter of compromise" [3].

Below is a recap of the listing of the thirteen dimensions of the Cognitive Dimensions Framework described by Green in [3].

- Abstraction Gradient - What are the minimum and maximum levels of abstraction?
- Closeness of Mapping - What programming "games" need to be learned?
- Consistency - When some of the language has been learned, how much of the remaining parts of the language can be inferred?
- Diffuseness - How many symbols or graphic entities are required to express a meaning?
- Error-proneness - Does the design of the notation induce "careless mistakes"?
- Hard Mental Operations - Are there places where the user needs to resort to fingers or penciled annotation to keep track of what's happening?
- Hidden Dependencies - Is every dependency overtly indicated in both directions? Is the indication perceptual or only symbolic?
- Premature Commitment - Do programmers have to make decisions before they have the information they need?
- Progressive Evaluation - Can a partially-complete program be executed to obtain feedback on "How am I doing?"
- Role-expressiveness - Can the reader see how each component of a program relates to the whole?
- Secondary notation - Can programmers use layout, colour, or other cues to convey extra meaning, above and beyond the 'official' semantics of the language?
- Viscosity - How much effort is required to perform a single change?
- Visibility - Is every part of the code simultaneously visible (assuming a large enough display), or is it possible to juxtapose any two parts side-by-side at will?

A more detailed description of each is given in the sections that follow.

Abstraction Gradient

Green describes the Abstraction Gradient as "What are the minimum and maximum levels of abstraction? Can fragments be encapsulated?" Furthermore, Green classifies languages based upon three categories: abstraction-hating, abstraction-tolerant, or abstraction-hungry depending upon the languages minimum starting level of abstraction and their readiness or desire to accept further abstraction.

That is, this dimension asks the questions: how much needs to be constructed and learned in order to begin making the programming environment perform a task? And how easy is it to add new abstractions?

The example Green gives of an abstraction-hating formalism is that of flowcharts, as they only allow decision boxes and action boxes. There is no way to "add" additional constructs or group related ideas within a flowchart. While they have a low minimum level of abstraction (there is not much to learn to begin using them), they have no readiness to accept further abstractions. Turing machines would be another example of a notation which is abstraction hating — each instruction can only have a current state, current symbol, next state, next symbol, and a direction to move on the tape.

The classic example of an abstraction-tolerant language would be C, as it has a higher minimum level of abstraction (one must learn some of the various keywords in the language, as well as how to write a `main()` routine for example), and it allows for new abstractions of various kinds to be created (for example, one can create new types with `typedef` and `struct`).

The example Green gives of an abstraction-hungry language is Smalltalk, as it has a high starting level of abstraction, and requires one to modify the class hierarchy in order to create new programs. That is, every program written in the language **requires** the introduction of new abstractions.

From the perspective of learning to program, it has been shown that oftentimes students new to computer programming struggle with languages which force one to learn and master several abstractions at once, giving them "a 'rubber hamburger' which has to be swallowed because it cannot be chewed"[3]. As such it is difficult to find the right balance here, requiring too many abstractions to be learned can negatively impact learning, but having limited abstraction mechanisms can make programs difficult to modify (see the section on Viscosity later). Green seems to indicate that abstraction-tolerant languages are the best fit for learning environments,

but admits that more research needs to be done in this area.

Closeness of Mapping

One could argue that programming is all about problem solving. Put another way, given a problem expressed in one particular domain (the problem domain) the role of the programmer is to come up with a mapping of this problem into the domain of the programming environment in use (the program domain). Closeness of mapping tries to measure the gap between the problem domain and the program domain. That is, how much extra "administrative stuff" does the environment impose upon the programmer in order to translate a solution to a problem into something that can be executed in the programming environment? Green refers to this "administrative stuff" as "programming games", or the little language/environment-specific idiosyncrasies that are imposed upon the programmer. The classic example of a programming game which is imposed on many first year computer science students is the `public static void main (String [] args)` method signature that must appear in any Java program. This line is (typically) completely absent from the problem domain, yet is a required artifact that students must produce in order to produce a solution to any problem given to them by their instructors when Java is the language of choice[1].

Closeness of Mapping also is to a lesser degree a measure of what the environment provides in terms of standard libraries and constructs. If a language provides a great deal of standard libraries, then intuitively it would seem that there would be less work for the programmer to do to create the mapping from problem domain to program domain. Related to this is the notion of whether or not the language allows constructs to be built that improve the Closeness of Mapping. That is, if functionality is not provided in the way of standard libraries, is it possible for one to add their own to better bridge the gap between problem and program domain.

Traditionally (and not surprisingly), domain-specific languages tend to score very well in this regard *when the problem lies within the target domain that the language was designed for*. It is less clear as to how well these languages score when attempting general problems that fall outside of that targeted domain.

[1] Assuming a development environment which consists of a plain text editor and the command-line based JDK from Sun

Consistency

Consistency tries to measure how easy it is to infer the remaining parts of the language once one has learned part of the language. Put another way, consistency assesses whether or not "similar semantics are expressed in similar syntactic forms" [80]. Green states consistency as a form of "guessability": "when a person knows some of the language structure, how much of the rest can be successfully guessed?"[3] An example that has been given for a way in which a language suffers from consistency problems is Pascal and its handling of reals and integers versus boolean variables. In Pascal, one can read and write to reals and integers, but boolean variables may only be read from even though syntactically booleans are essentially used and treated in the same way as other variables.

Another more modern example would be Java and its String data type. Strictly speaking, `String`s are object types in Java, but can be treated in much the same way as primitive data types (such as `int`s or `boolean`s). For example, one can use the concatenation operator to combine two strings together in infix notation as seen in Program 3. This is the only time in which object types may be used in this "operator overloaded" way, all other object types must resort to method calls (as seen in Program 4). "Special cases" such as this can greatly hurt the consistency of a language and make them harder to learn.

Program 3 Concatenating Two Java Strings Using an Operator in Infix Notation

```
String s1 = "Hello";
String s2 = " World";
String s3 = s1 + s2;  // create "Hello World"
```

Program 4 Concatenating Two Java Strings Using Method Calls

```
String s1 = "Hello";
String s2 = " World";
String s3 = s1.concat (s2); // create "Hello World"
```

Typically visual languages tend to score well in consistency, largely due to a simpler syntax. Many textual-based languages have struggled in regards to consistency as most "real-world" textual languages have grammars of substantial size, and generally speaking the larger the grammar, the more complex the syntax (and thus the more likely there will be inconsistencies that arise).

Additionally, as noted by Kelso an issue related to consistency is that of library regularity [53]. For example, Haskell generally keeps naming and argument ordering in libraries in regular order (thus scoring well in terms of consistency). C libraries however tend to not score so well, largely due to the fact that they have been developed over a long period of time by a large number of authors[2].

It has also been noted that consistency can be viewed from two different perspectives: the learner and the designer [81]. Ideally these two perspectives should be the same, but in practice they often are not. The distinction in Java between object types and primitive types is an example of this. From the designers perspective this is perfectly consistent – there are two fundamental categories of data types, and within the Java Virtual Machine (JVM) they are handled completely differently and separately. From a learner's perspective however, this distinction can create confusion. From a learner (or user of the language) it is all just data so the distinction is not as apparent and, for many new to the language, seems rather arbitrary.

At any rate, given that the Cognitive Dimensions framework was designed by Green to be used by designers of a language one might think that his intention was that consistency was to be applied from the designers perspective, however this is not explicitly stated. From our perspective as evaluators of a VPE from a pedagogical standpoint we are however more interested in the learners perspective, and our application of this dimension in evaluating the Gem Cutter will reflect this.

Diffuseness

Diffuseness is also sometimes referred to as terseness. That is, it tries to measure how many symbols (or graphic entities in the case of a VPE) are required to express a meaning. This is, of course, a trade off – if a language is too terse it can be hard to read, but if a language is too verbose it becomes difficult to "keep it all in your head" at once.

Traditionally functional languages have been quite terse, and scored well in that regard[3]. Visual languages also tend to require a small number of syntactic elements to express a meaning, so one would think that a visual functional programming environment would be the most terse of all environments, and Kelso found this to be

[2]As well, it could be argued that the fact that all functions in C reside in the same namespace negatively impacts consistency as well as it means that library designers must come up with their own unique naming schemes for functions to avoid naming conflicts.

[3]Some even criticize functional languages for being *too* terse, further illustrating that this dimension represents a trade-off.

true of his VFPE [53].

Measuring diffuseness/terseness is done by counting the number of syntactic lexemes that comprise a program. To give a meaningful comparison, Green uses a sample yardstick problem taken from [82] that he calls the "rocket trajectory problem" which he summarizes as:

> The rocket program computes the vertical and horizontal trajectory of a rocket on which the only forces acting are its thrust and gravity. At time zero the rocket stands stationary and vertical on level ground, with a mass of 10^4 pounds. Its engine develops a thrust of $4 \cdot 10^5$ foot-pounds, using up a mass of 50 pounds of fuel per second, until the fuel is exhausted after 100 seconds. It rises vertically for 10 seconds after which it adopts and retains an angle of 0.3941 radians (22.5 degrees) to the vertical. The downwards acceleration of gravity is $32 feet/sec^2$ [3].

He then goes on to give solutions to the problem in each of the environments he looked at, and Kelso did the same for Haskell and his VFPE. The unfortunate flaw in this approach is that it does not account for individual skill or familiarity with a language – the more familiar one is with a programming environment, the more likely it will be that one can use the "right" constructs and thus produce a solution which is the most terse. Conversely, if one is unfamiliar with an environment[4] one will likely try to use a subset of the full set of constructs available, and end up producing a solution which is overly "cluttered".

As well, as noted by Kelso in [53] it can be difficult to measure diffuseness in visual environments which have dynamic layout schemes. Oftentimes visual environments allow one to "collapse" code blocks into single elements. Or as Kelso puts it: "The level of diffuseness can in effect be dynamically traded off against the level of detail".

Error-Proneness

The fundamental question that the error-proneness dimension tries to answer is "does the design of the notation induce 'careless mistakes'?" Green draws a distinction between 'mistakes' and 'slips', where the former refers to the parts of program design which are deeply difficult irregardless of the notation[5], and the latter refers to those

[4] And particularly if the environment suffers from poor consistency

[5] For example, design issues such as how to decompose a larger problem into smaller ones would be an example of a "deeply difficult" design problem.

errors that one did not mean to do, where you knew better, but still somehow made a simple little "slip-up". Error-proneness tries to measure whether or not the notation itself oftentimes leads to or encourages these "slips".

An example from the world of textual languages is the use of textual identifiers, particularly in languages which are case-sensitive. It is very easy to accidentally misspell an identifier in languages which do not require identifiers to be declared before they are used, which can lead to subtle and difficult to debug errors in the program. The paired-delimiter system (braces in C/C++/Java, parenthesis in languages such as Lisp and Scheme, or the begin/end pair in languages like Pascal) can also lead to trivial errors which cause the code to fail to compile. Note that all of these are *syntax* errors which are all but avoided in visual programming languages. Thus, not surprisingly visual environments tend to avoid some of these common "slips", however, it would not be fair to say that visual languages are inherently less error-prone than textual languages.

Kelso in [53] also notes that issues related to data types can play a role here. That is, languages which have strong type checks that are enforced throughout a program will likely be less error-prone than languages which do not enforce such strong and rigorous type checking.

Hard Mental Operations

A programmer must already juggle a number of different ideas and concepts while implementing the solution to a task. What data structure should I use? How efficient is this algorithm? Can I access this piece of data within this context? Ideally we would like our programming environments to reduce the cognitive load imposed upon the programmer. That is, a programmer's job is hard enough, without having the tools in use add to the amount of things he or she must juggle. This is where the hard mental operations dimension comes in. Green defines a hard mental operation as having two properties:

1. it must lie at the notational level, rather than the semantic level, as the dimension is trying to measure shortcomings in the programming environment, rather than discover meanings which are inherently difficult to express irregardless of the notation

2. combining multiple hard mental operations vastly increases the difficulty

As such, this allows him to outline a "broad-brush" test for hard mental operations by asking the following two questions:

1. if I fit two or three of these constructs together, does it get incomprehensible?
2. is there a way in some other language of making it comprehensible? (Or is it just a really hard idea to grasp?)

If the answer to both is yes, then we have a hard mental operation. As an example application of this, consider the following declaration written in the C programming language:

```
int *(*(*b)())[10])();
```

This is taken from [83], and it is a declaration which declares a pointer to a function called b which returns a pointer to an array of 10 elements, each of which are pointers to functions which return pointers to ints.

If we examine "functions as parameters in C" as a construct, we would ask the question "if I fit two or three function pointers together, does it get incomprehensible?" It would appear that the answer to this is yes, a pointer to a function is difficult to mentally parse to begin with, but if we have pointers to functions of pointers to functions, they are even more difficult to decompose. Next we ask the question "is there a way in some other language of making function pointers comprehensible?" Given that higher order functions are essentially functions as parameters, most any functional programming language provide support for this mechanism in a much simpler and easier to understand form. Thus, it would appear that "functions as parameters" is an example of a hard mental operation in the C programming language.

Hard mental operations in particular is an important dimension for environments designed for learning, as presumably students learning a language already have to mentally process the new language or notation that they are learning, in addition to any fundamental ideas or concepts instructors are trying to convey. Put another way: hard mental operations increase the cognitive load upon students, and this can have a negative impact upon learning.

Hidden Dependencies

A hidden dependency occurs when there is a relationship between two components such that one is dependent upon the other, and for which this dependency is not fully visible or apparent. An example from the textual domain that is of particular interest to programmers working within the functional paradigm is that of the side effect. If we have a subroutine `a()` which alters a global variable, this represents a hidden dependency between the variable and the function. One of the hallmarks of the functional paradigm is the notion of referential transparency, whereby an expression can be replaced with its value without altering the meaning or semantics of the program as a whole, a definition which prohibits these sort of side effects.

Another common example includes the much maligned GOTO statement found in various programming languages. This is due to the absence of a "come-from" which means that the effect of changes to code making heavily use of GOTOs can be difficult to predict.

A much more systemic example of a hidden dependency is the construct that virtually all programming languages make use of – the named subroutine. If `a()` calls `b()` which calls `c()`, a change to `c()` has implications for both `a()` and `b()`, though that relationship will likely not be apparent unless one examines the code of those routines. Call graphs are commonly used to mitigate this concern.

Premature Commitment

The fundamental question to be asked in regards to this dimension is "do programmers have to make decisions before they have all the information they need?" Green explains premature commitment is akin to "writing the contents list - with page numbers - before you write the book"[3]. Premature commitment occurs when the following three criteria are met:

- the notation contains many internal dependencies
- the medium/environment constrains the order of doing things
- the order is inappropriate

For example, consider the C function defined in Program 5. In isolation, a source file containing only this code would not compile as there is an internal dependency between the `foo()` subroutine, and another routine called `bar()`. The typical C

programming environment constrains one from writing subroutines which depend on other subroutines until a definition of those subroutines are given[6]. Whether or not this constraint on the order of subroutine creation is appropriate or not shall be left for the reader to decide.

Program 5 A Hypothetical C subroutine

```
int foo ()
{
    return bar() * 42;
}
```

Furthermore, Green provides a number of categories of premature commitment, outlined as:

- Commitment to layout
- Commitment to connections
- Commitment to order of creation
- Commitment to choice of construct

Progressive Evaluation

This dimension tries to evaluate to what degree the environment in question allows one to evaluate a partially completed solution. That is, environments which are strong from the perspective of progressive evaluation will allow evaluating partially complete programs with greater frequency.

As such, traditional text-based, compiled languages tend to score poorly in this dimension – a Java program cannot be executed until it is compiled, and it cannot be compiled until it is a complete program. Interpreted environments can do better, as they oftentimes allow one to enter individual expressions and evaluate them at any time[7]. There are still limitations in that the expressions must be "self-contained",

[6]The common "work-around" for this is the stub routine, where the function prototype and an empty body (or a simple return statement for routines which have return types) is provided by the programmer, and "filled-in" later

[7]This is, in fact one of the arguments in favour of using Python as an introductory programming language, as the standard Python interpreter allows one to enter expressions and get immediate results

they cannot contain references to not yet identified values or functions for example. However, this is more an issue of premature commitment than one of progressive evaluation.

This dimension is of particular importance to students new to programming, as it is recognized that novices need to evaluate their progress frequently to assess whether or not they are appropriately understanding the new concept or idea.

Role-Expressiveness

Role-expressiveness concerns the environments support to answer a user's query of "what is this bit for?". That is, how easy is it for a user to see how various component parts of the program relate to the program as a whole. Green identifies four specific items which relate to role-expressiveness:

- meaningful identifiers
- well-structured modularity
- use of secondary notation (see below) to signal functionally-related groupings
- "beacons" which signify highly diagnostic code structures

For example, LabVIEW does not support identifiers of any kind, thereby increasing the difficulty of comprehending a given LabVIEW program. Different languages have different levels of support for modularity as well. For example, Java allows one to break up a program into classes, each of which contains methods. This modularity improves the ability for one to understand (particularly larger) programs. Code "beacons" are defined by Wiedenbeck as "key features [in a program] which indicate the presence of a particular structure or operation" and are identified as "idiomatic or stereotypical elements in program code" [84]. As an example, in looking at a sorting routine one typically expects to find some code which is responsible for swapping two elements that are incorrect order. The "swap" beacon helps programmers to understand the sorting program as a whole. It has been shown that recognition of beacons in code is strongly correlated to programmer experience. A study done by Wiedenbeck found that while 79% of experienced programmers recalled significantly more of the beacon lines in a program, only 14% of novices could do the same [84]. Related to role-expressiveness, some languages encourage certain beacons to appear and some do not. Using the same sorting example, the code in Program 6 shows the

Quicksort routine written in the Haskell programming language. Note that there is no "swap" beacon in this code.

Program 6 Quicksort in the Haskell Programming Language[2]

```
qsort []     = []
qsort (x:xs) = qsort (filter (< x) xs) ++ [x] ++
               qsort (filter (>= x) xs)
```

Green also notes that role-expressiveness is one of the most difficult dimensions to clarify, largely due to a lack of "studies on comparative comprehension of equivalent programs expressed in different notations" [3].

Secondary Notation

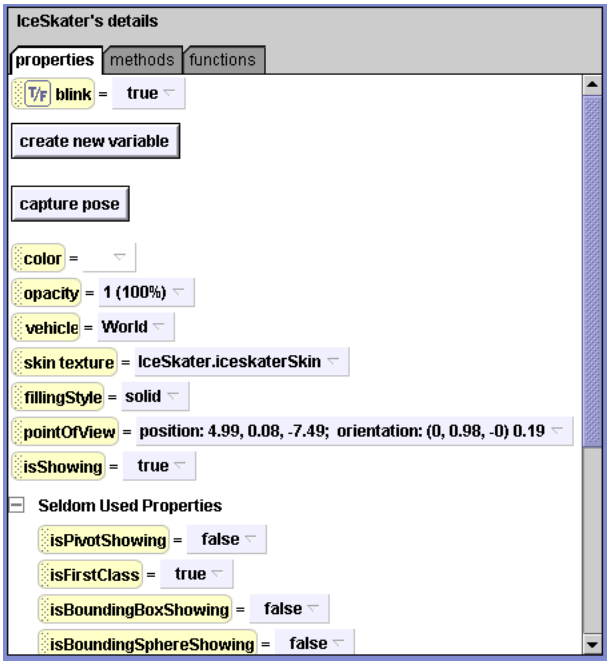
The dimension of secondary notation concerns how much extra information beyond the official syntax of a notation can be conveyed. The classic example of secondary notation from the world of textual programming is the comment. Comments have no bearing on the final program[8], and are present solely for the purpose of providing extra information to other programmers examining the code. Other examples include indentation, naming conventions, and grouping of related statements.

In the visual world, the analogues include things such as layout of graphical entities, the use of colour and/or shapes, or other cues to convey extra meaning. For example, perhaps one construct may uniformly always be coloured a particular way. For example, the Alice VPE makes extensive use of colour to convey extra information to the user. In Figure 5.1a we see that properties of an object[9] are listed in rectangular boxes that are coloured yellow, whereas in Figure 5.1b we see that methods of the same object are coloured a beige colour, and finally in Figure 5.1c we see functions are coloured a violet colour.

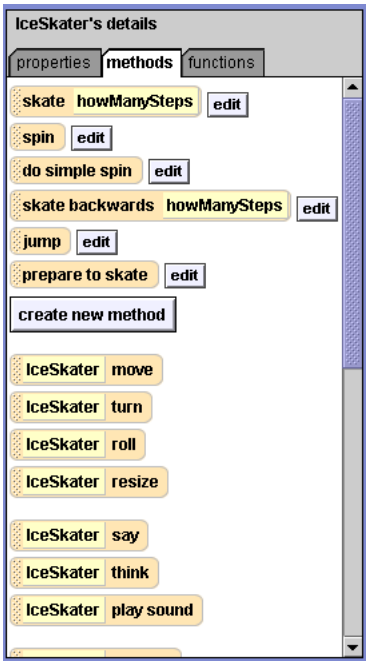
Secondary notation can be related to visibility (see below), in that a secondary notation may be present, but if it is not readily apparent then its usefulness is somewhat diminished. As well viscosity (see below) can impact secondary notation, as if a system is very viscous then changing the layout may be artificially difficult causing an aversion to changing the layout of a program after it has been initially constructed. As well, if the structure is changed, then the secondary notation can go out of date

[8] And are often removed completely during the compilation process

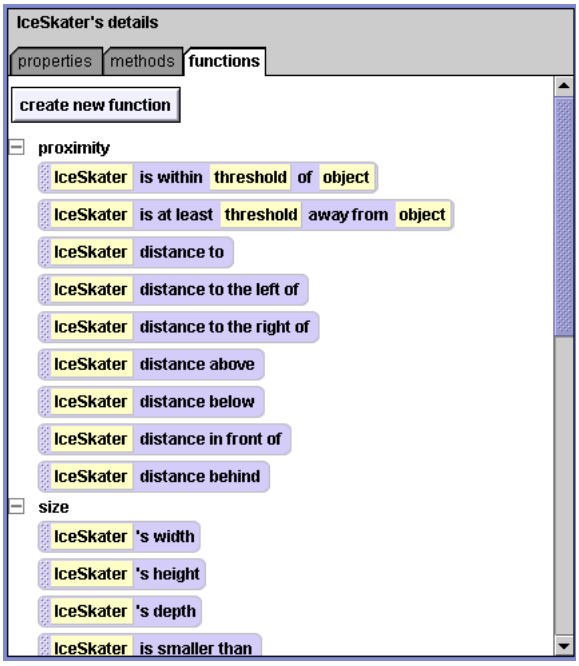
[9] The object in question is the "IceSkater" object from the default Alice tutorial



(a) Properties of an Object in Alice



(b) Methods of an Object in Alice



(c) Functions of an Object in Alice

Figure 5.1: Components of the Alice Interface

and become a detriment to aiding understanding of a program. Green also acknowledges that perhaps novices may benefit from a more constrained system in which secondary notation is minimized, but admits that this is pure conjecture and that more research needs to be done to confirm or deny this hypothesis [3].

Viscosity

Viscosity measures resistance to change. A system which is highly viscous is one where seemingly small changes require a relatively large amount of effort to undertake. This can of course be dependent upon the specific change in question, but certainly some environments make changes more difficult than others. Green identifies two particular types of viscosity:

- Repetition viscosity, where one change implies that the same change will have to be made elsewhere a number of times
- Knock-on viscosity, where one change implies many other changes to the whole to restore consistency

An example of repetition viscosity would be changing the name of an identifier in a textual language as each use of the identifier will have to be changed to the new name. An example of knock-on viscosity is changing the return type of a function, as this will require one to change all code which makes use of the return value of the function to make the program syntactically correct again.

Given how many software systems evolve over time viscosity is a rather important dimension. Additionally from the perspective of students learning to program, oftentimes students will make mistakes early on and correct them after the fact. If the system is highly viscous this can be problematic, as the effort to correct mistakes can become a deterring factor in a students motivation to produce a high quality solution rather than one that is "good enough".

Kelso notes that oftentimes VPE's suffer from viscosity problems due to the overhead of managing layout. There is an overhead associated with laying out and rearranging components in a way that conveys extra meaning (aiding secondary notation), and without making the layout seem "cluttered". He argues that this is an inherent problem in any VPE which requires manual layout and formation of links between

components [53]. Additionally, an editor which enforces program correctness[10] imposes an impediment on changing or rearranging components.

Viscosity can be related to premature commitment, as if one has to make a design decision too early it is probable that that choice will need to be revised. If the system is highly viscous then those changes will be difficult to incorporate, whereas if it has low viscosity then the changes should be easy to implement. That is, ideally a system which suffers from premature commitment should have low viscosity otherwise the problems related to premature commitment are amplified.

Given that the amount of time and effort to incorporate a change is dependent not only on the notation but also the specific change in question, viscosity has oftentimes been measured by implementing a small change to the rocket trajectory program[11] to take account of air resistance, by exerting a drag proportional to the square of the velocity, and measuring the amount of time it takes to implement this change. Because the goal of this test is to measure editing time and not the time taken to solve the problem, the test is conducted in two stages. First a solution to the new problem is created, a screen capture of the new solution is taken, and then the original rocket program along with the screen capture of the extended version is provided and the time measured is that of how long it takes to turn the original into the new modified version.

Visibility

Visibility concerns "whether required material is accessible without cognitive work; whether it is or can readily be made visible, whether it can readily be accessed in order to make it visible, or whether it can readily be identified in order to be accessed" [3]. Related to visibility is juxtaposability, the ability to view two or more components side-by-side. For example, can a user display two subroutine designs simultaneously so as to be able to compare and contrast them?

Visibility is in some senses similar to hidden dependencies, but while hidden dependencies measure whether relationships are manifest whereas visibility is concerned with the number of steps needed to make a given item visible.

In general VPE's tend to suffer from visibility problem insomuch as there is only a limited amount of screen space available to display constructs, and typically a graphical notation tends to take up more visual space than a textual one. While

[10] Which many VPE's do given that one of the goals of a VPE is the removal of syntax errors
[11] Described on page 53

graphical displays have increased in size and resolution over time, this can still remain a problem inherent to visual representations.

5.1.2 Application of the Cognitive Dimensions Framework to Other Environments

In this section we will briefly recap the application of the Cognitive Dimensions framework to three different visual programming environments: Prograph, LabVIEW, and the VFPE from [3, 53, 80, 85].

In terms of abstraction gradient, both Prograph and the VFPE were found to be abstraction tolerant. Both allowed the introduction of new abstractions, however it was argued that the VFPE had a lower minimum level of abstraction as little needs to be learned to begin working with the environment. LabVIEW however was found to be an abstraction hating language, as it provides no way of adding new abstractions past the "bundle of wires" construct.

As a domain specific language, LabVIEW has excellent closeness of mapping for problems in the electronics simulation domain, but outside of that domain there were difficulties expressed due to the fact common constructs required extra "administrative stuff" (such as looping requiring knowledge of the shift register to transfer values from the end of one iteration to the next). The library support in LabVIEW reflected the domain specific nature of the language as well, limiting the positive closeness of mapping largely to the domain of electronics simulation and circuit design. The VFPE and Prograph as general programming environments do not suffer from the same limitation. In particular the library support of the VFPE was found to be somewhat weak compared to other environments used in industry, and as such more code was required to achieve similar results, thereby hurting the closeness of mapping from problem to program domain.

VPE's in general have been found to score well in regards to consistency due to the simpler syntax. No specific problems related to consistency were noted about LabVIEW, Prograph, or the VFPE. Additionally, the VFPE standard library is based upon the Haskell standard library which when possible keeps argument naming and order consistent, thereby also improving this dimension in that environment.

In terms of diffuseness or terseness, VPE's have traditionally suffered due to the fact that graphical entities tend to consume more screen real estate than textual lexemes. The sample rocket program mentioned on page 53 was implemented in

all three environments to measure this dimension. The LabVIEW implementation consisted of 45 icons and 59 wires for a total of 104 graphical entities and fit easily onto a "medium sized screen" [3]. The Prograph implementation occupied 11 windows, with a total of 52 icons and 79 connectors to give a total of 131 graphic entities, and would not fit onto a single screen. Being a functional language, it is not particularly surprising that the VFPE was found to be quite terse and the sample implementation of the rocket program consisted of 98 graphic entities. The counts for the VFPE did not include node-joining lines as they are semantically redundant and drawn by the environment, not the user.

In terms of error-proneness, most VPE's essentially eliminate typographical and syntax errors due to the constrained interface and all three environments were found to share this property.

Hard mental operations were discovered in LabVIEW and Prograph. The difficulties in LabVIEW seemed to mostly arise due to the use of logic gates in conditionals, as users tried to "trace" through the connections between the gates. This was somewhat surprising as the notation for logic gates is very well known. Green speculated that perhaps this difficulty may be due to the fact that the notation evolved from a paper-based notation, where users can pencil in intermediate results as they trace through the various connections between gates. Since LabVIEW does not provide a facility for this, users have to keep in their head the intermediate results of conditional expressions. Difficulties in Prograph were noted surrounding control of loops. In terms of the VFPE no specific hard mental operations were identified, however it is not clear if this is an indication of the absence of hard mental operations, or simply that none were noted.

The box-and-wire representation used in many VPE's greatly helps to avoid hidden dependencies at the local level, as it makes the dependency between components visually explicit. All three environments displayed this. Past a local level however, Prograph in particular greatly suffers from issues related to hidden dependencies due to deep nesting and no overview of the nesting structure. One can navigate down a call graph by clicking methods to open up a window for that method, but the reverse is not true.

In regards to premature commitment, it was also noted that VPE's using the box-and-wire representation tended to allow fewer constraints on the order of creation of code. That is, none of the three visual environments surveyed were found to have any issues with respect to commitment to order of creation. LabVIEW was found to

have difficulty with commitment to layout, due to the fact that the environment was so viscous (changes after the fact were difficult to do). Prograph had great problems in regards to commitment to connections, as avoiding the "visual spaghetti" of wires crossing over one another took a great deal of lookahead on the part of the user. Again, viscosity was cited as a problem as if one chose a poor layout of components in a Prograph program which lead to overlapping wires, modifying it to improve the layout was difficult. The VFPE was found to be very strong in terms of commitment to layout and commitment to connections as both are controlled by the environment. The drawback to this is that secondary notation is negatively affected as layout cannot be used to convey extra meaning as would be the case in a more freeform layout scheme. It was also noted by Kelso that any syntax-directed editor (such as the VFPE) will always suffer from problems related to premature construction commitment problems unless it allows easy relocation of code into temporary "meaningless" locations.

Both Prograph and the VFPE were found to have strong support for progressive evaluation as both allowed the ability execute program fragments as opposed to requiring entire complete programs before execution can take place. Prograph in particular was quite strong in this regard as it allows program fragments to be executed which can even contain unfinished structures (the interpreter will execute up to the point where there is unfinished code). The VFPE allows any expression can be evaluated so long as it is complete. LabVIEW however required that a complete program must be constructed before any execution can take place.

LabVIEW was found to have significant problems in regards to role expressiveness, due to the absence of identifiers, poor secondary notation and "beacons" of code. Secondary notation was somewhat aided by the extensive control over program layout as well as the ability to add comments to components and wires[12]. Viscosity is hurt by this however, as the overhead of rearranging components made small changes more costly. Prograph programs contained identifiers to help role expressiveness, but this dimension also suffered from problems related to secondary notation, as the secondary notation in Prograph was found to be very limited. Comments could be applied to primitives and methods, but not groups of objects. Secondary notation is partially aided by the ability to control layout of components, however, this turns out to be diminished by the need to minimize wire crossings (which had a negative impact on viscosity). Support for role-expressiveness in the VFPE was found to be aided by the

[12]But not groups of components

ability to add as many or as few identifiers as one wishes. Kelso also identified some common "beacon" structures that tended to arise in the VFPE such as compositional pipelines appearing as a vertical string of beads, and arithmetic expressions tending to form into pyramids. There was however little support for secondary notation identified in the VFPE, short of the ability to add comments to any node (sub-expression), and even this was hurt by the lack of a visual indication of the presence of a comment. The automatic layout of components was found to aid viscosity as it was identified that the freedom to arrange components imposed an overhead on any change to the program as one would have to rearrange the components, however, this came at the cost of secondary notation as layout could not be used to convey extra meaning.

In terms of Viscosity, the "straw test" of making the small change to the rocket program was done with each of the three environments, and took 508.3 seconds in LabVIEW, 193.6 seconds in Prograph, and 105 seconds in the VFPE. Specifically, the LabVIEW version took so long due to the fact that connections had to be rebuilt when components were rearranged. It was also noted that all three visual environments took significantly longer to incorporate the change than doing so in a traditional textual environment, as the same rocket trajectory program was implemented in Microsoft BASIC, and the "straw test" change took only 63.3 seconds. Thus while visual environments tend to be rather terse (i.e. - have strong diffuseness), they also tend to suffer from viscosity problems.

LabVIEW was found to suffer from problems in regards to visibility and juxtaposibility, in particular the conditional was identified as being a significant problem. In LabVIEW, a conditional can only display one branch of the conditional at any time, a clear violation of juxtaposibility. The VFPE suffered from a similar problem with patterns in expressions. With patterns in the VFPE, only one "case" can be displayed at any given time, there is no way to simultaneously display all cases of a pattern. As well, oftentimes more complex expressions can be automatically laid out in a way that is impossible to view on a single screen, thereby making it impossible to juxtapose distant parts of the same expression.

5.2 Applying the Cognitive Dimensions Framework to the Gem Cutter

In this section we shall apply the Cognitive Dimensions framework developed by Green to the Gem Cutter VPE.

5.2.1 Abstraction Gradient

As described in Section 5.1.1 the Abstraction Gradient dimension tries to measure how much needs to be constructed and learned in order to begin making the programming environment perform a task, and additionally how easy is it to add new abstractions to the environment.

Discussion of Dimension

In regards to the Gem Cutter, much like the VFPE, as a functional language there is a relatively small set of constructs to master before making the environment perform a task. As a base minimum, one only needs to be familiar with the notion of a function, and how a gem is a visual representation of that concept, along with the mechanics of how to connect gems together on the tabletop. As one wishes to do more sophisticated tasks, introduction of the additional gem types[13] will be required, however, this is still a relatively low minimum level of abstraction. This is a great strength of the Gem Cutter, particularly from the perspective of a student learning to program, as little needs to be learned and mastered before beginning to make the environment perform basic tasks.

The Gem Cutter like most programming languages, would be an example of an abstraction-tolerant language, though only barely. The only mechanism for introducing new abstractions is the ability to define new gems and use those gems in other gem designs. Aside from this basic abstraction mechanic there is little or no support for introducing new abstractions. In particular the lack of ability to introduce new types as discussed in Section 3.3.2 greatly hinders ones ability to introduce new abstractions to reduce the complexity of larger problems. This is mitigated somewhat by the introduction of code gems which allow one to introduce CAL code snippets at any point into a gem design. Since CAL is very much an abstraction-tolerant language fully supporting the ability to introduce and define new types, this allows a

[13] Such as collector/emitter pairs, value gems, code gems, and record gems

"loophole" where one can bring all the expressive power of CAL to the Gem Cutter. However, code gems will only be useful to those who already understand the CAL language. This would be roughly analogous to requiring users of Alice to write Java code to create new object types to use in their Alice programs, which would seem rather cumbersome.

Remedies, Workarounds, And Trade-offs

Green suggests that a possible remedy for problems related to the abstraction gradient of an environment is to introduce incremental abstractions. This would be where the environment has a low starting abstraction barrier (ie - start with a low minimum level of abstraction where little needs to be learned to begin working with the environment), and to allow the introduction of new abstractions that will aid users later. To a certain degree the Gem Cutter does this already, as there is a relatively low minimum level of abstraction as mentioned above, and one can learn about the other gem types as they progress with the environment. What is missing are enhanced facilities for newer abstractions, most notably in regards to types. The introduction of a mechanism for defining a new type in a visual way would help to alleviate this aspect much more than the current "workaround" of using code gems.

5.2.2 Closeness of Mapping

As described in Section 5.1.1 the dimension of Closeness of Mapping tries to measure the gap between the problem domain and the program domain.

Discussion of Dimension

In terms of "programming games", the Gem Cutter imposes relatively few upon the user. As a visual functional language, the notation of the Gem Cutter is rather concise. Sometimes gem designs can become "cluttered" with an excessive number of visual entities, but this is more of an issue related to Diffuseness than of Closeness of Mapping.

Additionally, there are two other issues to consider from the perspective of Closeness of Mapping. The first is the issue of the language's standard library support, and the second is what constructs the language allows to be built to improve the Closeness of Mapping.

With respect to the Gem Cutter, there is an extensive set of predefined gems provided to the user. In particular, there is a great deal of library support in the Quark framework, CAL and the Gem Cutter for working with relational databases. The DatabaseMetadata, Sql, SqlBuilder, and SqlParser modules are all imported by default into the Gem Cutter, and provide extensive support to close the gap between a problem in the domain of relational databases and the Gem Cutter environment itself. In addition to these modules, the CAL language itself has a significant collection of modules for working in a variety of domains, any of which can be imported into the Gem Cutter. The library support in the OpenQuark framework is not as robust as some more "industry-proven" languages (such as Sun's Java API), but is much more comprehensive and varied than the Prograph or VFPE environments.

In terms of constructs to improve the Closeness of Mapping, much like the VFPE, the Gem Cutter can use higher-order functions to enable various programming "idioms" which can help greatly with changing the mapping from problem to program domain.

Remedies, Workarounds, And Trade-offs

There are no specific issues identified with the Gem Cutter with respect to Closeness of Mapping. Short of additional library support, there is little that can be introduced to the Gem Cutter to improve this dimension.

5.2.3 Consistency

As described in Section 5.1.1 the dimension of Consistency tries to address how easy is it to infer the remaining parts of the language, once one has learned part of the language.

Discussion of Dimension

As noted in Section 5.1.2, visual languages tend to be very consistent due to the much simpler syntax. The Gem Cutter is no different in this respect. In particular, once one masters the metaphor of "gem as function" with inputs connecting to the left side of the gem, and the single output leaving the right side, the rest of the environment becomes very easy to learn as all gems follow this pattern.

Additionally, another issue related to consistency is library regularity. Much like the Haskell language which inspired it, CAL (and as a result the Gem Cutter) keeps

argument ordering very consistent. For example, if a gem takes two arguments, one a function and the other a primitive type (such as an Integer), then the function argument will be the first (top-most) argument in the gem display. This follows the common functional programming convention of having higher order functions come before primitive types in argument lists. However, one very interesting thing to note however is that while this convention is followed, there is no restriction in the Gem Cutter in terms of in what order arguments are bound. For example, consider the `map()` function common to virtually all functional programming languages. The `map()` function is a function which takes two arguments, a function `f()`, and a list of items $(x, x_0, x_1, ..., x_n)$. The result returned is the list with the function applied to each element or the list $(f(x), f(x_0), f(x_1), ..., f(x_n))$.

In traditional textual functional languages such as Haskell or SML, one must bind the function argument before one binds the list argument. The consequence of this is that we can use `map()` to create new functions of one argument – a list. We cannot however use `map()` to create a new function of single function argument. In the Gem Cutter, because we can bind arguments in any order, we do not have this limitation. This would seem to be the best of both worlds, we have the flexibility and freedom to bind arguments in whichever order (thus allowing us greater expressivity), but we also have the consistency of argument ordering (though top down instead of left to right).

Remedies, Workarounds, And Trade-offs

The Gem Cutter is remarkably consistent, and as such there are no specific issues or ways that this dimension could be improved.

5.2.4 Diffuseness

As described in Section 5.1.1 the dimension of Diffuseness (or terseness) tries to measure how many symbols are required to express a given meaning in the notation being examined.

Discussion of Dimension

With respect to the Gem Cutter, it is worth noting that the sample implementation by Green of the rocket trajectory program written in BASIC (seen in Program 7) was done in an iterative style, and since the Gem Cutter is a functional language there

are no mechanisms for iteration instead requiring recursion to be used. Thus while the implementation in Gem Cutter of the rocket trajectory program is based upon the BASIC version it is structured significantly differently. Instead of having a single routine that encompasses the entire problem, we created a gem called `rocket1()` which given the current state of the rocket[14] at time t, calculates the new state of the rocket at time $t + 1$ and returns this as a 6-tuple. In some respects, this is a solution to the rocket trajectory problem by itself. However, to more closely match the semantics of the BASIC version, a second gem was created called `rocketTester()` which generates the state of the rocket from time 0 to whatever time the rocket's vertical distance becomes negative. Presumably, Kelso had to do something similar for the implementation in the VFPE of the rocket trajectory problem, however, the layout of his implementation was not supplied so we do not know how closely his approach matched that of the Gem Cutter implementation.

Program 7 The Implementation of the Rocket Trajectory Problem in BASIC[3]

```
Mass = 10000
Fuel = 50
Force = 400000
Gravity = 32
WHILE Vdist >= 0
      IF Tim = 11 THEN Angle = .3941
      IF Tim > 100 THEN Force = 0 ELSE Mass = Mass - Fuel
      Vaccel = Force*COS(Angle)/Mass - Gravity
      Vveloc = Vveloc + Vaccel
      Vdist = Vdist + Vveloc
      Haccel = Force*SIN(Angle)/Mass
      Hveloc = Hveloc + Haccel
      Hdist = Hdist + Hveloc
      PRINT Tim, Vdist, Hdist
      Tim = Tim + 1
WEND
STOP
```

The breakdown of graphic entities in the two gems can be seen in Table 5.1. The complete solution of both gems for the rocket trajectory problem required 161 different graphic entities, far more than the 131 for Prograph, 104 for LabVIEW, and 98 for the VFPE. Note that connector lines were included in the totals for the

[14]The vertical distance, vertical velocity, horizontal distance, horizontal velocity, and current mass

Table 5.1: The Breakdown of Graphic Entities in the Gem Cutter Solution to the Rocket Trajectory Problem

Gem Type	`rocket1()` **Count**	`rocketTester()` **Count**	**Total For Both**
Connectors	59	14	73
Emitter Gems	27	1	28
Function Gems	19	5	24
Value Gems	9	7	16
Collector Gems	12	1	13
Record Selection Gems	6	1	7
Totals	132	29	161

Gem Cutter implementation. This was different than the totals for the VFPE where connectors were not included in the total due to the fact that connection lines were not established by the user. Connection lines were included in the total for Prograph, however, this was due to the fact that when one wishes to move a component in a Prograph layout, they also need to manually adjust the connections. In Gem Cutter, while the connections between gems need to be established by the user, once they are established movement of gems around the tabletop does will cause the connections to be redrawn automatically. If we leave out the connections, then the total for the Gem Cutter implementation falls to 88 graphic entities, easily the most terse of the four considered.

Also note that the single target gem was not counted in these numbers as it is a graphic entity that always appears in any gem design (it is not added by the user). Given that there is only a single target gem for any gem design, the inclusion or exclusion of this entity will not significantly alter the graphic count either way.

The use of record selection gems to parse a tuple into subsequent parts incurs a significant number of entities as well. While the count in Table 5.1 does not seem to indicate this (as there are only 7 record selection gems), what is missing from this total is the fact that connected to the record selection gems are two connectors, and a collector and emitter pair for each (an emitter for the tuple being parsed, and a collector to "name" the parsed value). Thus breaking the 6-tuple rocket state into six separate values in `rocket1()` adds a total of 30 entities (including connectors, 18 if we omit connectors).

In terms of screen space, the implementation of the `rocket1()` gem would not fit on a single 1280x960 resolution screen, instead scrolling roughly halfway past the end

of the screen. The `rocketTester()` gem however fit easily on a single screen, and can be seen in Figure 5.2.

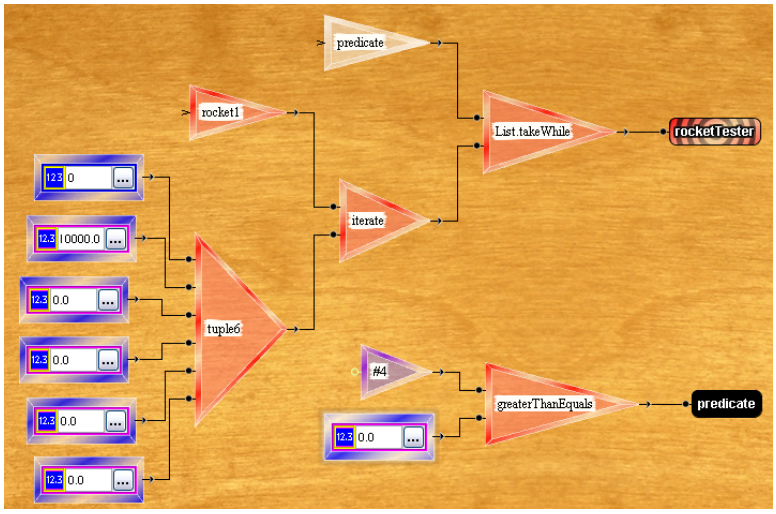


Figure 5.2: The `rocketTester()` Gem

Additionally, simple mathematical expressions tend to take up a disproportionate amount of space on the screen. It is a much more compact representation to have a single character such as '*' to denote multiplication, than a full gem with the word "multiply" written on it. One can alleviate this with code gems, however the use of code gems are not in the "spirit" of the visual paradigm.

Conditionals can also introduce seemingly redundant entities. Consider the if/else statement in Program 8 written in Java. This fragment contains a single if statement, and assigns values to two different variables depending on the value of someBoolean-Expression. This same fragment translated into Gem Cutter would be what we see in Figure 5.3. Note that there are now two if statements, one to determine the value of `var1`, and another to determine the value of `var2`. As there is no mechanism for "blocks" of code in the Gem Cutter, one has to separate the two collections of statements into separate if statements.

Remedies, Workarounds, And Trade-offs

Specifically, the biggest issue identified with respect to Diffuseness in the Gem Cutter is in regards to working with composite data types such as lists and tuples. As mentioned in the rocket program, the parsing of a tuple into separate values is cumbersome as it requires the use of a record selection gem for each value in the tuple,

Program 8 A Hypothetical if Statement in Java

```
if (someBooleanExpression)
{
      var1 = 10;
      var2 = 20;
}
else
{
      var1 = 20;
      var2 = 10;
}
```

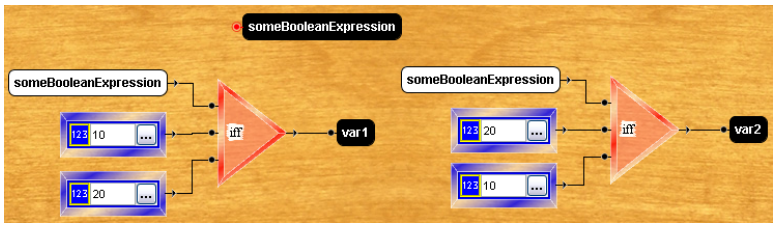


Figure 5.3: A Hypothetical if Statement in Gem Cutter

along with an emitter (to identify which tuple to draw from) and a collector to name the decomposed value[15].

Lists are somewhat better, in that there is a single gem for getting the head of a list. However if one needs a specific number of items from a list, doing so in the Gem Cutter becomes quite verbose, as it requires multiple collector/emitter pairs, as well as use of the head and tail gems to decompose the list.

Many textual functional languages have support for patterns, which would help to alleviate the verbosity of decomposing composite data types. The VFPE supports a very compact pattern mechanism, where multiple patterns to decompose a composite value can be specified in one single graphic entity. The trade off here is visibility, as in the VFPE there is no way to view all patterns simultaneously. Perhaps in the Gem Cutter a "tuple decomposition" gem could be added to the notation where a single tuple is input to the gem, and the gem would then have multiple outputs (one for each part of the tuple). The trade off with this approach however is Consistency, given that as it currently stands *all* gem types in the Gem Cutter have exactly one single output.

Conditionals are a tougher problem, due to the fact there can be only a single output from an `iff()` gem. If `iff()` gems were modified to allow for multiple outputs, Consistency would again suffer as a result as all other gems have a single output.

5.2.5 Error-Proneness

As described in Section 5.1.1 the dimension of Error-Proneness tries to measure if the notation itself induces or encourages careless mistakes.

[15]Strictly speaking the collector is optional, however, if the value is going to be used more than once, having a collector will reduce the overall number of gems required

Discussion of Dimension

As a visual environment, the Gem Cutter dramatically reduces the number of possible errors that can occur due to user error. Syntax errors are virtually removed, and the constrained nature of the environment where gems can only be connected when they are type compatible greatly help to avoid errors related to data type.

The biggest shortcoming with the Gem Cutter in regards to this dimension is the issue of type compatibility being broken by indirect modification to other gems. As mentioned in Section 3.3.2, if we have one gem `a()` which calls gem `b()`, then a change to `b()`'s type signature will break the design of `a()`.

Remedies, Workarounds, And Trade-offs

Aside from the type compatibility issue, there are no significant issues in regards to Error-Proneness with the Gem Cutter. To address the type compatibility issue, there needs to be some sort of mechanism in place that checks to see if changing the type signature of a gem will violate any type constraints in other gems, and to prevent the user from saving the new type signature until that dependency is broken.

5.2.6 Hard Mental Operations

As described in Section 5.1.1 the dimension of Hard Mental Operations tries to identify operations within the environment which are difficult to express due to deficiencies in the notation.

Discussion of Dimension

Remarkably, perhaps due to the relatively simple syntax, there are very few hard mental operations we identified in the Gem Cutter. Using the "two-step" test for a Hard Mental Operation, we could not identify any specific issues in the Gem Cutter which met the criteria for a hard mental operation.

The "box-and-wire" model can become difficult for a user to mentally "trace" execution, but only if the design of a gem becomes quite large. And given that like the VFPE, the Gem Cutter allows for the ability to introduce collections of named expressions anywhere in a gem design, or to decompose a problem into smaller parts and save them as separate gems, much of this difficulty is minimized/eliminated.

It is interesting to note that both functional VPE's discussed in this thesis (the Gem Cutter and the VFPE) had no identified Hard Mental Operations. An interesting question would be to explore if this is a consequence of the functional paradigm, or just properties of these two specific environments.

Remedies, Workarounds, And Trade-offs

No specific difficulties were identified with regards to this dimension, and as such there are no specific issues or ways that this dimension could be improved.

5.2.7 Hidden Dependencies

As described in Section 5.1.1 the dimension of Hidden Dependencies tries to answer the question "Is every dependency overtly indicated in both directions? Is the indication perceptual or only symbolic?".

Discussion of Dimension

In regards to the Gem Cutter, there are a few points where the software very much aids this dimension. One example is the use of diamond and line connections between gems. This visual representation of the dependency between two gems *within a single design* is made explicit via the notation. Put another way, the added visual cues to the notation which makes the dependency explicit improves this dimension of the Gem Cutter. This is very similar to the findings by Green in regards to LabVIEW and (on a local level) Prograph as discussed in Section 5.1.2, where the "linking wires" of LabVIEW served a similar purpose.

The use of visual cues to indicate types also aids this dimension. For example, there is a dependency between collector gems (which collect a value) and emitter gems (which emit the value collected). The fact that both collector and emitter gems are annotated and labeled provide a visual cue that there is a dependency between the collector and emitter. Additionally, collector and emitter gems share the same shape (though a different colour), further conveying the message that there is a dependency between them.

A problem with regards to hidden dependencies within Gem Cutter is the issue of type dependencies between separate gem designs discussed in Section 3.3.2. There is no mechanism within Gem Cutter to see what gems a particular gem is dependent

upon short of examining the gem design. Even if there were, the fact that the dependency relationship is transitive would require such a facility to be able to discover dependency relationships multiple levels deep. For example, if `a()` calls `b()` which calls `c()`, it is not enough to know that `c()` is dependent upon `b()`, but `a()` as well. There is the facility for determining what gems depend upon a given gem (that is, "what gems depend upon this gem?"), but not the other way around (that is, "what gems does this gem depend upon?"). To use the language of the cognitive dimensions framework: not every dependency is overtly indicated in *both* directions.

Remedies, Workarounds, And Trade-offs

Green suggests three possible remedies to the Hidden Dependencies dimension: (a) adding cues to the notation; (b) highlighting different information; and (c) providing extra tools.

In regards to the type dependency issue it is difficult to envision what possible additional cues could be added to the interface without "cluttering" the interface. The possibility of an extra tool which allows for one to see a tree-like structure of all gems which a given gem depends upon could help.

5.2.8 Premature Commitment

As described in Section 5.1.1 the dimension of Premature Commitment addresses if users of the environment are forced to make decisions before they have all the information they need.

Discussion of Dimension

There is no restriction in terms of commitment to layout or commitment to connections, as the Gem Cutter is completely "freeform", allowing users to arrange gems on the tabletop as they see fit, and connections maintain a consistent ordering. Minimizing wire crossings is simple due to the fact that all gems have a single output on the right hand side. That is, the fact that a gem graph is a tree and not a general directed graph (like in Prograph) eliminates any issues with regard to commitment to connection. The "Tidy Tabletop" option under the View menu in Gem Cutter will allow one at any time to instantly, automatically arrange all gems in a manner which eliminates wire crossings (though it might not make the best use of screen real estate).

In terms of commitment to construct, if one decides at any time that a particular gem is the incorrect gem to use then so long as the new gem to use is type compatible with the old it is a simple matter of deleting the original, replacing it with the new gem, and restoring the connections. If the type of the new gem is not compatible with the old, then one may need to introduce extra gems to convert the inputs and outputs of the gem to the newer types, but this is still a relatively simple process.

From the perspective of commitment to order of creation, the Gem Cutter allows one to create parts of gems in any order. The only restriction imposed upon the user is that a gem must be defined before it can be used. Like like most programming environments, if we have a function called `a()` which calls `b()`, then `b()` must be created before `a()` can be. Oftentimes in the textual domain, users will create "stub functions" (functions which have the correct type signature and return statement, but no body) for this purpose, and this is possible in the Gem Cutter as well. Before an emitter gem can be used, the corresponding collector gem must be defined. Aside from these relatively minor restrictions, the Gem Cutter allows a great deal of freedom with respect to order of creation within a single gem. The only significant problem with respect to order of creation arises *between gems*. As mentioned in Section 3.3.2, the issue of type dependency between gems can cause significant problems when one realizes that a design mistake has been made, essentially forcing the user to go back, and break any dependency between the various gems before making the necessary new change.

Remedies, Workarounds, And Trade-offs

The main remedies for issues related to premature commitment that Green suggests are (a) to allow one to decouple the portion being worked on from the rest of the problem, (b) to reduce viscosity, and (c) to reduce or remove constraints on the order of actions.

With respect to the issue of type dependencies between gems, currently it is *possible* to decouple the portion being worked on, but it is rather difficult to do so as it requires modifying any gems which depend on the gem being modified. That is, the high viscosity of Gem Cutter amplifies the problems related to commitment to order of creation. This is not surprising, as Premature Commitment most directly trades off against Viscosity, as if a system has low viscosity (that is, changes are relatively easy to make) then the impact of Premature Commitment problems can

be mitigated. It is difficult to envision a solution to this problem without sacrificing type safety. If the Gem Cutter were to allow one to modify an existing gem to have a different type signature, then gems which rely on the gem that was changed will no longer be syntactically correct. At the very least however, it would be helpful to have the Gem Cutter indicate which gems are dependent upon the current gem, so at least we can more easily identify which gems we need to decouple the current gem from if we wish to change it.

5.2.9 Progressive Evaluation

As described in Section 5.1.1 the dimension of Progressive Evaluation examines the environments facilities for allowing one to execute a partially completed solution.

Discussion of Dimension

This dimension is one of the strongest of Gem Cutter, particularly in regards to students learning to program with it. The Gem Cutter allows one to evaluate any program fragment at any time by right-clicking the gem fragment and choosing "Run Gem". As mentioned in Section 5.1.1, this ability to evaluate small fragments of code rather than having to create an entire program is extremely useful to students learning to program as they need to evaluate their progress frequently to assess if the code that they have written does what they think it should do.

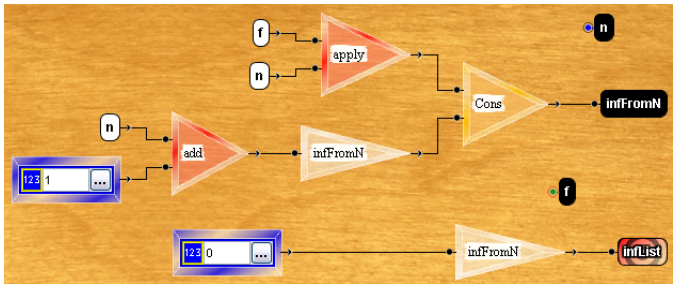
The only small limitation to the ability to run gems in this fashion is in regard to infinite lists. As a lazy (non-strict) language, CAL (and as a result the Gem Cutter) allows one to create lists of infinite length[16]. In Figure 5.4a we see a design for a gem called `infList()` which given a function `f()`which generates the n^{th} term of a series, will generate the infinite list $(f(0), f(1), f(2), ...)$. If in the design of `infList()` we pick "Run Gem" on the target gem, the gem will be run, and the Gem Cutter will continue to evaluate `infList()` until memory has been exhausted. If we wish to see if the gem is correct, we must create a new gem, and combine the `infList()` gem with the `take()` gem from `List` module and run that subexpression, as seen in Figure 5.4b[17].

Also note in Figure 5.4b that when a user selects "Run Gem", the environment will

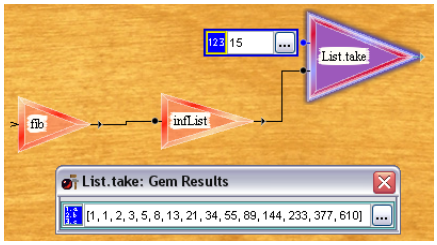
[16]Of course, this the lists are infinite from an abstract viewpoint, in reality there is a finite amount of memory in any machine

[17]In this example we are using the `infList()` gem to generate the first 15 Fibonacci numbers

prompt the user for any incomplete or unbound arguments. This is even better than in Prograph where once an unspecified argument was found execution was terminated.



(a) The InfList() Gem Definition



(b) Using the InfList() Gem To Generate Fibonacci Numbers

Figure 5.4: Using The InfList() Gem

Remedies, Workarounds, And Trade-offs

The support for Progressive Evaluation in the Gem Cutter is quite strong. The infinite list issue is relatively minor, only requiring one to create a new subexpression that makes use of the gem which generates the infinite list. Perhaps a small improvement would be if the user tries to run a gem which generates an infinite list directly, to have the Gem Cutter display the evaluated results rather than just a "out of memory" error message.

5.2.10 Role-Expressiveness

As described in Section 5.1.1 the dimension of Role Expressiveness tries to evaluate the environment's support for answering the user's question of "what is this bit for?".

Discussion of Dimension

The Gem Cutter has a number of aspects to support Role Expressiveness. For example, all gems have meaningful identifiers attached to them (the name of the function, etc). Modularity is supported through the use of modules and gems (functions), though while all gems from the standard libraries are organized into modules, all user-created gems go into the same module (the "GemCutterSaveModule" module) and creating new modules to use in the Gem Cutter is a nontrivial task (as mentioned in Section 3.3.2). In terms of code "beacons", we found the same structures identified

by Kelso in the VFPE, though rotated 90 degrees as the Gem Cutter arranges items from left to right as opposed to top down (as in the VFPE). For example, compositional pipelines took the form of a "vertical string of beads"[53] in the VFPE, where in the Gem Cutter they became a horizontal string of Gems, etc. In some ways, list processing created a "staircase" of gems, as most of the standard list processing gems (`map()`, `filter()`, `zip()`, etc) have two inputs, the top being a function and the bottom being a list. A common programming idiom in the functional world is to have the output of list processing functions fed into other list processing functions, which in the case of Gem Cutter results in this left to right, rising "staircase" of gems.

Remedies, Workarounds, And Trade-offs

The only additional support needed for modularity in regards to the Gem Cutter is the ability for one to more easily create their own modules. As it stands, one has to resort to writing CAL code by hand, then modifying the Gem Cutter environment to import the modules into the current workspace.

5.2.11 Secondary Notation

As described in Section 5.1.1 the dimension of Secondary Notation explores what support the environment has for conveying extra information to users beyond the official syntax of a notation.

Discussion of Dimension

Layout in the Gem Cutter is freeform, and as such can be used to convey meaning to users. This is very much like Prograph where layout was also freeform, and unlike the VFPE where layout was completely controlled by the environment. Colour and visual cues are used to convey extra meaning, function gems are coloured red, user-created local functions are coloured a yellow colour, record selection gems a violet colour, and collector gems are black, and emitters are coloured white. As well, the target gem that represents the gem's return value is coloured as a "bullseye" to convey the meaning that it is the overall "target" of the current gem.

Commenting is supported through the properties window of any collector gem or the target gem. The Properties window for the `rocket1()` gem described in Section 5.2.4 is shown in Figure 5.5. In the properties window one can describe and

annotate various aspects of the gem in question. As mentioned the same set of documentation can be done for any collector gem within a gem, thus allowing the ability to comment groups of gems. However, individual gems other than collector gems cannot be annotated with comments. In the case of value gems it is possible to "name" them by feeding a value gem into a collector gem and then naming the collector gem, however this increases the number of graphic entities on screen thereby negatively impacting Diffuseness. Additionally, though comments are supported through the properties window of a gem, like the VFPE there is no visual cue that such a comment exists.

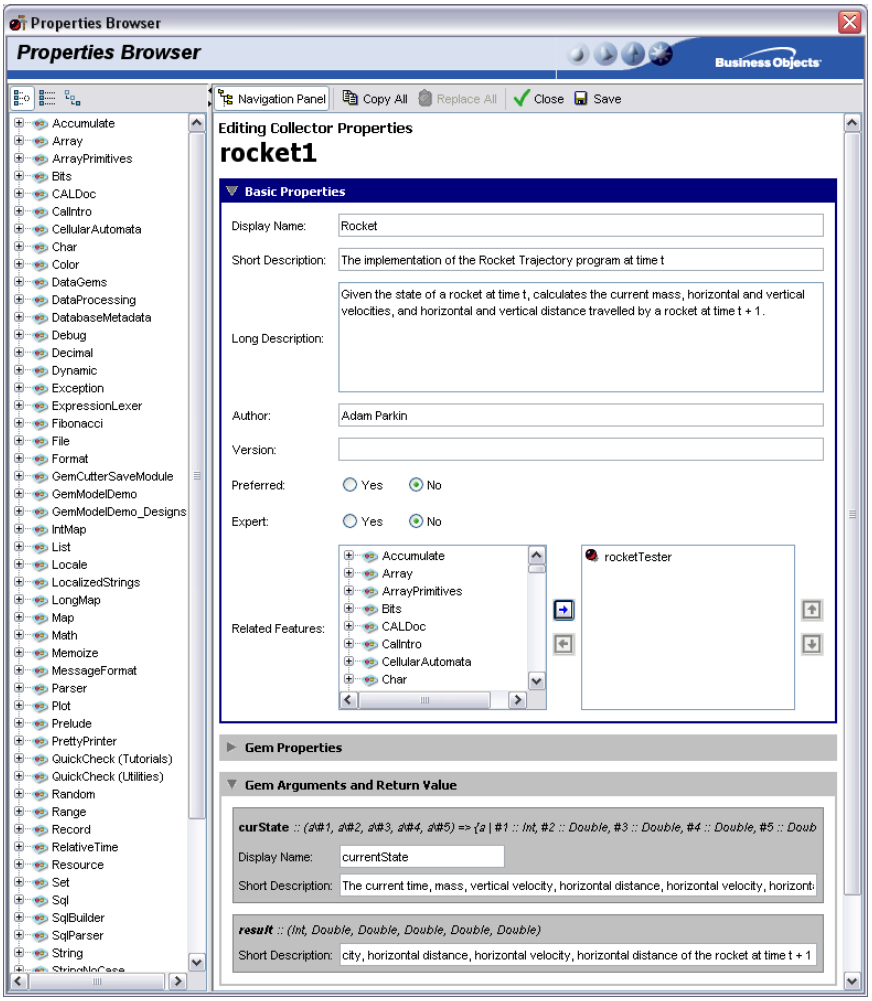


Figure 5.5: The `rocket1()` Gem's Properties Window

Remedies, Workarounds, And Trade-offs

The main remedy to problems associated with Secondary Notation that Green describes is to "provide tools in the system to allow components to be labeled and described and their relationships made explicit" [80]. The Gem Cutter does this to a certain degree with the Properties window, but it is very coarse grained – there is not enough support in the form of tools to allow sub-parts of gems to be documented and described.

It is also worth noting that Secondary Notation often trades off against Viscosity, as if the structure is changed, then the secondary notation becomes obsolete and also needs to be changed. In the case of the Gem Cutter when the arguments and return types of a gem change, the properties are automatically updated, which helps to mitigate this concern. Specifically, if arguments are removed from the gem definition, then they are also automatically removed from the properties window. If arguments are added then an entry for the argument will be added automatically to the properties of the gem, however a meaningful description still needs to manually be added by the user. Other metadata stored in the properties window such as author name, description of the gem, etc, will also have to be maintained manually by the user.

5.2.12 Viscosity

As described in Section 5.1.1 the dimension of Viscosity tries to measure an environment's resistance to change.

Discussion of Dimension

The biggest issue in terms of Viscosity in the Gem Cutter is the type dependency issue between gems described in Section 3.3.2. This imposes a serious impediment on a user to refactor existing gem designs.

The manual layout problem described by Kelso in [53] is somewhat mitigated by the ability in Gem Cutter to instruct the environment to automatically rearrange all items on the tabletop[18]. There is still overhead associated with making changes to expressions due to the having to make and break connections between gems, and move them around to make room for whatever changes need to be made. This became particularly evident while performing the "straw-test" of making the small change to

[18]Via the "Tidy Tabletop" menu command

account for air resistance to the rocket trajectory problem, as a significant time consuming aspect of the change in the Gem Cutter version of this problem was breaking the connection between the calculation of the new horizontal and vertical velocities and the collector gem which collected these values, and inserting the gems to incorporate the change to air resistance. In terms of time taken, incorporating the air resistance change into the Gem Cutter version of the problem was done three times, each application of the change was timed, and took 172 seconds on average to complete. This is dramatically better than the LabVIEW (508.3 seconds) and somewhat better than the Prograph (193.6 seconds) results, but worse than the VFPE (105 seconds).

One thing worth noting is that the change for air resistance was confined only to the `rocket1()` gem, and not the `rocketTester()` gem. Thus the type dependency issue was not encountered in this test. If `rocketTester()` did have to be changed due to the changes in `rocket1()` the amount of time taken likely would have been dramatically higher.

Remedies, Workarounds, And Trade-offs

Green notes than an important point to consider about Viscosity is that it is not always harmful [80]. High viscosity may encourage users to reflect more about the problem before "diving in" (though this would be an example of negatively impacting Premature Commitment). Low Viscosity may encourage users to make changes more often than need be, and as a result increase the Error-Proneness of the system.

The main workaround for extremely viscous environments is to detach from the environment itself. That is, to use a different environment with very low viscosity to plan out a solution to the problem, then to translate the solution in the low viscosity environment to the high viscosity environment. For example, one may use pen and paper to sketch out a gem design, changing as problems are encountered, and then recreate the design in Gem Cutter. This negatively impacts Premature Evaluation as one cannot "execute" a drawing on paper. Alternatively, a possible workaround used at times by the author of this thesis was to write a solution in a textual language that was less viscous, and then translate that text-based solution to the Gem Cutter. However, this would seem counterintuitive to the motivation behind using the Gem Cutter in the first place.

With respect to the rearranging the layout problem, a possible workaround is to

heavily decompose the problem into very small functions. Making a change to the internal structure of a gem is easy when the gem is small. Counterbalancing this is the type dependency issue – if another gem depends on the one being worked on then changing the type signature of the function is problematic, and the more gems there are, the greater the likelihood there will be a type dependency issue between them.

5.2.13 Visibility

As described in Section 5.1.1 the dimension of Visibility is concerned with the environments support for making required materials easily accessible.

Discussion of Dimension

In terms of Visibility, navigational visibility is generally strong in the Gem Cutter, as the number of actions needed to find a particular gem in a gem design is usually very small, at most being a simple scrolling of the window to find what one is looking for. If a gem's design becomes overly "messy" or "cluttered", a user can select the "Tidy Tabletop" option on the view menu to have the environment automatically arrange gem groupings in a top-down manner. Alternatively, the Scope Window in the top left corner (seen in Figure 5.6) of the Gem Cutter interface, gives a hierarchical, tree-like listing of all components currently on the tabletop. Clicking any item in this list will immediately center the tabletop on that item. So in the worst case finding any item in a gem design is a matter of one or two clicks. Note that as well, any subexpression in the Scope Window can be collapsed to allow one to focus on other parts of the gem design. In Figure 5.6 is seen with the "hdist", "hveloc", and "mass" collector gem expressions being collapsed, and all others being expanded.

Predefined functions from the standard libraries are organized into modules, and are easy to find via the Gem Browser. As well, as mentioned in Section 3.3.1 the Gem Browser also allows one to do a textual search for gems with particular names, input types, return types, or number of arguments.

A significant problem related to Juxtaposability in that at any given time only a single gem design can be open. This makes working on multiple gems at the same time artificially difficult, as to open up another gem's design one has to save the current gem, then open the design of the other gem.

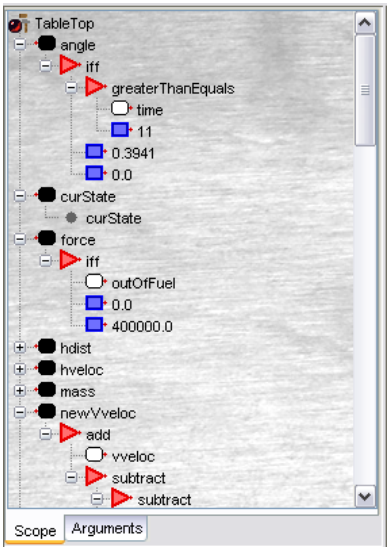


Figure 5.6: The Scope Window For The `rocket1()` Gem

Remedies, Workarounds, And Trade-offs

The obvious solution to the juxtaposibility problem would be to allow multiple gems to be open at once. Since the tabletop is a sub-window of the main Gem Cutter window, it would seem that there is no reason why this cannot take place as it would just mean allowing a second sub-window to be open. This would allow one to switch back and forth between multiple gem designs easily. However, given the fact that gem designs often take up a considerable amount of screen space, even if multiple gems could be open at the same time, it might be difficult or impossible to view them side-by-side. Short of being able to "zoom-out" on a gem design to make it take less space (which would make seeing the individual components more difficult) it would seem that there is no solution to this problem.

As it stands, the only workaround for users to the juxtaposibility problem is to take a screenshot of a gem's design from a separate program, and then refer back to the screenshot as needed[19]. A significant limitation of this is that it is not uncommon for gem designs to scroll off the screen, making taking a screenshot of an entire gem's design difficult or impossible.

[19]This is what was done when doing the viscosity "straw-test" of making the small change to the rocket trajectory program

Chapter 6

Conclusion And Future Work

These last two paragraphs do not claim to be convincing arguments. They should rather be described as "recitations tending to produce belief."

Alan Turing [86]

This thesis explored and examined a visual programming environment called the Gem Cutter from a pedagogical viewpoint for the purpose of being a starting-point for answering the question as to whether or not games combined with a visual programming environment can aid students learning to program.

A qualitative evaluation of the environment was performed using the Cognitive Dimensions framework, as well as a general discussion of the strengths and weaknesses of the environment. We showed how potential improvements to the environment could possibly be incorporated to improve the Gem Cutter for use in learning environments.

In addition, a framework developed in the environment for producing turn-based word games was produced along with some sample exercises for the purpose of helping instructors of introductory programming courses evaluate whether or not the Gem Cutter suits their needs. Furthermore, the Word Game Framework gives instructors a tool which allows for the creation of assignments for students which can be tailored to meet a broad range of learning objectives thereby showing that it is at least feature complete enough to be adequate for use in a learning environment. The combination of the use of games as a basis for an assignment means assignments which can be more engaging and interesting for students. We believe that this increased motivational power, combined with the removal of syntax errors due the visual nature of the Gem Cutter will help alleviate some of the hurdles that students face when learning to program.

6.1 Future Work

6.1.1 User Studies

Kelso outlined an experiment involving user studies for his VFPE in which he outlined two possible avenues of for empirical evaluation of the VFPE in [53]:

1. Evaluating the general adequacy of the environment
2. Investigating Visual Programming proper

The first is focused on conducting experiments to evaluate the general adequacy of the programming environment. Such experiments would focus on showing quantitatively that for programmers experienced with textual languages, that the visual environment is not significantly worse than the textual alternative. That is, that the environment is "feature-equivalent" to a given textual environment.

The second is for the purpose of exploring the textual/visual division. That is, experiments designed here would be for the purpose of identifying differences that are due solely to the respective modes of display and the programming environment tools. This requires a textual environment which is roughly equivalent to the VFPE. Given that the Gem Cutter truly is the visual representation of CAL code, it would seem that an experiment of this form with the Gem Cutter and CAL would be very telling of the differences between the visual and the textual representations of a program.

In addition to these, we can foresee other avenues for empirical evaluation of the Gem Cutter, most notably that of functional versus object-orientated programming. Many studies which have explored the differences between the two paradigms have largely been based in the textual world. Similar experiments using the Gem Cutter (a visual environment rooted in the functional paradigm) and one of the common object-orientated environments (such as Prograph, Alice, or Scratch) would be an interesting addition to the work exploring the difference between the two paradigms.

For our purposes as educators however, an open question is how effective the Gem Cutter can be as a learning tool for students. We have shown in the exercises making use of the Word Game Framework that the Gem Cutter can be used to design exercises for students new to programming, however two open questions are how effective these exercises are, and if the skills learned in doing the exercises in the Gem Cutter can be transferred to other (particularly textual) programming environments.

6.1.2 HCI Based Evaluation Strategies

The Cognitive Dimensions framework was chosen for this thesis for two primary reasons: (i) it is accessible to non-HCI specialists (ii) it explores issues that lie at the notational level, rather than the interface level . However, a rigorous HCI-style evaluation of the Gem Cutter would be of great use to identify improvements to the interface of the environment. Green suggests using the Cognitive Dimensions framework as a "broad-brush" overview of the environment, then following this up with a programming walkthrough, followed by a GOMS analysis. This thesis has done the first step of that approach, the programming walkthrough and GOMS analysis remain as future research.

6.1.3 Abstractions and Metaphors

In [87], Brown discusses how an inherent flaw in the Alice programming environment is that Alice has very visual abstractions and objects. For example, when the program involves making an ice skater do a flip, the user sees a visual representation of that ice skater on the screen. As such, he found that when students made the jump from Alice to Java they struggle with now having to make those abstractions work in their heads. The Gem Cutter has no such visual abstractions, while a gem is a visual metaphor for a function, it is still just as abstract as if one was writing it in a textual language. A useful question to answer would be to explore if Gem Cutter has the same drawback as Alice in regards to students becoming reliant upon visual "entities".

Appendix A

Exercises

In this section, we outline a series of exercises of increasing difficulty and complexity that make use of the Gem Cutter environment and the Word Game Framework. Instructors can use these exercises as-is, tailor them to their specific needs, or simply use them as a base to draw upon for their own learning environments.

A.1 Gem Cutter Basics

Learning Objectives

By the end of this lesson, participants will be able to:

- Construct and develop gems (functions) using the Gem Cutter environment, including those that make use of collector/emitter gems, value gems, and the result target gem.
- Execute and evaluate designed gems with various arguments

Task Type (Self-Directed or Guided)

Guided

Pre-Requisite Knowledge/Skills Needed

Participants must already know/be able to:

- Some basic high school mathematics (knowledge of functions)

Extra Environment Assumptions

- Access to overhead/whiteboard

Outline of Tasks

- Give outline of the gem metaphor, how "gem" means "function"
- Write on board some common mathematical functions and expressions they may have seen, such as: $cos(0) = 1$ and $f(x) = x + 3$, as well as $f(8)$, and its result.
- Draw an analogy between the mathematical world and the computer: how a computer can process information by applying functions to given input data
- Explain how in the Gem Cutter we do this by creating "gems" which are just friendly names for "functions"
- Show this by dragging the cos gem from the Math module to the tabletop, and then have them run the gem with a few values (in particular the value 0 to tie to the example written on the board)
- Now, walk through the process of creating the $f(x) = x + 3$ gem.
- Create a collector gem to "name" the x argument to the function. Stress this is one way we can provide names for the arguments to our gems.
- Create an emitter gem for the x within the function definition, and note to them how it has a single arrow pointing out of it to indicate it "outputs" or "emits" the value of x
- It might be desirable to draw a correlation between the symbol x appearing twice in the mathematical definition of $f()$ written on the board, and the fact there are two "x" gems on the tabletop (one to name a parameter, and one to represent its use within the function's definition).
- Add the "add" gem from the Prelude module to the tabletop. This gives one a chance to outline how related gems can be grouped together into collections called "modules", and how the "Prelude" module is one such collection.

- Note to them how the add gem has two inputs and a single output, or mathematically speaking it is a function with two arguments.
- Connect the emitter gem for x into one of the inputs of the add gem.
- Ask them what should go into the other input of the add gem. It is likely that they will say "3" or something similar to indicate that the other argument to the add gem should be the constant value 3
- Add a value gem to the tabletop, and explain how value gems are like emitters in that they produce or output a value, but they differ from emitter gems in that value gems will always produce the same value.
- Explain that to input a value into a value gem we must first indicate what kind of value to emit. This leads to a distinction between data types. One can motivate this by saying that for example numbers are different than words, and 3 is an example of a numeric value. Thus, we need to indicate that this value gem contains a numeric value. Then show how to specify a Double type for the gem (this choice is made for a reason - if we chose a type like "Int" they may later try to input non-whole numbers and become confused).
- Connect the value gem to the add gem.
- Then show how to "run" the gem by right-clicking on the add gem and picking "Run Gem". Note to them how the system will prompt them for a value for x, due to the fact it is a variable that can change. That is, it does not know what value to associate with x, so it prompts the user to supply one (just like the mathematical function, $f(x)$ is meaningless until we supply a value for x).
- Have them run the gem a few times with different values to see the results.
- Now note to them that we have not "named" this function. Explain how the specially coloured result target gem in the corner is used to indicate what the result of the function should be as well as where we name the function.
- Right click on the result target gem and pick "rename", then name the gem "f"
- Now connect the output of the add gem to result target gem
- Have them save the gem, with public access

- Now point out to them the GemCutterSaveModule module, and how there is now a gem called f there.
- Start a new gem layout, and have them drag their f gem onto the tabletop, and try running it.
- Emphasize how this now means we can create even more complex gems which make use of our f gem, just like we made use of the more primitive add gem while creating f.

A.2 Data Types And Basic Control Flow

Learning Objectives

By the end of this lesson, participants will be able to:

- Demonstrate an understanding of the different primitive data types in the Gem Cutter environment by constructing a few simple expressions.
- Develop hypotheses in regards to when gems can be connected together, then verify their claims by experimentation.
- Create a simple gem which makes use of branching control flow constructs.

Task Type (Self-Directed or Guided)

Guided

Pre-Requisite Knowledge/Skills Needed

Participants must already know/be able to:

- Have basic knowledge of the Gem Cutter environment (how to arrange gems on the tabletop, link gems together, find gems in the Gem Browser, etc)
- Understand what value gems are and how to use them

Data Type	Description	Examples
Int	Whole (non-decimal) numbers	3, 5, -2
Double	Fractional numbers	3.0, 2.1411, -3.14159
Char	A single character/letter	c, a, A, -
String	A series of characters	"Adam", "hello", "how are you?"
Boolean	A value which can be either true or false	true, false

Table A.1: Some primitive data types in CAL

Extra Environment Assumptions

- Access to overhead/whiteboard
- Optionally, one can previously create the partially completed gem from Part 1 to save some time.

Outline of Tasks

Part 1

- Begin with a discussion of some simple mathematical expressions. Write the expression $3 + 5 - 2$ on the board. Point out how this expression makes sense due to the fact that we associate certain operations or rules that we can apply to numbers (namely addition and subtraction). Then ask them for the answer to this expression (the number 6).
- Now change the middle operand to the string "hello". You should now have $3 +$ "$hello$" $- 2$ on the board. Now pose the question to them "Does this expression make sense and why?"
- Outline to them how things like words are different than numbers, yet both are data. This leads to a distinction of *types* of data and how different types of data support different operations, and in particular how some operations will only be valid on certain types of data (for example addition is only valid with numbers).
- Write on the board the table outlined in Table A.1

- Explain how in Gem Cutter gem connections are tied to specific types, and outputs of one gem can only be connected to the input of another if the types are *compatible*, that is only when the two are the same exact type, or an equivalent type as was the case with type variables (of course this is not exactly correct when type classes enter the picture, but is sufficient for our purposes).

- Now turn to the Gem Cutter and add two value gems to the tabletop. Point out that at this point the data type of these two gems are unknown so we must specify the type. Tell them that we want to put the sentance "They are the same!" into one, and the sentance "They are different!" into the other. Then pose the question "What data type should these two value gems be?" (the answer is String)

- Then proceed to set the types of the two gems to String, and enter the two sentances into the gems.

- Now have them drag the iff and equals gems from the Prelude module onto the tabletop. Point out that to find out the type of a connection on a gem (input or output) we can hover the mouse above it and a tooltip will display the type of that connection. Pose to them the question "What is the type of the topmost input to iff?" (the answer is Boolean)

- Now ask them "What is the type of the output of equals?" (the answer is Boolean)

- Now point out to them that since the two types are the same, we should be able to connect the output of the equals gem to the top input of the iff gem. Then do so.

- Now break the connection between the iff and equals gems. Then pose the question to them "Can we connect the first value gem to the top input of iff?" The correct answer is no, but have them guess if it is or is not possible. Then tell them to try it (it will not work since Strings and Booleans are not type compatible). As them to explain why the connection could not be made.

- Now ask what the type of the other inputs to iff are (they are `a`, meaning that they are *type variables*). Explain to them that a single lower-case letter as a data type is a *type variable* which means an arbitrary type. That is, the variable

a can be replaced with any specific data type, however all occurrences of a must be replaced with the same type.

- Now ask them if they can connect one of the value gems to the other inputs of iff. After they do so, ask what is the type of the other input to iff (it should now be string, since the type variable was bound to the String type).

Part 2

- Explain to them that we want to create a simple gem which displays the message "They are the same" if two given numbers are the same, and "they are different!" otherwise. Or more formally:

$$areSame\,(x, y) = \begin{cases} They\,are\,the\,same, & \text{if } x = y \\ They\,are\,different, & \text{if } x \neq y \end{cases}$$

- Ask them how would they construct the gem given all that is currently on the tabletop. If it seems like they will be able to do so, give them a couple minutes to do so, otherwise walk them through the procedure of linking the gems together.

- Have them save their gems with the name "areSame", and test it out with a few values.

A.3 Type Inference

Learning Objectives

By the end of this lesson, participants will be able to:

- demonstrate an understanding of data types in an environment which makes use of a type inference engine by answering a series of questions

- demonstrate a basic understanding of how type inference works by predicting the outcome of an action which involves the use of polymorphic types

- verify their understanding by experimentation

Task Type (Self-Directed or Guided)

Guided

Pre-Requisite Knowledge/Skills Needed

Participants must already know/be able to:

- Have basic knowledge of the Gem Cutter environment (how to arrange gems on the tabletop, link gems together, find gems in the Gem Browser, etc)
- Understand what value gems are and how to use them
- Know the difference between various primitive data types (specifically Int's, Double's, and String's)
- Understand what a list is, and the type notation used to denote them (ex - $[String]$ is a list of Strings)
- Understand what tuples are (or at least 2-tuples) and the type notation used to denote them (ex - $(Int, String)$ is a pair consisting of an Int and a String)

Extra Environment Assumptions

- Optionally one can pre-create the gem used in task 2 to save the time of having participants create the value gem which contains a list of Strings.

Outline of Tasks

Part 1

- Have participants place the Add gem from the Prelude module onto the tabletop
- Note to them how both inputs and the output share the same colour
- Demonstrate to them that if you hover the mouse over the inputs or the output that the type of that connection is displayed (in this case it is "$Num\ a => a$", which indicates this is a numeric type)
- Have them add a value gem which contains the Int value 1 to the tabletop. Ask them to note the colour of the output of the value gem.

- Connect this value gem to one of the inputs of the Add gem.
- Ask participants if they noticed any changes to the Add gem. They might have noticed:
  - the colour of the unconnected inputs and output of the Add gem changed to the colour of the value gem's output
  - that the type of the inputs and outputs are now "Int" instead of "$Numa => a$"
- At this point, one can give a series of questions or start a discussion how the Gem Cutter deduced the type of the other input and the output of the Add gem (type inference). If time is short, an explanation to the effect that "add takes two arguments of the same type, and since we fixed the type of one of the inputs to be Int, it *knows* that the other must also be of type Int" is sufficient for the remainder of this lesson.
- Have them add a value gem which contains the Double value 3.14159 to the tabletop. Again, have them note the colour of the output the value gem (it should be different than the colour of the previous value gem's output due to the different types).
- Ask them try to connect this value gem to the other input of the Add gem. They will not be able to (Int and Double are not the same type), so propose the question to them why they could not do this.
- You can additionally have them correct the problem by changing the Int value gem to become a Double and then have them run the Gem (producing the sum 4.14159).

Part 2

- Have participants place the zip gem from the List module onto the tabletop, and to take note of the types of inputs and outputs (two inputs, one of type $[a]$ and one of type $[b]$, and a single output of type $[(a, b)]$)
- Have participants create a value gem of type $[String]$, and to enter a few names into this gem. Alternatively one could previously create this gem and supply it to participants.

- Pose the question to participants: "If we connect the value gem to the first input of the zip gem, what will the type of the remaining inputs and outputs of the zip gem become?" The correct answer is that the other input will be unchanged (or changed from $[a]$ to $[b]$ which is equivalent), and the output will be changed to $[(String, a)]$ – a list of pairs containing strings and some other type)
- Have them try connecting the value gem to the zip gem, and see if they were right/wrong (that is, have them verify their hypothesis by experimentation)

A.4 Implementing Undo In Hangman

Learning Objectives

By the end of this lesson, participants will be able to:

- Demonstrate an understanding of conditional statements
- Decompose compound data structures (lists and tuples) into subsequent parts, and construct compound data structures from more atomic parts
- Construct a reasonably complicated gem which makes use of a variety of sub-gems

Task Type (Self-Directed or Guided)

Self-Directed

Pre-Requisite Knowledge/Skills Needed

Participants must already know/be able to:

- Have basic knowledge of the Gem Cutter environment (how to arrange gems on the tabletop, link gems together, find gems in the Gem Browser, etc)
- Know the difference between various primitive data types (specifically Int's, Char's, and String's)
- Understand what a list is, and how to work with them
- Understand what tuples are and how to create them using the tupleXX gems.

Extra Environment Assumptions

- Must have the Word Game Framework available within the Gem Cutter
- Must have the hangMan gem available within Gem Cutter under the GemCutterSaveModule module
- Must have the partially-completed hangmanUndo gem available within Gem Cutter under the GemCutterSaveModule module
- Must have the isCharInString gem available available within Gem Cutter under the GemCutterSaveModule module, or optionally can ask students to create it as part of the assignment (it is a simple application of String.stringToList and List.isElem)

Material To Supply To The Student

Your assignment is to complete the implementation of the "undo" feature of the Hangman game. As it currently stands, the command "undo" entered in the Hangman game will have no effect.

Your task is to try and modify the hangmanUndo gem to implement the undo functionality so that the undo command "undoes" the last guess made.

The hangmanUndo gem currently takes one single argument – the current state of the game. This is a tuple consisting of three parts: the word being guessed (a String), the number of guesses remaining (an Int), and a list of characters that are the previously made guesses in the order they were made (from most recent to least recent).

The hangmanUndo gem currently breaks the passed in "state" argument into the three separate parts for you (this is the field1, field2, and field3 gems).

There are three possible return values for the hangmanUndo gem, depending on the current state of the game:

Possibility #1 - no guesses have yet been made If a player has not yet made a guess, then the third field of the state will be an empty list. If this is the case, then we simply return the state of the game unchanged (there is nothing to undo).

Possibility #2 - the guess to undo is a letter that was in the word to be guessed If a player makes an correct guess in that he/she guesses a letter that is in the word being guessed, then we just need to return a new state with the word to be guessed, the same number of guesses, and the same list of guesses with the correct guess removed.

Possibility #3 - the guess to undo is not a letter that was in the word to be guessed If a player makes an incorrect guess in that he/she guesses a letter that is not in the word being guessed, then we have to remove the guessed letter as in the last case, but we also need to increase the number of remaining guesses by 1, due to the fact that when the player makes an incorrect guess, this number is decreased by 1 (this is how the player loses – when the number of remaining guesses becomes zero).

For possibilities 2 and 3, we need to break the list of guesses into two parts: the most recent guess (which is the one we need to undo), and the remaining previously made guesses (which we need to return). The most recent guess (that is, the one we are to undo) will be the first element (or head) of the list of guesses, and additionally the remaining guesses will be the remainder of the list of guesses. That is, the guess to undo is the head of the list, and the remaining guesses is the tail of the list.

Some hints:

- You will likely find the following gems useful:
  - The Nil, equals, add, tuple3, and iff gems in the Prelude module
  - The head, and tail gems in the List module
  - The isCharInString gem in the GemCutterSaveModule

A.5 Scoring Hangman

Learning Objectives

By the end of this lesson, participants will be able to:

- Decompose compound data structures (lists and tuples) into subsequent parts
- Perform a simple mathematical calculation using gems
- Construct a reasonably complicated gem which makes use of a variety of sub-gems

Task Type (Self-Directed or Guided)

Self-Directed

Pre-Requisite Knowledge/Skills Needed

Participants must already know/be able to:

- Have basic knowledge of the Gem Cutter environment (how to arrange gems on the tabletop, link gems together, find gems in the Gem Browser, etc)
- Know the difference between various primitive data types (specifically Int's, Char's, and String's)
- Understand what a list is
- Understand what tuples are and how to decompose them using the fieldXX gems.

Extra Environment Assumptions

- Must have the Word Game Framework available within the Gem Cutter
- Must have the hangMan gem available within Gem Cutter under the GemCutterSaveModule module

Material To Supply To The Student

As it currently stands you can play the Hangman game, but there is no mechanism for indicating whether a player has played well or poorly. That is, there is no "score" assigned to a player when they play a game. Your assignment is to create a gem which "scores" a given game of Hangman.

The output of the Hangman gem is a tuple consisting of three items: the word being guessed, the number of guesses remaining at the end of the game (which will always be 0), and the list of all characters that were guessed by the player.

We will calculate a score of Hangman by giving the player 20 points for each correctly guessed letter, and subtract 10 points for each incorrect guess. Or put another way:

$$score = numberOfCorrectGuesses \cdot 20 - numberOfIncorrectGuesses \cdot 10 \quad \text{(A.1)}$$

The question then becomes how does one determine the number of correct and incorrect guesses. To keep things simple, we will assume that the player always correctly guesses the word (that is, that they did not run out of guesses). Thus, one simple (but not entirely accurate) way would be to determine the number of correct guesses to be the length of the word being guessed, and the number of incorrect guesses to be equal to the length of the list of guesses. This is slightly inaccurate in that correct guesses will also be given a penalty of 10 points, but is good enough for our purposes.

Thus, your task is to create a gem called `scoreHangman` which accepts the output tuple of the Hangman gem and calculates a score based upon the output of the Hangman gem. That is, you should be able to use your gem in the manner displayed in Figure A.1

Figure A.1: Using the hangManScore Gem

The result returned by your `scoreHangman` gem should be an Int which is the final score as calculated.

Some hints:

- You will likely find the following gems useful:
  - The field1, field2, field3, subtract, and multiply gems in the Prelude module
  - The length gem in the List module
  - The length gem in the String module

A.6 Scoring Hangman Accurately With Higher Order Functions

Learning Objectives

By the end of this lesson, participants will be able to:

- Decompose compound data structures (lists and tuples) into subsequent parts
- Perform a simple mathematical calculation using gems
- Perform relatively sophisticated operations relating lists and strings
- Construct a reasonably complicated gem which makes use of a variety of sub-gems

Task Type (Self-Directed or Guided)

Self-Directed

Pre-Requisite Knowledge/Skills Needed

Participants must already know/be able to:

- Have basic knowledge of the Gem Cutter environment (how to arrange gems on the tabletop, link gems together, find gems in the Gem Browser, etc)
- Know the difference between various primitive data types (specifically Int's, Char's, and String's)
- Understand what a list is and how to decompose them
- Understand what tuples are and how to decompose them using the fieldXX gems.
- Understand how to make use of higher order functions (specifically creating a predicate for the `filter` function)
- Have attempted the previous `hangManScore` gem seen in Section A.5

Extra Environment Assumptions

- Must have the Word Game Framework available within the Gem Cutter
- Must have the hangMan gem available within Gem Cutter under the GemCutterSaveModule module
- Must have the isCharInString gem available available within Gem Cutter under the GemCutterSaveModule module, or optionally can ask students to create it as part of the assignment (it is a simple application of String.stringToList and List.isElem)

Material To Supply To The Student

As seen in the previous exercise we created a rather simplistic `scoreHangMan` gem. This gave a score which was not particularly accurate because:

- It assumed the player had won, when in fact he/she may not have
- It inaccurately calculated the number of correct guesses as being the length of the word, which would artificially inflate the score when a letter appears more than once in the word (they would recieve points for each occurrence of the letter, even though only all occurrences would be associated with a single guess)
- It penalized correct guesses.

For this task we will create a gem called `scoreHangmanAccurate` which more accurately assesses the number of correct and incorrect guesses, and uses these numbers to calculate the final score. Again, as in the last case, the score will be calculated as:

$$score = numberOfCorrectGuesses \cdot 20 - numberOfIncorrectGuesses \cdot 10 \quad \text{(A.2)}$$

The difference is that the "numberOfCorrectGuesses" and "numberOfIncorrectGuesses" will be calculated accurately. You are free to do this however you wish, however a hint would be to use the `filter` gem in the List module to determine how many correct guesses were made. Once you have the number of correct guesses, the number

of incorrect guesses would simply be the total number of guesses minus the number of correct guesses.

Thus, your task is to create a gem called `scoreHangmanAccurate` which accepts the output tuple of the Hangman gem and calculates an accurate score based upon the output of the Hangman gem. That is, you should be able to use your gem in the manner displayed in Figure A.2

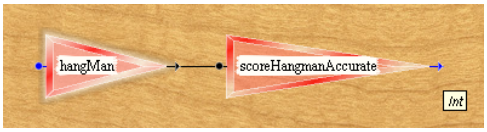


Figure A.2: Using the hangManScoreAccurate Gem

The result returned by your `scoreHangmanAccurate` gem should be an Int which is the final score as calculated.

Some hints:

- You will likely find the following gems useful:
  - The field1, field2, field3, subtract, and multiply gems in the Prelude module
  - The isCharInString gem in the GemCutterSaveModule
  - The length and filter gems in the List module

Appendix B

Word Game Framework Implementation Details

B.1 Helper Routines

In creating the *game*() routine of the Word Game Framework, we also devised a set of routines to handle the various I/O tasks. Each is outlined in detail in the subsequent sections.

B.1.1 session

session() is a routine which encapsulates the basic gameplay session. That is, it tests if the game has ended using the supplied gameOverTest predicate, and if so returns the final string displayed to the player and the final game state as a tuple. More formally session is defined as:

```
session::
       stateChangingFunction ::  a -> String -> (String, a)
       stateToStringFunction ::  a -> String
       initialState ::  a
       userPrompt ::  String
       gameOverTest ::  (String, a) -> Boolean
       returns (String, a)
```

Each parameter corresponds to one of the arguments passed to the main *game*() routine. A screenshot of the implementation of *session*() is shown in Figure B.1. Note that this function also has a local function called *endGameTest*() defined within it. This function is simply the logical or of the two possible ways of a game ending: either by the gameOverTest predicate returning true, or by the player entering the "quit" command.

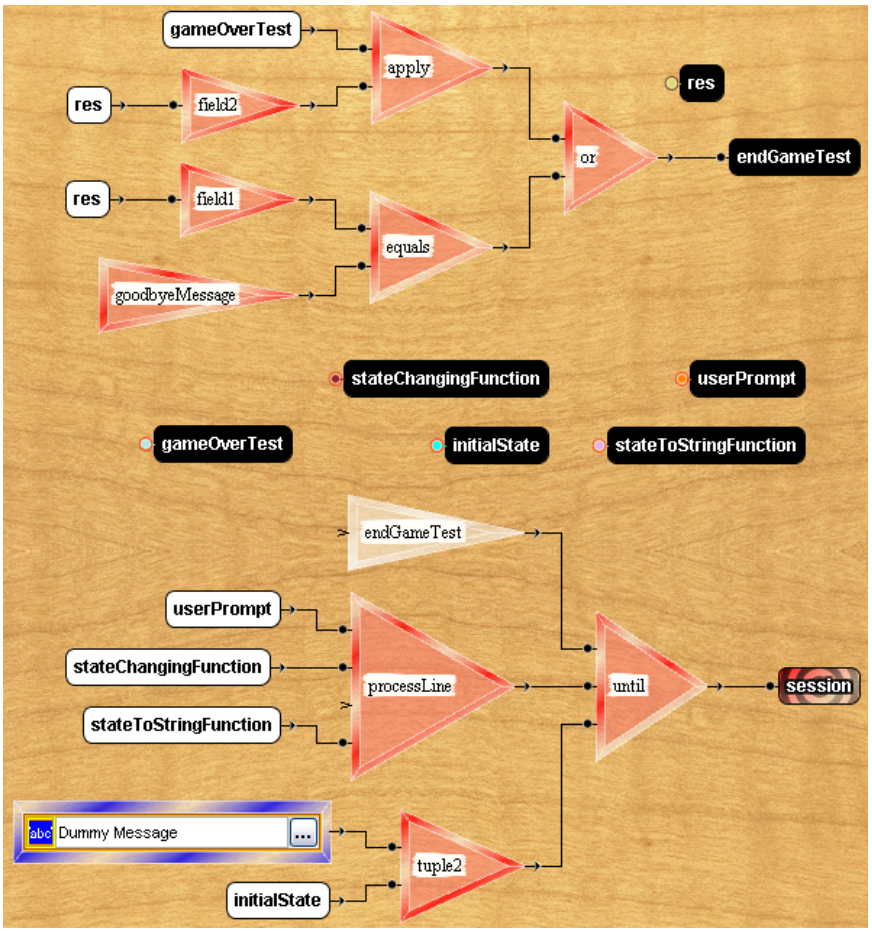


Figure B.1: Implementation of the session() gem

B.1.2 processLine

processLine() is a routine which encapsulates a single "move" of the game. That is, it represents one single application of the state transition function, and returns a tuple consisting of a message to display to the player and the new game state. More formally, processLine is defined as:

```
processLine::
            stateChangingFunction ::  a -> String -> (String, a)
            stateToStringFunction ::  a -> String
            previousState ::  (String, a)
            userPrompt ::  String
            returns (String, a)
```

Where `previousState` is a tuple consisting of a String[1] and the state which is transitioned from, and all other parameters correspond to those passed to the main *game*() routine. A screenshot of the implementation of *processLine*() is shown in Figure B.2. Note that this function also has a local function called *printMsg*() defined within it which is simply for the side effect of printing the resulting string message to STDOUT.

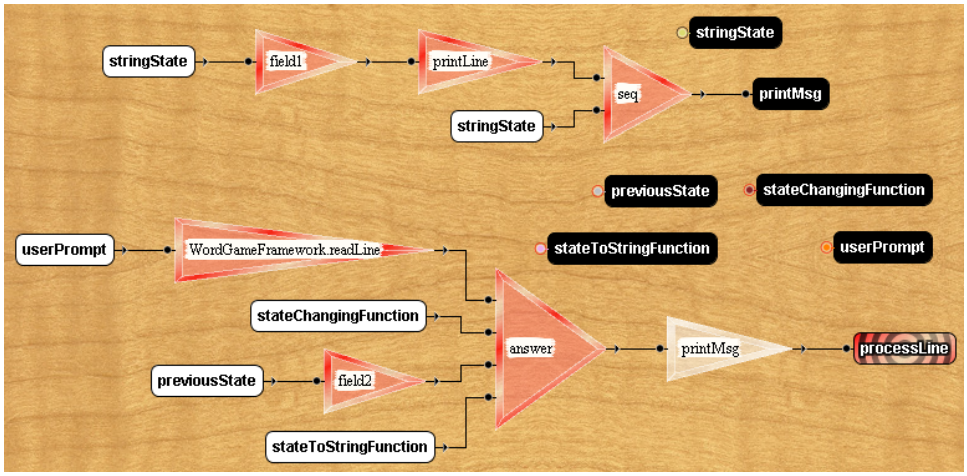


Figure B.2: Implementation of the processLine() gem

B.1.3 answer

answer() is a routine which applies the state transition function to the current state and current input from the player. Additionally, answer determines if the user has prematurely quit the game by entering the special "quit" command.[2] Like *processLine*(),

[1]Which is ignored, and only appears to be type compatible with the return type of `session`

[2]The "quit" command is the only specific hard-coded command in the Word Game Framework, all others are passed to the state transition function.

it returns a tuple consisting of a message to display to the player and the new game state. More formally, answer is defined as:

```
answer::
      stateChangingFunction ::  a -> String -> (String, a)
      stateToStringFunction ::  a -> String
      previousState ::  a
      userResponse ::  String
      returns (String, a)
```

Where `previousState` is the state which is transitioned from, `userResponse` is the response read from the player, and all other parameters correspond to those passed to the main *game*() routine. A screenshot of the implementation of *answer*() is shown in Figure B.3. The `joinStrings` code gem consists of the single line of CAL code:

```
"\n\n" ++ s ++ "\n\n" ++ stateString ++ "\n"
```

which simply serves the purpose of concatenating the two output strings together with a few newlines between them.

B.1.4 goodbyeMessage

goodbyeMessage() is a routine which simply returns the string to be displayed to the player when the game is over. It is used to determine if and when the player prematurely ended the game by entering the "quit" command. More formally, goodbyeMessage is defined as:

```
goodbyeMessage::
              returns String
```

A screenshot of the implementation of *goodbyeMessage*() is shown in Figure B.4.

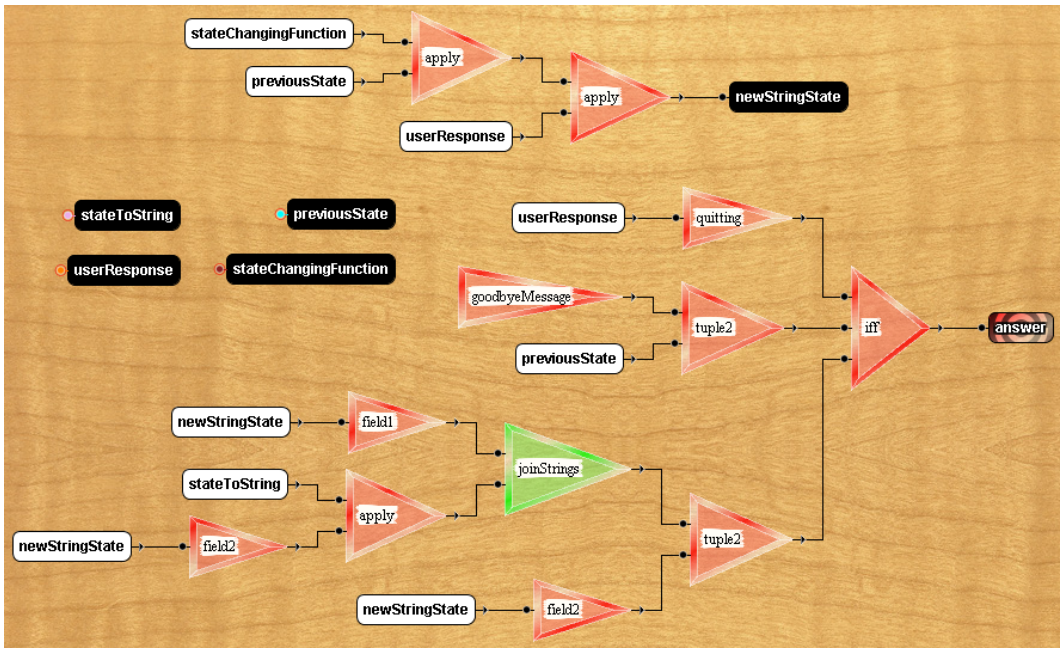


Figure B.3: Implementation of the answer() gem

Figure B.4: Implementation of the goodbyeMessage() gem

B.1.5 readLine

readLine() prints a supplied prompt message to STDOUT, and then reads a line of text (using newline as a delimiter) from STDIN and returns it as a String. More formally, readLine is defined as:

```
readLine::
        prompt ::  String
        returns String
```

A screenshot of the implementation of *readLine*() is shown in Figure B.5, which makes use of the Cal.Console.IO routines for reading from STDIN and printing to STDOUT.

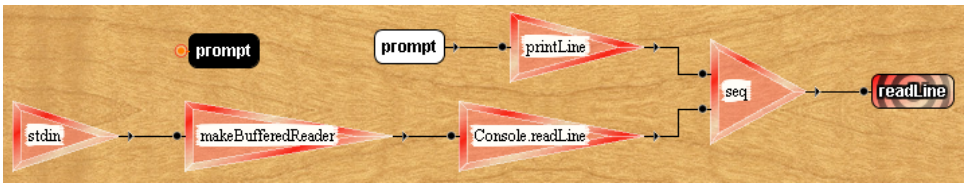


Figure B.5: Implementation of the readLine() gem

B.1.6 quitting

quitting is a predicate function which returns true if the supplied string is equal to "quit", which is the special command a player can issue to terminate a game prematurely. More formally, quitting is defined as:

```
quitting::
        s ::  String
        returns Boolean
```

A screenshot of the implementation of *quitting*() is shown in Figure B.6.

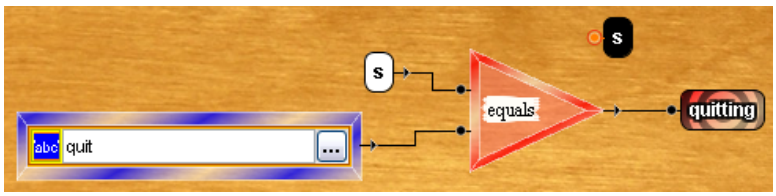


Figure B.6: Implementation of the quitting() gem

Bibliography

[1] C. The ACM Java Task Force, "The acm java task force - project rationale." `http://jtf.acm.org/rationale/rationale.pdf`, February 2004.

[2] "Haskell wiki: Introduction." `http://www.haskell.org/haskellwiki/Introduction`.

[3] T. R. G. Green and M. Petre, "Usability analysis of visual programming environments: A 'cognitive dimensions' framework," *Journal of Visual Languages and Computing*, vol. 7, pp. 131–174, June 1996.

[4] D. Church, "Decision support for managing security complexity in software development," Master's thesis, University Of Victoria, 2006.

[5] T. Jenkins, "On the difficulty of learning to program," in *3rd annual Conference of LTSN-ICS*, 2002.

[6] T. Beaubouef and J. Mason, "Why the high attrition rate for computer science students: Some thoughts and observations," *SIGCSE Bull.*, vol. 37, no. 2, pp. 103–106, 2005.

[7] C. D. Hundhausen, S. F. Farley, and J. L. Brown, "Can direct manipulation lower the barriers to computer programming and promote transfer of training?: An experimental study," *ACM Trans. Comput.-Hum. Interact.*, vol. 16, no. 3, pp. 1–40, 2009.

[8] C. Kelleher and R. Pausch, "Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers," *ACM Comput. Surv.*, vol. 37, no. 2, pp. 83–137, 2005.

[9] B. Manaris, "Dropping cs enrollments: Or the emperor's new clothes?," *SIGCSE Bull.*, vol. 39, no. 4, pp. 6–10, 2007.

[10] J. Vesgo, "Continued drop in cs bachelor's degree production and enrollments as the number of new majors stabalizes," *Computing Research News*, vol. 19, no. 2, 2007.

[11] B. Ward, "Computer science enrollments drop," *IEEE Computer*, vol. 41, no. 4, pp. 87–89, 2008.

[12] J. D. Bayliss, "Using games in introductory courses: Tips from the trenches," in *SIGCSE '09: Proceedings of the 40th ACM technical symposium on Computer science education*, (New York, NY, USA), pp. 337–341, ACM, 2009.

[13] "Alice - an educational software that teaches students computer programming in a 3d environment." `http://www.alice.org`.

[14] "Scratch." `http://scratch.mit.edu/`.

[15] Q. H. Mahmoud, W. Dobosiewicz, and D. Swayne, "Redesigning introductory computer programming with html, javascript, and java," in *SIGCSE '04: Proceedings of the 35th SIGCSE technical symposium on Computer science education*, (New York, NY, USA), pp. 120–124, ACM, 2004.

[16] A. Benander, B. Benander, and J. Sang, "Factors related to the difficulty of learning to program in Java — an empirical study of non-novice programmers," *Information and Software Technology*, vol. 46, no. 2, pp. 99–107, 2004.

[17] S. Bloch, "Teach scheme, reach java: Introducing object-oriented programmming without drowning in syntax," *J. Comput. Small Coll.*, vol. 23, no. 3, pp. 119–119, 2008.

[18] "The teachscheme/reachjava project." `http://www.teach-scheme.org/`.

[19] M. Felleisen, R. B. Findler, M. Flatt, and S. Krishnamurthi, "The teachscheme! project: Computing and programming for every student," *Computer Science Education*, vol. 14, no. 1, pp. 55–77, 2004.

[20] R. Berghammer and F. Huch, "From functional to object-oriented programming: A smooth transition for beginners," in *FDPE '05: Proceedings of the 2005 workshop on Functional and declarative programming in education*, (New York, NY, USA), pp. 3–8, ACM, 2005.

[21] L. Carter, "Why students with an apparent aptitude for computer science don't choose to major in computer science," in *SIGCSE '06: Proceedings of the 37th SIGCSE technical symposium on Computer science education*, (New York, NY, USA), pp. 27–31, ACM, 2006.

[22] T. Barnes, E. Powell, A. Chaffin, and H. Lipford, "Game2learn: Improving the motivation of cs1 students," in *GDCSE '08: Proceedings of the 3rd international conference on Game development in computer science education*, (New York, NY, USA), pp. 1–5, ACM, 2008.

[23] Y. Rankin, A. Gooch, and B. Gooch, "The impact of game design on students' interest in cs," in *GDCSE '08: Proceedings of the 3rd international conference on Game development in computer science education*, (New York, NY, USA), pp. 31–35, ACM, 2008.

[24] S. A. Curtis, "Word puzzles in haskell: Interactive games for functional programming exercises," in *FDPE '05: Proceedings of the 2005 Workshop on Functional and Declarative Programming in Education*, (New York, NY, USA), pp. 15–18, ACM, 2005.

[25] M. Overmars, "Learning object-oriented design by creating games," *Potentials, IEEE*, vol. 23, pp. 11–13, 2004-Jan. 2005.

[26] E. Sweedyk, M. deLaet, M. C. Slattery, and J. Kuffner, "Computer games and cs education: Why and how," in *SIGCSE '05: Proceedings of the 36th SIGCSE technical symposium on Computer science education*, (New York, NY, USA), pp. 256–257, ACM, 2005.

[27] E. W. Dijkstra, "On the cruelty of really teaching computing science," *CACM: Communications of the ACM*, vol. 32, 1989.

[28] A. Pears, S. Seidman, L. Malmi, L. Mannila, E. Adams, J. Bennedsen, M. Devlin, and J. Paterson, "A survey of literature on the teaching of introductory programming," in *ITiCSE-WGR '07: Working group reports on ITiCSE on Innovation and technology in computer science education*, (New York, NY, USA), pp. 204–223, ACM, 2007.

[29] "Slashdot - philip greenspun answers." `http://slashdot.org/interviews/00/04/27/107235.shtml`, April 2000.

[30] C. The Joint Task Force on Computing Curricula, "Computing curricula 2001," *J. Educ. Resour. Comput.*, p. 1, 2001.

[31] "Computer science curriculum 2008: An interim revision of cs 2001." `http://www.acm.org//education/curricula/ComputerScience2008.pdf`.

[32] B. S. Bloom, "Innocence in education," *School Review*, 1972. SO: School Review; 80, 3, 333-52, May 72.

[33] S. Cooper, W. Dann, and R. Pausch, "Teaching objects-first in introductory computer science," in *SIGCSE '03: Proceedings of the 34th SIGCSE technical symposium on Computer science education*, (New York, NY, USA), pp. 191–195, ACM, 2003.

[34] O. Astrachan, K. Bruce, E. Koffman, M. Kölling, and S. Reges, "Resolved: Objects early has failed," in *SIGCSE '05: Proceedings of the 36th SIGCSE technical symposium on Computer science education*, (New York, NY, USA), pp. 451–452, ACM, 2005.

[35] C. Hu, "Rethinking of teaching objects-first," *Education and Information Technologies*, vol. 9, no. 3, pp. 209–218, 2004.

[36] R. Lister, A. Berglund, T. Clear, J. Bergin, K. Garvin-Doxas, B. Hanks, L. Hitchner, A. Luxton-Reilly, K. Sanders, C. Schulte, and J. L. Whalley, "Research perspectives on the objects-early debate," in *ITiCSE-WGR '06: Working group reports on ITiCSE on Innovation and technology in computer science education*, (New York, NY, USA), pp. 146–165, ACM, 2006.

[37] G. J. Sussman, J. Sussman, and H. Abelson, *Structure and Interpretation of Computer Programs*. MIT Press, 2nd ed., 1996.

[38] M. Felleisen, R. B. Findler, M. Flatt, and S. Krishnamurthi, "The structure and interpretation of the computer science curriculum," *J. Funct. Program.*, vol. 14, no. 4, pp. 365–378, 2004.

[39] P. Wadler, "A critique of abelson and sussman or why calculating is better than scheming," *SIGPLAN Not.*, vol. 22, no. 3, pp. 83–94, 1987.

[40] L. Böszörményi, "Why java is not my favorite first-course language," *Software - Concepts and Tools*, vol. 19, no. 3, pp. 141–145, 1998.

[41] M. M. T. Chakravarty and G. Keller, "The risks and benefits of teaching purely functional programming in first year," *J. Funct. Program*, vol. 14, no. 1, pp. 113–123, 2004.

[42] L. Wittgenstein, "Tractatus logico-philosophicus," *London: Routledge, 1981*, 1922.

[43] K. B. Bruce, "Controversy on how to teach cs 1: A discussion on the sigcse-members mailing list," *SIGCSE Bull.*, vol. 37, no. 2, pp. 111–117, 2005.

[44] E. Roberts, K. Bruce, R. Cutler, I. James H. Cross, S. Grissom, K. Klee, S. Rodger, F. Trees, I. Utting, and F. Yellin, "The acm java task force: Status report," in *SIGCSE '05: Proceedings of the 36th SIGCSE technical symposium on Computer science education*, (New York, NY, USA), pp. 46–47, ACM, 2005.

[45] "Python programming language." `http://python.org/`.

[46] W. S. Daher, "Eecs revamps course structure." `http://tech.mit.edu/V125/N65/coursevi.html`, February 2006.

[47] A. Radenski, "Python first: A lab-based digital introduction to computer science," in *ITICSE '06: Proceedings of the 11th annual SIGCSE conference on Innovation and technology in computer science education*, ACM, 2006.

[48] C. Shannon, "Another breadth-first approach to csi using python," *SIGCSE Bull.*, vol. 35, no. 1, pp. 248–251, 2003.

[49] K. K. Agarwal and A. Agarwal, "Python for cs1, cs2 and beyond," *J. Comput. Small Coll.*, vol. 20, no. 4, pp. 262–270, 2005.

[50] K. K. Agarwal, A. Agarwal, and M. E. Celebi, "Python puts a squeeze on java for cs0 and beyond," *J. Comput. Small Coll.*, vol. 23, no. 6, pp. 49–57, 2008.

[51] R. Pausch, "Alice: A dying man's passion," in *SIGCSE '08: Proceedings of the 39th SIGCSE technical symposium on Computer science education*, (New York, NY, USA), pp. 1–1, ACM, 2008.

[52] M. M. Burnett, M. J. Baker, C. Bohus, P. Carlson, P. J. V. Zee, and S. Yang, "The scaling-up problem for visual programming languages," tech. rep., Corvallis, OR, USA, 1994.

[53] J. Kelso, *A Visual Programming Environment for Functional Languages*. PhD thesis, Department of Engineering, Murdoch University, 2002.

[54] "Ni labview." `http://www.ni.com/labview/`.

[55] M. Guzdial and E. Soloway, "Teaching the nintendo generation to program," *Commun. ACM*, vol. 45, no. 4, pp. 17–21, 2002.

[56] D. C. Cliburn, "The effectiveness of games as assignments in an introductory programming course," in *Frontiers in Education Conference, 36th Annual*, pp. 6–10, Oct. 2006.

[57] M. Overmars, "Game design in education," Tech. Rep. UU-CS-2004-056, Department of Information and Computing Sciences, Utrecht University, 2004.

[58] S. A. Wallace and A. Nierman, "Addressing the need for a java based game curriculum," *J. Comput. Small Coll.*, vol. 22, no. 2, pp. 20–26, 2006.

[59] S. Leutenegger and J. Edgington, "A games first approach to teaching introductory programming," *SIGCSE Bull.*, vol. 39, no. 1, pp. 115–118, 2007.

[60] J. Murray, I. Bogost, M. Mateas, and M. Nitsche, "Game design education: Integrating computation and culture," *Computer*, vol. 39, no. 6, pp. 43–51, 2006.

[61] M. Zyda, "Guest editor's introduction: Educating the next generation of game developers," *Computer*, vol. 39, no. 6, pp. 30–34, 2006.

[62] D. C. Cliburn and S. Miller, "Games, stories, or something more traditional: The types of assignments college students prefer," in *SIGCSE '08: Proceedings of the 39th SIGCSE technical symposium on Computer science education*, (New York, NY, USA), pp. 138–142, ACM, 2008.

[63] B. J. DiSalvo and A. Bruckman, "Questioning video games' influence on cs interest," in *FDG '09: Proceedings of the 4th International Conference on Foundations of Digital Games*, (New York, NY, USA), pp. 272–278, ACM, 2009.

[64] D. Gürer and T. Camp, "An acm-w literature review on women in computing," *SIGCSE Bull.*, vol. 34, no. 2, pp. 121–127, 2002.

[65] G. Carmichael, "Girls, computer science, and games," *SIGCSE Bull.*, vol. 40, no. 4, pp. 107–110, 2008.

[66] C. Kelleher, *Motivating Programming: Using Storytelling To Make Computer Programming Attractive To Middle School Girls*. PhD thesis, School of Computer Science, Carnegie Mellon University, 2006.

[67] L. Evans, B. Ilic, and E. Lam, "The gem cutter: A graphical tool for creating functions in the strongly-typed lazy functional language CAL." `http://openquark.org/Documents/gemcutter-techpaper.pdf`, 2007.

[68] L. Evans and N. Corkum, "Business objects gem cutter manual." `http://openquark.org/Documents/gem_cutter_manual.pdf`, 2006.

[69] "About open quark." `http://openquark.org/Open_Quark/About.html`.

[70] "The scala programming language." `http://www.scala-lang.org/`.

[71] L. Evans, E. Lam, R. Cameron, M. Byne, and J. Wong, "Java meets quark." `http://openquark.org/Documents/java_meets_quark.pdf`, 2007.

[72] B. Ilic, "Cal for haskell programmers." `http://openquark.org/Documents/cal_for_haskell_programmers.pdf`, 2007.

[73] T. Gray, "Elegy written in a country churchyard," 1751.

[74] A. Parkin, "Seng 330 openquark prelab." `http://webhome.csc.uvic.ca/~aparkin/quark/s330prelab.pdf`, 2008.

[75] B. Objects, "The open quark framework for java and the cal language homepage." `http://labs.businessobjects.com/cal/`.

[76] J. P. Carse, *Finite and Infinite Games — A Vision of Life as Play and Possibility*. Ballantine Books, 1987.

[77] A. T. Schreiner and J. E. Heliotis, "Sudoku: A little lesson in oop," *SIGCSE Bull.*, vol. 40, no. 2, pp. 44–47, 2008.

[78] B. Alliet, "Sudoku solver." `http://darcs.brianweb.net/sudoku/Sudoku.pdf`, 2005.

[79] D. Adams, *So Long And Thanks For All The Fish*. Pan Macmillan UK, 1984.

[80] T. R. Green and A. F. Blackwell, "A tutorial on cognitive dimensions." `http://www.cl.cam.ac.uk/users/afb21/publications/CDtutSep98.pdf`, 1998.

[81] P. Reisner, "Apt: A description of user interface inconsistency," *International Journal of Man-Machine Studies*, vol. 39, no. 2, pp. 215–236, 1993.

[82] B. Curtis, S. B. Sheppard, E. Kruesi-Bailey, J. W. Bailey, and D. A. Boehm-Davis, "Experimental evaluation of software documentation formats," *Journal of Systems and Software*, vol. 9, no. 2, pp. 167–207, 1989.

[83] E. Giguere, "Reading c declarations: A guide for the mystified." `http://www.ericgiguere.com/articles/reading-c-declarations.html`, 1987.

[84] S. Wiedenbeck, "The initial stage of program comprehension," *Int. J. Man-Mach. Stud.*, vol. 35, no. 4, pp. 517–540, 1991.

[85] T. R. G. Green, "Instructions and descriptions: Some cognitive aspects of programming and similar activities," in *AVI '00: Proceedings of the working conference on Advanced visual interfaces*, (New York, NY, USA), pp. 21–28, ACM, 2000.

[86] A. M. Turing, "Computing machinery and intelligence," *Mind*, vol. 59, pp. 433–60, October 1950.

[87] P. H. Brown, "Some field experience with alice," *J. Comput. Small Coll.*, vol. 24, no. 2, pp. 213–219, 2008.