A Programming System for Children that is Designed for Usability

John F. Pane

CMU-CS-02-127
May 3, 2002

School of Computer Science
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Also appears as: CMU-HCII-02-101

This research was sponsored in part by the National Science Foundation under Grant No. IRI-9900452. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect those of the National Science Foundation.

Keywords: Natural Programming, HANDS, End-User Programming, Psychology of Programming, Empirical Studies of Programmers, Educational Software, Children, User Interface Design, Programming Environments, Programming Language Design, Usability, Human-Computer Interaction.

Abstract

A programming system is the user interface between the programmer and the computer. Programming is a notoriously difficult activity, and some of this difficulty can be attributed to the user interface as opposed to other factors. Historically, the designs of programming languages and tools have not emphasized usability.

This thesis describes a new process for designing programming systems where HCI knowledge, principles and methods play an important role in all design decisions. The process began with an exhaustive review of three decades of research and observations about the difficulties encountered by beginner programmers. This material was catalogued and organized for this project as well as for the benefit of other future language designers. Where questions remained unanswered, new studies were designed and conducted, to examine how beginners naturally think about and express problem solutions. These studies revealed ways that current popular programming languages fail to support the natural abilities of beginners.

All of this information was then used to design HANDS, a new programming system for children. HANDS is an event-based system featuring a concrete model for computation based on concepts that are familiar to non-programmers. HANDS provides queries and aggregate operations to match the way non-programmers express problem solutions, and includes domain-specific features to facilitate the creation of interactive animations and simulations. In user tests, children using the HANDS system performed significantly better than children using a version of the system that lacked several of these features. This is evidence that the process described here had a positive impact on the design of HANDS, and could have a similar impact on other new programming language designs.

The contributions of this thesis include a survey of the knowledge about beginner programmers that is organized for programming system designers, empirical evidence about how non-programmers express problem solutions, the HANDS programming system for children, a new model of computation that is concrete and based on familiar concepts, an evaluation of the effectiveness of key features of HANDS, and a case study of a new user-centered design process for creating programming systems.

Acknowledgements

I would like to extend my heartfelt appreciation to Brad Myers for his insight and guidance throughout my Ph.D. work. I am very grateful for the many hours that he spent discussing and critiquing my work.

I am also very thankful for the feedback and support of my co-advisor, David Garlan, and the other members of my committee: Jim Morris, Clayton Lewis and Albert Corbett. Albert was especially generous in the time he spent helping me to design the user studies and analyze the results.

Many other faculty at CMU and elsewhere gave me valuable feedback and suggestions along the way. In particular, I would like to thank Bonnie John, Ken Koedinger, Wayne Gray, Margaret Burnett, Alan Blackwell, and Thomas Green.

I am especially grateful to Bonnie John, Dana Scott and Phil Miller, who were influential in my decision to become a Ph.D. student, and who helped me gain admission to the program.

I would like to thank the undergraduate and master's students who helped me develop the ideas in HANDS and who worked on the user studies: Leah Miller, Chotirat "Ann" Ratanamahatana, John Chang, Gabe Brisson, Luis Cota, and Ruben Carbonnell. Thanks to Joonhwan Lee for creating the graphics in the HANDS system. Thanks to Rob Miller for contributing his code for multi-level undo in the text editor.

Many thanks to Bernita Myers for acting as liaison to the East Hills Elementary school. Thanks to Mr. Niklos, the principal, as well as the teachers who allowed us to work in their classrooms: Carol Beavers and John Meighan. Also, thanks to Laurie Heinreicher at Winchester-Thurston school. Thanks to Michael Pane for his assistance in pilot testing the user study evaluating HANDS, and Melody Mostow for doing this and also starring in the HANDS video. Thanks to Ryan and Reid Myers, who helped us recruit volunteers for one of the studies. And, special thanks to Ryan for his insightful comparison of HANDS with Stagecast. And of course, thanks to all of the participants in my studies.

Thanks to Gary Perlman for working with me to develop and evaluate the search interface for the HCI Bibliography.

Thanks to many additional friends and fellow students who have helped me in various ways, especially Neil Heffernan, Chuck Rosenberg, Laurie Hiyakumoto, Herb Derby, Eugene Ng, Adam Berger, Matt Zekauskas, Maria Ebling, Chris Long, and David Eckhardt. A special thanks to Drew Morgan whose friendship and counsel was essential to my ability to make it through this project.

Especially, I would like to express my gratitude to my family for their support and encouragement. Most of all, thanks to my wife Barbara, who has given me her patient loving support throughout. Without this I may have never made it. I hope the rest of our lives will give me sufficient opportunity to reciprocate. Finally, thanks to Lorenzo, our next big project and one of the compelling motivations to finish this one.

Contents

CHAPTER 1

Introduction

Only a very small proportion of users can program their computers. However, most could benefit in some way from this powerful capability, whether to customize and interconnect their existing applications or to create new ones. As with writing, “the significance of programming derives not only from the carefully crafted works of a few professionals, but also from the casual jottings of ordinary people” [diSessa 1986, p. 859]. For ordinary people, understandability, familiarity, ease of performing small tasks, and user interface are more important features in a programming system than technical objectives such as mathematical elegance, efficiency, verifiability, or uniformity.

Many of the people who try to learn to program are quickly discouraged because it is very difficult. In fact, it is even challenging for more experienced people who have received formal training. Why is programming so difficult? Part of the problem is that it requires problem solving skills and great precision, but this does not fully explain the difficulty. Even when a person can envision a viable detailed solution to a programming problem, it is often very hard to express the solution correctly in the form required by the computer. This is a user-interface problem that has long been recognized but neglected.

1.1 Historical Context

In 1971, Gerald Weinberg published *The Psychology of Computer Programming*, with the stated goal to trigger a new field that studies computer programming as a human activity [Weinberg 1971]. At the time, there was little scientific literature about the human aspect of programming, and most of it appeared in technical reports and other obscure publications. The field began to grow quickly after Allen Newell addressed the third ACM *CHI Conference on Human Factors in Computing Systems*, and later published his comments in an article with Stuart Card:

> Millions for compilers, but hardly a penny for understanding human programming language use. Now, programming languages are obviously symmetrical, the computer on one side, the programmer on the other. In an appropriate science of computer languages, one would expect that half the effort would be on the computer side, understanding how to translate the languages into executable form, and half on the human side, understanding how to design languages that are easy or productive to use. Yet we do not even have an enumeration of all of the psychological functions programming languages serve for the user. Of course, there is lots of programming language *design*, but it comes from computer scientists. And though technical papers on languages contain many appeals to ease of use and learning, they patently contain almost no psychological evidence nor any appeal to psychological science. [Newell 1985, p. 212]

Soon two workshop series were started, which have become focal points for research in the usability of programming languages: the *Psychology of Programming Interest Group* (PPIG) explores the cognitive aspects of computer programming; and the *Empirical Studies of Programmers* (ESP) group focuses on empirical studies of beginners and experts.

Over the past three decades, many researchers have worked to understand the cognitive demands of programming and the sources of difficulty in existing programming languages and tools. In addition to the proceedings of the PPIG and ESP workshop series, relevant work has appeared in the *International Journal of Human-Computer Studies* (formerly *International Journal of Man-Machine Studies*), the proceedings of the ACM *CHI* conference and the IEEE *Human-Centric Computing* (formerly *Visual Languages*) conference, and the books *Studying the Novice Programmer* [Soloway 1989b], *Psychology of Programming* [Hoc 1990a], and *Software Design: Cognitive Aspects* [Détienne 2001].

1.2 A User Centered Design Process for Programming Systems

It is disappointing that the knowledge gathered over the past thirty years has had so little influence on the designs of new programming systems (in this document, the term programming system is used to encompass the programming language as well as the tools for viewing, editing, debugging and running programs). In order to help remedy this, I have organized the prior work that studied beginner programmers so that it might be readily included among the guidelines and strategies that are used by future programming system designers. Generally, language designers have focused on technical goals for their systems, such as to build systems that are scalable, efficient, reusable, provably correct, or that have mathematical elegance. When they face a design decision that is not determined by these criteria, they usually choose a solution that is similar to existing languages or one that appeals to their intuition. Usability has rarely been adopted as a formal objective.

I believe that usability should always be included among the criteria that are considered during the design of programming systems. Depending on the constraints of a particular project and target audience, usability may be given more or less weight. However, it is always worth considering for at least those decisions that are not already determined by other design criteria

In this thesis, I exemplify a new design process for programming systems, where usability is treated as a first-class objective:

1. *Identify the target audience* and the domain, that is, the group of people who will be using the system and the kinds of problems they will be working on.
2. *Understand the target audience*, both the problems they encounter and the existing recommendations on how to support their work. This includes an awareness of general HCI principles as well as prior work in the psychology of programming and empirical studies. When issues or questions arise that are not answered by the prior work, conduct new studies to examine them.
3. *Design the new system* based on this information.

4. *Evaluate the system* to measure its success, and understand any new problems that the users have. If necessary, redesign the system based on this evaluation, and then re-evaluate it.

In this design process, all of the prior knowledge about the human aspects of programming are considered, and the strategy for addressing any unanswered questions is to conduct user studies to obtain design guidance and to assess prototypes. For my new programming system for children, I adopted an extreme position by giving usability precedence over other objectives.

While my focus has been on beginner programmers, I believe this approach also applies to experts, and that it can have positive impacts on training and productivity as well as the reliability of professional software systems. Improving the programming systems used by experts will also affect beginners, because although these systems may not be the best choices for learning to program, they are often chosen because they are widely available and familiar to mentors. Everyone would benefit if these programming languages and tools were more usable.

1.3 Motivation

The goal of this thesis is to *enable* more beginners to learn to program for their personal purposes, with minimal training. There is no explicit goal to teach any particular computer science concepts, such as recursion, unless the concept is essential to the users achieving their goals. There is also no requirement for the new programming language produced by this work to match existing programming languages. Ideally, the new system will be general and powerful enough that many people will achieve their objectives without having to move to other new languages. Hopefully, the need to learn some of the harder computer science concepts can be deferred or eliminated. For those who do move on to other languages or even to become computer scientists, their early success with this first language should ease their difficulties in learning the harder computer science concepts.

1.4 Thesis Statement

The thesis statement for this work is:

this user-centered design process, incorporating principles from human-computer interaction, psychology of programming, and empirical studies, will result in a unique programming system that is easier to learn and use than more conventional programming systems.

1.5 Target Audience and Domain

The target audience for my new programming system is children in fifth grade (about ten years old) or older. I chose to build a system for children because they often have an interest in learning how to program, but can be quickly discouraged when they try. Their goals are creative and ambitious – they would like to make programs that are similar to the applications they use, such as games and simulations. These applications are graphically rich and highly interactive, unlike the first programs they are likely to create in many professional programming systems, such as to display “hello world” on the screen. My goal is to provide an easy entry into creating these interactive graphical programs. However, to the extent possible, I also tried to create a general purpose language that scales well, so that it is not inherently limited to creating toy programs.

1.6 Understanding the Target Audience

In addition to general design principles that are applicable to all users, there is a wealth of information available about how beginner programmers work and the problems they encounter. This section summarizes the prior work and briefly describes the new studies I conducted to examine additional questions.

1.6.1 General Design Principles

The field of *Human Computer Interaction* (HCI) has general principles and heuristics that can be applied to programming system design [Nielsen 1994]:

- simple and natural dialog – user interfaces should be simplified, and should match the user’s task in as natural a way as possible, such that the mapping between computer concepts and user concepts becomes straightforward.
- speak the user’s language – the terminology in user interfaces should be based on the user’s language, instead of using system-oriented terms or attaching non-standard meanings to familiar words.

- minimize user memory load – the system should take over the burden of memory from the user.
- consistency – the same command or action should always have the same effect.
- feedback – the system should continuously inform the user about what it is doing and how it is interpreting the user's input.
- clearly marked exits – the system should offer the user an easy way out of as many situations as possible, including ways to undo.
- shortcuts – the system should make it possible for experienced users to perform frequently used operations quickly.
- good error messages – the system should report errors politely in clear language, avoid obscure codes, use precise rather than vague or general explanations, and include constructive help for solving the problem.
- prevent errors – where possible, the user interface should be structured to avoid error situations.
- help and documentation – the help system and documentation should provide a quick way for users to find task-specific information when they are having a problem.

Many of these principles are routinely violated by programming systems – several examples are presented in Chapter 2.

When designing and evaluating programming systems, it is also useful to consider the more specific evaluation criteria in the *Cognitive Dimensions of Notations* framework (Cognitive Dimensions, for short) [Blackwell 2000, Green 1996]:

- viscosity – the system should not resist change; it should not require many user actions to accomplish one small goal.
- visibility – the information needed by the programmer at any particular time should be visible or very easy to access.
- premature commitment – the system should not force the user to go about the job in a particular order, or make a decision before the needed information is available.
- hidden dependencies – important links between entities should be visible.

- role expressiveness – the purpose of an entity should be readily apparent.
- error proneness – the notation should protect against slips and errors.
- closeness of mapping – the system's operations should closely match the way users think about problem solutions.
- secondary notation – the system should allow the programmer to communicate additional information with comments, typography, layout, etc.
- progressive evaluation – the system should permit users to test partial programs.
- diffuseness – small goals should not require extraordinarily long solutions or large amounts of screen space.
- provisionality – the system should allow the user to sketch out uncertain parts of their solution.
- hard mental operations – none of the system's operations should require great mental effort to use.
- consistency – similar notations should mean similar things, and vice versa.
- abstraction management – the system should provide a way to define new facilities or terms that allow the user to express ideas more clearly or succinctly, but it should not force users to use this capability right from the start.

These factors are sometimes in conflict, so improving the system along one dimension can result in reduced performance on another. Tradeoffs are necessary, and in making these tradeoffs it is useful to consider cognitive models and observations from empirical studies.

1.6.2 Observations about Existing Programming Languages

The principles of *simple and natural dialog, speak the user's language* and *closeness of mapping* are reinforced by cognitive models that define programming as a process where the user translates a mental plan into one that is compatible with the computer [Hoc 1990b]. The language should minimize the difficulty of this translation by providing operators that match those in the plan, including any that may be specific to the topic or domain of the program. "The closer the programming world is to the problem world, the easier the problem-solving ought to be.... Conventional textual languages are a long way from that goal"

[Green 1996, p. 146]. Hix & Hartson describe the general usability guideline to *use cognitive directness* [Hix 1993, p. 38] to "minimize the mental transformations that a user must make. Even small cognitive transformations by a user take effort away from the intended task." If the language does not provide these high-level operators, programmers have to assemble lower-level primitives to achieve their goals. This synthesis is one of the greatest cognitive barriers to programming [Lewis 1987].

Programmers are often required to think about algorithms and data in ways that are very different than the ways they already think about them in other contexts. For example, a typical C program to compute the sum of a list of numbers includes three kinds of parentheses and three kinds of assignment operators in five lines of code:

```
sum = 0;
for (i=0; i<numItems; i++) {
     sum += items[i];
}
return sum;
```

In contrast, this can be done in a spreadsheet with a single line of code using the *sum* operator [Green 1996]. The mismatch between the way a programmer thinks about a solution and the way it must be expressed in the programming language makes it more difficult not only for beginners to learn how to program, but also for people to carry out their programming tasks even after they become more experienced. One of the most common bugs among professional programmers using C and C++ is the accidental use of "=" (assignment) instead of "==" (equality test). This mistake is easy to make and difficult to find, not only because of typographic similarity, but also because "=" operator does indeed mean equality in other contexts such as mathematics.

Soloway, Bonar & Erlich [Soloway 1989a] found that the looping control structures provided by modern languages do not match the natural strategies that most people bring to the programming task. Furthermore, when novices are stumped they try to transfer their knowledge of natural language to the programming task. This often results in errors because the programming language defines these constructs in an incompatible way [Bonar 1989]. For example, *then* is interpreted as *afterwards* instead of *in these conditions*.

1.6.3 Naturalness

There are two ways to improve closeness of mapping. One is to teach people to think more like computers; the other is to make the programming system's operations match how users think. The latter approach is preferred in this thesis. A primary goal of my programming system is to support the *natural* ways that non-programmers think about problem solutions, instead of making them learn new and often unnatural ways to accomplish their objectives. In this context, natural means *expected or accepted*. If people have a viable approach to solving problems, the ideal programming system would support that solution directly, without requiring the programmer to learn anything new or perform additional work in translating their ideas into program code.

By this definition, naturalness is not universal for all humans. People from different backgrounds and cultures, or from different points in history, are likely to bring different expectations and methods to the programming task. Therefore, a programming system that is designed to be natural for a particular target audience is unlikely to be universally optimal. This is why identifying the target audience is an intrinsic part of the design process, and why the process itself is important. It will have to be applied over and over again, in order to best support the particular characteristics of the people who will use each new programming system.

Striving for naturalness does not necessarily imply that the programming language should use natural language. Programming languages that have adopted natural-language-like syntaxes, such as Cobol [Sammet 1981] and HyperTalk [Goodman 1987], still have many usability problems. For example, HyperTalk often violates the principle of consistency [Thimbleby 1992]. There are also many ambiguities in natural language that are resolved by humans through shared context and cooperative conversation [Grice 1975].

Novices attempt to enter into a human-like discourse with the computer, but programming languages systematically violate human conversational maxims because the computer cannot infer from context or enter into a clarification dialog [Pea 1986]. The use of natural language may compound this problem by making it more difficult for the user to understand the limits of the computer's intelligence [Nardi 1993].

However, these arguments do not imply that the algorithms and data structures should not be close to the ways people think about the problem. In fact, leveraging users' natural-language-like knowledge in a more formalized syntax can be an effective strategy for designing end-user-programming languages [Bruckman 1999].

There are additional motivations for why a more natural programming language might be better. A programming language is a type of user interface, and user interfaces in general are recommended to be *natural* so they are easier to learn and use, and will result in fewer errors. Naturalness is closely related to the concept of directness which, as part of *direct manipulation*, is a key principle in making user interfaces easier to use. Hutchins, Hollan & Norman describe *directness* as the distance between one's goals and the actions required by the system to achieve those goals [Hutchins 1986]. Reducing this distance makes systems more direct, and therefore easier to learn. User interface designers and researchers have been promoting directness at least since Shneiderman identified the concept [Shneiderman 1983], but it has not been a consideration in most programming language designs.

1.6.4 Studies of Naturalness in Problem Solving

This thesis presents two studies examining the language and structure that children and adults naturally use before they have been exposed to programming (Chapter 3). In these studies, I gave programming tasks to non-programmers and they solved these problems by writing and sketching their answers on paper. The tasks covered a broad set of essential programming techniques and concepts, such as control structures, storage and manipulation of data, arithmetic, Boolean logic, searching and sorting, animation, interactions among objects, etc. In posing the problems, I was careful to minimize the risk that my materials would influence the answers, so I used pictures and very terse captions.

Some observations from these studies were:

- An event-based or rule-based structure was often used, where actions were taken in response to events. For example, "when pacman loses all his lives, it's game over."
- Aggregate operators (acting on a set of objects all at once) were used much more often than iterating through the set and acting on the objects individually. For example, "Move everyone below the 5th place down by one."

- Participants did not construct complex data structures and traverse them, but instead performed content-based queries to obtain the necessary data when needed. For example, instead of maintaining a list of monsters and iterating through the list checking the color of each item, they would say “all of the blue monsters.”
- A natural language style was used for arithmetic expressions. For example, “add 100 to score.”
- Objects were expected to automatically remember their state (such as motion), and the participants only mentioned changes in this state. For example, “if pacman hits a wall, he stops.”
- Operations were more consistent with list data structures, rather than arrays. For example, the participants did not create space before inserting a new object into the middle of a list.
- Participants rarely used Boolean expressions, but when they did they were likely to make errors. That is, their expressions were not correct if interpreted according to the rules of Boolean logic in most programming languages.
- Participants often drew pictures to sketch out the layout of the program, but resorted to text to describe actions and behaviors.

1.6.5 Study of Methods to Specify Queries

Because content-based queries were prevalent in non-programmers’ problem solutions, I began to explore how this might be supported in a programming language. Queries are usually specified with Boolean expressions, and the accurate formulation of Boolean expressions has been a notorious problem in programming languages, as well as other areas such as database query tools [Hildreth 1988, Hoc 1989]. In reviewing prior research I found that there are few prescriptions for how to solve this problem effectively. For example, prior work suggests avoiding the use of the Boolean keywords *AND*, *OR*, and *NOT* [Greene 1990, McQuire 1995, Michard 1982], but does not recommend a suitable replacement query language.

Therefore I conducted a new study to examine the ways untrained children and adults naturally express and interpret queries, and to test a new tabular query form that I designed called *match forms* (shown in Figure 1-1). This study confirmed that relying on the Boolean keywords, as well as parentheses for grouping, would result in poor usability. Textual alternatives that avoided the Boolean keywords were not reliably better. However, the match forms were successful.

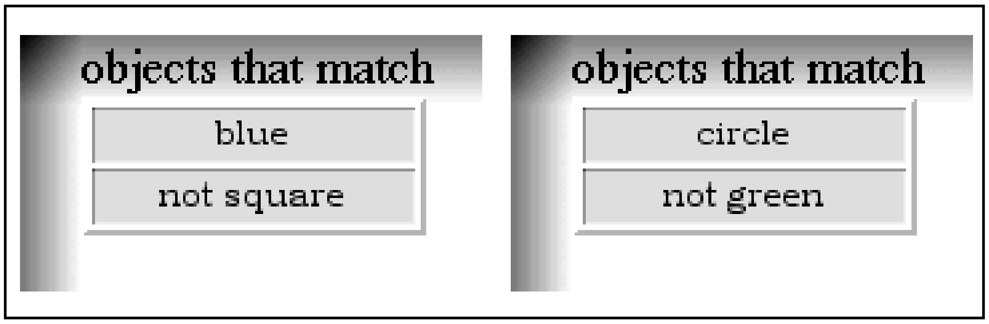


Figure 1-1. Match forms expressing the query: (blue and not square) or (circle and not green)

Each match form contains a vertical list of slots. Conjunction is specified by placing terms into these slots, one term per slot. Negation is performed by prefacing a term with the *NOT* operator, and disjunction is specified by placing additional match forms adjacent to the first one. This design avoids the need to name the *AND* and *OR* operators, provides a clear distinction between conjunction and disjunction, and makes grouping explicit. Match forms are suitable for incorporation into programming systems. When compared with textual Boolean expressions, users performed significantly better when they expressed their queries using match forms. When interpreting already-written queries, performance was about equal using either language. Chapter 4 contains full details about match forms and this study, as well as an application of this work to the search interface for the online *HCI Bibliography*.

1.6.6 Model of Computation

One of the biggest challenges for new programmers is to gain an accurate understanding of how computation takes place. Traditionally, programming is described to beginners in completely unfamiliar terms, often based on the von Neumann model, which has no real-world counterpart [du Boulay 1989a, du Boulay 1989b]. Beginners must learn, for example, that the program follows special rules of control flow for procedure calls and returns. There are complex rules that govern the lifetimes of variables and their scopes. Variables may not exist at all when the program is not running, and during execution they are usually invisible, forcing the programmer to use print statements or debuggers to inspect them. This violates the principle of visibility, and contributes to a general problem of memory overload [Anderson 1985, Davies 1993].

Usability could be enhanced by providing a different model of computation that uses concrete and familiar terms [Mayer 1989, Smith 1994]. Using a different model of computation can have broad implications beyond beginners, because the model influences, and perhaps limits, how experienced programmers think about and describe computation [Stein 1999].

Section 1.7.1 introduces the new model of computation I invented to address this problem.

1.6.7 Visual vs. Textual

In visual languages, graphics replace some or all of the text in specifying programs. Proponents of visual programming languages often argue that reducing or eliminating the text in programming will improve usability [Smith 1994]. However, much of the underlying rationale for this expectation is suspect [Blackwell 1996]. User studies have shown mixed results on the superiority of visual languages over text (e.g. [Green 1992]), and the advantage of visual languages tends to diminish on larger tasks. It is useful to note that one of the most successful end-user programming systems to date is the spreadsheet, which is mostly textual [Nardi 1993].

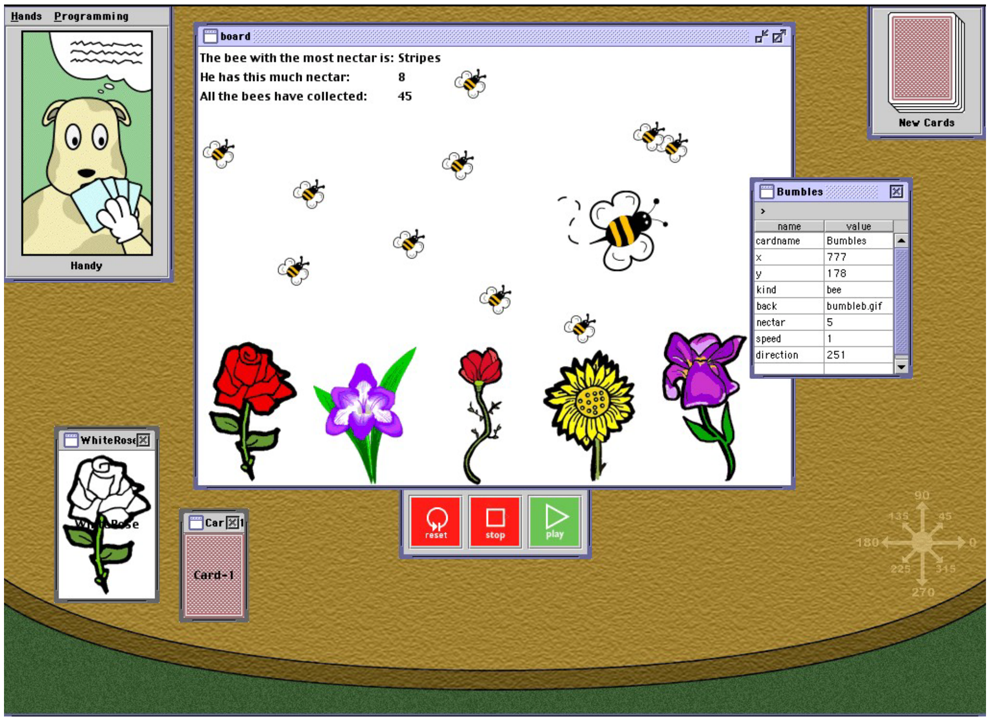
My new programming system supports the hybrid graphical-textual approach used by the participants in my studies, and relies on the programming environment to alleviate some of the difficulties of textual languages. For example, during program entry, context-sensitive menus like those in Microsoft's Visual Studio can make it easier to know what choices are available and to help the user to enter the program correctly. This support could be augmented with a drag-and-drop syntax-directed editor, as seen in Squeak's eToys interface [Steinmetz 2001] and other systems. The system can also provide visual representations for textual elements that are difficult, such as the match forms mentioned in Section 1.6.5.

1.7 The HANDS Programming System Design

All of these observations have influenced the design of my new programming system, which is called HANDS (Human-centered Advances for the Novice Development of Software). HANDS uses an event-based language that features a new concrete model for computation, provides queries and aggregate operators that match the way non-programmers express problem solutions, has high-visibility of program data, and includes domain-specific features for the creation of interactive animations and simulations. The HANDS system is detailed in Chapter 5.

1.7.1 Computational Model

In HANDS, the computation is represented as an agent named Handy, sitting at a table manipulating a set of cards (see Figure 1-2). All of the data in the system is stored on these cards, which are global, persistent and visible on the table. Each card has a unique name, and an unlimited set of name-value pairs, called properties. The program itself is stored in Handy's *thought bubble*. To emphasize the limited intelligence of the system, Handy is portrayed as an animal – like a dog that knows a few commands – instead of a person or a robot that could be interpreted as being very intelligent.



```
The bee with the most nectar is: Stripes
He has this much nectar:         8
All the bees have collected:     45
```

name	value
cardname	Bumbles
x	777
y	178
kind	bee
back	bumbleb.gif
nectar	5
speed	1
direction	251

Figure 1-2. The HANDS system portrays the components of a program on a round table. All data is stored on cards, and the programmer inserts code into Handy's thought bubble at the upper left corner. When the play button is pressed, Handy begins responding to events by manipulating cards according to the instructions in the thought bubble. This is described in more detail in Chapter 5.

1.7.2 Programming Style and Model of Execution

HANDS is event-based, the programming style that most closely matches the problem solutions in my studies. A program is a collection of event handlers that are automatically called by the system when a matching event occurs. Inside an event handler, the programmer inserts the code Handy should execute in response to the event.

1.7.3 Aggregate Operations

In my studies, I observed that the participants used aggregate operators, manipulating whole sets of objects in one statement rather than iterating and acting on them individually. Many languages force users to perform iteration in situations where aggregate operations could accomplish the task more easily [Miller 1981]. Requiring users to translate a high-level aggregate operation into a lower-level iterative process violates the principle of *closeness of mapping*.

HANDS has full support for aggregate operations. All operators can accept lists as well as singletons as operands, or even one of each. For example,

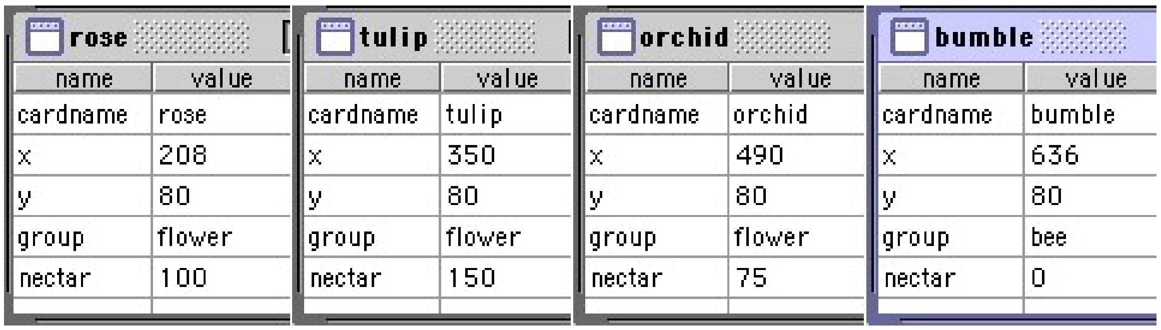
- `1 + 1` evaluates to `2`
- `1 + (1,2,3)` evaluates to `2,3,4`
- `(1,2,3) + 1` evaluates to `2,3,4`
- `(1,2,3) + (2,3,4)` evaluates to `3,5,7`

1.7.4 Queries

In my studies, I observed that users do not maintain and traverse data structures. Instead, they perform queries to assemble lists of objects on demand. For example, they say “all of the blue monsters.” HANDS provides a query mechanism to support this. The query mechanism searches all of the cards for the ones matching the programmer’s criteria.

Queries begin with the word *all*. If a query contains a single value, it returns all of the cards that have that value in any property. Figure 1-3 contains cards representing three flowers and a bee to help illustrate the following queries.

- `all flowers` evaluates to `orchid, rose, tulip`



rose

name	value
cardname	rose
x	208
y	80
group	flower
nectar	100

tulip

name	value
cardname	tulip
x	350
y	80
group	flower
nectar	150

orchid

name	value
cardname	orchid
x	490
y	80
group	flower
nectar	75

bumble

name	value
cardname	bumble
x	636
y	80
group	bee
nectar	0

Figure 1-3. When the system evaluates the query `all flowers` it returns `rose, tulip, orchid`

- `all bees` evaluates to `bumble`
- `all snakes` evaluates to the empty list

HANDS permits more complex queries to be specified with traditional Boolean expressions, however the intention is to eventually incorporate match forms into the system as an option for specifying and displaying queries.

Queries and aggregate operations work in tandem to permit the programmer to concisely express actions that would require iteration in most languages. For example,

- `set the nectar of all flowers to 0`

1.7.5 Domain-Specific Support

HANDS has domain-specific features that enable programmers to easily create highly-interactive graphical programs. For example, the system's suite of events directly supports this class of programs. The system automatically detects collisions among objects and generates events to report them to the programmer. It also generates events in response to input from the user via the keyboard and mouse. It is easy to create graphical objects and text on the screen, and animation can be accomplished without any programming.

1.8 Evaluation

To examine the effectiveness of three key features of HANDS: *queries, aggregate operations, and data visibility*, I conducted a study comparing the system with a limited version that lacks these features. In the limited version, programmers could achieve the same results but had to use more traditional programming techniques. Fifth-grade children were able to learn the HANDS system during a three-hour session, and then use it to solve programming problems. Children using the full-featured HANDS system performed significantly better than their peers who used the reduced-feature version. This is evidence that this set of features improves usability over the typical set of features in programming systems.

In a separate informal study, a high-school student compared hands with Stagecast, a commercial programming environment for children [Earhart 1999]. He implemented a game in both systems, and concluded that HANDS was easier to use, enabled him to implement more features, and required fewer lines of code. In addition, several more experienced programmers have used HANDS to implement a broad variety of programs to explore its range of capabilities.

Evaluation of the HANDS system is detailed in Chapter 6.

1.9 Contributions

The contributions of this thesis are:

- a case study of a new *design process* for creating programming systems, where usability is a first class objective;
- the *HANDS programming system* for children, which has a unique set of features due to its user-centered design, several of which were demonstrated to be more usable than those found in typical programming systems;
- a new *model of computation,* or way of thinking about programs, that is concrete and based on familiar concepts, unlike the traditional Turing machine or von Neumann machine models;
- a general-purpose *programming language* that offers database-style access to the program’s data, and in which all operators can be applied to singletons and lists;
- *match forms,* a tabular method for expressing queries that was compared to textual expressions and shown to improve beginners’ performance;
- a new *query interface* for the HCI Bibliography (www.hcibib.org), based on match forms, which reduces user errors in comparison to the old interface;

- *empirical evidence* about how non-programmers express problem solutions, which can be used to help designers generate and select programming system features that provide a close mapping between those problem solutions and their expression in program code;
- *empirical evidence* characterizing the kinds of errors made by inexperienced users of textual Boolean expressions;
- *a user study* demonstrating the effectiveness of queries, aggregate operations, and high-visibility of data, in comparison to the typical features sets of programming systems; and,
- a *broad survey* of the prior work on beginner programmers, organized in a form that can be used by other programming system designers (appears in Appendix C).

1.10 Overview of Thesis

The remainder of this thesis is organized as follows: Chapter 2 describes the prior empirical work on beginner programmers as well as other programming systems for beginners and children; Chapter 3 describes the first two studies examining the language and structure in non-programmers solutions to programming problems; Chapter 4 describes the third study, examining methods for specifying queries, and provides details about match forms; Chapter 5 details the design of the HANDS system; Chapter 6 describes a fourth study, to evaluate features of HANDS, as well as other less formal evaluations; Chapter 7 discusses the implications of this work and some ideas for future work; and Chapter 8 gives some concluding remarks.

Supplemental materials are contained in appendices. Appendix A contains a formal specification of the HANDS language syntax; Appendix B contains some example programs implemented in HANDS; Appendix C contains the full text of my technical report surveying usability issues in programming systems for beginners; Appendix D contains the materials used in the first study; Appendix E contains the materials used in the second study; Appendix F contains the materials used in the third study; and Appendix G contains the materials used in the fourth study.

CHAPTER 2 Related Work

This chapter surveys other programming systems for beginners and children.

2.1 Usability Issues in Programming Systems for Beginners

Appendix C contains the full text of my technical report surveying usability issues in the design of programming systems for beginners, covering the prior work in *Empirical Studies of Programmers* and *Psychology of Programming* [Pane 1996]. Throughout this thesis, additional relevant related work is cited in context.

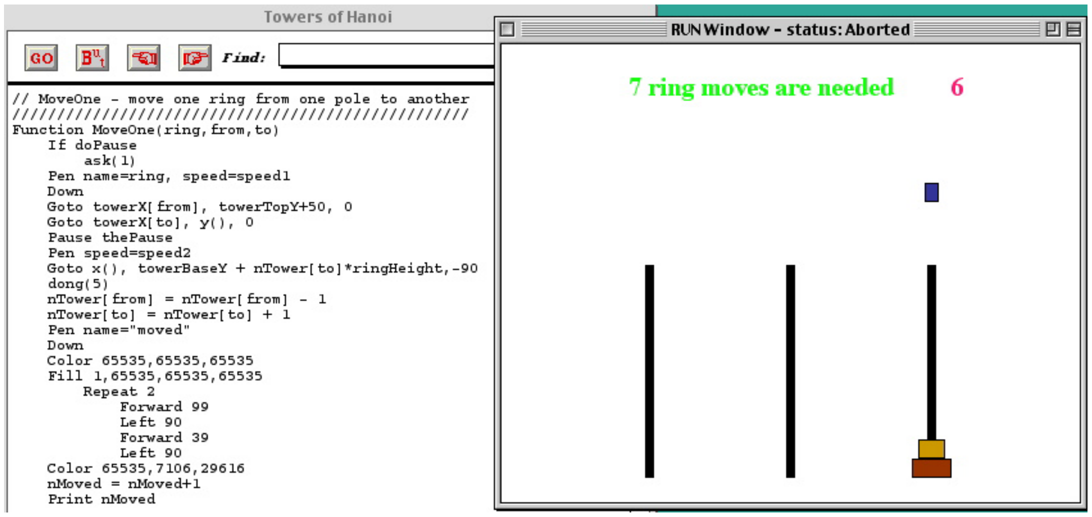
2.2 Systems for Beginners and Children

This section briefly summarizes the numerous systems that have been created for beginners and children.

2.2.1 The Logo Family

Logo [Papert 1980] is a successful and popular language for children. Its textual language is based on Lisp, with a syntax that was redesigned to be easier to learn and read, yet it does use unusual punctuation and cryptic names for commands. Logo uses a turtle metaphor for a drawing pen. There have been many implementations of Logo. StarLogo extends the metaphor to parallel processing [Resnick 1994]. Figure 2-1 shows an solution to the Towers of

Hanoi problem implemented in LogoMation. LEGO/Logo links the popular LEGO construction kit with the Logo programming language [Martin 1993]. Children build machines out of LEGO pieces, including newer pieces such as gears, motors and sensors, and then write computer programs to control the machines.



```
// MoveOne - move one ring from one pole to another
////////////////////////////////////////////////////
Function MoveOne(ring,from,to)
    If doPause
        ask(1)
    Pen name=ring, speed=speed1
    Down
    Goto towerX[from], towerTopY+50, 0
    Goto towerX[to], y(), 0
    Pause thePause
    Pen speed=speed2
    Goto x(), towerBaseY + nTower[to]*ringHeight,-90
    dong(5)
    nTower[from] = nTower[from] - 1
    nTower[to] = nTower[to] + 1
    Pen name="moved"
    Down
    Color 65535,65535,65535
    Fill 1,65535,65535,65535
        Repeat 2
            Forward 99
            Left 90
            Forward 39
            Left 90
    Color 65535,7106,29616
    nMoved = nMoved+1
    Print nMoved
```

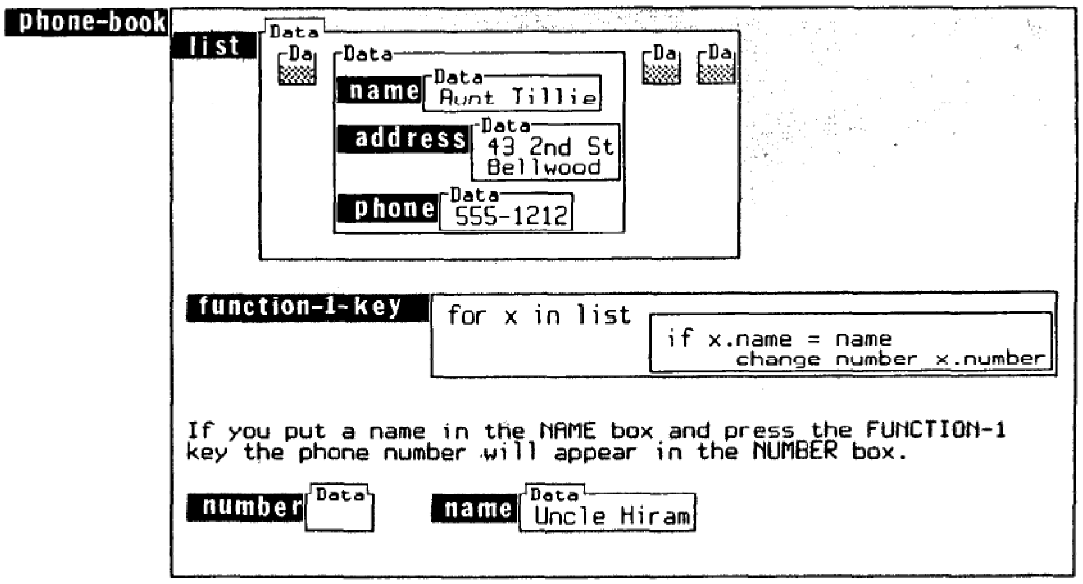
Figure 2-1. A solution to the Towers of Hanoi problem, implemented in LogoMation.

2.2.2 Boxer

Boxer [diSessa 1986] uses a two-dimensional, visible, concrete metaphor where boxes and their spatial relations represent the computation (Figure 2-2). Variables can be modified by direct manipulation, and the state and code for graphical objects is packaged with their graphical representation. Boxer's textual language is very similar to Logo, with extensions to broaden its range of capabilities.

2.2.3 ToonTalk

ToonTalk is a children's programming language based on a video game metaphor [Kahn 1996]. Its cartoon world provides concrete realizations of all of the concepts required in concurrent constraint programming. For example, birds and nests represent communication, a robot represents a statement guarded by a condition, a balance scale represents comparison tests, a wand can be used to create data, a vacuum cleaner can be used to delete data, and a bomb represents termination of a process (see Figure 2-3). Programs are constructed by using video-game controls to train the robots. For example, if a robot drops a number 2



```
for x in list
    if x.name = name
        change number x.number
```

If you put a name in the NAME box and press the FUNCTION-1 key the phone number will appear in the NUMBER box.

Figure 2-2. In Boxer, the computation is represented by boxes and their spatial relations. The language is based on Logo. This figure originally appeared in [diSessa 1986].

on top of a number 7, a character named Bammer appears and smashes the numbers with a sledgehammer, causing the 7 to be replaced with the sum 9. It is a fascinating system, however its low-level primitives present a challenge for beginners, who may have great difficulty in figuring out how to compose the primitives to accomplish their higher-level goals. For example, a single robot cannot accomplish the test `x+y>100`; a team of robots must be programmed to do this simple test. As another example, a research paper devotes more than four pages (including illustrations) to explain how a programmer would program ToonTalk to append two lists [Kahn 1999].

2.2.4 AgentSheets

AgentSheets [Repenning 1995] is the first in a family of rule-based graphical programming environments based on a spreadsheet metaphor. In these systems, program objects occupy cells in a grid, and interact with the objects in neighboring cells. These interactions are specified by graphical rewrite rules, which are before-and-after pictures (see Figure 2-4).

2.2.5 Stagecast

Stagecast (formerly KidSim and Cocoa) extended this metaphor with the capability to use programming by demonstration to create the graphical rewrite rules (see Figure 2-5) [Joers

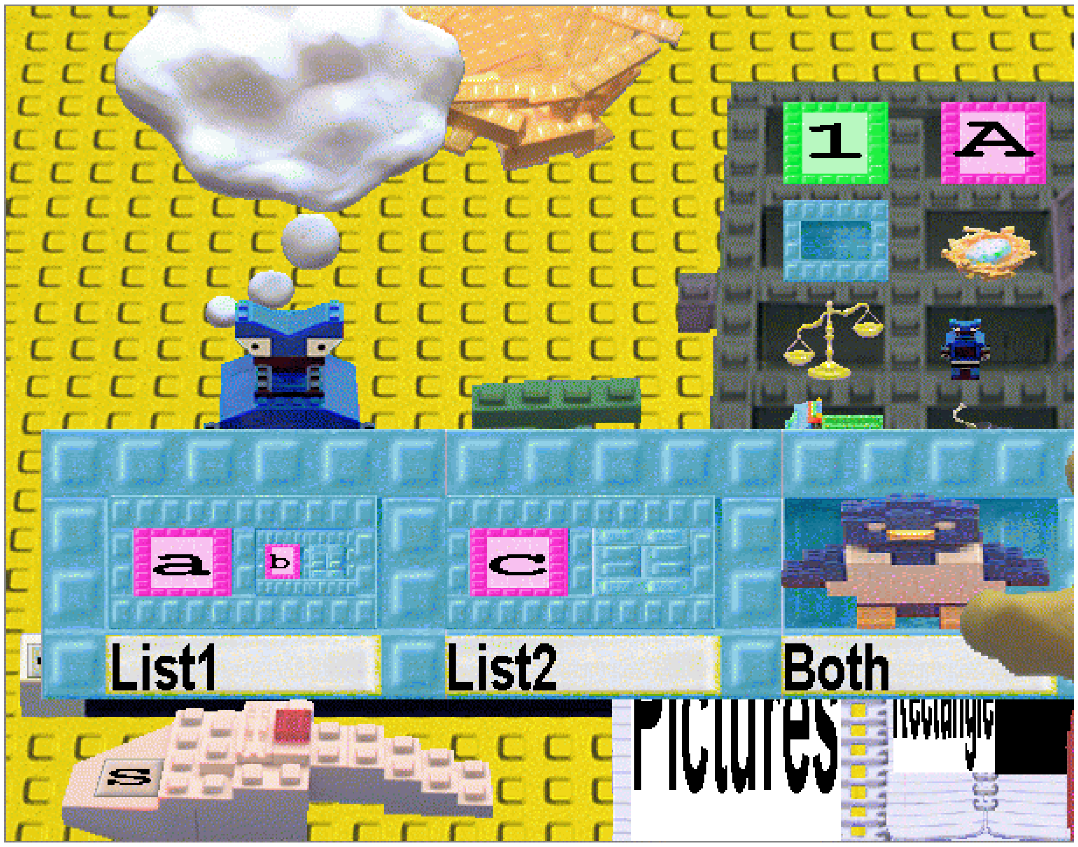


Figure 2-3. ToonTalk has concrete representations for computational concepts. For example, birds and nests represent communication, a robot represents a statement guarded by a condition, a balance scale represents comparison tests, a wand can be used to create data, a vacuum cleaner can be used to delete data, and a bomb represents termination of a process. This figure originally appeared in [Kahn 1999].

1999, Smith 1994]. While beginners can quickly create some interesting programs, some kinds of games and simulations are difficult to implement. For example, the grid makes it difficult to implement smooth motions in arbitrary directions. Also, graphical rewrite rules are local – a region of the grid surrounding the object is considered in determining whether a rewrite rule matches. Therefore, these rules by their nature are not suited to interactions at a distance. As more objects populate the system, and as the grid region is expanded to include more cells, there can be a combinatorial explosion in the number of situations that must be considered. Often, multiple similar rules must be programmed in order to handle the various situations.

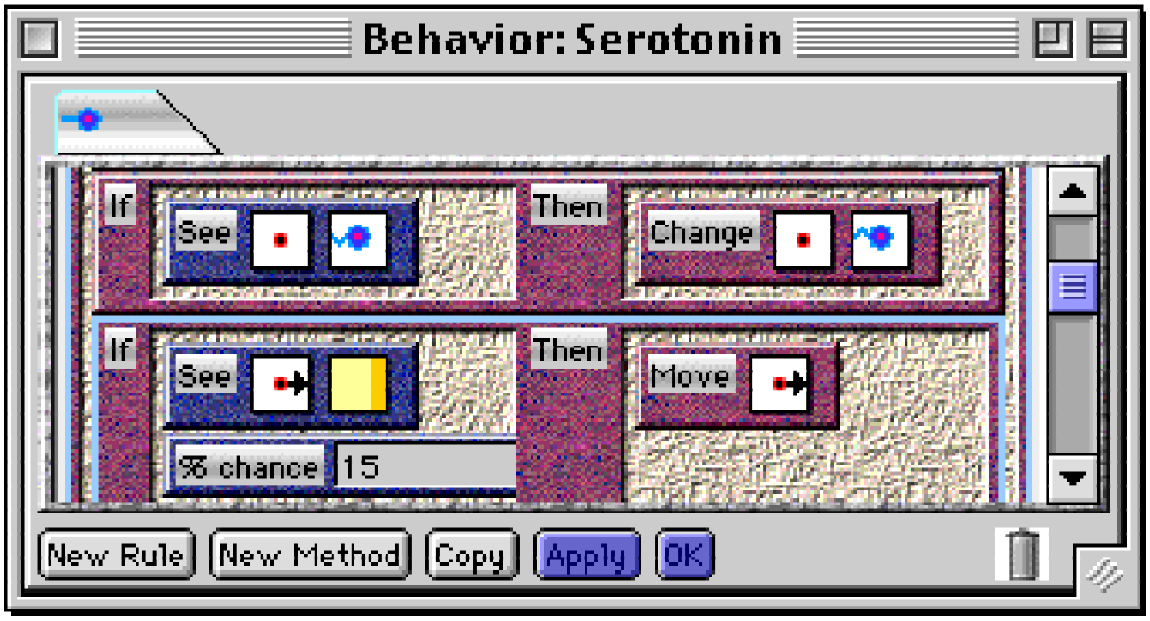


Figure 2-4. AgentSheets uses rewrite rules. In this example, the top rule specifies that if the serotonin has a certain appearance it will change to another appearance; and the bottom rule specifies that if there is a membrane to the right, the serotonin will move to the right with at 15% probability. This figure originally appeared in [Repenning 2000].

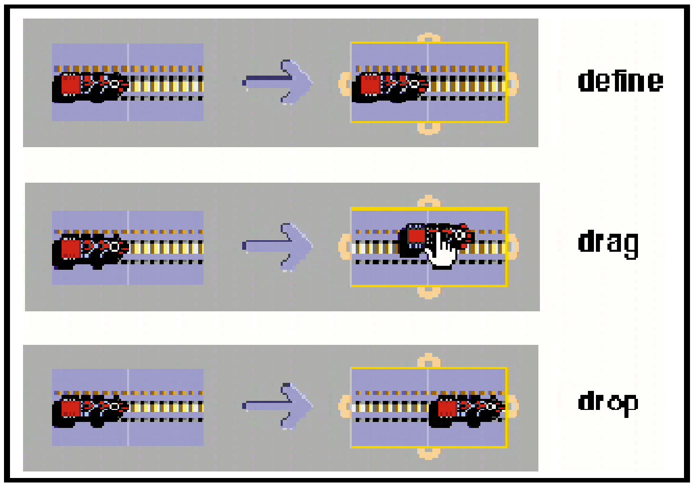
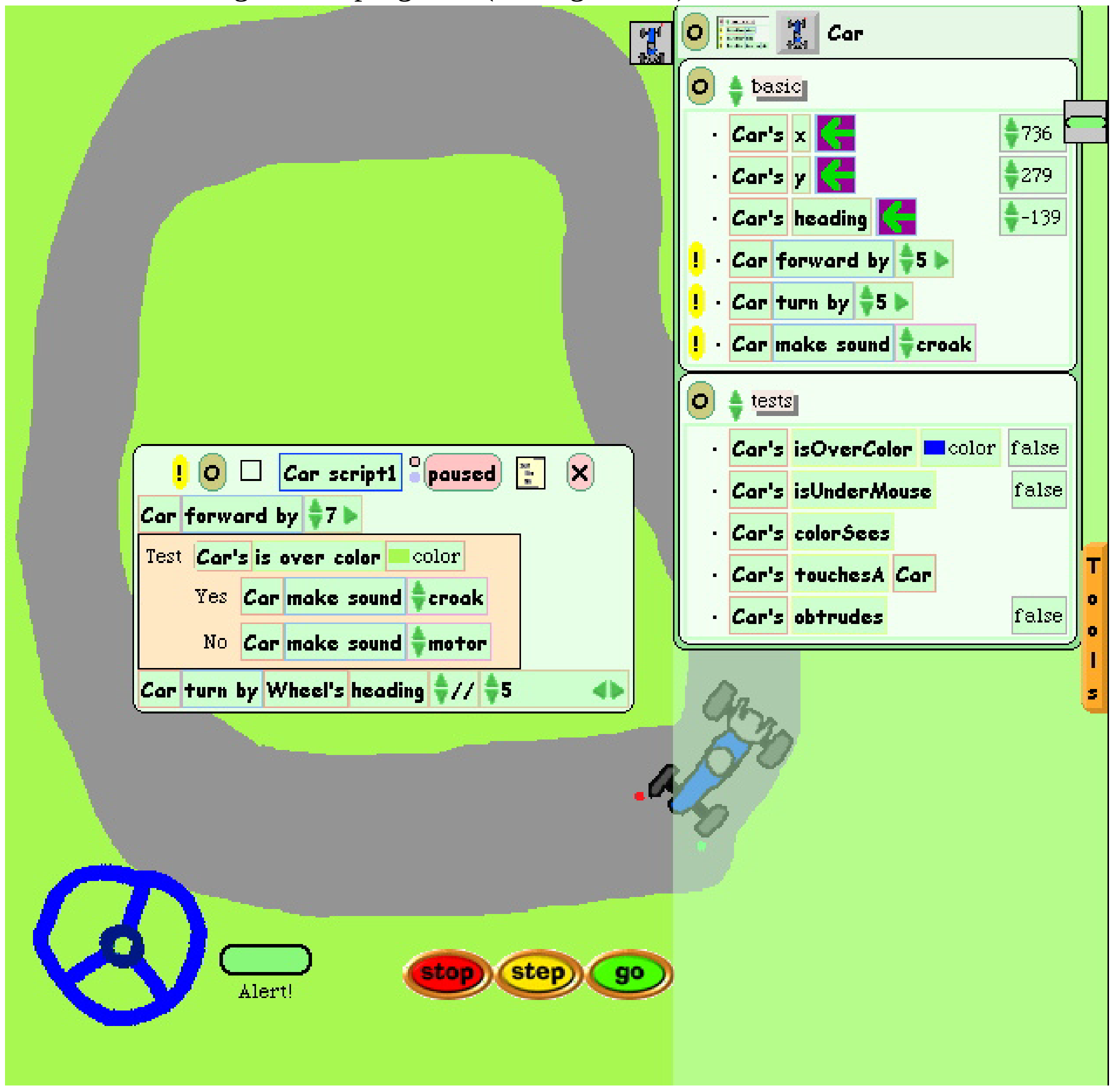


Figure 2-5. In Stagecast, the rewrite rules can be specified by demonstration, by directly manipulating the objects. For simple rules, no text is required. In this figure, the programmer is defining a rule that moves a vehicle forward (to the right) along a track. This figure originally appeared in [Smith 2000].

2.2.6 SmallTalk and eToys

SmallTalk is an exploratory object-oriented language that was designed to be accessible to non-technical people [Ingalls 1981]. A recent portable implementation named Squeak includes a learning environment interface for children. This system, called eToys, has support for children to add behaviors to objects, using an interface that features a subset of the SmallTalk language in a more verbose style, with a tile-based drag-and-drop interface to assist in constructing correct programs (see Figure 2-6) [Steinmetz 2001].



```
Car script1 paused
Car forward by 7
Test Car's is over color color
     Yes Car make sound croak
     No  Car make sound motor
Car turn by Wheel's heading // 5
```

```
Car
basic
· Car's x            736
· Car's y            279
· Car's heading      -139
! · Car forward by 5
! · Car turn by 5
! · Car make sound croak
tests
· Car's isOverColor color   false
· Car's isUnderMouse        false
· Car's colorSees
· Car's touchesA Car
· Car's obtrudes            false
```

Figure 2-6. The eToys interface in Squeak provides a tile-based drag-and-drop interface for constructing programs in a verbose version of SmallTalk.

2.2.7 Alice

Alice [Conway 2000, Conway 1997] is an authoring tool for scripting and prototyping 3D object behaviors. By writing simple scripts, Alice users can control object appearance and behavior, and while the scripts are executing, objects respond to user input via mouse and keyboard. Alice is designed to be simple enough that it can be used by people who don't necessarily call themselves programmers. A new version of Alice that is currently being

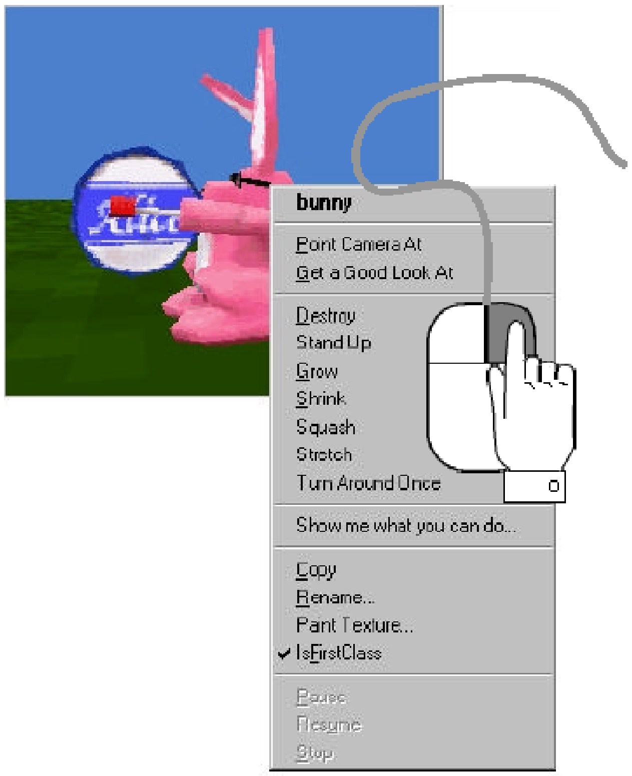


Figure 2-7. Alice is an authoring tool for scripting and prototyping 3D object behaviors. This figure originally appeared in [Conway 1997].

built has been influenced by this thesis. For example, Alice team has tried to reduce punctuation in their language, and to avoid the Boolean keywords AND and OR.

2.2.8 Rehearsal World

Rehearsal World is a system that uses a theatre metaphor, where the programming process consists of moving “performers” around on “stages” and teaching them how to interact by sending “cues” to one another [Finzer 1993].

2.2.9 Karel the Robot

Numerous “mini-languages” have been created over the years for teaching programming in an environment that is intentionally limited for simplicity [Brusilovsky 1997]. These languages are used for a short time to allow beginners to learn some programming, before they move to more complete programming languages. Usually these languages are very similar to existing languages, so they generally do not break substantial new ground in language design. For example, Karel the Robot was designed as a simple introduction to the Pascal language [Pattis 1995].

2.2.10 GRAIL

GRAIL stands out among mini-languages because its creators relaxed this constraint, and adopted usability principles and a pedagogical theory to guide its design [McIver 2001]. GRAIL is an imperative language with an English-like syntax. It has no pointers or references, and it has a single numeric type. GRAIL uses non-ascii characters to typographically represent the symbols that students are used to seeing in other domains (such as ÷), A user study showed that students made significantly fewer errors when using GRAIL than with Logo.

2.2.11 HyperTalk

HyperTalk is the scripting language in the HyperCard system [Goodman 1987]. HyperTalk is a verbose English-like language, with optional extra words to enhance readability. Code is located inside the user interface object it is associated with. This makes it difficult for a programmer to find all of the code in a large or unfamiliar program, or to determine why the code is not working as expected. Even if the programmer knows where to look, the system requires many steps to make a desired item visible [Green 1990, Green 1996]. Many inconsistency issues and other usability problems have been reported about HyperTalk [Thimbleby 1992].

2.2.12 AppleScript

AppleScript is a scripting language built into the Apple Macintosh operating system. It has a similar English-like flavor to HyperTalk. The system has a programming by demonstration capability, where the programmer can turn on a *record* mode and then perform the actions using the standard user interface of the application, such as menus and buttons. When the record mode is turned off, an AppleScript script to perform those same actions appears.

2.2.13 SK8Script

SK8Script is a similar scripting language in Apple's SK8 authoring tool. SK8Script has a query-like mechanism for locating objects matching certain criteria, and has the ability to act on them in aggregate. For example, `set the fillColor of every Rectangle whose height > 30 in DrawWindow to Red`.

2.2.14 Chart ‘n’ Art

Chart ‘n’ Art is a programmable tool for creating charts, graphs, and other kinds of graphics [DiGiano 2001, DiGiano 1996]. The tool is *self-disclosing*. As the user performs interactive commands the system volunteers information about how the same commands can be performed programmatically, and also offers a list of possible commands that could be issued next. This is done in a non-intrusive way, so that the user might learn to accomplish their tasks more efficiently through programming than they could interactively. Chart ‘n’ Art uses SK8Script, so it also has aggregate operations.

2.2.15 cT

cT [Sherwood 1988] is a multimedia authoring tool for creating simulations. When the user interactively defines and manipulates graphical objects, the corresponding cT code is automatically written or revised to reflect the changes. The cT language has few extraneous brackets, and indenting is used to indicate scope, instead of begin-end blocks or brackets.

2.2.16 LabView

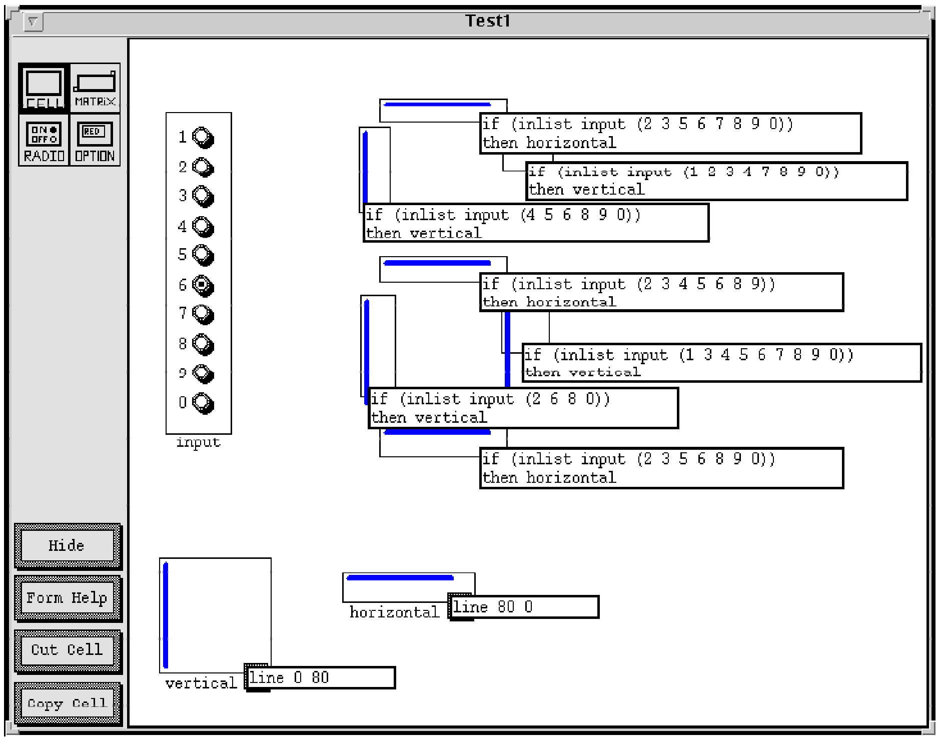
LabView is a visual programming environment for creating interfaces to scientific instruments. A study comparing LabView with a textual language concluded that visual programs were more difficult to comprehend, even by programmers who were experienced with the language [Green 1992].

2.2.17 Forms/3

Forms/3 [Hays 1995] is a general purpose, declarative, spreadsheet-based visual programming language (see Figure 2-8). Its goal is to provide computational and expressive power in a language featuring a simple, concrete programming style with immediate feedback. In Forms/3 the programmer creates cells by direct manipulation and then defines formulas for the cells.

2.2.18 Visual Basic

Visual Basic is a popular end-user programming system for non-programmers. It is a textual event-based language with domain-specific support for forms, dialog boxes and tables,



```
if (inlist input (2 3 5 6 7 8 9 0))
then horizontal

if (inlist input (1 2 3 4 7 8 9 0))
then vertical

if (inlist input (4 5 6 8 9 0))
then vertical

if (inlist input (2 3 4 5 6 8 9))
then horizontal

if (inlist input (1 3 4 5 6 7 8 9 0))
then vertical

if (inlist input (2 6 8 0))
then vertical

if (inlist input (2 3 5 6 8 9 0))
then horizontal

line 80 0

line 0 80
```

Figure 2-8. Forms/3 is a visual programming environment based on a spreadsheet paradigm. This example shows the program for an LED digit. This figure originally appeared in [Wilcox].

which are common in business tasks. The programming language itself is based on the original Basic language, which has many well-known usability problems [Pane 1996].

2.2.19 Java and C#

Java and Microsoft's C# contain many usability refinements over C and C++. These changes were based on common problems experienced by programmers using the earlier languages. For example, these languages prohibit assignment in the tests of conditional structures because this was a common place of errors in C and C++. However the designers were constrained in how far they could deviate from the predecessors, so programmers could more easily switch to the new languages.

2.2.20 MacGnome

MacGnome is a family of structure-editor based programming environments for beginner programmers [Miller 1994]. The structure editor keeps the program syntactically correct, and offers context-sensitive menus showing all of the legal constructs at any point in the program.

These systems started out enforcing syntactic correctness at all times, which has many benefits for beginners who otherwise might make many syntactic errors and not discover them until much later. However, this made editing difficult because it was not possible to temporarily go through syntactically-incorrect states while making a change. Later, these systems added the ability for the user to edit portions of the program textually, and tried to make the transition between textual and structure editing as smooth as possible. For example, to prevent users from naively staying in textual mode and not receiving the benefits of structure editing, the system automatically returned to structure mode at the end of each statement (which included a check for correct syntax), but would go back into text mode again if the user kept typing.

These systems also performed incremental semantic analysis, so that errors such as type errors or using an undeclared variable could be flagged while the programmer was editing. They also had integrated support for running and debugging programs. As the program ran, the executing code was highlighted in the editing window. A graphical portrayal of the call stack showed all program data and was updated as the program ran. Common data structures, such as linked lists and binary trees, were automatically recognized and displayed similar to the way teachers draw them on the blackboard.

2.2.21 Programming by Demonstration

In programming by demonstration (PBD) systems, the user performs actions interactively to demonstrate the desired behavior and the system generalizes these actions to form a program [Myers 1992]. One of the difficulties in PBD systems is this generalization step and getting the user to provide the right examples to make the correct inferences possible. Often there is no visible representation of the program itself, but Pursuit [Modugno 1995] provided a comic-strip metaphor to represent programs. Gamut [McDaniel 1999] was a system that permitted children to create entire board-game applications by demonstration. This thesis does not make use of PBD techniques.

2.2.22 Hank

Hank [Mulholland 2000] is a cognitive modelling language for non-programmers. It contains a database with “fact cards,” represented graphically as small spreadsheets, and “instruction cards,” represented as flowcharts. The system uses a comic strip storyboard to represent the model's behavior, and includes a query system for finding information on fact cards.

CHAPTER 3

The Language and Structure in Problem Solutions Written by Non-Programmers

In Chapter 1 it is argued that programming systems should provide operations that closely match the ways people naturally think about achieving their goals. While the prior work provides general design guidance, and identifies specific areas where people are known to have difficulty, it offers little prescriptive information about which features should be included in programming systems. This chapter describes a pair of studies seeking to obtain this prescriptive guidance, by examining the ways that people solve programming-like tasks before they have been influenced by an exposure to programming.[1] These studies were designed to provide insight into the concepts people use when thinking about algorithms, the kinds of structures they used to organize their solutions, and the vocabulary they use to express their answers. The programming system can then be designed to directly support the natural methods that are observed.

The first study focuses on children because they are the audience for my new programming language. In addition, children are less likely to be programmers, so their responses should reveal problem solving techniques that have not been influenced by programming experience. The exercises in this study are drawn from the domain of computer games and ani-

1. Portions of this chapter were previously reported in [Pane 2001].

mated stories, because children are often interested in building these kinds of programs. The second study then examines how the results of the first study generalize to a broader range of ages that includes adults, and to a different domain that incorporates database access scenarios that are typical of business programming tasks.

3.1 Comparison to Lance Miller's Studies

These studies are similar to a series of studies by Lance Miller in the 1970s [Miller 1974, Miller 1981]. Miller examined natural language procedural instructions generated by non-programmers and made a rich set of observations about how the participants naturally expressed their solutions. This resulted in a set of recommended features for computer languages. For example, Miller suggested that *contextual referencing* would be a useful alternative to the usual methods of locating data objects by using variables and traversing data structures. In contextual referencing, the programmer identifies data objects by using pronouns, ordinal position, salient or unique features, relative referencing, or collective referencing [Miller 1981, p 213].

Although Miller's approach provided many insights into the natural tendencies of non-programmers, there have only been a few studies that have replicated or extended that work. Biermann, Ballard & Sigmon confirmed that there are many regularities in the way people express step-by-step natural language procedures, suggesting that these regularities could be exploited in programming languages [Biermann 1983]. Galotti & Ganong found that they were able to improve the precision in users' natural language specifications by ensuring that the users understood the limited intelligence of the recipient of the instructions [Galotti 1985]. Bonar & Cunningham found that when users translated their natural-language specifications into a programming language, they tended to use the natural-language semantics even when they were incorrect for the programming language [Bonar 1988]. It is surprising that the findings from these studies have apparently not had any direct impact on the designs of new programming languages that have been invented since then.

A risk in designing these studies is that the experimenter could bias the participants with the language used in asking the questions. For example, the experimenter cannot just ask: "How would you tell the monsters to turn blue when the PacMan eats a power pill?" because this may lead the participants to simply parrot parts of the question back in their answers. This would defeat the prime objective of these studies, to examine users' unbiased responses.

The studies reported in this chapter differ from Miller's studies in several ways:

1. *Eliminating possible bias.* Miller's studies used verbose textual problem statements, increasing the risk that the language used in the participants' responses was biased by the materials. In fact, one of the frequently-observed keywords in Miller's results actually appeared in the problem statement that was given to the participants. The current studies take great care to minimize this kind of bias by using terse descriptions along with graphical depictions of the problem scenarios.
2. *Fewer constraints on the form of the solutions.* Miller's studies placed constraints on the participants' solutions, such as: they were broken into *steps*, each a line of text limited to 80 characters; steps had to be retyped completely in order to edit them; and, a minimum of five steps was required in a solution. The current studies are much less constrained, allowing users to write or draw as much or as little text and pictures as they need to convey their solutions.
3. *Broader range of tasks.* Miller's tasks were typical database problems common to the era of his studies. The current studies investigate a broader range of tasks that incorporate modern graphical user interfaces and media such as animations.
4. *Broader age range of participants.* Miller's participants were all college students. The current studies investigate a broader age range, including children.

Thus, the current studies may yield more reliable information about the natural expressions of a wider audience, on a broader range of algorithms and domains.

3.2 Overview of the Studies

In these studies, participants were presented with programming tasks and asked to solve them on paper using whatever diagrams and text they wanted to use. Before designing the tasks, a list of essential programming techniques and concepts was enumerated, covering various kinds of applications. These include: use of variables, assignment of values, initialization, comparison of values, Boolean logic, incrementing and decrementing of counters, arithmetic, iteration and looping, conditionals and other flow control, searching and sorting, animation, multiple things happening simultaneously (parallelism), collisions and interactions among objects, and response to user input.

Because children often express interest in creating games and animated stories, the first study focused on the skills that are necessary to build such programs. In this study, the PacMan video game was chosen as a fertile source of interesting problems that require these skills. Instead of asking the participants to implement an entire PacMan game, various situations were selected from the game because they touch upon one or more of the above concepts. This allowed a relatively small set of exercises to broadly cover as many of the concepts as possible in the limited amount of time available. Many of the skills that were not covered in the first study were covered in the second, which used a set of spreadsheet-like tasks involving database manipulation and numeric computation.

To minimize the form of bias described in Section 3.1 on page 32, a collection of pictures and QuickTime movie clips were developed to depict the various scenarios using very terse captions. This enabled the experimenter to show the depictions to the participants and ask general, open-ended questions to prompt their responses. An example from Study One is shown in Figure 3-1. Copies of the materials are available in Appendix D and Appendix E.

3.3 Study One

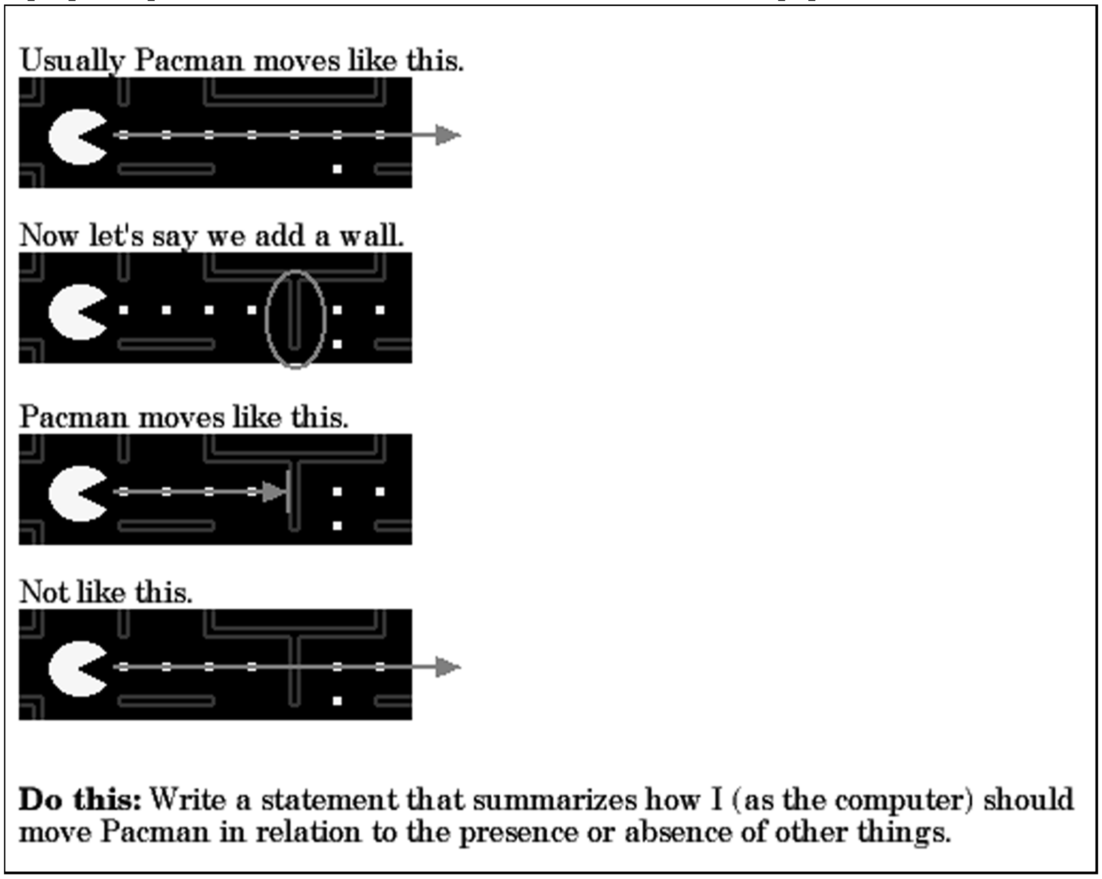
The first study examines children's solutions to a set of tasks that would be necessary to program a computer game.

3.3.1 Participants

Fourteen fifth graders at a Pittsburgh public elementary school participated in this study. The participants were equally divided between boys and girls, were racially diverse, and were either ten or eleven years old. All of the participants were experienced computer users, but only two of them (both boys) said they had programmed before. All of the analyses in this article examine only the twelve non-programmers. The participants were recruited by sending a brief note and consent form to parents. The participants received no reward other than the opportunity to leave their normal classroom for a half hour, and the opportunity to play a computer game for a few minutes.

3.3.2 Materials

A set of nine scenarios from the *PacMan* game were chosen, and graphical depictions of these scenarios were developed, containing still images or animations and a minimal amount of text. The topics of the scenarios were: an overall summary of the game, how the user controls PacMan's actions, PacMan's behavior in the presence and absence of other objects such as walls, what should happen when PacMan encounters a monster under various conditions, what happens when PacMan eats a power pill, scorekeeping, the appearance and disappearance of fruit in the game, the completion of one level and the start of the next, and maintenance of the high score list. Figure 3-1 shows one of the scenario depictions and the rest are in Appendix D. The participants viewed the depictions on a color laptop computer, and wrote their solutions on blank unlined paper.



Usually Pacman moves like this.

Now let's say we add a wall.

Pacman moves like this.

Not like this.

Do this: Write a statement that summarizes how I (as the computer) should move Pacman in relation to the presence or absence of other things.

Figure 3-1. Depiction of a problem scenario in study one.

3.3.3 Procedure

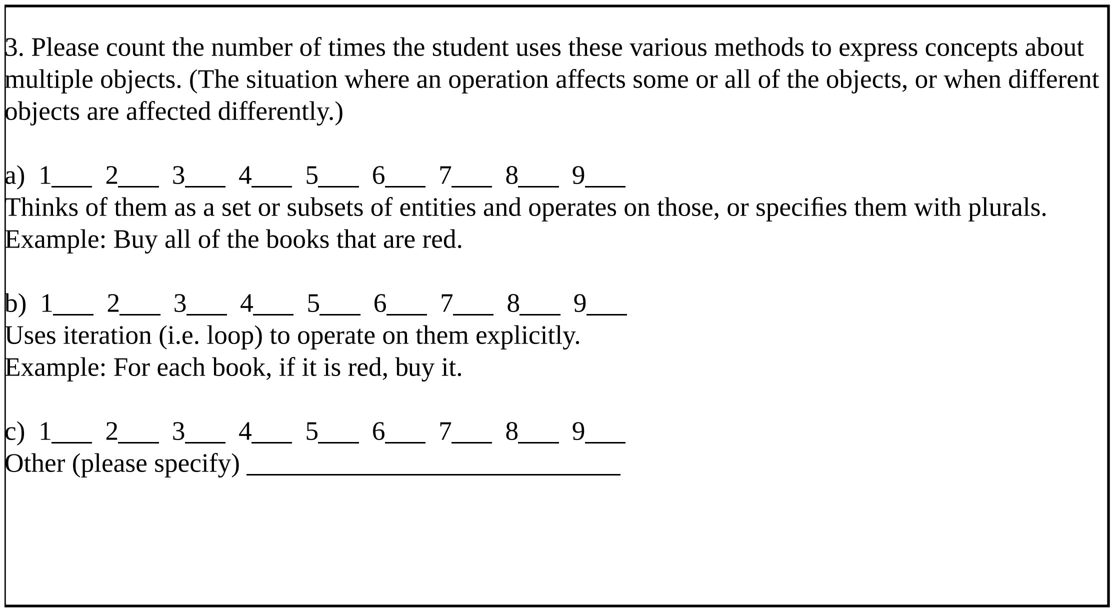
After a brief interview to gather background information, participants were shown each scenario and asked to write down in their own words and pictures how they would tell the computer to accomplish the scenario. When a response was judged to be incomplete or unsatisfactory, the experimenter attempted to elicit additional information by asking the participant to give more detail, by demonstrating an error in the existing answer, or by asking questions that were carefully worded to avoid influencing the responses. The sessions were audiotaped.

3.3.4 Content Analysis

A rating form was developed to be used by independent raters to analyze each participant's responses. Each question on the form addressed some facet of the participant's problem solution, such as the way a particular word or phrase was used, or some other characteristic of the language or strategy that was employed. Many of these questions arose from the results of a pilot study. In addition, a preliminary review of the participant data revealed trends in the solutions that seemed important, so the rating form was supplemented with questions to explore these as well.

Each question was followed by several categories into which the participant's responses could be classified. The rater was instructed to look for relevant sentences in the participant's solution, and classify each one by placing a tickmark in the appropriate category, also noting which problem the participant was answering when the sentence was generated. Each question also had an *other* category, which the rater marked when the participant's utterance did not fall into any of the supplied categories. When they did this, they added a brief comment. Figure 3-2 shows one of the questions from the rating form for study one, and the rest can be found in Appendix D.

Five independent raters categorized the participants' responses. These raters were experienced computer programmers, who were recruited by posting to Carnegie Mellon University's electronic bulletin boards, and were paid for their assistance. They were given a one-page instruction sheet describing their task. Each rater filled out a copy of the 17-question rating form for each of the participants. The raters reported that this was a very tedious pro-



3. Please count the number of times the student uses these various methods to express concepts about multiple objects. (The situation where an operation affects some or all of the objects, or when different objects are affected differently.)

a) 1___ 2___ 3___ 4___ 5___ 6___ 7___ 8___ 9___
Thinks of them as a set or subsets of entities and operates on those, or specifies them with plurals.
Example: Buy all of the books that are red.

b) 1___ 2___ 3___ 4___ 5___ 6___ 7___ 8___ 9___
Uses iteration (i.e. loop) to operate on them explicitly.
Example: For each book, if it is red, buy it.

c) 1___ 2___ 3___ 4___ 5___ 6___ 7___ 8___ 9___
Other (please specify) ____________________________

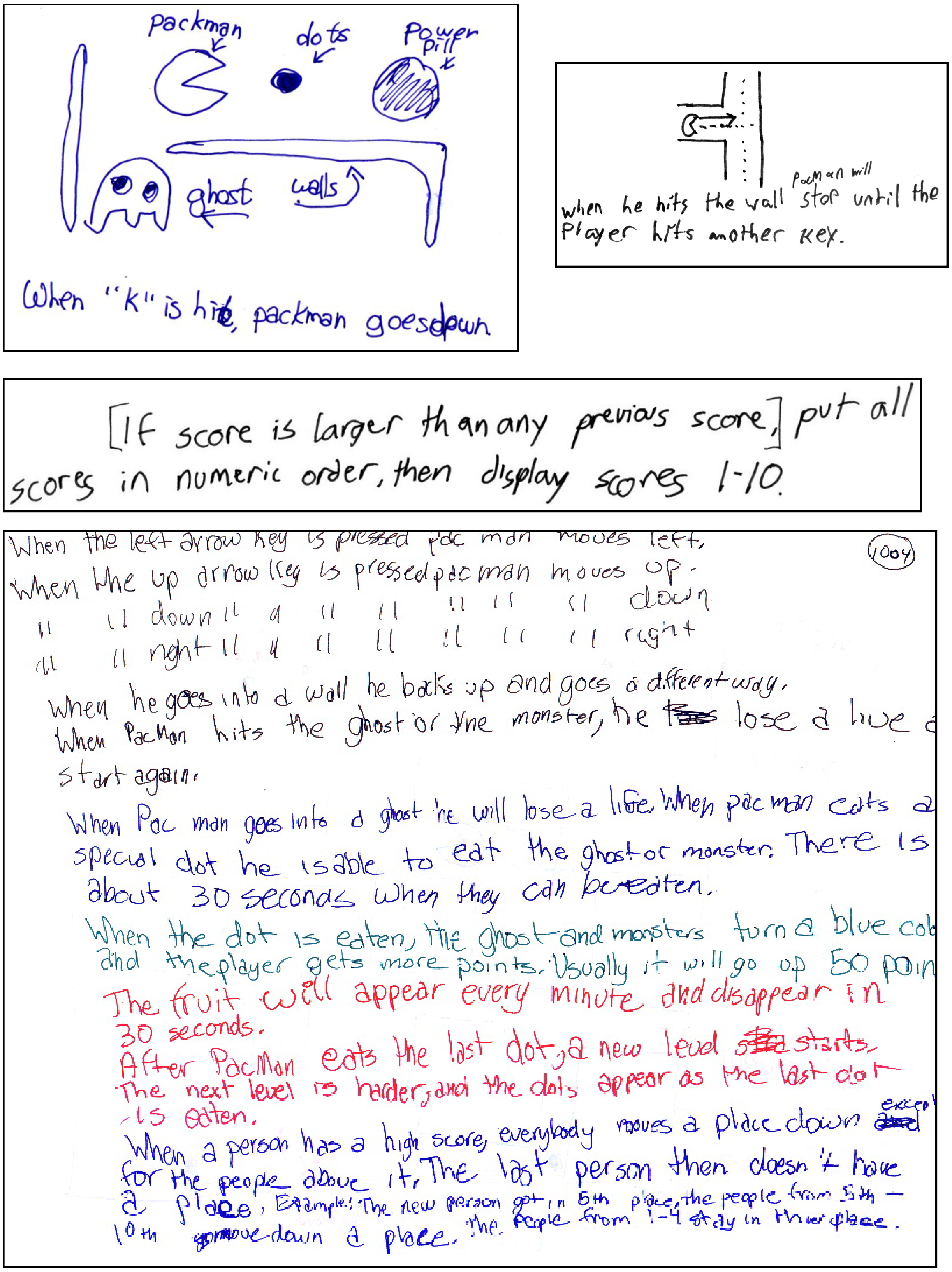
Figure 3-2. A question from the rating form for study one. The nine blanks on each line correspond to the nine tasks that the participants solved.

cess that took each of them 8-10 hours to answer all of the 17 questions for the fourteen participants.

3.3.5 Results

The participants' solutions ranged from one to seven pages of handwritten text and drawings. Some excerpts from the solutions are shown in Figure 3-3. The raters were instructed to use each utterance (statement or sentence) as the unit of text to analyze. Since each rater independently partitioned the text into these units, the total number of tickmarks differed across raters, so the results are normalized by looking at the proportion of the tickmarks credited to each category rather than the raw counts. Although there was variance among the results from individual raters, their ratings were generally similar. So the results are reported as averages across all raters and all of the non-programmer participants.

The results for each rating form question are summarized with an overall *prevalence* score followed by *frequency* scores for each category sorted from most frequent to least frequent. The prevalence score measures the average count of occurrences that each rater classified for each participant when answering the current question. In study one, this score varies from 1.0 to 23.2, indicating the relative amount of data that was available to the raters in


packman
dots
Power Pill
ghost
walls
Pacman will

When "k" is ~~hit~~ hit, packman goes~~d~~down

when he hits the wall Pacman will stop until the Player hits another key.

[If score is larger than any previous score,] put all scores in numeric order, then display scores 1-10.

1004

When the left arrow key is pressed pac man moves left,
when the up arrow key is pressed pac man moves up.
" " down " " " " " " " down
" " right " " " " " " " right
when he goes into a wall he backs up and goes a different way.
When Pacman hits the ghost or the monster, he ~~[illegible]~~ lose a live a
start again.

When Pac man goes into a ghost he will lose a life. When pac man eats a
special dot he is able to eat the ghost or monster. There is
about 30 seconds when they can be eaten.
When the dot is eaten, the ghost and monsters turn a blue col
and the player gets more points. Usually it will go up 50 poin
The fruit will appear every minute and disappear in
30 seconds.
After PacMan eats the last dot, a new level ~~[illegible]~~ starts.
The next level is harder, and the dots appear as the last dot
is eaten.
When a person has a high score, everybody moves a place down ~~and~~ except
for the people above it. The last person then doesn't have
a place. Example: The new person got in 5th place, the people from 5th –
10th ~~go~~ move down a place. The people from 1-4 stay in their place.

Figure 3-3. Excerpts from the participants' solutions in Study 1.

answering the question. The frequency scores then show how those occurrences were apportioned across the various categories, expressed as percentages. The frequencies may not sum to exactly 100% due to rounding errors. The examples are quoted from the participants' solutions. Table 3-1 summarizes the results that follow, which are sorted into four general categories: the overall structure of the solutions, the ways that certain keywords are used, the kinds of control structures that are used, and the methods used to effect various aspects of computation. These results are presented from most frequent to least, which is generally not the order that they appeared on the rating form, and frequencies below 5% do not appear in the summary table.

	Overall Structure	
Programming Style	*Perspective*	*Modifying State*
54% Production rules / events	45% Player or end-user	61% Behaviors built into objects
18% Constraints	34% Programmer	20% Direct modification
16% Other (declarative)	20% Other (third-person)	18% Other
12% Imperative		
		Pictures
		67% Yes
	Keywords	
AND	*OR*	*THEN*
67% Boolean conjunction	63% Boolean disjunction	66% Sequencing
29% Sequencing	24% To clarify or restate a prior item	32% “Consequently”, or “in that case”
	8% “Otherwise”	
	5% Other	
	Control Structures	
Operations on Multiple Objects	*Complex Conditionals*	*Looping Constructs*
95% Set / subset specification	37% Set of mutually exclusive rules	73% Implicit
5% Loops or iteration	27% General case, with exceptions	20% Explicit
	23% Complex boolean expression	7% Other
	14% Other (additional uses of exceptions)	
	Computation	
Remembering State	*Mathematical Operations*	*Insertion into a Data Structure*
56% Present tense for past event	59% Natural language style - incomplete	48% Insert first then reposition others
19% “After”	40% Natural language style - complete	26% Insert without making space
11% State variable		17% Make space then insert
6% Discuss future events	*Motions*	8% Other
5% Past tense for past event	97% Expect continuous motion	
		Sorted Insertion
Tracking Progress	*Randomness*	43% Incorrect method
85% Implicit	47% Precision	28% Correct non-general method
14% Maintain a state variable	20% Uncertainty without using “random”	18% Correct general method
	18% Precision with hedging	
	15% Other	

Table 3-1. Summary of results from the first study. Items with frequencies below 5% do not appear.

3.3.6 Overall Structure

3.3.6.1 Programming Style

The raters classified each statement or sentence in the solutions into one of the following categories based on the style of programming that it most closely matches.

Prevalance: 22.7 occurrences per participant.

- 54% - production rules or event-based, beginning with *when*, *if*, or *after.*
  Example: When PacMan eats all the dots, he goes to the next level.
- 18% - constraints, where relations are stated which should always hold.
  Example: PacMan cannot go through a wall.
- 16% - other (98% of these were classified by the raters as declarative statements).
  Example: There are 4 monsters.
- 12% - imperative, where a sequence of commands is specified.
  Example: Start with this image. Play this sound. Display "Player One Get Ready."

3.3.6.2 Perspective

Beginners sometimes confuse their role or perspective while they are developing a program. Instead of thinking about the program from the perspective of the programmer, they might adopt the role of the end-user of the program, or in the case of games and stories, one of the characters portrayed by the program. The raters classified the participants' statements according to the perspective or role that they indicated.

Prevalance: 23.2 occurrences per participant

- 45% - player's or end-user's perspective.
  Example: When I push the left arrow PacMan goes left.
- 34% - programmer's perspective.
  Example: If arrow for Player 1 is "left" move PacMan left.
- 20% - other (99% of these were classified by the raters as *third-person perspective*).
  Example: If he eats a power pill and he eats the ghosts, they will die.

3.3.6.3 Modifying State

The raters examined places where the participants were making changes to an entity.

Prevalance: 4.6 occurrences per participant.

- 61% - behaviors were built into the entity, in an object-oriented fashion.
  Example: Get the big dot and the ghost will turn colors...
- 20% - direct modification of the properties of entities.
  Example: After eating a large dot, change the ghosts from original color to blue.
- 18% - other.

3.3.6.4 Pictures

In addition to the above classifications done by the raters, the experimenter examined each solution to determine whether pictures were drawn as part of the solution.

- 67% - included at least one picture.
- 33% - used text only.

3.3.7 Keywords

3.3.7.1 AND

The raters examined the intended meaning when the participants used the word *AND*.

Prevalance: 6.3 occurrences per participant.

- 67% - Boolean conjunction.
  Example: If PacMan is travelling up and hits a wall, the player should...
- 29% - for sequencing, to mean *next* or *afterward.*
  Example: PacMan eats a big blinking dot, and then the ghosts turn blue.
- 3% - other
  Example: Every level the fruit should stay for less and less seconds.

3.3.7.2 OR

The raters examined the intended meaning when the participants used the word *OR*.

Prevalance: 1.5 occurrences per participant.

- 63% - Boolean disjunction.

  Example: To make PacMan go up or down, you push the up or down arrow key.

- 24% - clarifying or restating the prior item.

  Example: When PacMan hits a ghost or a monster, he loses his life.

- 8% - meaning *otherwise*.

- 5% - other.

3.3.7.3 THEN

The raters examined the intended meaning when the participants used the word *THEN*.

Prevalance: 2.2 occurrences per participant.

- 66% - sequencing, to mean *next* or *afterward*.

  Example: First he eats the fruit, then his score goes up 100 points.

- 32% - meaning *consequently, or in that case.*

  Example: If you eat all the dots then you go to a higher level.

- 1% - to mean *besides* or *also.*

- 1% - other.

3.3.8 Control Structures

3.3.8.1 Operations on Multiple Objects

The raters examined those statements that operate on multiple objects, where some or all of the objects are affected by the operation.

Prevalance: 6.1 occurrences per participant.

- 95% - set and subset specifications.

  Example: When PacMan gets all the dots, he goes to the next level.

- 5% - loops or iteration.

  Example: #5 moves down to #6, #6 moves to #7, etc. until #10 which is kicked off the high score list.

3.3.8.2 Iteration or Looping Constructs

The raters examined those statements that were either implicit or explicit looping constructs.

Prevalance: 1.6 occurrences per participant.

- 73% - implicit, where only a terminating condition is specified.
  Example: Make PacMan go left until a dead end.
- 20% - explicit, with keywords such as *repeat*, *while*, *and so on*, etc.
- 7% - other.

3.3.8.3 ELSE or Equivalent Clauses

The raters looked for occurrences of *ELSE* clauses or equivalent constructs in the participants' solutions. They simply counted these, without classifying them further.

Prevalance: 0.4 occurrences per participant.

3.3.8.4 Complex Conditionals

The raters examined those statements that specify conditions with multiple options.

Prevalance: 2.3 occurrences per participant.

- 37% - a set of mutually exclusive rules.
  Example: When the monster is green he can kill PacMan. When the monster is blue PacMan can eat the monster.
- 27% - a general condition, subsequently modified with exceptions.
  Example: When you encounter a ghost, the ghost should kill you. But if you have a power pill you can eat them.
- 23% - Boolean expressions.
  Example: After eating a blinking dot and eating a blue and blinking ghost, he should get points.
- 14% - other (95% of these either listed the exception first, or did not list a general case).
  Example: If he gets a [power pill] then if you run into them you get points.

3.3.9 Computation

3.3.9.1 Remembering State

The raters examined the methods used to keep track of state when an action in the past should affect a subsequent action.

Prevalance: 4.1 occurrences per participant.

- 56% - using present tense when mentioning the past event.
  Example: When PacMan eats a special dot he is able to eat the ghosts.
- 19% - using the word *after.*
  Example: After using up the power pill, the ghosts can eat PacMan again.
- 11% - using a state variable to track information about the past event.
  Example: When the monster is blue PacMan can eat the monster.
- 6% - mentioning the future event at the time of past event.
  Example: When PacMan gets a shiny dot, then if you run into the ghosts, you get points.
- 5% - using the past tense when mentioning the past event.
  Example: In about 10 seconds, if PacMan didn't eat it take it off again.
- 4% - other.

3.3.9.2 Tracking Progress

The raters examined the methods used to keep track of progress through a long task.

Prevalance: 2.0 occurrences per participant.

- 85% - all or nothing, where tracking is implicit or done with sets.
  Example: When PacMan gets all the dots, he goes to the next level.
- 14% - using counting, where a variable such as a counter tracks the progress.
  Example: When PacMan loses 3 lives, it's game over.
- 1% - other.

3.3.9.3 Mathematical Operations

The raters examined the kinds of notations used to specify mathematical operations.

Prevalance: 3.4 occurrences per participant.

- 59% - natural language style, missing the amount or the variable.
  Example: When he eats the pill, he gets more points...
- 40% - natural language style, with no missing information.
  Example: When PacMan eats a big dot, add 100 points to the score.
- 0% - programming language style (count = count + 20)
- 0% - mathematical style (count + 20)

3.3.9.4 Motions

The raters examined the participants' expectations about whether motions of objects should require explicit incremental updating.

Prevalance: 7.8 occurrences per participant.

- 97% - expect continuous motion, specifying only changes in motion.
  Example: PacMan stops when he hits a wall.
- 2% - continually update the positions of moving objects.
- 1% - other.

3.3.9.5 Randomness

The raters examined the methods used by the participants' in expressing events that were supposed to happen at uncertain times or with uncertain durations.

Prevalance: 1.4 occurrences per participant.

- 47% - using precision, where no element of uncertainty is expressed.
  Example: Put the new fruit in every 30 seconds.
- 20% - using words other than *random* to express the uncertainty.
  Example: The fruit will go away after a while.
- 18% - using precision with hedging to express uncertainty.
  Example: After around 3 or 4 more seconds the fruit disappears.
- 15% - other (often the action was tied to another event).
  Example: Put a fruit on the screen when PacMan is running out of power.
- 0% - used the word *random.*

3.3.9.6 Insertion into a Data Structure

The raters examined the methods used by the participants to insert an element into the middle of an existing sequence of elements.

Prevalance: 1.0 occurrences per participant.

- 48% - inserting first, repositioning other elements afterwards.
- 26% - no mention of making room for the inserted element.
- 17% - making space by repositioning others, then inserting the element.
- 8% - other.

3.3.9.7 Sorted Insertion

The raters examined the methods used by the participants to determine the correct place to insert an element into a sorted list.

Prevalance: 1.1 occurrences per participant.

- 43% - using an incorrect method, with missing or incorrect details.
- 28% - a method that is correct for the current data, but not a correct general solution.
- 18% - a correct general method that would work for any data.
- 10% - other

3.3.10 Discussion

Combined discussion of the two studies appears in Section 3.5 on page 58.

3.4 Study Two

To see whether the observations from the first study would generalize to other domains and other age groups, a second study was conducted. This study used database access scenarios that are more typical of business programming tasks, and was administered to a group of adults as well as a group of children similar to the participants in study one.

3.4.1 Participants

Nineteen adults from the Carnegie Mellon University community, ranging in age from 18 to 34, participated in the study (10 men, 9 women). In addition, 22 fifth graders, ages 10 or

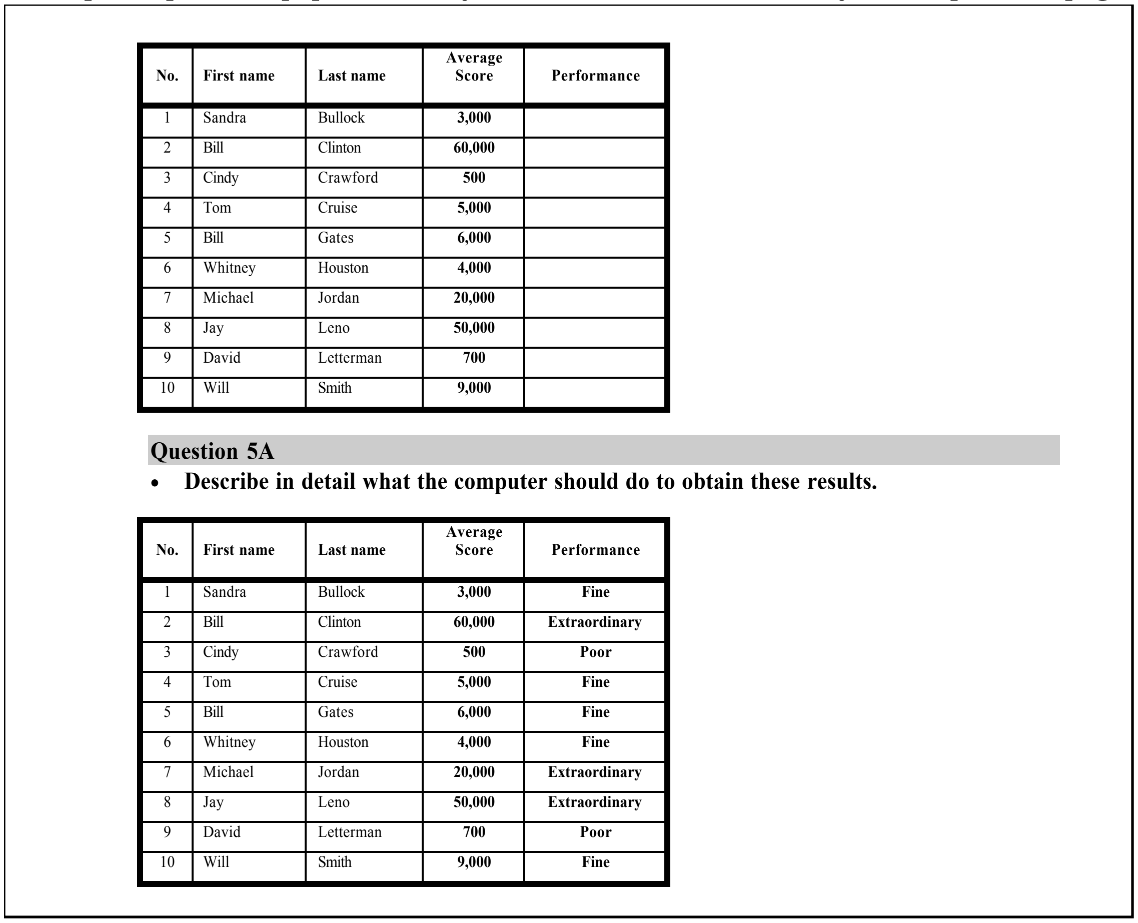
11, participated (13 boys, 9 girls). These fifth graders were recruited from the same Pittsburgh public elementary school as study one, but it was a new academic year so none of the participants from study one were involved in study two. The participants were racially diverse. Although the children spanned a range of academic abilities, all of the Carnegie Mellon participants had strong academic backgrounds.

Of the adults, only five had never programmed before (2 men, 3 women). Of the children, fourteen said they had never programmed before (11 boys, 3 girls). However, there is reason to believe that some of the children who claimed to be programmers did not accurately answer this question because they did not really seem to know what programming is. Nonetheless, only those participants who said they never programmed (5 adults, 14 children) were included in the analysis that follows.

The adult participants were recruited by word of mouth, and signed the usual human subject consent forms. The children were recruited by sending a brief note and consent form to parents. The adult participants received no reward for their participation; the children had an opportunity to leave their normal classroom for a half hour, and were given a snack at the end of their participation.

3.4.2 Materials

A set of eleven scenarios were created, representing a progression of problems that a programmer might encounter in the process of creating and manipulating a database of names and numeric values. These scenarios were chosen to cover some of the essential concepts of programming that were not addressed in study one, and to further elucidate some of the results from that study. As in study one, graphical depictions of these scenarios were developed. In this case they contained before and after pictures of database values in a tabular layout, with graphical annotations highlighting the differences between the before and after pictures, along with a minimal amount of text that was carefully chosen to avoid biasing the participants' responses. The topics of the scenarios were: entering values into the correct rows of a table, adding certain values in each row to produce a column of sums, discarding the smallest or largest value from each row when calculating the sum, assigning nominal values to each row depending on textual attributes or numeric ranges, producing a numerically sorted summary table with entries for only the rows with the highest sums, adding or subtracting a fixed value to every value in a column, deleting rows from the table or adding rows to it, and zeroing all of the values in a column. Figure 3-4 shows one of the scenario depictions, and the rest are shown in Appendix E. The depictions were displayed to the participants on paper, and they wrote their solutions directly on the problem pages.



No.	First name	Last name	Average Score	Performance
1	Sandra	Bullock	**3,000**	
2	Bill	Clinton	**60,000**	
3	Cindy	Crawford	**500**	
4	Tom	Cruise	**5,000**	
5	Bill	Gates	**6,000**	
6	Whitney	Houston	**4,000**	
7	Michael	Jordan	**20,000**	
8	Jay	Leno	**50,000**	
9	David	Letterman	**700**	
10	Will	Smith	**9,000**	

Question 5A

- **Describe in detail what the computer should do to obtain these results.**

No.	First name	Last name	Average Score	Performance
1	Sandra	Bullock	**3,000**	**Fine**
2	Bill	Clinton	**60,000**	**Extraordinary**
3	Cindy	Crawford	**500**	**Poor**
4	Tom	Cruise	**5,000**	**Fine**
5	Bill	Gates	**6,000**	**Fine**
6	Whitney	Houston	**4,000**	**Fine**
7	Michael	Jordan	**20,000**	**Extraordinary**
8	Jay	Leno	**50,000**	**Extraordinary**
9	David	Letterman	**700**	**Poor**
10	Will	Smith	**9,000**	**Fine**

Figure 3-4. Depiction of a problem scenario in study two.

3.4.3 Procedure

The same procedure was used as in study one, except the sessions were not audiotaped.

3.4.4 Content Analysis

Once again a form was developed, similar to the one used in study one, so that independent raters could analyze the data (see Appendix E). This rating form had 18 questions. Because the performance of the five analysts in the first study was satisfactory, there was general agreement among them, and the task was very tedious, it was decided that three analysts were sufficient for the second study. The analysts from the first study were permitted to return for this study because there was no reason to expect their prior participation to have a material affect on the results. Therefore, three analysts from the prior study analyzed the participants' responses in this study.

3.4.5 Results

The participants answers typically consisted of one to five sentences in response to each of the eleven questions. Once again, there was general agreement among the raters. The performance of adults was generally similar to the performance of children. Therefore, the results reported below are averages across the raters (n=3) and all of the non-programmer participants (n=19, 5 adults and 14 children).

As in study one, the results for each question are summarized with an overall *prevalence* score followed by *frequency* scores for each category. The prevalence score measures the average count of occurrences that each rater classified for each participant when answering the current question. In study two, this score varies from 0.2 to 11.5, indicating the relative amount of data that was available to the raters in answering the question. The frequency scores then show how those occurrences were apportioned across the various categories, expressed as percentages. The frequencies may not sum to exactly 100% due to rounding errors. The results are presented from most frequent to least, which is generally not the same order as they appeared on the rating form. The examples are quoted from the participants' solutions. Table 3-2 summarizes the results that follow, which are sorted into three general categories: the ways that certain keywords are used, the kinds of control structures that are used, and the methods used to effect various aspects of computation. In the summary table, items with frequencies below 5% do not appear.

3.4.6 Keywords

3.4.6.1 AND

The raters examined the intended meaning when the participants used the word *AND*.

Prevalance: 6.1 occurrences per participant.

- 47% - Boolean conjunction.

	Keywords	
AND	*OR*	*BUT*
47% Boolean conjunction	100% Boolean Disjunction	92% To mean “except”
43% Sequencing		8% Other
5% Other	*NOT*	
	100% Low precedence	*THEN*
AND as a boolean operator		91% Sequencing
76% Incorrect		7% “Consequently”
24% Correct		
	Control Structures	
Operations on Multiple Objects	*Complex Conditionals*	
97% Sets and subsets, including plurals	45% Set of mutually exclusive conditions	
	36% Dependent clause cannot stand alone	
	16% Nested conditions	
	Computation	
Set Construction	*Specifying Open Intervals*	*Sorting*
46% Plurals	35% “Above” is exclusive	37% “Alphabetical”, etc.
18% “Each” or “every”	22% “Above” is inclusive	36% “From A to Z”, etc.
16% Naming a column of the table	22% Powers of ten	11% Concrete example
14% “All”	15% Other	9% Provide a key to a sort operator
	5% Mathematical notation	
Set Manipulation		*Deleting an Element from a Data Structure*
45% Set inverse	*Specifying Closed Intervals*	73% No hole expected after deletion
29% Set difference	35% “From ... to” is inclusive	25% Repaired a hole after deletion
22% Disjoint or mutually exclusive sets	19% Powers of ten	
5% Other	10% Mathematical notation	*Inserting an Element into a Data Structure*
	9% Other	75% Insert without making space
Complete Specification of Ranges	9% "Between" used inconsistently	16% Make space then insert
50% Correct	7% “From ... to” used inconsistently	6% Insert then make space
50% Incorrect	6% "Between" is inclusive	
	5% Ends of interval specified separately	*Sorted Insertion*
		46% Incorrect method
	Mathematical Operations	34% Correct non-general method
	52% Natural language style - complete	13% Correct general method
	40% Other	6% Insert then sort

Table 3-2. Summary of results from the second study. Items with frequencies below 5% do not appear.

Example: Erase Bill Clinton and Jay Leno.

- 43% - sequencing, meaning *next* or *afterward*.

  Example: Crossed out the highest score, and added the lower scores.

- 5% - other.

- 4% - to specify a range.

  Example: Fine is between 3,000 and 20,000.

3.4.6.2 AND as a Boolean Operator

The raters examined the answers to two questions that were likely to elicit Boolean expressions. If the word *AND* appeared in a Boolean expression, the raters determined whether it was used correctly.

Prevalance: 0.6 occurrences per participant.

- 76% - incorrect, interpreting as Boolean conjunction would not give the intended result.
  Example: Everybody whose name starts with the letter G and L would be in the black group.
- 24% - correct, Boolean conjunction is intended meaning.
  Example: If AvgScore ≥1000 and <10000, say Fine.

3.4.6.3 OR

The raters examined the places where the participants used the word *OR* as a Boolean operator to see if it was used correctly.

Prevalance: 0.6 occurrences per participant.

- 100% - correct.
  Example: [Score] in the hundreds or less is poor.

3.4.6.4 NOT

The raters examined the places where the participants used the word *NOT* as a Boolean operator to see what operator precedence was intended.

Prevalance: 0.1 occurrences per participant.

- 100% - low precedence: *NOT A or B* means *NOT (A or B)*.
  Example: The Gold group [contains the people] with the first two letters in their last name that are not Le or Ga.

3.4.6.5 BUT

The raters examined the intended meaning when the participants used the word *BUT*.

Prevalance: 0.2 occurrences per participant.

- 92% - to mean *except.*

Example: Add every element in the row, but the maximum.

- 8% - other.
- 0% - to mean *and*.

3.4.6.6 THEN

The raters examined the intended meaning when the participants used the word *THEN*.

Prevalance: 1.3 occurrences per participant.

- 91% - sequencing, to mean *next* or *afterward*.
  Example: Add up all the scores in each row, then subtract the lowest score in each row.
- 7% - to mean *consequently, or in that case*.
  Example: If their name begins with a G or an L then put them in the Black group.
- 1% - besides or also.
- 1% - other.

3.4.7 Control Structures

3.4.7.1 Operations on Multiple Objects

The raters examined statements that operate on multiple objects, where some or all of the objects are affected by the operation.

Prevalance: 11.5 occurrences per participant.

- 97% - set or subset specifications, including the use of plurals.
  Example: Select the four highest scores of the participants.
- 3% - loop or iteration.
  Example: Match the last name and fill the score until there is no more input.
- 1% - other.

3.4.7.2 Complex Conditionals

The raters examined statements specifying conditions with multiple options.

Prevalance: 1.3 occurrences per participant.

- 45% - a set of mutually exclusive conditions.

Example: If Average Score is less than 1000, performance is poor. If Average Score is between 1000 and 10000, performance is fine. If Average Score is more than 10000, performance is extraordinary.

- 36% - a condition with a dependent clause that cannot stand alone.
  Example: If the people’s last name start with G or L they are on the black team. If not they are on the gold team.
- 16% - nested conditions
  Example: If average score is in the hundreds it's poor. Less than ten thousand is fine.
- 3% - other.

3.4.8 Computation

3.4.8.1 Set Construction

The raters examined the places where sets are used, to determine how those sets were constructed.

Prevalance: 11.0 occurrences per participant.

- 46% - using plurals.
  Example: Add the scores of 3 rounds.
- 18% - using the words *each* or *every.*
  Example: Add the score in every round.
- 16% - naming a column of the table.
  Example: Add 10,000 points to Round 1 and Round 3.
- 14% - using the word *all*.
  Example: Subtract [20,000 from] all elements in Round 2...
- 4% - enumerating the members of the set.
- 1% - other.

3.4.8.2 Set Manipulation

The raters examined the ways that subsequent sets are created after an initial related set has been created.

Prevalance: 2.7 occurrences per participant.

- 45% - using set inverse, where the leftover items are operated on.
  Example: If the last name begins with G or L, they are in the Black group. The rest are in the Gold group.
- 29% - set difference, where some items are removed from the specified set.
  Example: Add all the Rounds up except the highest score to get TOTAL.
- 22% - constructing disjoint or mutually exclusive sets.
  Example: Black is for G and L. Gold is for B, C, H, J, and S.
- 5% - other.

3.4.8.3 Complete Specification of Ranges

The raters examined the participants' statements that specify a range of integers, to see whether all of the possibilities were covered without holes or overlaps.

Prevalance: 1.3 occurrences per participant.

- 50% - correct.
  Example: Scores below 1000 are Poor. Scores from 1000 - 10,000 are Fine. Any scores above 10,000 are Extraordinary.
- 50% - incorrect.

3.4.8.4 Specifying Open Intervals

The raters examined the participants' statements specifying open intervals, where all values beyond a single boundary are specified.

Prevalance: 2.0 occurrences per participant.

- 36% - words such as *above, below, greater than* or *less than* were intended to be exclusive.
  Example: The performance of the person with the average scores below 1000 is considered as poor (the participant then used *good* for 1000).
- 22% - words such as *above, below, greater than* or *less than* were intended to be inclusive.

Example: Poor would be below 999 (the participant then used *poor* for 999).

- 22% - powers of ten were used to specify the range.
  Example: If your score is in the hundred's your performance is poor.
- 15% - other.
- 5% - mathematical notation, with inequality operators such as “>” or “≤”.
  Example: if score < 1000, performance = poor.

3.4.8.5 Specifying Closed Intervals

The raters examined the participants’ statements specifying closed intervals, where both boundaries are specified for a range of values.

Prevalance: 1.2 occurrences per participant.

- 35% - *from ... to*, the symbol “-”, or similar notations are intended to be inclusive.
  Example: The performance of ones whose average scores from 1000 up to 10,000 is considered as a fine performance (the participant then assigned *fine* to both 1000 and 10,000).
- 19% - powers of ten were used to specify the range.
  Example: If your score is in the thousands, you are fine.
- 10% - mathematical notation, with inequality operators such as “>” or “≤”.
  Example: 1000 < x < 9999; performance = fine.
- 9% - other.
- 9% - *between* is used with an inconsistent meaning at each end of the interval.
  Example: If the average score is between 1000 and 10,000, the performance is fine (the participant then assigned *fine* to 1000, and *extraordinary* to 10,000).
- 7% - *from ... to*, the symbol “-”, or similar notations are used with an inconsistent meaning at each end of the interval.
  Example: Scores from 1000 - 10,000 are fine (the participant then assigned *fine* to 1000, and *extraordinary* to 10,000).
- 6% - *between* is intended to be inclusive.

Example: Score between 1000 and 10,000 is fine (the participant then assigned *fine* to both 1000 and 10,000).

- 5% - specified each end of the interval separately.
- 0% - *between* is intended to be exclusive.
- *0% - from ... to*, the symbol “-”, or similar notations are intended to be exclusive.

3.4.8.6 Mathematical Operations

The raters examined the kinds of notations used by the participants’ in specifying mathematical operations.

Prevalance: 5.4 occurrences per participant.

- 52% - natural language style, with no missing information.
  Example: Add 10,000 points to the scores in Round 1 and Round 3.
- 40% - other (which includes natural language style, with missing amount or variable).
  Example: Add up the scores of each person but don't add the highest number (missing variable).
- 4% - mathematical notation.
  Example: Column for r2 = x - 20000.
- 4% - programming language notation.

3.4.8.7 Sorting

The raters examined the participants’ solutions to see how sorting operations were expressed.

Prevalance: 1.3 occurrences per participant.

- 37% - using keywords such as *alphabetical* or *numerical*.
  Example: Sort the table alphabetically.
- 36% - using expressions like *from A to Z* or *from lowest to highest*.
  Example: Put the 4 highest scores ... in a different table from highest to smallest.
- 11% - using a concrete example from the current situation.

Example: Put him in number 6 because his last name comes before Jordan but after Houston.

- 9% - using a sort key, such as *sort according to score*.
  Example: Insert Elton John in order of the last name.
- 4% - using words like *ascending* or *descending.*
  Example: Sort "total score" column in descending order.
- 4% - other.

3.4.8.8 Deleting an Element from a Data Structure

The raters examined the methods used to delete an element from the middle of an existing sequence of elements, to see whether they expected a hole to be left behind.

Prevalance: 1.0 occurrences per participant.

- 73% - no hole was expected after the deletion.
  Example: Take out Bill and Jay then put Elton John in.
- 25% - fixed a hole after the deletion.
  Example: Delete Row 2 and 8, moving everyone down to any unoccupied Rows.
- 2% - other.

3.4.8.9 Insertion into a Data Structure

The raters examined the methods used to insert an element into the middle of an existing sequence of elements to see whether they expected that items would have to be arranged to make space for the new element.

Prevalance: 1.0 occurrences per participant.

- 75% - no mention of making room for the new element.
  Example: Put Elton John in the records in alphabetical order.
- 16% - make room for the element before inserting it.
  Example: Use the cursor and push it down a little and then type Elton John in the free space.
- 6% - make room for the element after inserting it.

- 4% - other.

3.4.8.10 Sorted Insertion

The raters examined the methods used to determine the correct place to insert an element into a sorted sequence of elements.

Prevalance: 1.0 occurrences per participant.

- 46% - using an incorrect method, with missing or incorrect details.
  Example: Insert row between number 5 and 7 and name it Elton John.
- 34% - a method that is correct for the current data, but not a correct general solution.
  Example: Put him in number 6 because his last name comes before Jordan but after Houston.
- 13% - a correct general method that would work for all data.
  Example: Insert Elton John into the table in alphabetical order of the last name.
- 6% - insert then sort.
  Example: Add Elton John, and then sort the table alphabetically.
- 2% - other.

3.5 Discussion of Results

This section contains a discussion of the combined results from the two studies. In addition to interpretation of the results, this section includes some recommendations on how the programming system might be made more natural.[1]

3.5.1 Programming Style

The majority of the statements written by the participants were in a production-rule or event-based style, beginning with words like *if* or *when*. However, the raters observed a significant number of statements using other styles, such as constraints, other declarative statements (that were not constraints), and imperative statements.

1. These recommendations are not restricted to the programming *language* in isolation, but encompass the entire programming *system*, which includes the programming environment (editor, debugger, etc.) as well as the language. In modern programming systems these components all work in tandem, so it is most useful to consider how the findings of this study might impact the entire system.

The dominance of rule- or event-based statements suggests that a primarily imperative language may not be the most natural choice. One characteristic of imperative languages is explicit control over program flow. Although imperative languages have *if* statements, they are evaluated only when the program flow reaches them. The participants’ solutions seem to be more reactive, without attention to the global flow of control. When imperative statements were used, it was usually for local flow of control. The declarative style seems to have been primarily used for setting up the scenario (data, characters, objects, etc.) of the program. Many of the constraints that were observed in this study were graphical in nature, such as objects that had certain fixed positions relative to one another, or limitations on where those objects could go. The event-based style is used by several popular end-user programming environments such as Visual Basic, Lingo for Macromedia’s Director, and HyperTalk for HyperCard, although these systems have usability problems of their own [see, for example, Thimbleby 1992].

This mix of styles suggests that designers might be able to improve usability by not limiting the language to a single style. Different styles seem to be more natural for different parts of the programming task.

HANDS supports an event-based style, with imperative statements for local flow of control (see Section 5.3 on page 96). Data, characters and objects can be created declaratively (see Section 5.2 on page 89).

3.5.1.1 Operations on Multiple Objects

The results from both studies highlight an important area where today’s popular programming languages differ from the natural expressions used by the participants: the way that operations are performed on multiple objects. The participants strongly preferred to use set and subset expressions, or plurals, to specify the operations in aggregate. Miller made similar observations in his studies [Miller 1974, Miller 1981].

Although aggregate operations have appeared in some languages, such as Lisp, APL, SETL, and Perl, most popular languages require iterative operation on the objects, one at a time. It has been well established in the literature that loops are a hotspot of difficulty and errors for novice programmers [du Boulay 1989a]. And in many cases, a loop is a more complicated and contorted way to specify operations that the participants were able to express easily and succinctly with aggregate set operations. Languages should support these aggregate operations, thus eliminating many of the cases where loops would otherwise be necessary.

Another requirement imposed by loops is the need to use extra variables to count iterations, flag terminating conditions, or hold the current object being operated upon. This is even true in “high level” looping constructs such as *mapcar* in Lisp. The aggregate operations preferred by the participants reduce the need for these variables, which are another known area of difficulty for beginners [du Boulay 1989a]. Spreadsheets provide a few aggregate operators, such as *sum*, but this feature is not generalized across all of the operators.

However, the participants did use looping constructs in a few cases, and the language should support these as well. Often, these loops use *until* to specify a terminating condition, while other times the terminating condition is implicit in phrases such as *and so on* or *etc*. In deciding the exact loop control structures to provide, the language designer should consider prior empirical studies which found that novices expect the terminating condition to be checked continuously, and the loop to halt the instant the condition is satisfied, rather than waiting until all of the subsequent statements inside the loop have been executed one last time [du Boulay 1989a].

HANDS supports aggregate operations (see Section 5.9 on page 112), and also provides a high-level iteration construct (see Section 5.13 on page 119).

3.5.1.2 Set Construction and Manipulation

Study two illustrates a variety of ways that the participants construct sets: using plurals, the keywords *each*, *every* or *all*, or by naming columns in a table. Once they had created a set, the participants often used operations such as inverse or difference to create related sets. However, they sometimes preferred to create a separate disjoint set from scratch.

HANDS supports queries for the creation of sets of objects (see Section 5.10 on page 113).

3.5.1.3 Complex Conditionals and NOT

The participants used a number of ways to avoid writing complex Boolean conditionals. For example, they often wrote a series of mutually exclusive simple rules instead of a more complex conditional.

Also, they would sometimes express a general case followed by exceptions, as in:

```
if A do something unless B
```

Notice that the equivalent Boolean expression that would be required to accomplish this in many programming languages involves not only a conjunction, but also the negation of the exception clause:

```
if A and not B do something
```

The try...catch exception mechanisms in C++, Java, Lisp and other languages support this tendency by putting the general case first and listing the exceptions later, but other control structures in these languages do not. It might be useful to support the use of *unless* clauses throughout the language.

The raters found very few uses of negation. This is consistent with earlier findings that expressing negative concepts is more difficult than affirmative ones [Wason 1959].

When the participants did use the *not* operator they gave it low precedence, which is contrary to the precedence that it has in most programming languages. Operator precedence errors were among the high frequency bugs observed by Spohrer & Soloway in novice programs in a traditional programming language [Spohrer 1986]. However, in a recent study of a natural language style programming language, Bruckman & Edwards found that operator precedence errors were very infrequent [Bruckman 1999].

Chapter 4 explores these issues further and details *match forms*, which avoid many of the problems with forming expressions with the Boolean operators.

3.5.1.4 Mathematical Operations

In study one, all of the mathematical operations were expressed in a natural language form; the raters found no mathematical or programming language notations. In study two, they found a very small amount of mathematical and programming notations among the adults' solutions. The vast preference for natural language mathematical operations should be supported by the programming language. However, more concise mathematical notation may be still necessary for calculations that are more complex than the ones required by the tasks in these studies.

Many of the mathematical expressions were missing either the variable on which to operate, or the amount of the operation. This might be solved by providing slots that make the missing information more obvious, or by entering into a dialog with the user, with questions such as *how much?* or *to what?*

HANDS provides both natural language and mathematical notations for mathematical operations (see Section 5.2 on page 89).

3.5.1.5 Specifications of Ranges and Intervals

In study two, the raters found that the participants were only about 50% successful in specifying ranges without holes or overlaps. Adults were more successful than children, possibly because they had mathematical notations for inequality in their arsenal. The children never made use of these mathematical notations. Instead they used powers of ten, or natural language expressions of inequality such as *above* or *greater than*. However, the participants were inconsistent about whether these latter terms were inclusive or exclusive. Adults achieved 100% accuracy when they used mathematical notations, suggesting that these are a better choice for audiences that understand them.

HANDS uses mathematical notation since it is more accurate (Section 5.8.1 on page 110).

3.5.1.6 Tracking Progress and Remembering State

The participants often avoided the use of variables to track progress in a task. This is not surprising because, as mentioned above, variables are an area of difficulty for novice programmers [du Boulay 1989a]. Instead of variables, the participants preferred to use terms like *all* or *none* to detect when the task is finished. When they needed to use historical information to make decisions about present actions (or present information to make decisions about future actions), the participants usually did not use state variables to record the information. Instead they used future and past tenses to refer to the needed information. State variables are the only way accomplish this in most programming languages. The challenge for language designers is to find ways to accommodate the more natural preferences.

In HANDS, queries can be used to determine when all objects, or no objects, meet certain criteria (see Section 5.10 on page 113).

3.5.1.7 AND, OR and BUT

The raters found that often the word *and* was used as a sequencing word rather than as a Boolean operator. Also, in study two, the raters examined the Boolean uses of *and,* and found that 75% were used in situations where the *or* operator would be required to achieve the desired effect in today's programming languages, as well as the query languages used for most database search engines. For example, a subject said, "if you score 90 and above," but the score cannot simultaneously be 90 *and* greater than 90. Because the natural uses of *and* have such diverse meanings, and most of them are inconsistent with the Boolean operator, designers of future language should consider substituting a different name or symbol for this operator.

Or and *but* appeared too rarely in these studies to draw firm conclusions without further research. When *or* was used, a Boolean interpretation would result in correct results. The infrequent use of *or* may be because disjunctive expressions are cognitively more difficult than conjunctive ones [Bourne 1966].

Chapter 4 explores these issues further.

3.5.1.8 THEN

The raters found that the most popular use of the word *then* is for sequencing, or specifying that an action should happen after finishing a prior action. This is inconsistent with its use in most programming languages, where it means *consequently*. This confirms an earlier observation by du Boulay [du Boulay 1989a].

HANDS does not address this problem.

3.5.1.9 Data Structure Operations: Insertion, Deletion, Sorting

When the participants were inserting and deleting data elements, they often did not consider issues about storage space that come up when working with the array data structures in most popular programming languages. This suggests that a built-in list-like data structure such as in the Lisp language may be more natural.

The participants seemed to expect sorting to be a basic operator that they could utilize in their solutions, using expressions like *alphabetical* or *from A to Z*. When they were asked to provide an algorithm for sorting, they were rarely able to do this in a correct general way.

HANDS provides list data structures, and high-level list operators to accomplish tasks such as sorting (see Section 5.4 on page 101, and Section 5.12 on page 114).

3.5.1.10 Randomness and Uncertainty

The raters did not find any uses of the word *random* in study one. Instead, the participants either expressed things with precision, or used other ways of expressing uncertainty. Sometimes they tied the uncertain event to some other event that would happen at some unknown time. Perhaps a system could supply the uncertainty that is implicit in phrases like *about 3 seconds*.

HANDS provides a basic random operator, but no other methods for expressing uncertainty.

3.5.1.11 Object Oriented

Some aspects of object-oriented programming were apparent in the participants' solutions. Entities were treated as if they have state and an ability to respond to requests for action. However, there was no evidence in these studies of other aspects of object-oriented programming such as inheritance or polymorphism. Cypher & Smith found in user studies that inheritance hierarchies cause difficulty for children [Cypher 1995]. Even among professional programmers, researchers have found that full-fledged object-oriented programming is not necessarily natural [Détienne 1990, Glass 1995].

In HANDS, objects encapsulate state, but not code (see Section 5.18 on page 143). There is no inheritance mechanism. However, operators are provided for making copies of existing objects, so a programmer could use a prototype-instance style for managing objects.

3.5.1.12 Motion and Other Domain Specific Needs

The participants expected objects to move on their own, so their behaviors were similar to real-world objects. This is in contrast to the incremental way that animation is accomplished in many systems. This may not come up on all programming tasks, and thus might not be considered a language issue. But similar issues can arise in other domains, and the usability of the programming system can benefit from analysis of the specific needs of the particular domains in which it will be used [Green 1990]. One way to do this in a general way is to allow the language to be customized with domain-specific features.

HANDS provides domain-specific support for interactive graphical programs (see Section 5.14 on page 123).

3.5.1.13 Pictures

In study one, the experimenter counted how many participants used pictures or diagrams in their solutions, and found that two-thirds of them did. All of these pictures appeared early in the solutions, when setup and layout were being defined. Programming systems should accommodate this form of graphical specification in addition to textual specification.

HANDS permits setup and layout to be done by direct manipulation (see Section 5.2 on page 89).

3.6 Summary of These Studies

A large part of the programming task is to take a mental plan for solving a problem and transform it into the particular programming language being used. These studies attempt to capture these plans before they undergo the transformation into a programming language. Ideally, the distance between the plans and the programming language should be minimized. However, these studies identify many places where an unnecessarily large gap is imposed by the features and requirements of today’s programming languages.

Programming is a task of precision, and one reason that the programming languages may differ from these natural language solutions is that programming languages are more formal and facilitate the expression of solutions with more precision. Indeed, there is a large amount of imprecision and underspecification in the participants’ work, and it is important to find ways to help beginners to make their specifications more complete. In many cases, however, the structure and algorithms of the natural language solutions are satisfactory, but are in a different style than is allowed in today’s programming languages.

HANDS has been influenced significantly by these studies. For example, it supports an event-based style of programming as well as aggregate data access and queries for creating the sets to be operated on. In order for queries to be effective, it is necessary to improve the accuracy of query specification, but these studies have shown many serious problems with the Boolean operators *and*, *or* and *not*. Chapter 4 proposes and tests several alternatives to textual Boolean expressions.

These studies, along with the results of other human-centered research about programming, are resources that can be used to guide and evaluate programming language designs. In addition to HANDS, this approach could result in effective new language designs for other domains where it would be useful for non-programmers to have the capabilities of programming.

CHAPTER 4

Methods for Expressing Queries

The studies described in Chapter 3 suggest that programming languages should provide mechanisms for users to perform queries. This capability would reduce the need for the creation, maintenance, and traversal of data structures. However, the accurate specification of queries as Boolean expressions is a notorious problem area in programming languages and other activities such as web searching, library catalog searching, and other database retrieval tasks [Hildreth 1988]. This chapter introduces and evaluates a new tabular query form that I invented to avoid many of the common problems with the Boolean operators, and also explores the effectiveness of alternative textual methods for specifying queries.[1]

4.1 Overview

Despite the great difficulty that users have demonstrated with using the operators *AND, OR,* and *NOT* to construct Boolean expressions, no universally better alternatives have been discovered. Therefore most programming languages continue to rely on them, including many visual and forms-based languages (e.g., [Hays 1995, Pictorius 1996]). Early web search engines also used these operators, although many have now turned to less expressive query languages (for example, the plus and minus unary operators for inclusion and exclusion).

1. Portions of this chapter were previously reported in [Pane 2000].

Newsweek reported in 1999 that even with these simplifications, most web users are dissatisfied with search engines, and less than 6% manage to use these operators in their searches [Tanaka 1999]. Google has become very successful by returning useful results when users enter simple keyword searches, without any operators at all.

The problems with Boolean queries are exemplified in the studies described in Chapter 3. For example, it was very common for participants to use the word *AND* where the word *OR* is the correct Boolean operator. Instead of saying something like “count the cars with license plates from Georgia **or** Louisiana” they would say “count the cars with license plates from Georgia **and** Louisiana.” The latter version refers to an empty set of license plates when interpreted according to Boolean logic, but in English it is usually interpreted to mean the union of the two states’ license plates. This ambiguity in how to interpret the word that means “and” also appears in many other natural languages.

These studies also found that the words *OR* and *NOT* rarely appeared, suggesting that Boolean expressions are not natural. The participants often used other words and sentence structures to specify their queries accurately. For example, rather than saying “if I get up late and I’m not very hungry I skip breakfast,” they might say “if I get up late I skip breakfast unless I’m very hungry.” This latter construction avoids both the *AND* and *NOT* operators.

This chapter describes a study that examines several of these alternative formulations to see whether they are more usable than traditional Boolean expressions. In addition, because prior research suggests that non-textual query languages may be more effective than textual syntaxes [Young 1993], the study compared these textual alternatives against a proposed new query language that uses tabular forms, which could be integrated into a primarily textual programming language.

The study used a grid of nine colored shapes, where a subset of the shapes could be marked (see Figure 4-1). The participants were given two kinds of problems: *code generation* problems, where some shapes were already marked and they had to formulate a query to select them; and *code interpretation* problems, where they were shown a query and had to mark the shapes selected by the query. They solved all of these problems twice, once using purely textual queries, and once using the proposed tabular forms. Additional examples appear in Figure 4-3, and the full materials from this study can be found in Appendix F.

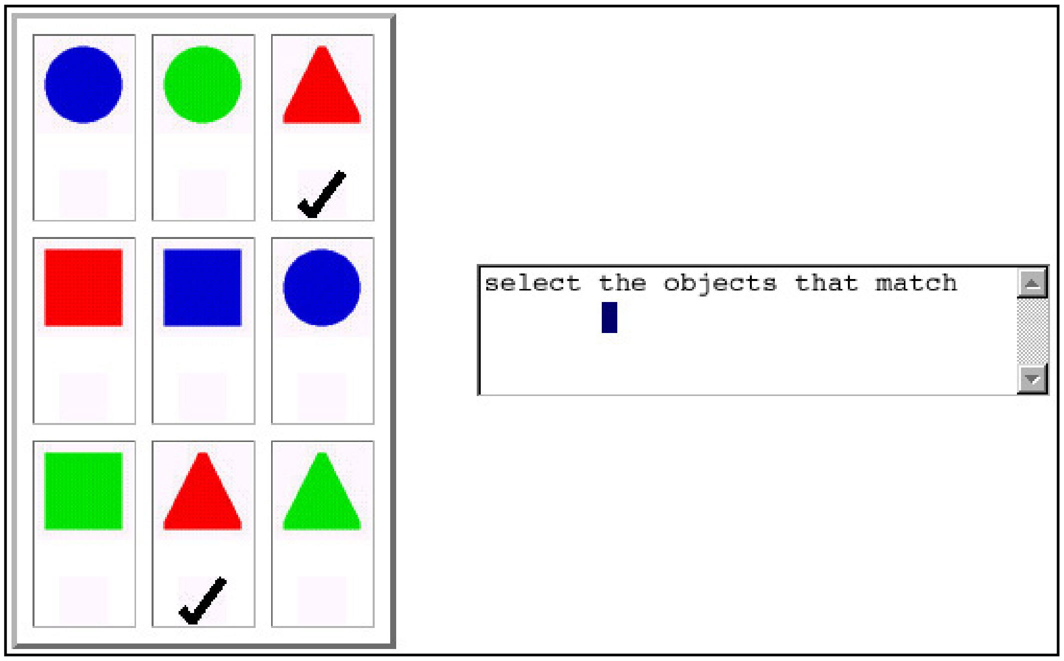


Figure 4-1. Sample problem from the study. In this problem, the participant is asked to write a textual query to select the objects that are marked. The color of each object is red, green or blue on the computer screen.

The results suggest that a tabular language for specifying Boolean expressions can improve the usability of a programming or query language. On code generation tasks, the participants performed significantly better using the tabular form, while on code interpretation tasks they performed about equally in the textual and tabular conditions. The study also uncovered systematic patterns in the ways participants interpreted Boolean expressions, which contradict the typical rules of evaluation used by programming languages. These observations help to explain some of the underlying reasons why Boolean expressions are so difficult for people to use accurately, and suggest that refining the vocabulary and rules of evaluation might improve the learnability and usability of textual query languages. A general awareness of these contradictions can help designers of future query systems adhere to the HCI principle to speak the user’s language [Nielsen 1994].

4.2 Prior work on Boolean Queries

Many researchers have noted that Boolean query languages using the *AND*, *OR*, and *NOT* operators are not very effective in programming languages or database retrieval (e.g. [Hildreth 1988, Hoc 1989]). Several researchers have noted that the common usage of these operators in natural language causes errors in queries, such as the substitution of *AND* for *OR* [Greene 1990, Michard 1982]. It has also been noted that the intended scope of the *NOT* operator is ambiguous in natural language [McQuire 1995].

The difficulties of Boolean expressions are intensified when several operators must be combined to form the query [Essens 1992]. Parentheses improved performance in that study, but other studies have shown that beginners have difficulty with parentheses, especially if they are nested [Greene 1990, Michard 1982].

Replacing the Boolean query language with a different subset of natural language, using other words for the operators, is still likely to be inadequate [Kohl 1987]. Many systems that permit unrestricted natural language queries have been shown to be effective for information retrieval tasks (e.g. [Turtle 1994]).

These problems have led researchers to develop graphical interfaces for queries. For example, truth tables and Venn diagrams have been shown to be effective for specifying simple queries [Jones 1998, Michard 1982, Thomas 1975]. Another system used tiles in a two-dimensional grid, where one dimension represented union and the other represented intersection, although these implicit semantics were found to be confusing [Anick 1990]. A system that used the graphical metaphor of water flowing through filters was found to be superior to Boolean expressions [Young 1993], however the screen space required for this tool might limit its effectiveness in a larger context such as a programming language.

4.3 Design alternatives for Boolean queries

My earlier studies (Chapter 3) analyzed the natural language solutions that non-programmers provided to solve programming problems, and identified some common trends in the ways that Boolean queries were expressed. The vocabulary and syntax of these solutions were unconstrained, so they provide insight into how people prefer to express their answers. I speculated that a programming language that closely matches these natural preferences would be more usable than one that requires users to translate their natural solutions into a less natural form. With this in mind, I proposed several alternate ways to express textual queries and compared them in this study. In addition, I also proposed a tabular format for queries.

4.3.1 Tabular query forms

Although some graphical query methods had been shown to be more effective than Boolean expressions, many of them were limited to expressing very simple queries. I wanted a solution that is fully expressive. Also, many of the graphical systems would not integrate well into a programming language, where the entire computer screen cannot be devoted to this one subtask of the programming process. I required a format that is compact and readable in the context of a larger program. With these points in mind, I designed a tabular form that is fully expressive and compatible with the programming language I was developing.

Since the HANDS programming language represents data on *cards* containing attribute-value pairs, I designed the query form to also use a card metaphor. For the purposes of this study, I simplified the forms by leaving out the attribute names, and limiting the number of terms to three. I called these *match forms* (see Figure 4-2). Criteria are placed into the slots, one term per slot. All of the terms on a single form implicitly form a conjunction. Negation is specified by prefacing a term with the *NOT* operator. Disjunction is specified by including an additional match form adjacent to the first one.

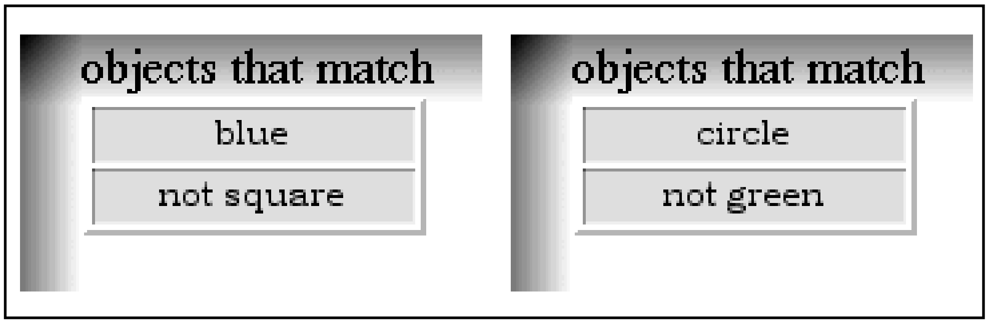


Figure 4-2. Match forms expressing the query: (blue and not square) or (circle and not green)

This two-dimensional layout is similar to the grid of tiles described by Anick et al. [Anick 1990] – one dimension implements intersection and the other implements union. However, match forms provide cues to help users remember which operator uses each dimension, such as the text in the form heading and the visual grouping. In addition, the scope of the NOT operator is made explicit by confining it to a single term. This proposed query language can express arbitrarily complex queries, although some queries have to be formulated in a less concise way than pure Boolean expressions would allow.

4.4 Hypotheses

The study tests nine hypotheses. The first seven hypotheses examine various textual alternatives to traditional Boolean expressions, and the last two hypotheses examine the tabular design alternative.

4.4.1 AND vs. nested IF

Hypothesis 1: Users will interpret nested IF statements more accurately than a Boolean expression using AND.

In the prior studies, people frequently nested an *IF* statement inside another *IF* statement. Instead of saying, “if a and b then ... ,” they would say, “if a then if b then ... .” The use of nested *IF*s may be easier to use and understand because it avoids using the confusing *AND* operator for conjunction, and keeps the Boolean expression simpler.

4.4.2 NOT vs. Unless

Hypothesis 2: Users will interpret an Unless clause more accurately than a Boolean expression that uses AND and NOT.

In the prior studies, people often wrote a simple conditional statement and then stated an exception at the end. For example, they would write, “if a then ... unless b” This is an alternative to “if a and not b then ... .” In addition to avoiding the *AND* operator, the *Unless* clause permits the user to express a negated term without using the *NOT* operator.

4.4.3 Location of Unless

Hypothesis 3: Users will interpret an Unless clause more accurately when it appears at the very end of the statement.

Although in English it may be natural to say “if a then ... unless b,” in programming languages those ellipsis (...) may be filled with a large block of code. If the *Unless* clause will appear at the very end of the *IF* statement, it will be far removed from the part of the query that is specified in the *IF* clause. Because this violates the principle of locality [Cordy 1992], it may reduce usability. While the principle of locality could not be tested directly with my simple stimuli, I wanted to investigate whether the *Unless* clause is sensitive to its placement within the query, so I also tested queries of the form “unless b, if a then ...”

4.4.4 Context-dependent interpretation of AND

Hypothesis 4: Users will interpret AND as Boolean conjunction in some contexts but not in other contexts.

People often use *AND* in places where the correct Boolean operator is *OR*. This may be because interpretation of *AND* in the English language depends on its context. In some cases it is interpreted to be a further restriction on a query (Boolean conjunction or set intersection), while in other cases it is interpreted to expand the query (Boolean disjunction or set union). For example, these two statements are usually interpreted differently: “pick up the boxes that are blue and green” vs. “pick up the boxes that are blue and the boxes that are green.” I attempted to demonstrate this context-sensitive interpretation of the *AND* operator.

4.4.5 Verbose AND vs. OR

Hypothesis 5: Users will interpret a verbose AND expression as Boolean disjunction more accurately than an OR expression.

If Hypothesis 4 is confirmed, it would be useful to characterize the contexts in which *AND* is interpreted as a Boolean disjunction instead of conjunction. If certain constructions consistently lead to disjunctive interpretations, perhaps they can reliably replace the rarely-used *OR* operator. I hypothesized that a more verbose expression that restates part of the query is more likely to induce a disjunctive interpretation (see the example in Section 4.4.4).

4.4.6 Operator precedence of NOT

Hypothesis 6: Users will interpret the NOT operator with lower precedence than the other Boolean operators.

People often interpret the *NOT* operator with lower precedence than the other Boolean operators. This is opposite to the rules of interpretation in most programming languages, where *NOT* has higher precedence than the other Boolean operators. That is, in “not a and b,” programming languages associate the *NOT* tightly with the “a”, while I expect people to first interpret the expression “a and b” and then apply the *NOT* operator to the result.

4.4.7 Parentheses for expression grouping

Hypothesis 7: Users will misinterpret parenthesized expressions.

Regardless of the precedences chosen for the Boolean operators, a mechanism is required for the user to clarify or override them. Programming languages typically use parentheses to explicitly group sub-expressions, but research has shown that beginners have difficulty with parentheses.

4.4.8 Tabular vs. textual

Hypothesis 8: Users will interpret queries that use match forms more accurately than equivalent textual queries.

Hypothesis 9: Users will generate more accurate queries using match forms than they generate using text.

I investigated the relative usability of match forms compared with text on both interpretation and generation of queries. I expected match forms to be effective because they eliminate many of the problems with text that are discussed above. By avoiding the words *AND* and *OR*, any confusion with the meaning of these words in English is avoided. Also, the precedence or grouping of the operators becomes less ambiguous.

4.5 Method

Before beginning the study, participants filled out a questionnaire that collected basic demographic information. Then they answered a set of problems that were divided into four sections. In two of the sections, the *writing* sections, the participants generated queries to match a result that I supplied. In the other two sections, the *reading* sections, participants interpreted queries that I supplied. I label the two writing sections *WT* (writing text) and *WF* (writing forms), and the two reading sections *RT* (reading text) and *RF* (reading forms).

There were five *WT* questions and five identical *WF* questions. Comparing the performance across these two conditions is the basis for testing hypothesis 9. By random assignment, half of the participants answered the WT questions first, and the other half answered the WF questions first, to control for any effect of presentation order. All of the writing questions were presented before any of the reading questions, so that the queries that were displayed in the reading sections would not bias their responses in the writing sections.

There were eleven *RT* questions, forming the basis for testing hypotheses 1-7. The first five hypotheses can be evaluated by comparing relative student accuracy across a pair of questions. Hypotheses 6 and 7 can be evaluated by examining which of two interpretations the participants used in answering a single question. To control for any effect of presentation order, participants were randomly assigned to a path through the questions. The paths were constructed so that, for every pair of questions I intended to compare, the number of times that either question appeared first was balanced.

The eleven *RF* questions were constructed by translating the *RT* questions into the tabular language. So, each participant answered the same question twice, once with text and again with match forms. Comparing the performance in these two conditions is the basis for testing hypothesis 8. By random assignment, half the participants solved the *RF* questions first, and the other half solved the *RT* questions first, to control for any effect of presentation order.

There were a total of 32 questions in the four reading and writing sections. After this, participants answered a survey of seven preference questions. Each of these showed a query result along with two or more queries that would correctly generate the result. The participants were asked to select the one they liked the best.

4.5.1 Participants

In addition to examining these hypotheses with children who are the target audience of my programming language, I were interested in how the results would generalize to other ages. So, I recruited both children and adults to participate in the study.

Of the 33 volunteers who participated, 17 adults were recruited by sending a message to an email list for fans of a musician, and they participated by accessing the study over the web.

The rest of the participants were recruited by two of my advisor’s children, who invited their friends to come to CMU for one hour in exchange for $5 and ice cream. These participants gathered in a classroom full of computers, and accessed the study over the web. 13 of these participants were children and three were adults (parents of children who participated).

Overall, 13 children (ages 10-14), and 20 adults (ages 18-46) participated. 14 were male and 19 were female. All but two were native speakers of English. 7 participants reported that they had written computer programs (4 adults, 3 children). 27 reported that they had some experience with web search engines, and 18 had used advanced searching features (such as AND, OR, NOT, +, -, etc.). Two adults were experienced with the SQL database query language.

4.5.2 Materials

The 32 problems were presented in a web browser, one problem per web page. Each of the problem groups was preceded by an instruction page explaining how the query language or query forms work and introducing the format of the exercises. The *WT* and *WF* instruction pages were constructed to be as similar to each other as possible, as were the *RT* and *RF* pages. The web server managed the random assignment of participants to a path through the problems, the presentation of the problems in the order determined by that path, and the collection of the data anonymously. Figure 4-3 contains an example problem from each of the four problem groups.

4.5.3 Procedure

Participants began on the demographic questionnaire page and proceeded at their own pace through the materials. They were instructed to be as accurate as possible and were told that there was no time limit. When they submitted an answer, the server recorded it and presented them with the next page in the sequence. The server performed some basic syntactic checks (for example, it made sure the user provided a Boolean operator on the WT tasks, and that they did not put multiple criteria into a single slot in the WF tasks). If this check failed, an error message asked the participant to go back and fix the answer. Any time par-

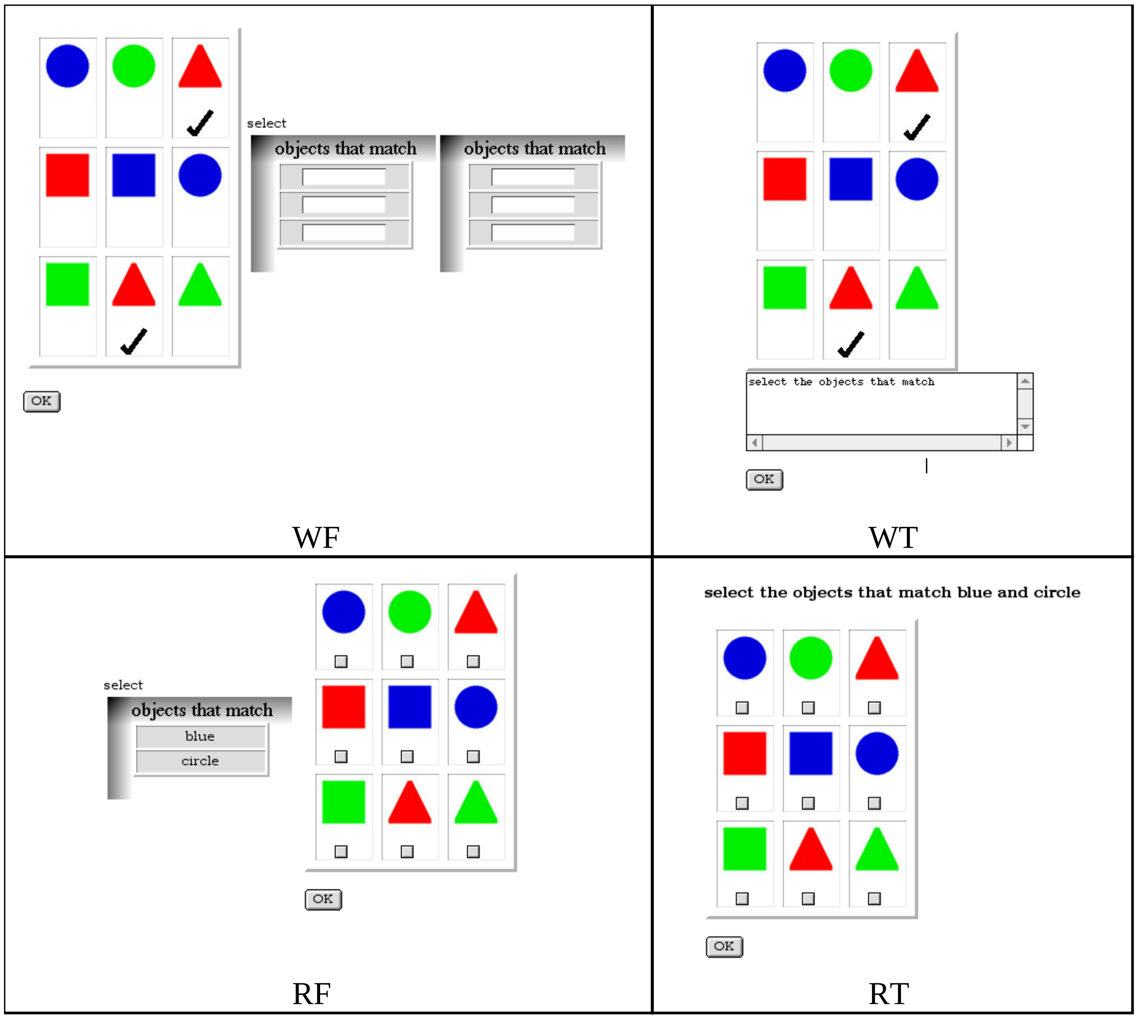


Figure 4-3. Example problems from each of the four problem groups (*WF, WT, RF, and RT*) before the answers are filled in. The color of each object is red, green or blue on the computer screen.

ticipants returned to a previous page to revise an answer, I recorded all of the answers but only used the final one for the results presented here.

Each participant's answer was scored as correct or incorrect according to the following policy. Spelling errors were tolerated, as were additional words such as *an*, *a*, or *the*. Plural and singular forms of all words were accepted. Consistent use of an incorrect color name that did *not* actually appear in the study (e.g. orange for red) was tolerated. But, any incorrect replacement of one color or shape with another one that *did* appear in the study (e.g. blue for green) was marked as incorrect. Except where otherwise noted, textual answers were interpreted the way a programming language would interpret them. Invented shorthand notations were marked as incorrect. Redundant or overly complex answers were scored as correct if they resulted in the correct selection. Finally, on the problems where I gave special instructions, answers that did not follow the instructions were marked as incorrect even if they resulted in a correct selection (e.g. one of the questions in *WF* and *WT* asked the users include the word *NOT* in their answers).

Because I simplified the match forms for this study, some of the problems became more complex when they were translated from *RT* to *RF*. For example, the lack of a way to negate a whole match form causes the expression "not (a and b)" to be translated into the tabular equivalent of "(not a) or (not b)." These question pairs were discarded from the comparison of *RT* to *RF* in testing hypothesis 8.

4.6 Results

No significant differences were detected between children and adults, between males and females, or between programmers and non-programmers, so the results are aggregated across all of the participants. The numbers shown are percentages.

In evaluating hypotheses 1-5, I performed within subject comparisons on pairs of questions from the *RT* problem group. Statistical significance in these comparisons was evaluated with a non-parametric sign test. To test hypotheses 6 and 7, I examined which of two interpretations participants used in answering a single question. Statistical significance in these comparisons was evaluated with a binomial test. In evaluating hypotheses 8 and 9, I compared pairs of questions between *RT* & *RF* and between *WT* & *WF*, respectively. These comparisons were within subject, and statistical significance was evaluated with a non-parametric sign test. In all of the statistical tests, p<.05 was used as the threshold for significance.

Hypothesis 1 is not confirmed. Users will interpret nested IF statements more accurately than a Boolean expression using AND.	% correct
Nested IF *select the objects that match red, if the objects match triangle*	94
AND *select the objects that match blue and circle*	85

not significant

Hypothesis 2 is not confirmed. Users will interpret an Unless clause more accurately than a Boolean expression that uses AND and NOT.	% correct
Unless *select the objects that match blue, unless the objects match square*	97
AND NOT *select the objects that match square and not red*	94

not significant

Hypothesis 3 is confirmed. Users will interpret an Unless clause more accurately when it appears at the very end of the statement.	% correct
Unless at end *select the objects that match blue, unless the objects match square*	97
Unless earlier *unless the objects match green, select the objects that match circle*	76

$p<.05$

Hypothesis 4 is confirmed. Users will interpret AND as Boolean conjunction in some contexts but not in other contexts.	% con-junction
select the objects that match blue and circle	85
select the objects that match blue and the objects that match circle (55% of the participants interpreted this as Boolean disjunction)	36

$p<.0001$

Hypothesis 5 is disconfirmed. Users will interpret a verbose AND expression as Boolean disjunction more accurately than an OR expression.	% dis-junction
select the objects that match blue and the objects that match circle	55
select the objects that match square or green	82

$p<.05$

Hypothesis 6 is disconfirmed for NOT with AND. Users will interpret the NOT operator with lower precedence than the other Boolean operators. *select the objects that match not red and square*	%
precedence of NOT is higher than AND interpreted as: *(not red) and square*	64
precedence of NOT is lower than AND interpreted as: *not (red and square)*	9

$p<.001$

Hypothesis 6 is confirmed for NOT with OR. Users will interpret the NOT operator with lower precedence than the other Boolean operators. *select the objects that match not triangle or green*	%
precedence of NOT is higher than OR interpreted as: *(not triangle) or green*	9
precedence of NOT is lower than OR interpreted as: *not (triangle or green)*	67

$p<.001$

Hypothesis 7 is confirmed. Users will misinterpret parenthesized expressions. *select the objects that match (not circle) or blue*	%
ignore parentheses, NOT has low precedence interpreted as: *not (circle or blue)*	39
observe parentheses interpreted as: *(not circle) or blue*	12
	p<.05

As mentioned above, three of the RF problems were not well-matched to the corresponding RT problems, so these pairs were discarded in analyzing hypothesis 8. This left eight pairs of reading problems to test hypothesis 8. All five pairs of writing problems were used to test hypothesis 9.

Hypothesis 8 is not confirmed. Users will interpret queries that use match forms more accurately than equivalent textual queries.	% correct
Match Forms (RF)	71
Text (*RT*)	74
	not significant

Hypothesis 9 is confirmed. Users will generate more accurate queries using match forms than they generate using text.	% correct
Match Forms (*WF*)	94
Text (*WT*)	85
	p<.0001

The following table breaks down the individual problems in *WF* vs. *WT*, showing the percent correct. The problems are labeled with my canonical text solutions.

	red and triangle	square and not red	(blue and circle) or (red and triangle)	circle or blue	square and not red[a]
Match Forms (*WF*)	94	73	91	42	33
Text (*WT*)	94	64	12	18	21
	n.s.	n.s.	p<.0001	p<.01	n.s.

a. The word *NOT* was required in the solution to this problem

4.7 Discussion

Although the results help to explain some of the reasons why Boolean queries using *AND*, *OR*, and *NOT* are so difficult, the textual alternatives that I proposed did not improve performance. On the other hand, the proposed tabular query forms did improve performance on writing tasks, while performing about the same on reading tasks.

4.7.1 Textual query variations

Hypothesis 1 was not confirmed. The participants performed about the same using nested *IF* statements as they did using a Boolean expression with the *AND* operator. On the preferences survey, the majority of the participants preferred the Boolean expression.

Hypothesis 2 was not confirmed. The participants performed about the same using an *Unless* clause as they did using a Boolean expression with the *AND* and *NOT* operators. On the preferences survey, the majority of the participants preferred the Boolean expression.

Hypothesis 3 was confirmed. The participants performed significantly better with the *Unless* clause at the end than they did with the *Unless* clause earlier in the statement. However, given the result of Hypothesis 2, the importance of this result is questionable. Also, the very simple problems used in this study did not provide a good way to test the situation where I speculated that the *Unless* would violate the principle of locality. On the preferences survey, most of the participants preferred the *Unless* at the end.

Hypothesis 4 was confirmed. Two slightly different queries using *AND* resulted in significantly different interpretations. 85% of the participants interpreted the *AND* in “select the objects that match blue and circle” as a conjunction operator. But only 36% of them interpreted it that way in “select the objects that match blue and the objects that match circle.” Instead, 55% of them interpreted the *AND* in the second statement as a disjunction operator. This result helps to explain the frequently observed error where users incorrectly use *AND* instead of *OR*.

Hypothesis 5 was disconfirmed. Despite the fact that the majority of the participants interpreted *AND* as a disjunction operator in “select the objects that match blue and circle,” they are significantly more accurate in interpreting disjunction if the *OR* operator is used. On the preferences survey, the majority of the participants preferred *OR* over a verbose *AND* statement to express disjunction.

In the surprising results of hypothesis 6, I measured reliable effects in opposite directions depending on context. The hypothesis was disconfirmed when comparing the precedence of *NOT* with *AND*. 64% of the participants treated *NOT* with **higher** precedence than *AND*, matching the common usage in programming languages. However, the hypothesis was confirmed when comparing the precedence of *NOT* with *OR*. In this case, 67% of the participants treated *NOT* with **lower** precedence than *OR*. Since consistency is an important human-computer interaction principle [Nielsen 1994], this reversal in the natural interpretation of precedence suggests that it is unwise to rely on implicit precedence rules.

Hypothesis 7 was confirmed. Users ignored parentheses significantly more often than they observed them. The query was, “select the objects that match (not circle) or blue.” The results on hypothesis 6 suggest that without the parentheses, most participants would have applied the *NOT* operator to the expression “circle or blue.” The parentheses were not able to override this tendency.

4.7.2 Match forms vs. text

Hypothesis 8 was not confirmed. On reading tasks, the participants performed about as well with match forms as they did with text. However, hypothesis 9 was confirmed. On writing tasks, the participants performed significantly better with the match forms than they did with text. This disparity, where a positive effect is stronger on generation tasks than interpretation tasks, has also been observed in other systems (e.g. [Modugno 1996]). On the preferences survey, which was a reading task, the participants’ choices were about equally divided between text and match forms.

Match forms were not superior for code interpretation, but they did not have a detrimental impact on that task. Thus the overall effect of using match forms should be positive due to the strong gains on generation tasks, despite the lack of an effect on interpretation tasks.

The breakdown of individual questions in the query generation task shows that the participants performed about the same in the two conditions when the queries were simpler, but as the queries became more complex, the differences in favor of the match forms increased. While the trend in favor of match forms was present in all cases, only the queries that involved disjunction revealed significant differences between match forms and text. As expected, the most common error on these problems was the substitution of *AND* where *OR* was required.

The three problems that were excluded from the reading comparison were among the more complex queries. Since the advantage of query forms is stronger on more complex queries, excluding this data may have reduced any positive effect of match forms on the reading task. Thus it would be less likely for the study to be able to confirm hypothesis 8. Further research into this question is warranted.

The strong effect of match forms came with very little training. It is unlikely that the participants had used an equivalent tabular query language before, and they only viewed a brief instruction page with a few examples before beginning to solve the problems. While the instructions for the textual problems were similarly brief, the participants brought knowledge from a lifetime using the words *AND*, *OR*, and *NOT* in English. This may have interfered with the programming language interpretation, or made them less careful in reading the instructions.

4.8 Summary

Based on the results of this study, I can make the following recommendations to designers of programming languages, scripting tools, and search engines that incorporate query mechanisms:

- Do not use the word *AND.*
- Do not rely on parentheses for grouping.
- Do not rely on implicit operator precedence rules.
- Consider using match forms or similar tabular query forms instead of pure text.

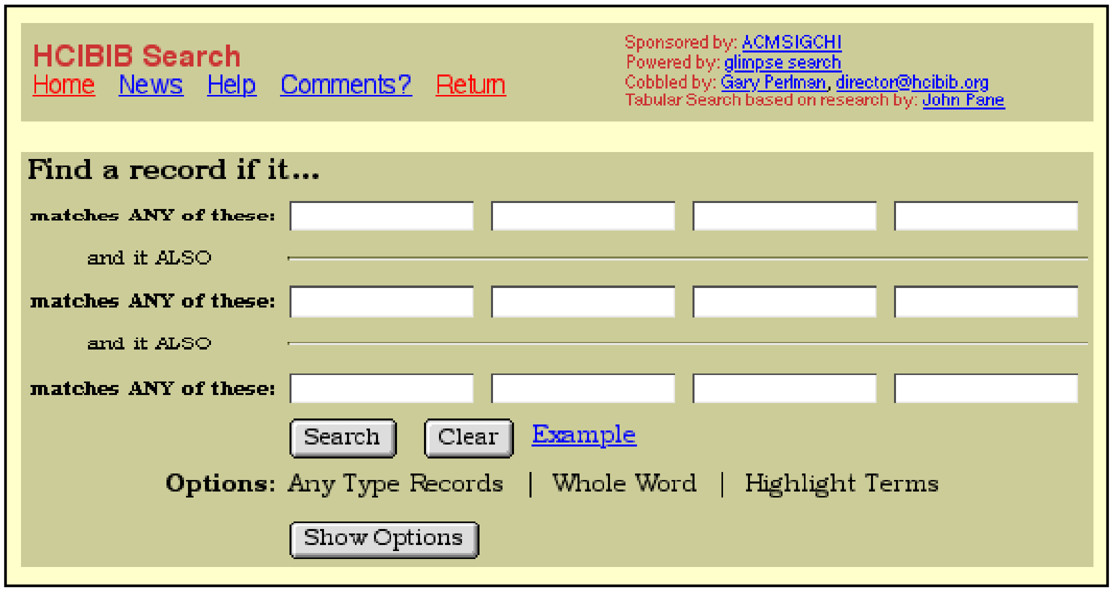
The success of match forms can be attributed to several factors: *AND* and *OR* are avoided; the scope of *NOT* is unambiguous; parentheses are not needed for grouping; and cues help to disambiguate conjunction and disjunction.

This study of what is natural for untrained users provides an empirical basis for choosing among design alternatives in query tools for beginners. The strategy employed here can be also used by developers to assist in the design of other kinds of tools for programmers.

4.9 Application of Results

The HCI Bibliography (www.hcibib.org) is a web database of HCI related resources, maintained by Gary Perlman. This site uses the *glimpse* tool (glimpse.cs.arizona.edu) for its search engine. Glimpse has an unusual query syntax that was causing many errors and complaints from users. For example, the glimpse syntax for the query *(Pane or Myers) and Boolean* is *{pane,myers};{boolean}*. I proposed to use match forms for queries on this site, and created a prototype. When the user submitted a query using this match form interface, JavaScript translated the query into the glimpse syntax and sent it to the server.

Mr. Perlman liked the idea, but felt that conjunctive normal form was more useful for searching bibliographic databases. We worked together to create a conjunctive normal derivative of match forms that preserves their important features. The result is shown in Figure 4-4, and is installed on the HCI Bibliography website.


HCIBIB Search
Home News Help Comments? Return

Sponsored by: ACMSIGCHI
Powered by: glimpse search
Cobbled by: Gary Perlman, director@hcibib.org
Tabular Search based on research by: John Pane

Find a record if it...
matches ANY of these:
and it ALSO
matches ANY of these:
and it ALSO
matches ANY of these:

Figure 4-4. The new tabular search interface for the HCI Bibliography (www.hcibib.org).

Note that the descriptive text along the left edge of the form includes the word *and*. We felt this use was acceptable because it is part of the longer phrase, *and it ALSO*, which reinforces the conjunctive interpretation of the word *and*.

Mr. Perlman recently analyzed user performance on the new tabular search interface in comparison to the older glimpse-syntax interface, and concluded that users make fewer errors when using the new tabular interface. He has also received favorable comments about the new interface, such as, “What a great search interface! I like how easy it is to figure out how to search using ‘and’ or ‘or.’” Our conclusion is that this new search interface is a success.

CHAPTER 5

The HANDS System

This chapter describes the details of the HANDS system design. It begins with a review of the high-level motivating factors for the design, and then describes the computational model that is portrayed in HANDS, followed by details of the syntax and features of the HANDS language and environment. A full syntax chart for the language appears in Appendix A.

5.1 Motivating Factors in the HANDS Design

The various components of the system were designed in response to the observations in my studies as well as prior work:

- Beginners have difficulty learning and understanding the highly-detailed, abstract and unfamiliar concepts that are introduced to explain how most programming languages work. HANDS provides a simple concrete model based on the familiar idea of a character sitting at a table, manipulating cards.
- Beginners have trouble remembering the names and types of variables, understanding their lifetimes and scope, and correctly managing their creation, initialization, destruction and size, all of which are governed by abstract rules in most programming languages. In HANDS, all data is stored on cards, which are familiar, concrete, persistent,

and visible. Cards can expand to accommodate any size of data, storage is always initialized, and types are enforced only when necessary, such as when performing arithmetic.

- Most programming languages require the programmer to plan ahead to create, maintain, and traverse data structures that will give them access to the program's data. Beginners do not anticipate the need for these structures, and instead prefer to access their data through content-based retrieval as needed. HANDS directly supports queries for content-based data retrieval.
- Most programming languages require the programmer to use iteration when performing operations on a group of objects. However, the details of iteration are difficult for beginners to implement correctly, and furthermore, beginners prefer to operate on groups of objects in aggregate instead of using iteration. HANDS uniformly permits all operations that can be performed on single objects to also be performed on lists of objects, including the lists returned by queries.
- Despite a widespread expectation that visual languages should be easier to use than textual languages, the prior work finds many situations where the opposite is true (see Section 1.6.7 on page 13). In my studies, pictures were often used to describe setup information, but then text was used to describe dynamic behaviors. HANDS supports this hybrid approach, by permitting objects to created and set up by direct manipulation but requiring most behaviors to be specified with a textual language. This design assumes that the environment will provide syntax coloring and other assistance with syntax. These features are commonly available in programming environments, but re-implementing them in HANDS was beyond the scope of this thesis.
- Programming language syntax is often unnatural, laden with unusual punctuation, and in conflict with expectations people bring from their knowledge in other domains such as mathematics. The HANDS language minimizes punctuation and has a more natural syntax that is modeled after the language used by non-programmers in my studies.
- The prior research offers few recommendations about which programming paradigm might be most effective for beginners (imperative, declarative, functional, event-based, object-oriented, etc.). In my studies of the natural ways beginners expressed problem

solutions, an event-based paradigm was observed most often, and program entities were often treated with some object oriented features. HANDS therefore uses an event-based paradigm. Cards are the primary data structure, and they have some object-like properties: they are global, named, encapsulated, persistent, and have some autonomous behaviors.

- The prior work recommends that programming systems should provide high-level support for the kinds of programs people will build, so they do not have to assemble more primitive features to accomplish their goals. In my interviews with children, they said they wanted to create interactive graphical programs like the games and simulations they use every day. HANDS provides domain-specific support for this kind of program.

5.2 Representation of the Program

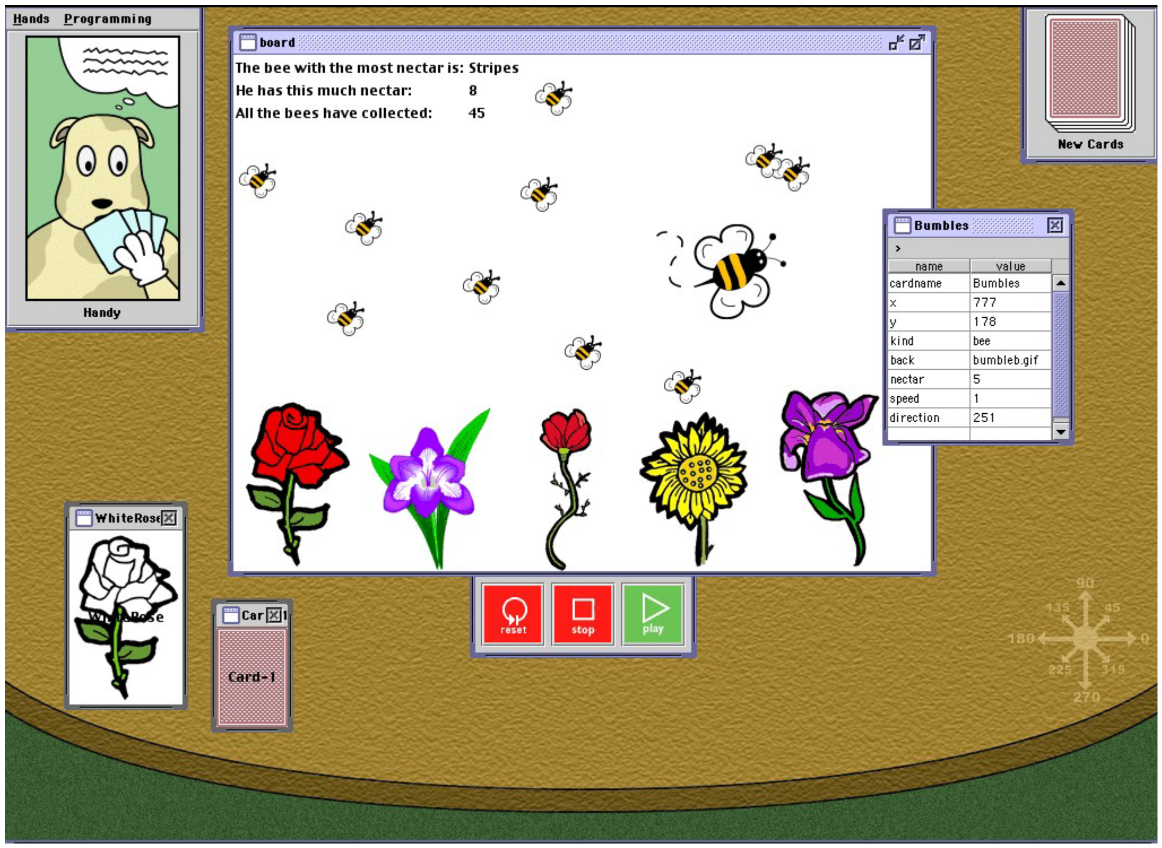
The HANDS system introduces a new model of computation that is concrete, uses concepts that are familiar to children, and provides high visibility of program data. In HANDS, an agent named Handy sits at a table, manipulating information on cards (see Figure 5-1).

5.2.1 Cards for Data Storage

All of the data in the system is stored on cards, which are global, persistent and visible on the table. Each card must have a unique name, which is not case sensitive. The name must be an identifier, which is generally a word without spaces. The exact definition of an identifier is included in Appendix A. When a new card is created, it is given a unique name by the system unless the programmer specifies a name. The names of cards can be changed at any time, and an error dialog comes up immediately if the programmer attempts to use a name that is already in use.

5.2.1.1 Properties

The front of each card has a list of properties, which are name-value pairs (Figure 5-2). Names of properties must be identifiers, are not case sensitive, and must be unique within each card. See Section 5.4 on page 101 for information about what types of information can be stored into the value of a property. Several properties are always present: the `cardname` property holds the card's name, and the `x` and `y` properties contain the card's position coordinates. The card's name is also shown in the title bar at the top of the card. The pro-



The bee with the most nectar is: Stripes
He has this much nectar: 8
All the bees have collected: 45

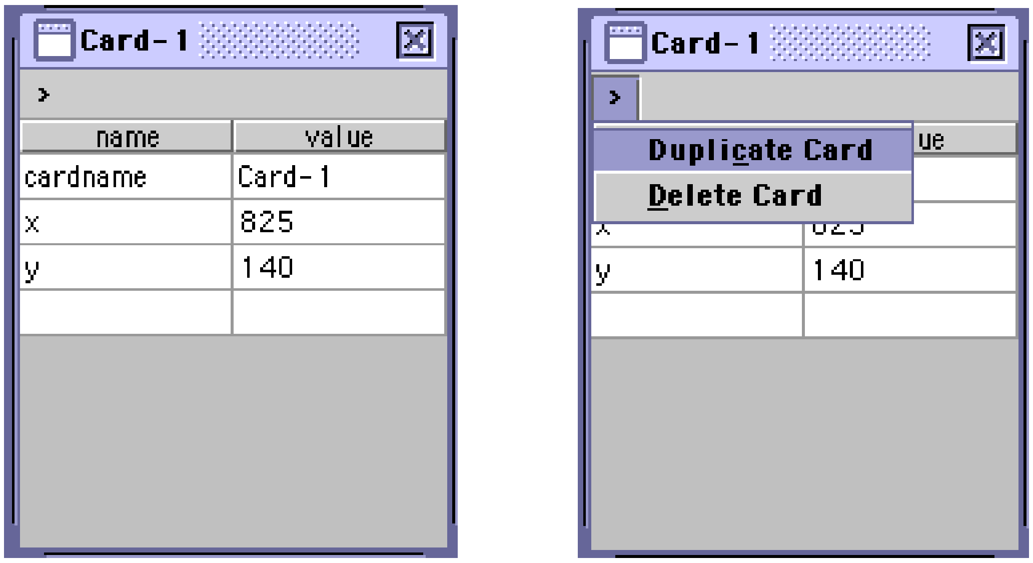
name	value
cardname	Bumbles
x	777
y	178
kind	bee
back	bumbleb.gif
nectar	5
speed	1
direction	251

Figure 5-1. The HANDS system portrays the components of a program on a round table. All data is stored on cards, which can be drawn from the pile at the top right and dragged into position. At the lower left, two cards are shown face-down on the table. One has a generic card back and the other has been given a picture by the programmer. In the center of the table is a board, where the cards are displayed in a special way where only the contents of the back are displayed. Each picture, string, and number on the board is a card. At the right, one of the cards has been flipped face-up, where its properties can be viewed and edited. The programmer inserts code into Handy's thought bubble, by clicking on Handy's picture in the upper left corner. When the play button is pressed, Handy begins responding to events by manipulating cards according to the instructions in the thought bubble. The stop button halts the program, and the reset button will restore all cards to their state at the time the play button was last pressed. For reference, a compass is embossed on the table at the lower right.

grammer can add more properties as needed, so cards are similar to records (or structs) in other languages. When a new property is created, and the programmer does not specify a name, the system automatically generates a unique name. An error dialog comes up immediately if the programmer attempts to use a property name that already exists on the card.

5.2.1.2 Direct Manipulation of Cards and Properties

In addition to the usual ways that data can be manipulated by the code of a running program, cards and their properties can be managed by direct manipulation, even before or after the program has run. New cards can be drawn from the pile of new cards in the top



name	value
cardname	Card-1
x	825
y	140

Figure 5-2. In HANDS, all data is stored on cards. The fronts of cards have an unlimited set or properties, or name-value pairs. The `cardname`, `x`, and `y` properties are always present. Additional properties can be added by the running program, or by making an entry into the blank slot that is always available at the bottom. A popup menu on the card enables the programmer to duplicate or delete the card.

right corner of Figure 5-1, and existing cards can be cloned by choosing “Duplicate Card” from a popup menu on the card (see the right side of Figure 5-2). The clone is identical to the original, except a unique name is generated from the original name. The popup menu also allows cards to be deleted. Cards can be moved around on the screen to set their position (`x` and `y` properties). Property names and values can be edited by typing directly into the slots on the card. New properties can be added by inserting entries into the blank row that is always present at the bottom of each card. If only a value is inserted, the system generates a unique name for the property. If only a property name is specified, the property is initialized to the value `empty`. If the programmer edits the `x` or `y` properties, the card is automatically moved to the new position.

When a card’s properties are showing, it is considered to be *face-up*. The other side of the card is called its back, which can contain a picture or other information. A face-up card can be flipped face-down to show its back by clicking the close box at the top right corner. When face-down, clicking the close box, or anywhere on the back of the card, flips it face-up again.

5.2.1.3 The `back` Property

The `back` property controls what is displayed on the back of the card when it is face-down. If there is no `back` property on a card, a picture of a generic card back is displayed (shown at left in Figure 5-3). If the value of the `back` property is the name of a file containing an

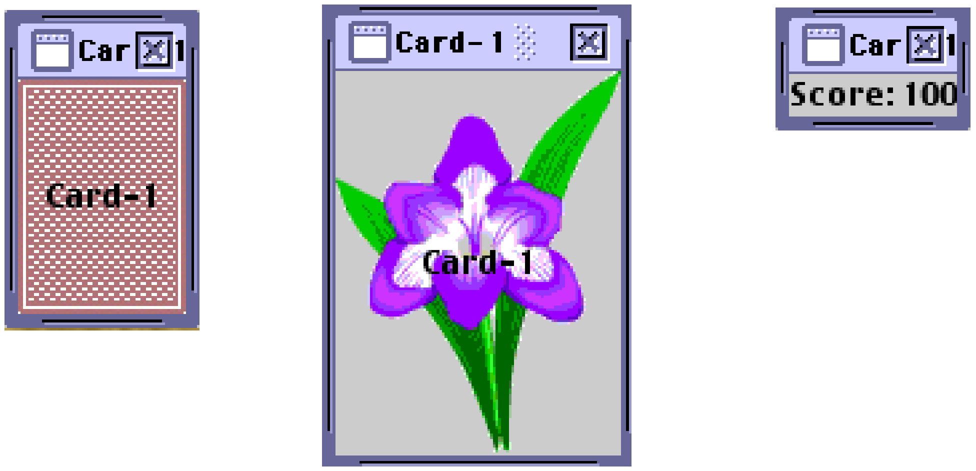


Figure 5-3. The backs of cards are controlled by the `back` property. If this property is absent, a generic card back is shown (left). If the name of an image file is placed into the back property, the image is displayed on the back of the card (center). In either of these cases, the card name is superimposed on the back. If a string or number is placed into the back property, it is displayed on the card back (right), and the card name is not superimposed.

image, the image is displayed on the back of the card (shown at center in Figure 5-3). The program automatically searches for an image file with the specified name in the same directory as the HANDS program, and if not found, it then searches in the “Graphics” subdirectory. Paths relative to these directories may also be used, as well as full pathnames. It would be a straightforward extension to allow the programmer to select an image file using a file browser. If the specified file is located, and it contains a JPEG or GIF graphic, the image is automatically loaded, the card is resized to the match the size of the image, and the image is displayed on the back of the card. When an image is shown on the card back (either the generic card back image or another image), the name of the card is superimposed on the image. If the `back` property contains any other value that cannot be resolved into the name of an image file, the value is displayed as a string on the back of the card. Quotes are removed if they delimit the string, and the card’s size is adjusted to match the size of the string (shown at right in Figure 5-3). All cards are automatically flipped so the back shows when the program starts running.

5.2.1.4 The Game Board

When creating a program for other people to use, usually some of the program's data is visible to the end-user (such as characters in a game), while other data is invisible (such as intermediate calculations, or characters that are not supposed to be visible at a certain time). In traditional programming systems, all data is invisible unless it is explicitly printed or drawn onto the screen. In a system like HANDS, where all data has a visible representation, another mechanism is needed to separate these two kinds of data. In Rehearsal World [Gould 1984], which uses a *stage* metaphor, the hidden data is placed in the *wings*. The *game board* addresses this issue in HANDS, and also provides a way for programmers to display information that does not look like a card (such as characters, scenes, text and numbers).

The white area in the center of the screen in Figure 5-1 is the game board. The game board represents the part of the program that would be visible to an end user of a program developed in HANDS. When cards are face-down on the board, only the back is displayed, without a containing window or the name of the card. The container seems to magically disappear and appear as the programmer drags a card on and off the board. Unlike face-down cards, cards that are face-up on the board look and behave the same as if they were off the board. Since there is no containing window once a card has been dropped face-down onto the board, it can be flipped only by clicking within the bounding rectangle of its back. Also, HANDS does not currently implement a way to begin dragging a card that is face-down on the board, so it must first be flipped over to gain access to its window title bar.

Figure 5-4 shows how the three cards in Figure 5-3 are displayed when they are on the board. A feature that is not yet implemented would allow the programmer to create an end-user version of the program, where only the game board is displayed, and any cards that are off the board would be invisible.

These features make it relatively easy for the programmer to display graphics, animations and text on the screen. For example, no code is necessary to create the classic program that displays "hello, world" on the screen. Any time a running program changes contents of the `back` property, the back of the card is immediately updated. A program can cycle a series of images into the `back` property to animate the appearance of an object.

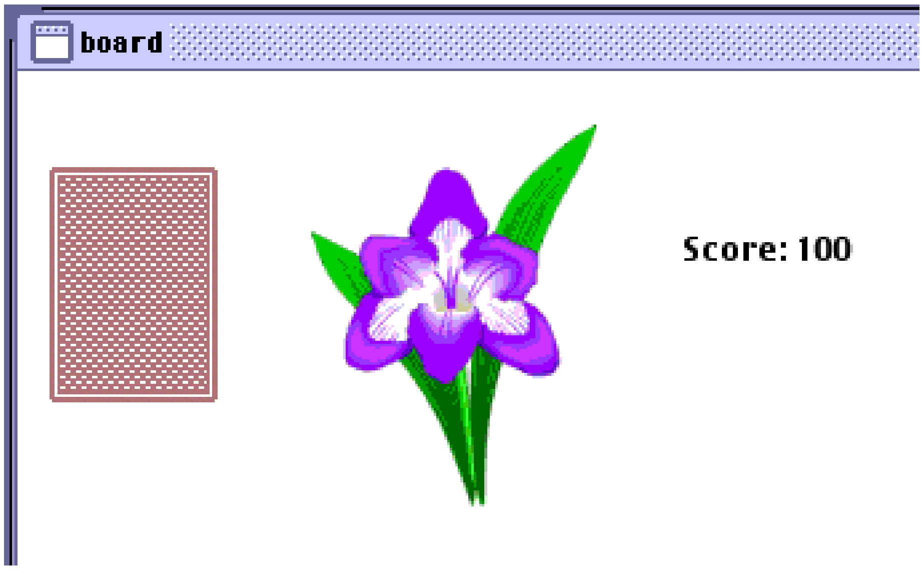


Figure 5-4. When cards are on the board, only the back is displayed, without a containing window and without the card's name. This makes it very easy to display text and graphics on the screen.

Objects are drawn onto the board in alphabetical order by cardname, so cards that fall later in the alphabet are drawn on top of any overlapping cards that fall earlier in the alphabet. While this gives the programmer some control over the layering of objects, it is at the expense of flexibility in naming cards. HANDS could be easily extended to allow the programmer to specify a drawing order independently of the card's name. For example, cards could be displayed in the order created, with explicit commands to move them in front of or behind other cards.

5.2.1.5 Models for the Cards in HANDS

Although the backs of cards have a generic design that looks like the backs of playing cards, children are often familiar with other kinds of cards that have a strong resemblance to the the cards in HANDS. For example, baseball cards have large quantities of information on them, like the lists of properties in HANDS. Also, popular games such as *Magic the Gathering* (www.magicthegathering.com) use cards with rich graphics and a set of properties defining the characteristics of game entities.

5.2.2 Computation is Performed by Handy

Beginners often expect the computer to be very intelligent and to be able to make inferences, so they are often lax in describing important details. Galotti & Ganong [Galotti

1985] found that they were able to improve the precision in procedural specifications by ensuring that users understood the limited intelligence of the recipient of the instructions. To emphasize the limited intelligence of HANDS, Handy is portrayed as an animal– like a dog that knows a few commands – instead of a person or a robot that could be interpreted as being very intelligent.

Below the game board are *play, stop,* and *reset* buttons. When the play button is pressed, the program begins to run. The image of Handy (at the top left corner of Figure 5-1), which is a static picture when the program is not running, begins to animate. The animation shows Handy picking up and putting down miniature cards, and the squiggly lines in his “thought bubble” wiggle around to indicate that he is thinking. This animation helps to confirm that the program is indeed running, even if no other visible action is taking place on the screen.

5.2.2.1 Handy Watches for Events and Executes the Code in His Thought Bubble

The program itself is stored in Handy’s “thought bubble,” which can be accessed from a menu, or by clicking on Handy. The thought bubble provides a central location for all of the program code, unlike some other beginner systems such as Hypercard [Goodman 1987], where the code is scattered around and it may be difficult for the programmer to determine where the code is that is causing something to happen or preventing it from happening.

When the program is running, Handy watches for events, and if his thought bubble contains a corresponding event handler he responds by executing the code in the event handler. There is more information about events and event handlers in Section 5.3 on page 96, and the browser for viewing event handlers is described in Section 5.15.2 on page 131. Handy is also responsible for some domain-specific functions that do not require any programming, such as animating cards that have speed and direction slots (see Section 5.14 on page 123).

5.2.2.2 Stop and Reset

The stop button halts the program. Any changes to the cards are preserved if play is hit again. In this sense, stop is more like the pause button on a CD player. If the programmer would like to restore all of the cards to their state at the last time the play button was pressed, the reset button can be used. The reset button also halts the program if it isn't already halted. The reset feature is accomplished by saving all of the cards to a checkpoint file when play is pressed, and reading them back in when reset is pressed, after first displaying a confirmation dialog.

An earlier version of HANDS saved and restored the program code as well as the cards, but this had an undesirable effect that was uncovered in early testing. During program development and debugging, a typical sequence of operations was to run the program, stop it, make changes to the code, and then hit reset right before running the program again. When used in this way, the reset operation was discarding the changes the programmer had made to the code. For this reason, the reset operation was modified to only restore the cards.

5.2.2.3 Handy's Hand

Handy can be directed to pick up cards off the table and put them into his hand. When he picks up a card, it disappears from the table, but all of the card's properties remain unchanged, including its position properties. Therefore, a card that is picked up from a certain location on the table will be put down at that same location on the table, unless the program changes its location properties in the interim. Picking up and putting down cards is analogous to the visibility property of objects in other systems. A menu command switches the system to a view that shows only the cards in Handy's hand, at their appropriate places on the table. Section 7.2.2 on page 171 describes a multiple-agent extension to the HANDS system where an agent's hand could also represent private data that is not visible to other agents, and the passing of cards could be used for private communication between agents.

5.3 Programming Style and Model of Execution in HANDS

HANDS is event-based, to match the style of programming that I observed in my studies. A program is a collection of event handlers that are automatically called by the system when a matching event occurs. Inside an event handler, the programmer inserts one or more imperative statements to execute in response to the event. After these statements have executed, control returns to the system, where the next event is dispatched.

5.3.1 Structure of Event Handlers

All event handlers have this structure:

```
when <event>
     <statements>
end when
```

The keywords `when` and `if` both appeared frequently at the start of code fragments in my studies. One reason `when` was selected for the event handler keyword is because it is more suggestive of one-time evaluation at or near the occurrence of an event, whereas `if` might engender a continuous-evaluation interpretation. Also, I felt it was better to have a keyword that was distinct from the conditional statement, which uses the `if` keyword. I also considered using `at the time` as the keyword for event handlers, but this was rejected, mainly because it suggests more precise timing than an event-oriented system provides, where the execution of the event handler may actually be delayed by the processing of other events already in the queue. Furthermore, it was too verbose.

5.3.2 Event Dispatch

The system automatically maintains the event queue. All of the supported events are listed below in Section 5.3.3. When the system observes one of these events taking place, it inserts a record of the event into the queue. Meanwhile, an event dispatcher continuously removes the first event in the queue, calls all of the appropriate event handlers (there may be zero, one, or more than one), and then discards the event and moves on to the next one. When the event queue is empty, the dispatcher synthesizes an idle event and calls the appropriate handler if it exists.

When there is more than one event handler matching a particular event, they are all executed, one at a time in the order they appear in the event browser, which is alphabetical. There could be many handlers for a particular event. For example, seven event handlers would be called if the bee `Bumbles` collided into the flower `Rose`, and there were handlers for each of: `Bumbles collides`, `Bumbles collides into any flower`, `Bumbles collides into Rose`, `any bee collides`, `any bee collides into any flower`, `any bee collides into Rose` and `anything happens`.

Ideally, HANDS would give the programmer better control over which order the event handlers are called in.

The event dispatcher periodically allows the user interface and animation engine to run, by releasing control to them between event dispatches. This ensures that no event handler can create a deadlock, for example, by repeatedly inserting a new event that will cause the same handler to be called again. Even if the code were written this way, objects on the screen would continue to be animated, screen updates would occur, the user interface would respond to the user, and other kinds of events would be inserted into the queue between the ones by the offending code, such as collisions, objects changing, and the user clicking on objects or typing on the keyboard.

5.3.3 The Events

The following events are defined:

- `program starts`

  This event is always the first event that is dispatched when the program begins to run.

- `program stops`

  This event is always the last event that is dispatched when the program stops. This event has priority over any other events in the queue, which are discarded when the program stops.

- `<identifier> appears`

  A card named <identifier> is created.

- `<identifier> disappears`

  A card named <identifier> is destroyed. A runtime error will occur if the event handler for this event attempts to access the card named <identifier>, unless it was replaced in the interim.

- `<identifier> changes`

  A card named <identifier> has any of its properties set. If there is already a change event for this card in the event queue, another one is not inserted. Otherwise it would be too easy for events to be inserted into the queue faster than they could be removed. For example, without this protection, the problem would occur if the event handler for `Card-1 changes` assigned to more than one property of `Card-1`.

- `<identifier> collides`

  A card named <identifier> collides into any object (see Section 5.14.4 on page 127).

- `<identifier> collides into <identifier2>`

  A card named <identifier> collides into another card named <identifier2>.

- `<identifier> is clicked`

  A card named <identifier> is clicked on by the mouse. The object must be on the board, and the mouse click must be within the bounding box of the object. At this time, only the mouse-down event using the left mouse button is supported. Support for the right mouse button, and "still down," "mouse moved," and "mouse released" events should be added in the future.

- `<key> is typed`

  The keystroke <key> is typed on the keyboard.

- `nothing happens`

  The system is idle; there are no events in the event queue. It is possible that this event is never dispatched if the event queue always has at least one event in it.

- `anything happens`

  Any event except `nothing happens` will cause an handler for this event to be dispatched. The only time this handler is not called is when the system is completely idle.

5.3.4 Event Patterns

The programmer can use patterns of the form `any <identifier> <event>` or `any key is typed`. This enables cards to be treated as groups by giving them a common string in a property such as `kind` or `group`:

- `any <identifier> appears`

  A card with `<identifier>` in one of its properties is created.

- `any <identifier> disappears`

  A card with `<identifier>` in one of its properties is destroyed.

- `any <identifier> changes`

  A card with `<identifier>` in one of its properties has any of its properties set.

- `any <identifier> collides`

  A card with `<identifier>` in one of its properties collides into any object.

- `any <identifier> collides into any <identifier2>`

  A card with `<identifier>` in one of its properties collides into another card with `<identifier2>` in one of its properties. This pairwise collision event can also be specified with one each of `<identifier>` and `any <identifier>`, in either order.

- `any <identifier> is clicked`

  A card with `<identifier>` in one of its properties is clicked by the mouse.

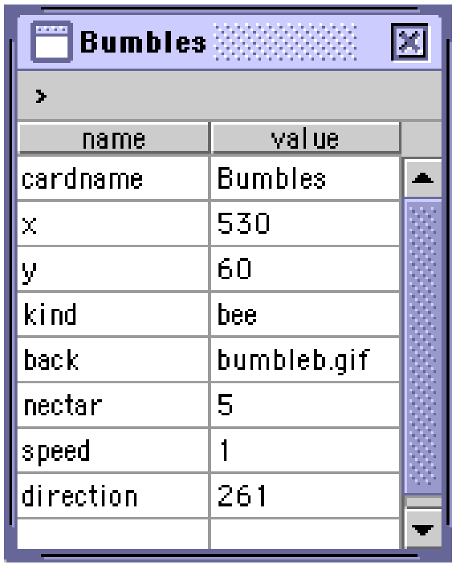
- `any <key> is typed`

  Any keystroke is typed on the keyboard.

5.3.4.1 Temporary Variables for Event Patterns

Within these event handlers that use `any` patterns, the system automatically creates a temporary variable for the duration of the event, binding the `<identifier>` or `<key>` to the specific object that was involved in the event. For example, consider the card `Bumbles` from Figure 5-1 and repeated in Figure 5-5. If there is an event handler for `any bee collides`, it would be called when `Bumbles` collides into any object, because one of the properties of `Bumbles` contains the string `bee`. During the execution of the event handler, the identifier `bee` is bound to `Bumbles`.

In the case of the collision event handler where there are two identifiers to bind, for example, `any bee collides into any flower`, both `bee` and `flower` are bound to the appropriate objects. For example, if `Bumbles` collides into the flower `Rose`, the identifier `bee` is bound to `Bumbles`, and the identifier `flower` is bound to `Rose`. If both identifiers are the same, the prefixes `first-` and `second-` are attached to the identifiers. For example, in the event handler `any bee collides into any bee`, the two cards in the collision are bound to `first-bee` and `second-bee`.



name	value
cardname	Bumbles
x	530
y	60
kind	bee
back	bumbleb.gif
nectar	5
speed	1
direction	261

Figure 5-5. If there is an event handler for `any bee collides`, it would be called if the card `Bumbles` collides into any object. During the execution of event handler, the identifier `bee` is bound to `Bumbles`.

5.3.5 Event Cards

Like data, each event is represented by a card, with a unique `cardname`, an `x`, `y` position, a `type` (always `eventcard`), a `group` indicating the type of event (e.g. `collision`), and a `value` property that contains information specific to the event. When a single object is involved in the event, its name is in the `value` property. When two objects are involved in a collision, both are listed in the `value` property. When the event is a keystroke, the key that was typed is listed in the `value` property.

During the execution of the event handler, the temporary variable `event` is bound to the current event card, so that its properties can be accessed. However, for efficiency reasons, events do not appear on the table while the program is running, unless there is a runtime error (see Section 5.16 on page 138). A future extension to HANDS could offer a “slow” debugging mode where the current event as well as the event queue are displayed continuously on the table.

5.4 Data Types

Values in HANDS can have the following types:

- an identifier, such as the name of a card, which is not case-sensitive;
- a string literal delimited by quotes;
- an integer or floating point number, which can be used interchangeably;
- a Boolean literal, either `yes` or `no`;
- a list of zero or more data elements that do not necessarily have to be all of the same type. The empty list is represented by `empty`.

The vocabulary for literal constants, `yes`, `no` and `empty`, was selected for simplicity and familiarity, instead of choosing terms or symbols that are less familiar to non-programmers, such as `true`, `false` or `null`.

The system does not enforce types until necessary. All data values are treated as strings until they are used in an operation that requires a different type, such as arithmetic. This includes identifiers, so many single-token strings do not have to be quoted in HANDS. Identifiers must begin with a letter and cannot have spaces, but they can contain hyphens, underscores, periods, and slashes. The last two of these are accepted so that many file names and paths can be stored into card slots without quoting.

However, this flexibility is not universal, because strings that are not legal identifiers, such as multiple-word strings or keywords of the language, must be quoted. The strategy for addressing this issue in HANDS is to provide good error messages, both interactively and from the parser. For example, the interactive message for when a user tries to insert multiple values into a card slot without separating them with commas is, “A property cannot have the value `one two three`, unless it is quoted or there are commas separating the items.”

Lists have properties like traditional lists, such as being unbounded and permitting insertion without making space, and properties like arrays, such as index-based access. The syntax of lists is described in Section 5.6.6 on page 105.

5.5 Numeric Values and Calculations

In HANDS there is no operational distinction between integer and floating-point numbers. All numeric calculations are conducted using double-precision, and the full-precision result is maintained. However, floating point numbers that end in "`.0`" have this suffix removed so that the result is converted into an integer for display. This ensures that the results of addition, subtraction, and multiplication on integers are not converted into floating point numbers as a side effect of the double-precision math.

5.6 Language Syntax

The syntax of HANDS was designed to match the common ways that participants expressed operations in my studies, in order to observe the principles of *simple and natural dialog, speak the user's language,* and *closeness of mapping*.

5.6.1 Natural-Language Style

The syntax for accessing properties on cards uses a natural-language style instead of the dot's, arrows, and brackets used in many other languages. The programmer can refer to the `nectar` property of a card named `flower` with either of these syntaxes: `nectar of flower` or `flower's nectar`. These choices are provided so the programmer can select whichever is more readable in context. However, the "`'s`" syntax is not adapted for any special cases such as plurals (see Section 5.6.2 on page 103). Another example is the natural-language style that is provided for arithmetic: `add 100 to the game's score`, in addition to the usual more mathematical style: `set the game's score to 100 + game's score`.

The language is uniformly case-insensitive. Overall, the language has a verbose conversational style, similar to HyperTalk [Goodman 1987], AppleScript, and Macromedia's Lingo [Gross 1999]. To improve readability, the system allows the words `the` or `is` to be placed anywhere in the code – they are ignored. For example, `set the nectar of the bee to 0` is the same as `set nectar of bee to 0`. The word `a` is not ignored however, because this word is often used by programmers to name variables.

5.6.2 Plurals

The HANDS language does not attempt to correctly handle special cases with plurals. Instead it uses simple fixed rules for creating possessives to access properties (see Section 5.2.1.1 on page 89) and the plurals used in queries (see Section 5.10 on page 113).

For example, if there were several cards with `ox` in one of their slots, the query to find them would be `all oxs`, not `all oxen`. While the system could have been designed to automatically handle plurals, it would require fairly sophisticated processing, and would likely be imperfect [Conway 1998, McIver 2001]. Furthermore, it is unlikely that the children using HANDS would know all of the correct rules for pluralization.

5.6.3 Control Structure Terminators

Many languages use a common symbol (such as “`}`” or “`end`”) to terminate many different kinds of control structures. When several structures are nested, it can be difficult to figure out which terminator belongs to which structure. If the terminator is optional, additional ambiguities can result. For example, Pascal and HyperTalk have the dangling-else problem, where the system may attach an else clause to a different if statement than the user intended. For these reasons, all structures in HANDS that allow statements to be nested require a matching terminator that incorporates the name of the structure. For example, the `when` statement is terminated with `end when`. To reduce the amount of typing required, the system automatically inserts some of the terminators.

5.6.4 Statement Terminators

Where possible, HANDS avoids requiring punctuation or other symbols if their only purpose was to make it easier to parse. These syntactic elements are a distraction from the semantically-important parts of the program, and are a common source of errors. For example, no semicolon is required to terminate or separate statements. Because each statement in HANDS begins with a keyword, and the last statement in any context must be followed by the keyword `end`, the system is able to parse the code without help from statement terminators.

5.6.5 Parentheses Are Required to Indicate Precedence Explicitly

In my studies, I observed that beginners have many problems with implicit precedence rules, so HANDS requires parentheses when expressions are nested. The studies also determined that parentheses were confusing, but I could not figure out a good alternative to their use. To reduce confusion, the language uses only one kind of parentheses: “`(`” and “`)`”.

5.6.6 List Syntax

In early implementations of HANDS, the syntax required special symbols at the beginning and the end of lists. I experimented with `list(1 2 3)` and `list 1 2 3;`, but was unsatisfied with the extra required symbols for opening and terminating the lists. After much experimentation and re-organization of the parser, I was able to achieve a more realistic syntax for lists, where commas are used to separate list items but no opening and closing delimiters are required: `1, 2, 3`. Lists can be nested by using parentheses. For example, `(1, 2, (9, 8, 7), 4, 5)`. The empty list is represented by the keyword `empty`, instead of using symbols such as `()` or `null`. Another alternative might have been to use the absence of anything to indicate an empty list, but this would introduce parsing difficulties.

There is no distinction between single-item lists and singletons. This feature is extremely convenient, for example, when working with query results where the number of items in the result may be unpredictable (queries are described in Section 5.10 on page 113). However, it does mean nested lists containing one item lose their nested structure. For example, the list `1, (((3))), 5` becomes `1, 3, 5` as soon as it is evaluated by HANDS.

Although HANDS does not support subranges, it would be a natural extension. This would allow longer lists such as `1, 2, 3, 4, 5, 6, 7, 8, 9, 10` to be abbreviated as: `1..10`.

5.6.7 Consistency Between Values on Cards and in Program Code

In HANDS, the syntax for entering and displaying values on cards is identical to the syntax used in program code. This is in contrast to LISP, for example, where values must be quoted in program code. This consistency eliminates the need for people to learn two different formats, and enables values to be copied and pasted from one place to the other. Also, data of any type can be displayed on the board in a readable format, simply by storing it into the back slot of a card.

5.6.8 Comments, Indenting, and White Space

The HANDS language is not sensitive to white space or indenting. Two kinds of comments are provided, like those found in Java and C: one line comments beginning with `//`, and multi-line comments delimited by `/*` and `*/`. It is assumed that the program editor would provide syntax coloring, to flag situations where the programmer accidently comments out more code than intended.

5.6.9 Choices for Keywords and Special Identifiers

The names of special words in the language were chosen to be easy to type in and spell correctly. For example, where several systems use the word `appearance`, HANDS uses the word `back`. Similarly, `speed` was chosen over `velocity`. However, conciseness is not taken to an extreme as it is in some languages, where the programmer is forced to memorize the meanings of many short abbreviations. For example, in StarLogo [Resnick 1994] the `cg` command clears all of the graphical patches, but the mnemonically-similar but much more dangerous command `ca` clears all of the graphical patches and also destroys all of the turtle objects. All of the HANDS keywords are full names. Menus are available to help users enter them correctly (see Section 5.15.2.2 on page 132).

HyperTalk [Goodman 1987] binds the temporary variable `it` to the result of the most recent computation. There are several places in the HANDS language where the system automatically generates temporary variables for referring to a computed value (see Section 5.3.4.1 on page 100 and Section 5.13.1.1 on page 119). I considered using the name `it` for these variables and in some other situations where the programmer might want a shorthand notation for accessing a recently computed value. However, when I wrote some sample code using this feature, I found that the ambiguity of what `it` actually referred to was very confusing. I would write the code with a particular binding in mind, but when I read the code later I would use a different binding. For this reason, the idea was rejected. Instead, the default binding is the name of the property value searched for (see Section 5.3.4 on page 99) or else the user can assign a name (Section 5.13.1 on page 119).

5.7 Statements

This section summarizes the operators in the HANDS language. This is followed by a discussion of expressions in Section 5.8 on page 109. Error messages are discussed in Section 5.16 on page 138.

5.7.1 Operations on Cards

The programmer can instruct Handy to pick up and put down cards, or flip them over:

- `pickup <identifier>`

  The card named `<identifier>` is picked up into Handy's hand, making it invisible.

- `putdown <identifier>`

  The card named `<identifier>` is put down onto the table from Handy's hand, making it visible.

- `flip <identifier>`

  The card named `<identifier>` is flipped over to show the back or front. This is mainly used for interactive debugging, to locate a card. There is no way for program code to determine which side of the card is showing at any given time.

Cards can be created, duplicated, and destroyed:

- `make new card <identifier>`

  A new card is created with the name `<identifier>`.

- `make duplicate of <identifier>`

  A new card is created by cloning all of the properties of the card named `<identifier>`, except it is given a unique name that is generated from the original card's name (for example, a duplicate of the card `rose` is named `rose-copyN`, where `N` is an integer that makes the card name unique).

- `putdown <identifier> onto discard pile`

  The card named `<identifier>` is destroyed.

A runtime error is generated if the program attempts to operate on a card that does not exist, or to create a card with a name that is already in use.

5.7.2 Operations on Card Properties

The programmer can instruct Handy to manipulate properties of cards:

- `set <property> to <expression>`

  The expression is evaluated, and the resulting value is stored into the specified card property. If the card does not already have the named property, it is created. `Set` is a less confusing alternative to using = for assignment.

- `add <expression> to <property>`
- `subtract <expression> from <property>`

  The expression is evaluated and then added to or subtracted from the value in the specified card property, and the result is stored into the property. If the card does not already have the named property, or if the expression and the property are not both numeric values, an error occurs.

- `multiply <property> by <expression>`
- `divide <property> by <expression>`

  The expression is evaluated, and it is multiplied or divided by the value in the specified card property, and the result is stored into the property. Errors are handled the same way they are for `add` and `subtract`.

- `append <expression> to <property>`

  The expression is evaluated and then appended onto the end of the list in the specified card property, and the result is stored into the property. If the card does not already have the named property, an error occurs.

5.7.2.1 Specifying a Property

A `<property>` is specified with a card name and a property name, using one of these two formats:

- `<propertyname> of <cardname>`
- `<cardname>'s <property>`

It is also possible to determine a cardname indirectly, as in this example:

- `nectar of (buzzy's favoriteFlower)`

  Refers to the nectar property of a card whose name is stored in the `favoriteFlower` property on the card named `buzzy`.

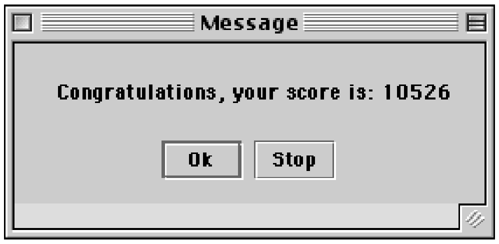
The evaluation of the property dereference operators (`of` and `'s`) has higher precedence than any other operator.

5.7.3 Output Statements

In addition to the ability to display messages on the screen, by placing information into the back properties of cards, HANDS also provides two output statements:

- `tell <expression>`

  The expression is evaluated, any quotes are removed, and if it is a list, the list items are concatenated with a blank space between each item and without the commas. The resulting string is displayed in a modal dialog box Figure 5-6. Execution pauses while the dialog box is up, and the user is offered two choices: to continue running the program or to stop the program. The stop option is necessary because after the user dismisses the dialog box, the system could execute another tell statement so quickly that the user have no time to access the stop button or any of the other controls in the user interface. An expression for getting input from the user is described in Section 5.8.6 on page 112.



Congratulations, your score is: 10526

Figure 5-6. A dialog box similar to this will come up when the statement `tell "Congratulations, your score is:", score's back` is executed. Execution pauses while the dialog box is showing. If the Ok button is pressed, execution continues; if the Stop button is pressed, the program is halted.

- `beep`

  When this statement is executed, the computer beeps.

5.7.4 Other Statements

The two remaining statements, the `with` and `if` control structures, are discussed in Section 5.13 on page 119.

5.8 Expressions

Anytime an expression with more than one operator appears, it must be parenthesized to indicate the order of evaluation. In addition to the expressions listed here, the list operators described in Section 5.12 on page 114 are also accepted anywhere expressions are accepted.

5.8.1 Relational Operators

The following binary relational operators work on all types. If both operands are numeric, a numeric comparison is performed; if both operands are Boolean, a Boolean comparison is performed, where `yes` is greater than `no`; otherwise the operands are treated as strings and are compared lexicographically. These operators each have both mathematical and natural-language variants, separated by vertical bars (`|`) in the lists below:

- `= | equals | equal`
- `> | greater than`
- `>= | greater than or equal`
- `<= | less than or equal`
- `< | less than`
- `<> | not equal`

The word `is` can be used in these expressions because it is ignored, and the word `to` can optionally be appended to these operators. Here are some examples:

- x = 0
- x equals zero
- x is equal to 0

The use of the `=` symbol for the equality predicate matches the way equality is written in other situations such as mathematics, in contrast to the `==` in C and Java.

5.8.2 Boolean Operators

The Boolean operators require Boolean operands:

- `and`
- `or`
- `not`

The first two are binary infix operators, and the latter is a unary prefix operator. Note, we do not expect these to be used except by experienced programmers.

5.8.3 Card Existence Predicate

The programmer can check whether a card exists anywhere in the program, or at a particular location:

- `<cardname> exists`

  Returns true if the card exists anywhere in the system.
- `<cardname> exists in hand`

  Returns true if the card exists, and is in Handy's hand.
- `<cardname> exists on table`

  Returns true if the card exists, and is on the table.

5.8.4 Mathematical Operators

The following infix binary math operators work only on numbers, except where noted:

- `+ | plus`

  This operator will perform string concatenation on its operands if either or both of them are not numbers.
- `- | minus`
- `* | times`
- `/ | divided by`
- `% | modulo`

5.8.5 Random

The `random` function returns an integer between the specified lower bound and upper bound, inclusive. The bounds must be integers. The words `from` and `to` are optional:

- `random [from] <lowerbound> [to] <upperbound>`

5.8.6 Expression for Getting Input from User

The programmer can use `ask` to bring up a dialog box requesting an expression from the user, analogous to the `tell` statement described in Section 5.7.3 on page 109. As with `tell`, execution pauses while the dialog box is up, and the user is offered two choices: to continue running the program or to stop the program. The string argument is displayed as a prompt in the dialog box (see Figure 5-7). The user's input is treated as a string, and is quoted unless it is a legal value without quotes (such as a number).

- `ask <string>`

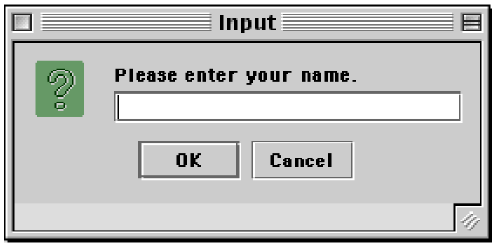


Figure 5-7. If the code `set winner's back to ask "Please enter your name."` is executed, this dialog box comes up. If the user types a value and presses the Ok button, the value is stored into `winner's back`. If the user presses the Cancel button, the program is halted.

5.9 Aggregate Operations

In my studies, I observed that the participants used aggregate operators, manipulating whole sets of objects in one statement rather than iterating and acting on them individually. Most languages require the programmer to use iteration, forcing them to use control structures that are very difficult for beginners (see Chapter 2), and violating the principle of closeness of mapping.

HANDS has full support for aggregate operations. Every operation in HANDS accepts lists as well as singletons for its operand(s), using exactly the same syntax. Binary operators even accept one list and one singleton as operands. The user does not have to correctly anticipate the number of items in the data being operated on.

Here are several examples using the + operator:

- `1 + 1` evaluates to `2`
- `1 + (1,2,3)` evaluates to `2,3,4`
- `(1,2,3) + 1` evaluates to `2,3,4`

- `(1,2,3) + (2,3,4)` evaluates to `3,5,7`

If the operands are lists of unequal length, the items in the shorter list are cycled through.

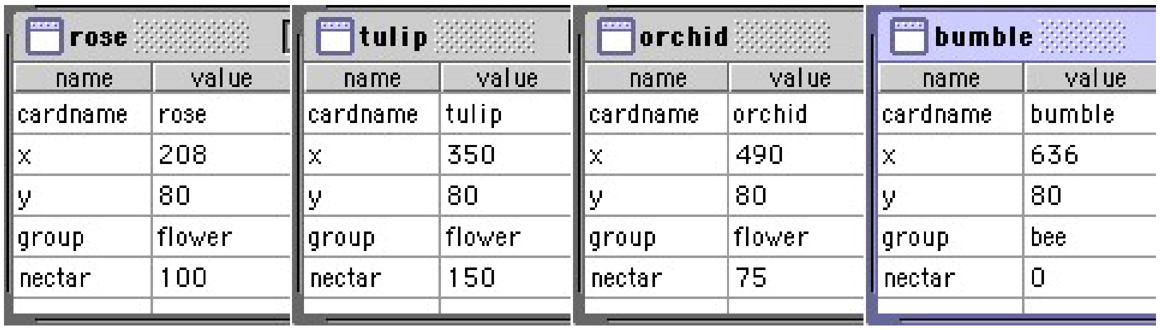
- `(1,10) + (4,5,6,7,8)` evaluates to `5,15,7,17,9`

Additional list operators are discussed in Section 5.12 on page 114.

5.10 Queries

In my studies, I observed that users do not maintain and traverse data structures. Instead they perform queries to assemble lists of objects on demand. For example, they say "all of the blue monsters." HANDS provides a query mechanism to support this. The query mechanism searches all of the cards for the ones matching the programmer's criteria.

Queries begin with the word `all`. If a query contains a single value, it returns all of the cards that have that value in any property. If the value is a word ending in `s`, it will also match cards that have the value without the trailing `s`. Although this simplistic rule does not cover all of the special cases for plurals in English, it seems to work well in practice (see Section 5.6.2 on page 103).



name	value
cardname	rose
x	208
y	80
group	flower
nectar	100

name	value
cardname	tulip
x	350
y	80
group	flower
nectar	150

name	value
cardname	orchid
x	490
y	80
group	flower
nectar	75

name	value
cardname	bumble
x	636
y	80
group	bee
nectar	0

Figure 5-8. When the system evaluates the query all flowers it returns `orchid, rose, tulip.`

Queries return a list of cards satisfying the query, in alphabetical order by cardname. Figure 5-8 contains cards representing three flowers and a bee to help illustrate the following queries.

- `all cards` evaluates to `bumble, orchid, rose, tulip`
  `Cards` is a special keyword that matches every card.

- `all flowers` evaluates to `orchid, rose, tulip`
  `Flowers` is not a keyword, so HANDS searches all of the cards in the system, and returns a list of all cards that have the words `flower` or `flowers` in any slot.
- `all bees` evaluates to `bumble`
  Note that there is no difference between a singleton and a list with one item.
- `all snakes` evaluates to `empty`
  `Empty` is the value representing the empty list in HANDS
- `all (flower and (nectar < 100))` evaluates to `orchid`
  Chapter 4 describes Match Forms, a more effective method for specifying more complex queries like this example. If Match Forms are incorporated into a future version of HANDS, the programmer would have the option to enter this query as it is shown above, or using Match Forms, with the same results.

5.11 Queries and Aggregates in Combination

Queries and aggregate operations work in tandem to permit the programmer to concisely express actions that would require many lines of code in most other languages. For example,

- `set the nectar of all flowers to 0`

This statement is evaluated as follows:

1. the query `all flowers` returns a list of all of the cards containing `flower` or `flowers`.
2. `nectar of` is applied to this list, resulting in a list of properties (see Section 5.7.2.1 on page 108).
3. the `set` statement sets each property to zero in the list of properties.

5.12 List Operators

The system provides a basic set of list operators. Like all operators in HANDS, these operators also accept empty lists and singletons where they accept lists. These operators are non-destructive – they do not modify their operands. If the result of an operation is a list, a new copy is returned. The names of these operators were selected to indicate this fact.

The following list reduction operators accept a list and return a single value (unless there are nested lists). The words inside the brackets (`[ in | of ]`) are optional:

- `Sum [ in | of ] <list>`

  Returns the sum of the items in the list if all items in the list are numeric; otherwise generates a runtime error.

- `AllYes [ in | of ] <list>`

  Returns `yes` if all items in the list are `yes`. Returns `yes` if the argument is `empty`.

- `AllNo [ in | of ] <list>`

  Returns `yes` if all items in the list are `no`. Returns `yes` if the argument is `empty`.

- `FirstItem [ in | of ] <list>`

  Returns the first item in the list. If the argument is a singleton, it is returned; if the argument is `empty`, `empty` is returned.

- `LastItem [ in | of ] <list>`

  Returns the last item in the list. If the argument is a singleton, it is returned; if the argument is `empty`, `empty` is returned.

- `NumberOfItems [ in | of ] <list>`

  Returns the number of items in the list. If the argument is a singleton, `1` is returned; if the argument is `empty`, `0` is returned.

- `ConcatenateItems [ in | of ] <list>`

  Treats all elements in the list as strings, and concatenates them into a single string. Commas are not included in the result, and no extra spaces are inserted between the items.

- `AnyItemIs <expression>[ in | of ] <list>`

  Returns `yes` if any item in the list is equal to the `<expression>`.

- `ItemAtPosition <index> [ in | of ] <list>`

  Returns the item at position `<index>` in the list. If the index is out of range, `empty` is returned.

The following list reduction operators accept a list and return a single value unless there is a tie, in which case a list of the tied items is returned. The programmer does not have to check the number of items returned, because all subsequent operations will accept lists as well as singletons. If the programmer wants a single result, the `FirstItem` operator can be used to extract the first result (and the `ShuffledCopy` operator can be used to randomize which item is extracted by `FirstItem`). Once again, if the list contains nested lists, the value returned may be a list:

- `GreatestItem [ in | of ] <list>`

  Returns the item(s) in the list that is(are) numerically greatest if all items in the list are numeric; otherwise treats all elements as strings and returns the item(s) in the list that is(are) lexicographically greatest.

- `LeastItem [ in | of ] <list>`

  Returns the item(s) in the list that is(are) numerically smallest if all items in the list are numeric; otherwise treats all elements as strings and returns the item(s) in the list that is(are) lexicographically smallest.

The following list operators accept lists and generally return lists:

- `AllButFirstItem [ in | of ] <list>`

  Returns all but the first item in the list. If the argument is a singleton or `empty`, `empty` is returned.

- `AllButLastItem [ in | of ] <list>`

  Returns all but the last item in the list. If the argument is a singleton or `empty`, `empty` is returned.

- `Round [ in | of ] <list>`

  If all items in the list are numeric, a new list is returned where all of the items are rounded; otherwise generates a runtime error.

- `SortedCopy [ in | of ] <list>`

  If all items in the list are numeric, a new list is returned where the items are sorted numerically from least to greatest; otherwise treats all elements as strings and returns a new list is returned where the items are sorted lexicographically from least to greatest.

- `ShuffledCopy [ in | of ] <list>`

  A new list is returned where the items are randomly shuffled.

- `ReversedCopy [ in | of ] <list>`

  A new list is returned where the items are in the reverse order of their appearance in the original list.

- `ConnectedCopy <first list> [ to ] <second list>`

  A single new list is returned where the items of the second list are concatenated to the end of the first list.

The following operators accept a property name and a list of cards, and expect each card in the list to have the named property. The property is inspected on each card to determine an ordering of the cards. Numeric ordering is used if the all of the property values are numeric, otherwise the property values are treated as strings and the order is lexicographical. Once again, if there is a tie, all of the tied items are returned. The ordering is then applied to the list of cards as follows:

- `CardWithGreatest <propertyname> [ in | of ] <cardlist>`

  Returns the card(s) in the list that has(have) the greatest value in the named property, according to the ordering.

- `CardWithLeast <propertyname> [ in | of ] <cardlist>`

  Returns the card(s) in the list that has(have) the least value in the named property, according to the ordering.

- `CardsSortedBy <propertyname> [ in | of ] <cardlist>`

  Returns a new list of cards that is sorted according to the ordering.

Here are some examples of the list operators, referring again to Figure 5-8:

- `FirstItem of all the flowers` evaluates to `orchid`
  `all the flowers` returns a list of the cards containing `flower` or `flowers`, and then `FirstItem` returns the first item in that list.
- `AllButFirstItem of all the flowers` evaluates to `rose, tulip`
  `all the flowers` returns a list of the cards containing `flower` or `flowers`, and then `AllButFirstItem` returns all but the first item in that list.
- `GreatestItem in the nectar of all the flowers` evaluates to `150`
  `all the flowers` returns a list of the cards containing `flower` or `flowers`, `nectar of` returns a list of the nectar properties of those cards, and then `Greates-tItem` returns the greatest item in that list.
- `CardWithGreatest nectar of all the flowers` evaluates to `orchid`
  `all the flowers` returns a list of the cards containing `flower` or `flowers`, `nectar of` returns a list of the nectar properties of those cards, which `CardWith-Greatest` then uses to select the card(s) holding the greatest nectar value(s), which is(are) returned. The difference between GreatestItem and CardWithGreatest is that the former returns the greatest nectar value while the latter returns the name of the card holding the greatest nectar value.
- `SortedCopy of the nectar of all the flowers` evaluates to `75,100,150`
- `Sum the nectar of all the flowers` evaluates to 325

It is interesting to compare the last example (Sum) with how it might look in a typical programming language:

```
int sum = 0;
for (i=0; i<cards.length(); i++) {
      if (cards[i].containsValue("flower")) {
            sum += cards[i].nectar;
      }
}
return sum;
```

This solution requires the programmer to create and maintain a data structure allowing access to all of the flower objects, plus two temporary variables, three kinds of parentheses or brackets, two kinds of punctuation, and the complexities of iteration, function calls, and array indexing.

5.13 Loop and Conditional Control Structures

This section describes the iteration and conditional control structures in HANDS. There is only one of each, but they are flexible enough to cover a variety of control structures in other languages. Each of these structures can accept multiple statements inside each clause, without the need for a `begin-end` block or other grouping mechanism (see Section 5.6.3 on page 104 and Section 5.6.4 on page 104).

5.13.1 Iteration Control Structure

The aggregate and list operators, described in Section 5.9 on page 112 and Section 5.12 on page 114, greatly reduce the need for iteration in HANDS. However, one high-level loop control structure is available, if needed.

```
with <list>
     <statements>
end with
```

The statements inside the `with` statement are evaluated once for each item in the list. If the list is empty, the statements are not executed at all.

5.13.1.1 Automatic Temporary Variable for Iteration

On each iteration, the read-only variable `item` is bound to successive items from the list. I considered heuristically naming the this temporary variable, based on the `<list>` expression, similar to the way temporary names are formed in event patterns (see Section 5.3.4 on page 99). For example, if the `<list>` expression is a query such as `all flowers`, the temporary variable could be named `flower`. However, the `<list>` expression is much less restricted than the event patterns – it could be a more complex query, a literal list, an expression, or a property that contains a list. It was impossible to come up with a heuristic naming scheme that would provide useful and predictable names in all of these cases, so the generic name `item` was chosen.

As an example, the following with statement will put up three dialog boxes, with the messages “1”, “2”, and “3”, beeping right before each dialog box comes up:

```
with 1,2,3
     beep
     tell item
end with
```

If loops are nested, the inner binding will mask an outer binding to the same identifier, so the default identifier `item` cannot be used for both the outer and inner loops, if both need to be accessible inside the inner loop. In this case, the `calling each` variant must be used for at least one of the two `with` statements (see Section 5.13.1.2).

5.13.1.2 Programmer-Specified Temporary Variable for Iteration

The programmer can bind the list items to a different temporary variable name:

```
with <list> calling each <identifier>
     <statements>
end with
```

On each iteration, the read-only variable `<identifier>` is bound to successive items from the list. The words `calling each` were selected to emphasize that the items in the list are individually assigned to the variable. The alternative `calling it` was also considered, but it was rejected because programmers might have naturally interpreted this clause to mean that the variable is an alias for the entire list – where the whole list is assigned to the variable at once – rather than holding the individual items of the list one at a time.

To illustrate this variant, this example will also put up three dialog boxes, with the messages “1”, “2”, and “3”, beeping right before each dialog box comes up:

```
with 1,2,3 calling each i
     beep
     tell i
end with
```

The following example illustrates the use of nested lists. Here nine dialog boxes will come up, with the messages “1 a”, “1 b”, “1 c”, 2 a”, “2 b”, “2 c”, 3 a”, “3 b”, and “3 c”, with a beep right before each dialog comes up:

```
with 1,2,3 calling each i
     with a,b,c calling each j
          beep
          tell i,j
     end with
end with
```

The `with` statement can be used to create temporary bindings, even if iteration is not needed. This example, binds highScore to `game’s score`:

```
with game’s score calling each highScore
     tell “you have the new high score:”, highScore
end with
```

5.13.1.3 Iteration vs. Aggregates

In many situations, aggregates can be used instead of iteration (see Section 5.9 on page 112). However, sometimes iteration is necessary. Consider this example, which sets the nectar properties of the orchid, rose, and tulip cards in Figure 5-8 to separate random values between 50 and 100:

```
with all the flowers
     set nectar of the item to random from 50 to 100
end with
```

This achieves a different effect than if an aggregate operation had been used. In this aggregate assignment, `random` is evaluated only once and all of the flowers would receive the same random value:

```
set nectar of all flowers to random from 50 to 100
```

5.13.2 Conditional Control Structure

A general `if` statement is available in HANDS which incorporates the functionality of `if` statements as well as the `case` and `cond` statements in other languages. These three control structures were unified for consistency. There is no good reason that they should be distinguished as three different structures with unique names and syntaxes. The keyword `otherwise` was chosen in order to have a uniform keyword that emphasizes that the clause is executed only if all of the prior conditions do not execute.

When there are multiple conditions, only the first one to evaluate to `yes` is executed. The `otherwise` clause is optional, and is executed only if none of the previous conditions were executed.

The variants of the `if` statement are:

- A variant that looks like an `if-then-else` statement:

```
if <boolean expression> then
     <statements>
otherwise
     <statements>
end if
```

- A variant that looks like a `cond` statement:

```
if
     <boolean expression>  then <statements>
     <boolean expression>  then <statements>
     otherwise             <statements>
end if
```

- A variant that looks like a `case` statement:

```
if <expression> <relational operator> …
     <expression> then <statements>
     <expression> then <statements>
     otherwise  <statements>
end if
```

The ellipsis in the last variant is necessary to help the parser distinguish it from the other variants. Even with this assistance, the parser has difficulty providing useful error messages when there is a syntax error in this control structure. This is a good example of a trade-off between flexibility and good error messages. It is easier for the system to provide good error messages when the language has a more rigid syntax.

`Otherwise` was selected instead of `else` or `default`, because it makes sense in all of the control structure's variants, helps to convey the idea that it is only evaluated if all of the prior conditions were not evaluated, and is more natural than the others – in natural language, people would usually say, "or else", and would probably never simply say "default".

Here are some examples using the `if` control structure:

```
if flower’s nectar > 0 then
     subtract 1 from the flower’s nectar
     add 1 to the bee’s nectar
end if


if the temperature of the simulation is greater than 10 then
     subtract 10 from the temperature of the simulation
otherwise
     beep
end if


if
     taxform’s amount > 0  then tell “please remit payment“
     taxform’s amount < 0  then tell “refund will be sent”
     otherwise             tell “we’re even”
end if


if player’s score > …
     10000      then tell “your score is excellent!”
     1000       then tell “your score is very good!”
     100        then tell “your score is fair!”
     otherwise  tell “you need more practice!”
end if
```

5.14 Domain-Specific Support

HANDS has domain-specific features that enable programmers to easily create highly-interactive graphical programs. The system's suite of events directly supports this class of programs. The system automatically animates objects, generating events to report collisions among objects as well as input from the user via the keyboard and mouse.

5.14.1 Graphical Objects

It is easy to create graphical objects and text on the screen, as described above in Section 5.2 on page 89. All cards have x and y positions which specify where the top left corner off the object is located. See Section 5.14.5 on page 129 for information about the coordinate system. A back property can be created to specify the appearance of the object. The system automatically determines the extent, or bounding box, of the object for use by the mouse detection and collision detection algorithms. This information is not accessible to the programmer, although it would often be useful for the programmer to have access to this information. For example, to programmatically position an object just to the left of a wall, it is necessary to know where the right edge of the object is. Making this information accessible through additional properties would be a straightforward extension to HANDS.

5.14.2 Animation

Any card that contains integer or floating point numbers in the properties named `speed` and `direction` is automatically animated by the system without any programming. `Speed` is a relative value indicating how many pixels the object should be moved during each time step, and can be a positive or negative. Numbers in the range of about +/- 5 are most useful, because larger numbers cause the object to move too far in each time step, which can make the motion appear jerky and could interfere with collision detection (see Section 5.14.4 on page 127). `Direction` is an angle specified in degrees (0 to 360, but larger and negative values wrap correctly), adopting the convention from math that zero points to the right and the angle increases in a counter-clockwise direction. Since some users may not be familiar with this convention, an image of a compass is shown on the table in HANDS for the user to refer to when working with directions (see Figure 5-1).

5.14.2.1 Method Used by Animation Engine

Each time the animation engine runs, it processes every card that has legal speed and direction values. The new position for the card is calculated as follows:

```
newX = oldX + (speed * cosine(direction))
newY = oldY + (speed * sin(direction))
```

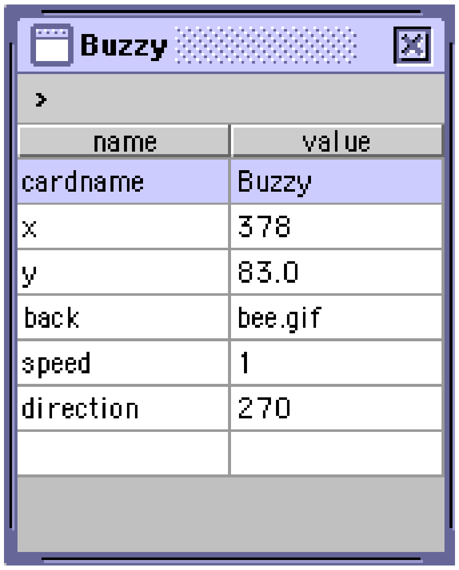
This calculation uses double-precision, and the new floating point position is stored into the `x` and `y` properties of the card. These floating point values are rounded to determine the new pixel location for the object. For small speeds, the animation engine may have to run multiple times before the object moves enough to change its pixel location on the screen.

This sub-pixel method for computing and storing the object's position gives the programmer fine-grained control over the directions of objects, and their relative speeds. This makes it possible to create some kinds of programs such as molecular simulations that are not directly supported in systems that do not have sub-pixel positioning. For example, Stagecast [Joers 1999] uses a grid for positioning objects, and graphical rewrite rules for deciding whether to move an object and to where to move it. Even if the grid is one-pixel in size, it offers only eight adjacent cells for the graphical rewrite rules to move an object into, producing horizontal, vertical and diagonal motions. To implement motions along other angles would require the programmer to supplement the graphical rewrite rules with manual calculation and storage of sub-pixel locations, using formulas similar to the ones built into HANDS.

As long as speed is no greater than 1, the object is guaranteed to move at most one pixel during each run of the animation engine. Faster speeds have an impact on collision detection (see Section 5.14.4 on page 127).

5.14.2.2 Examples of Using Animation

For example, consider the card in Figure 5-9. This card has a speed of 1 and a direction of 270, so when the program is running it would move down slowly. If the speed was changed to 5, it would move faster. It would move up if its speed was changed to -5, or its direction was changed to 90.



name	value
cardname	Buzzy
x	378
y	83.0
back	bee.gif
speed	1
direction	270

Figure 5-9. This card has a speed of 1 and a direction of 270, so it would move down slowly when the program is running. Changing the speed to 5 would make it go down faster, and changing the direction to 90 would make it go up.

This combination of features permits the programmer to implement sophisticated behaviors with only a few lines of code. For example, the following event handlers make the card shown in Figure 5-9 respond to the U, D, L, and R keys to go up, down, left, and right, respectively:

```
when U is typed
     set Buzzy’s direction to 90
end when

when D is typed
     set Buzzy’s direction to 270
end when

when L is typed
     set Buzzy’s direction to 180
end when

when R is typed
     set Buzzy’s direction to 0
end when
```

Another example is, after giving the bees in Figure 5-1 initial speeds and directions, the programmer can use this event handler to make them fly around like bees:

```
when any bee changes
     add random from -5 to 5 to the bee’s direction
end when
```

Each time the system moves one of the bees, an event is generated indicating that the card has changed. This event handler responds to that change by making a small random change to the bee’s direction in the range of -5 to 5 degrees. Note that this change causes another `changes` event to be inserted into the event queue for this bee card. When this new event is eventually removed from the queue, this event handler will run again, once again making a small change to the bee’s direction.

5.14.3 Mouse Click Detection

When the mouse is clicked on the board, mouse click events are generated for each object that is located under the click location. This is determined by checking whether the click is within the bounding box of the image or string that is displayed on the card’s back. For images that are not rectangular and oriented parallel to the x and y axes, clicks near the image but not appearing to touch it may actually be within the bounding box and will therefore generate a clicked event for that object. We hope to fix this in the future.

5.14.4 Collision Detection

The collision detector is responsible for generating events when two objects collide into one another. Once a collision has been reported between a pair objects, no further collisions are reported until they have moved apart.

5.14.4.1 Method Used by Collision Detector

The collision detector is run each time an object’s position is changed. This can happen if the `x` or `y` properties are modified by an event handler, if a card is dragged around while the program is running, or when Handy uses the `speed` and `direction` values to move the object. Only the new position of the object is used in the collision detection calculation.

For each object, the system maintains a collision list holding other objects that are currently in a collided state with this object. If the collision detector determines that there is a collision with an object that is not on this collision list, the collision is reported and the other object is added to the collision list. If the collision detector determines that there is a collision with an object that is already on the collision list, the collision is not reported. If the collision detector determines that an object on the collision list is no longer colliding with this object, the other object is removed from the collision list.

Collisions are detected and reported in a pairwise fashion. The objects that have collided are both listed in the `value` property of the collision event. If three objects collide at once, three separate collisions are reported, one for each pair of objects. This works correctly in the case of a ball striking two walls at a corner, but further investigation is required to determine if this works correctly in more complex multi-object collisions such as with billiard balls.

5.14.4.2 Limitations of the Collision Detector

Collision detection uses bounding boxes, and is subject to the same issue as mouse clicks, where the bounding box may include area that appears to be outside the image. A better collision detection algorithm would use the actual shape of the object to determine when it collides, but that work was outside the scope of this thesis.

When the speed of an object is no more than one pixel at a time, the collision is detected when the objects are butted against one another according to their bounding boxes. A problem arises when an object has a large value in its speed slot. As described in Section 5.14.2.1 on page 124, the animation engine may move the object by multiple pixels in a single step. If the object jumps completely past another object in one step, no collision is detected or reported. Even if the collision is detected, the objects may move enough in one step to penetrate one another, preventing the programmer's collision handler from executing exactly when the objects make contact. Also, in the current system it is difficult for the programmer to determine the direction of the collision or which surfaces of the object collided.

A better method would be for the animation engine and collision detection algorithm to work more closely, calculating a path for the object instead of simply calculating a new position. The collision detector could check for collisions along the entire path, and the algorithm could provide additional information such as the actual point of impact. Other researchers have already addressed these problems (e.g. [Baraff 1989]), but improving the collision detection algorithm was beyond the scope of this thesis.

5.14.5 Coordinate System

In HANDS, the origin of the coordinate system is the top left corner of the screen. X values increase as you move to the right, and y values increase as you move down. I considered tying the coordinate system to the lower left corner of the board, to better match the coordinate system children learn in math class. However, issues arose about what to do when the board is moved or resized: should the cards move with the board or should this cause the coordinates of every card to change? Also, the possibility of negative coordinates seemed to be an unnecessary complexity. In the end, I decided to leave the coordinate system the way it is in virtually all other computer systems, but to do internal transformations to make the angles (directions) work the same as they do in math class: zero degrees to the right, and increasing in a counter-clockwise direction.

5.15 Programming Environment

The HANDS programming environment includes some basic support for building, running, testing, and debugging programs.

5.15.1 System-wide Menu Commands

The following menu commands are always available in the HANDS menu:

- New

  Creates a new blank program, by removing all cards and event handlers from the system. If there are unsaved changes, the user is first given an opportunity to save them.
- Open...

  Displays a file browser dialog box, allowing the user to select a program file to be loaded into the system. Before loading the program, all existing cards and event handlers are removed. If there are unsaved changes, the user is first given an opportunity to save them.
- Import...

  Displays a file browser dialog box, allowing the user to select a program file to be imported into the existing program. All existing cards and event handlers are kept, unless they are duplicated in the imported file. If the imported program has a card with the same name as an existing card, the user is given a choice of replacing the old card, having the system rename the new card to have a unique name, or ignoring the new card. If an imported event handler handles the same event as an existing event handler, the user is given a choice of replacing the old event handler, having the system automatically merge the code of the two event handlers, or ignoring the new event handler. The system merges event handlers by producing an event handler that first lists all of the statements from the original event handler, followed by a few blank lines, and then all of the statements from the imported event handler.

- Save

  Saves the entire program, including all cards and event handlers, to its file. If the program has not yet been associated with a file, this command acts like the "Save As..." command.
- Save As...

  Displays a file browser dialog box, allowing the user to specify a file name and location. The program, including all cards and event handlers, is saved to this file.
- Revert

  After confirmation from the user, restores the program to its state at the last time it was saved. If the program has never been saved, this command will revert the program back to a blank program.
- Quit

  Quits HANDS. If there are unsaved changes, the user is first given an opportunity to save them.

The following menu commands are always available in the Programming menu:

- Open Handy's Thought Bubble

  Brings up the Event Browser (Figure 5-10 on page 131).
- Open Testing Window

  Brings up the Testing Window (Figure 5-13 on page 136).

- Show Card List

  Brings up the Cards window (Figure 5-14 on page 138).

- Show Handy's Hand

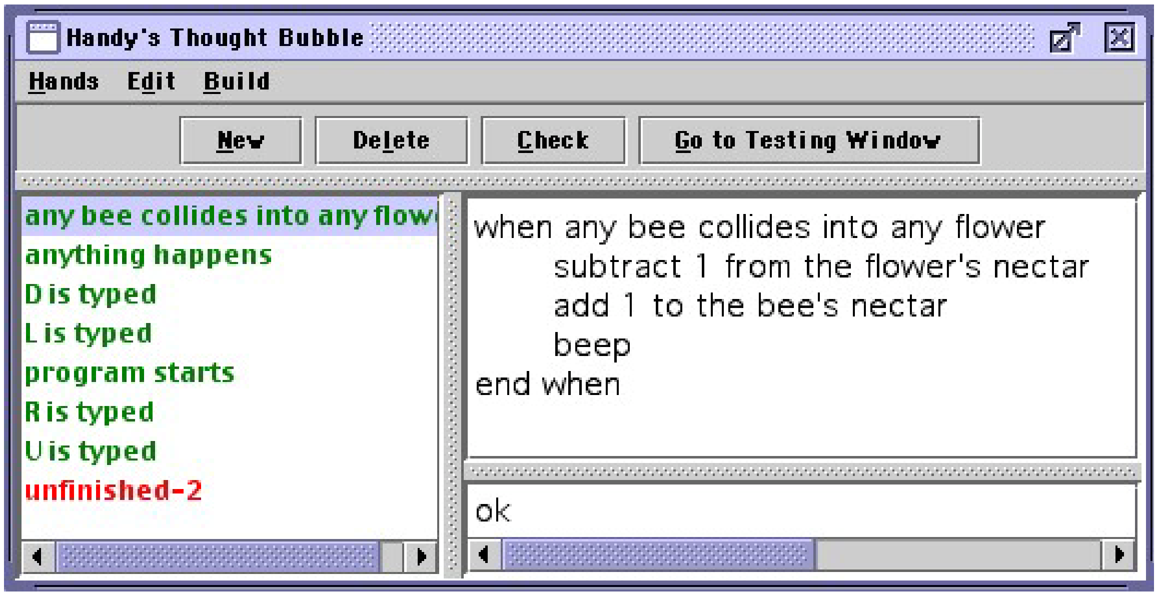
  Hides the table and all objects on the table – including the board, card pile, and the cards that are on the table – and shows Handy's hand (Figure 5-15 on page 139).

- Animate Handy

  This menu item toggles whether Handy's animation is shown when the program is running (see Section 5.2.2 on page 94). This setting defaults to on, but the programmer can turn it off if the animation is distracting.

5.15.2 Event Browser

Figure 5-10 shows the inside of Handy's "thought bubble", which is the browser for event handlers. The left pane of the event browser lists all of the complete and syntactically-correct event handlers in green. Any event handlers with parsing errors are listed in red, and are ignored when the program is run. The code for the selected event handler is shown in the top right pane, and any error is shown in the bottom right pane.



any bee collides into any flow
anything happens
D is typed
L is typed
program starts
R is typed
U is typed
unfinished-2

```
when any bee collides into any flower
      subtract 1 from the flower's nectar
      add 1 to the bee's nectar
      beep
end when
```

```
ok
```

Figure 5-10. HANDS is an event-based system. The left pane lists seven complete (syntactically correct) event handlers, and one that is marked in red because it is not finished (`unfinished-2`). The upper right pane shows the code for when any bee collides into any flower. The lower right pane would report any error messages for this event handler.

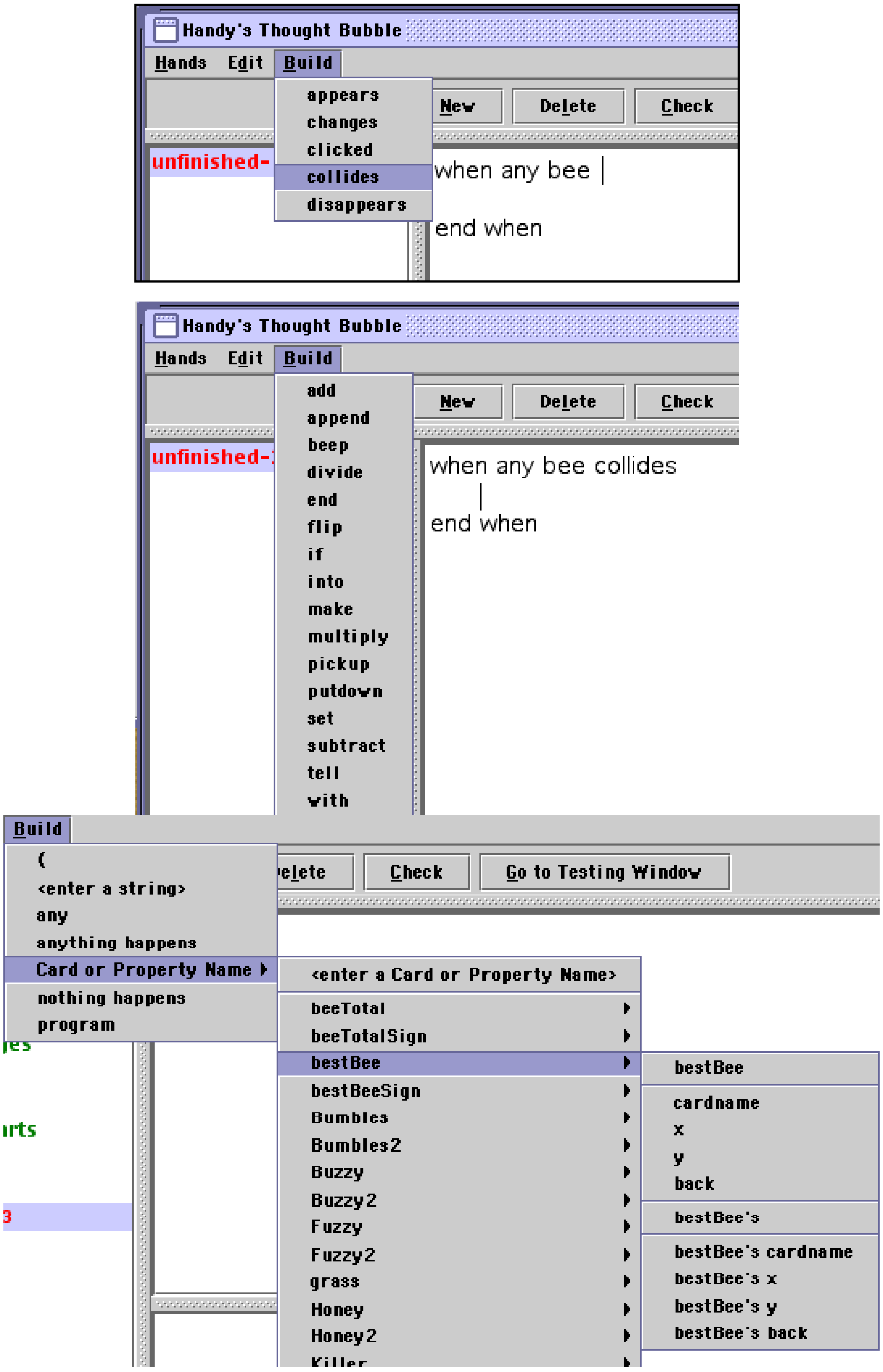
5.15.2.1 Editing Code in the Event Browser

The top right pane of the Event Browser supports freeform text editing, with commands for cut, copy, paste, and multi-level undo and redo. It also allows the programmer to enter program text by selecting items from context-sensitive menus, which saves typing and offers assistance with the language syntax (see Section 5.15.2.2 on page 132). This hybrid approach offers many of the benefits of structure editors (e.g. MacGnome [Miller 1994]), but it is less restrictive. For example, unlike most structure editors, the system allows the programmer to spend large portions of an editing session with syntactically-incorrect code while working towards a solution.

5.15.2.2 Context-Sensitive Menus

The Build menu is context-sensitive, listing the syntactically legal choices at the insertion point (see Figure 5-11). When one of these menu items is selected, the system inserts the text into the program at the insertion point, ensuring that it is surrounded by spaces to keep it separated from adjacent code.

These menus are constructed on the fly when the user clicks on the menu bar, by submitting all of the program text up to the insertion point, but nothing past the insertion point, to the parser. Since this will be an incomplete event handler, the parser will generate an error message. A list of legal choices is extracted from this error message. A small lookup table is used to make some basic transformations to this data, such as coalescing multiple-word sequences into a single menu choice (for example, after selecting “anything” from the context-sensitive menu, the next menu would always have a single choice: “happens”; so the lookup table changes the first menu to: “anything happens”). The choice “identifier” is replaced with a submenu containing all of the cards and properties in the system. Sets of related items such as the list operators are also grouped into submenus. The main menu is sorted alphabetically to help the programmer find the desired choice.The disadvantage of using the parser to generate this menu is that it does not work if there is a syntax error in the text before the insertion point. The JavaCC parser stops at the earlier error and does not continue parsing. In this case, an explanatory message is placed into the menu instead of a list of possible choices. However, any errors after the insertion point do not affect the system’s ability to generate this menu.



```
when any bee

end when
```

```
when any bee collides

end when
```

Figure 5-11. HANDS has context-sensitive menus to assist in constructing correct programs. When a menu item is selected, the text is entered into the program.

5.15.2.3 Parsing Code that has been Edited

The program can be edited whether or not it is running. Edits have no effect on the program until the parser is invoked. This happens when the programmer presses the Check button (see Section 5.15.2.4 on page 134), selects a different event handler in the left pane, or closes the thought bubble window. If the event handler parses correctly, the name of the event in the pane at the left is updated with text describing the event name, and the pane at the bottom right shows the message “ok”.

If there is a parsing error, the name of the event handler is not changed in the left pane, except that it is made red if it was green. The lower right pane shows the error message from the parser. In the upper right pane, the insertion point is moved to the location in the code where the error occurred. The JavaCC-generated parser only reports the first error; subsequent errors are reported only after the first one has been eliminated. Several example error messages are shown in Figure 5-12. The system does some manipulation of the error messages received from the JavaCC parser. For example, the introductory text “There is a problem” is prepended to the error message, quotes are stripped from around each item in the list of expected tokens, and instead of reporting an “end of file error” the message says “I expected more text after this.” More work could be done to improve these messages. For example, the message that lists expected tokens could be modified so that “<INTEGER_ LITERAL>” and “<FLOATING_ POINT_ LITERAL>” are replaced with “a number”.

5.15.2.4 Command Buttons in Event Browser

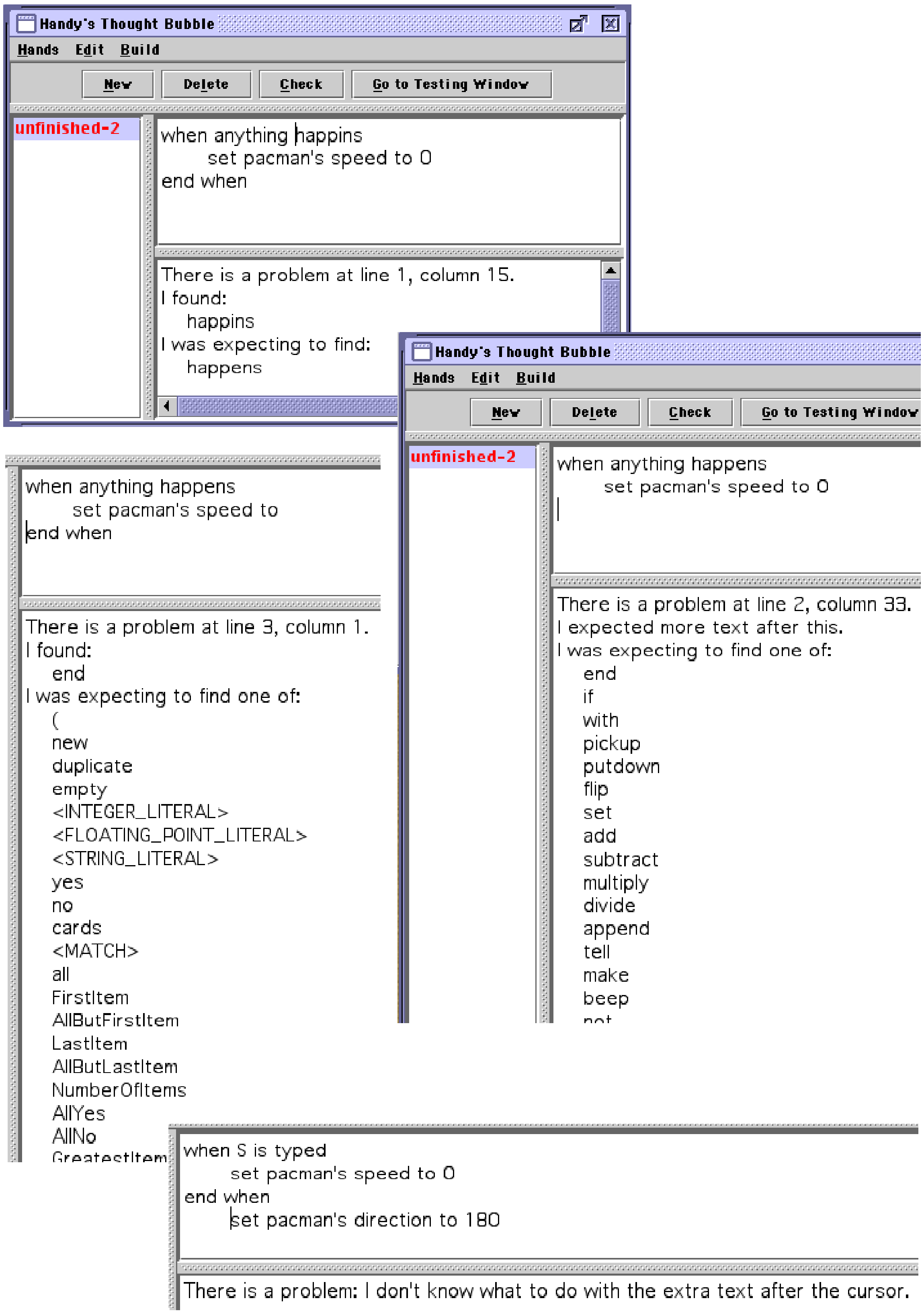
The “New” button at the top of the event browser creates a new event handler in the list on the left, with a name like “unfinished-2”. The name is shown in red because the new event handler is not complete. It starts with this skeleton of an event handler:

```
when

end when
```

The “Delete” button can be used to remove a selected event handler from the program. The system first prompts the user for confirmation.

The “Check” button can be pressed by the programmer at any time. This invokes the parser on the event handler as described in Section 5.15.2.3 on page 134.



```
when anything happins
        set pacman's speed to 0
end when
```

```
There is a problem at line 1, column 15.
I found:
    happins
I was expecting to find:
    happens
```

```
when anything happens
        set pacman's speed to
end when
```

```
There is a problem at line 3, column 1.
I found:
    end
I was expecting to find one of:
    (
    new
    duplicate
    empty
    <INTEGER_LITERAL>
    <FLOATING_POINT_LITERAL>
    <STRING_LITERAL>
    yes
    no
    cards
    <MATCH>
    all
    FirstItem
    AllButFirstItem
    LastItem
    AllButLastItem
    NumberOfItems
    AllYes
    AllNo
    GreatestItem
```

```
when anything happens
        set pacman's speed to 0
```

```
There is a problem at line 2, column 33.
I expected more text after this.
I was expecting to find one of:
    end
    if
    with
    pickup
    putdown
    flip
    set
    add
    subtract
    multiply
    divide
    append
    tell
    make
    beep
    not
```

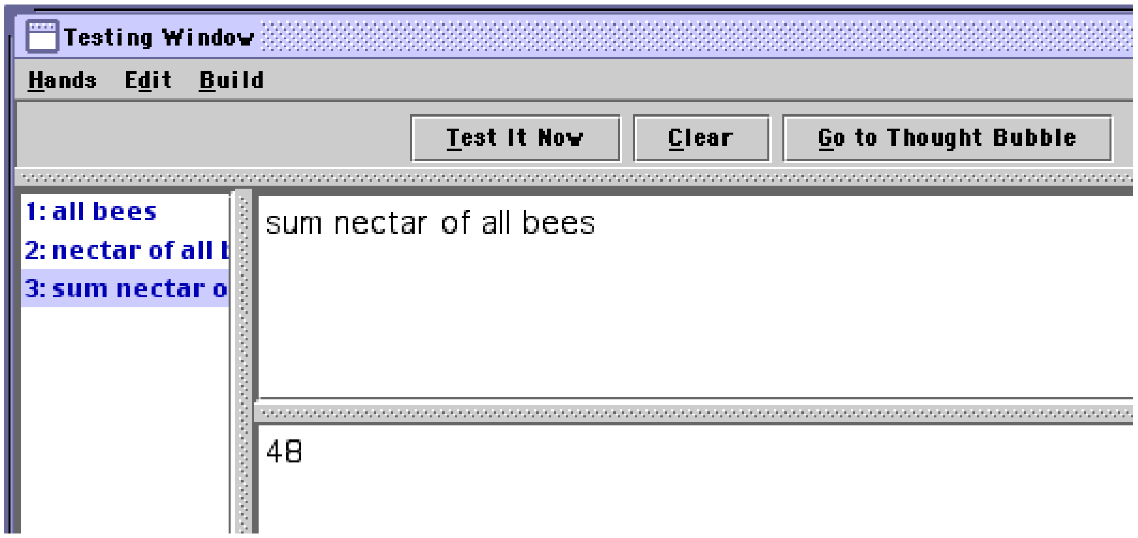
```
when S is typed
        set pacman's speed to 0
end when
        set pacman's direction to 180
```

```
There is a problem: I don't know what to do with the extra text after the cursor.
```

Figure 5-12. This figure shows several examples of the error messages that are displayed when there is a parsing error.

5.15.3 Testing Window

When developing a program, it is useful to be able to test code without having to put the code into an event handler, run the program, and wait for the event to occur. The system provides a Testing Window for this purpose. The Testing Window can be brought up by selecting “Open Testing Window” command from the Programming menu, or by clicking the “Go to Testing Window” button in the Event Browser. The Testing Window is shown in Figure 5-13. Any statement or expression can be entered into the top right pane of this window. When the “Test It Now” button is pressed, the code is parsed, and if there are no parsing errors, then it is evaluated. The lower right pane of this window reports the result of the evaluation if the code was an expression, or the message “ok” if the code was a statement. If there are errors, the lower right pane shows the error message, similar to the Event Browser.



```
1: all bees
2: nectar of all b
3: sum nectar o
```

```
sum nectar of all bees
```

```
48
```

Figure 5-13. The testing window allows statements and expressions to be evaluated immediately. The code is entered in the upper right pane, and the result is shown in the lower right pane. The left pane contains a history of the code that has been tested.

The Testing Window keeps a full history of the code that is evaluated and the evaluation results. The left pane shows this history in a numbered list containing the code that was evaluated. When one of the items in this history list is selected, the right two panes are restored to their state right after that item was evaluated – the code in the top pane and the result of the prior evaluation in the lower pane. The programmer can then execute the code again, with or without editing it, by pressing the “Test It Now” button. In either case, a new entry is made at the end of the history list and the result window is updated with the new result of evaluating the code.

After each press of the “Test It Now” button, as well as any time an item is selected from the history list, the full text in the upper right pane is selected. This is for the convenience of the programmer, who may want to copy it for pasting into an event handler, or to type over it with new code. The “Clear” button can also be used to clear the contents of the top right pane.

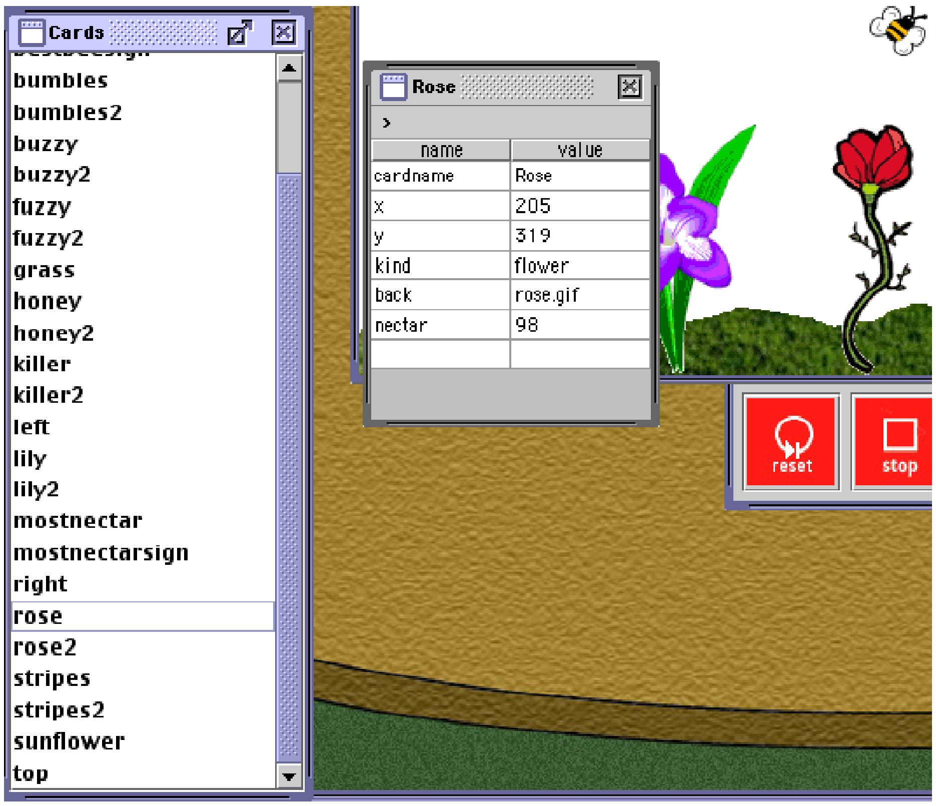
The same menus are provided as in the Event Browser, including menus for cut, copy and paste, multi-level undo and redo, and the context-sensitive Build menus. The “Go to Thought Bubble” and “Go to Testing Window” buttons can be used to quickly move back and forth between the Event Browser and the Testing Window.

5.15.4 Cards Window

The Cards window, shown in Figure 5-14, lists all of the cards in the program. It can be accessed by selecting the “Show Card List” command in the Programming menu. Clicking on one of the cards in this list flips it face-up. This is useful if a card is difficult to locate, is behind another card, or is too small to click on accurately. Using this window to flip cards has no effect on the visibility of a card, which is controlled by whether it is in Handy’s hand (Section 5.15.5) or on the table.

5.15.5 Handy’s Hand

When Handy picks up cards from the table, they become invisible. The menu command “Show Handy’s Hand” allows the programmer to look at the cards in Handy’s hand (Figure 5-15). In this view, the table, board, new card pile, and all the cards that are not in Handy’s hand are invisible. Handy’s picture is changed to highlight his hand. The cards in his hand are shown at the screen positions where they would be if they were put back down without moving them first. The card list window can be used to flip these cards, in the same way it is used in the normal view. The system automatically returns to the normal view when the user selects any menu item (including toggling the “Show Handy’s Hand” menu item) or clicks on Handy’s picture.



bumbles
bumbles2
buzzy
buzzy2
fuzzy
fuzzy2
grass
honey
honey2
killer
killer2
left
lily
lily2
mostnectar
mostnectarsign
right
rose
rose2
stripes
stripes2
sunflower
top

name	value
cardname	Rose
x	205
y	319
kind	flower
back	rose.gif
nectar	98

Figure 5-14. The Cards window (at left) lists all of the cards in the program. When the user clicks one of the card names in this list, the card is flipped face-up.

5.16 Runtime Errors

When there is a runtime error during program execution, the event card that caused the error is displayed face-up on the table, and the error is reported in a dialog box. If the error was generated from the test window, there is no event card. If the error relates to any other card, it is also flipped face-up.

The error messages in HANDS are tuned to be as specific and helpful as possible. For example, in user testing with children, I observed that the most common error was a misspelling or other typographical mistake. This often causes failure to find a card or property that is identified in the code, so the messages for this kind of error were customized to suggest a possible spelling problem. An example is shown in Figure 5-16. In this case, the programmer misspelled the word “nectar”, so the program attempted to access a property that did not exist.

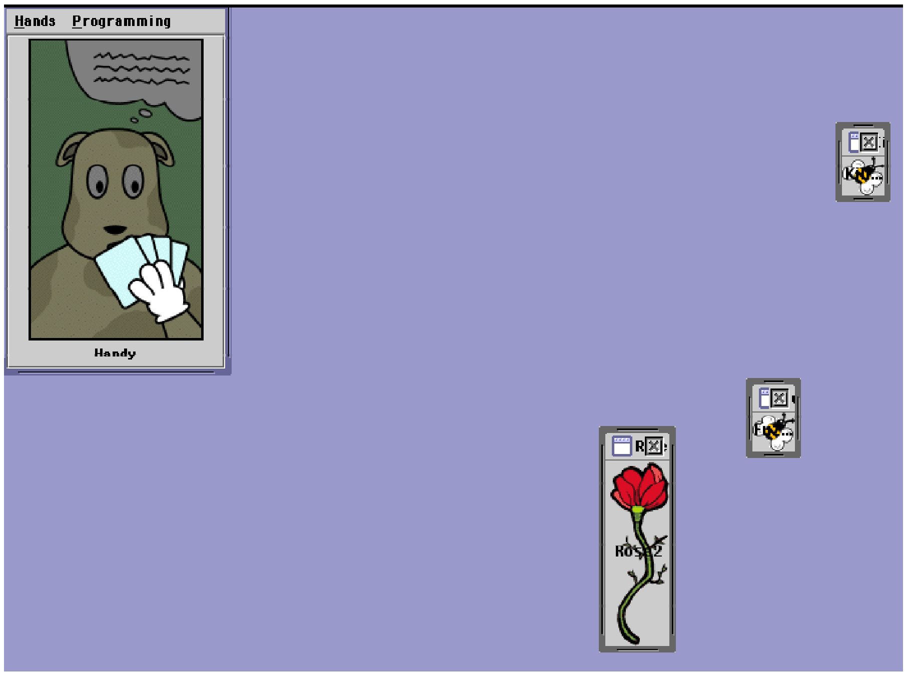
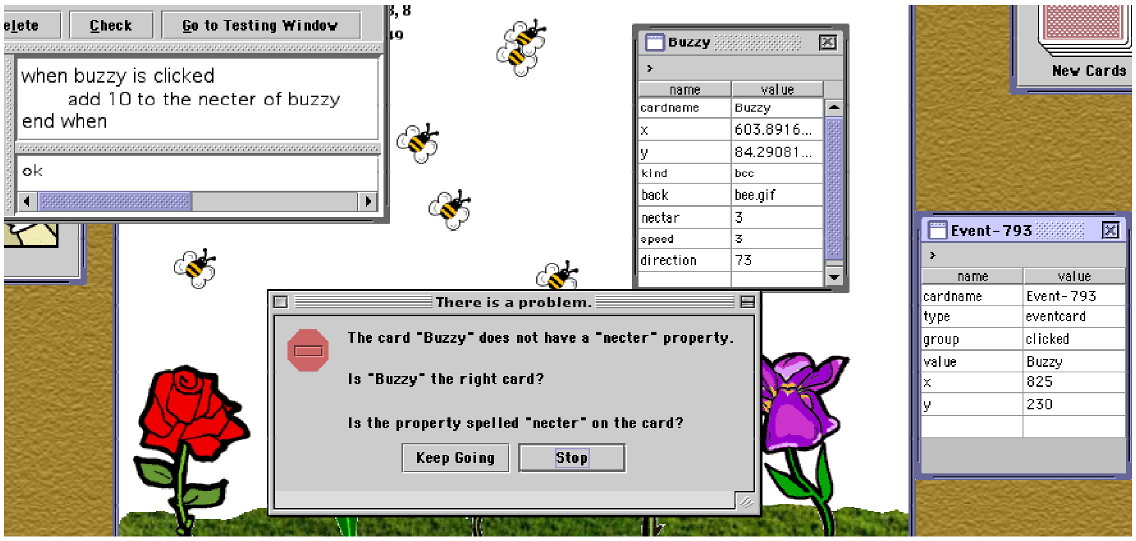


Figure 5-15. When the programmer display's Handy's hand, only the cards that are in his hand are shown, in the positions they would be in on the table. The table, board, new card pile, and all other cards are invisible in this view. Handy's picture is changed to highlight his hand.

The error dialog box offers two choices, to stop the program or to keep going. If the user chooses to keep going, the erroneous expression evaluates to the string ERROR. The ERROR value is propagated to subsequent calculations that depend on the value, similar to the error values in spreadsheets.



```
when buzzy is clicked
      add 10 to the necter of buzzy
end when
```

```
ok
```

Buzzy

name	value
cardname	Buzzy
x	603.8916...
y	84.29081...
kind	bee
back	bee.gif
nectar	3
speed	3
direction	73

Event-793

name	value
cardname	Event-793
type	eventcard
group	clicked
value	Buzzy
x	825
y	230

There is a problem.

The card "Buzzy" does not have a "necter" property.

Is "Buzzy" the right card?

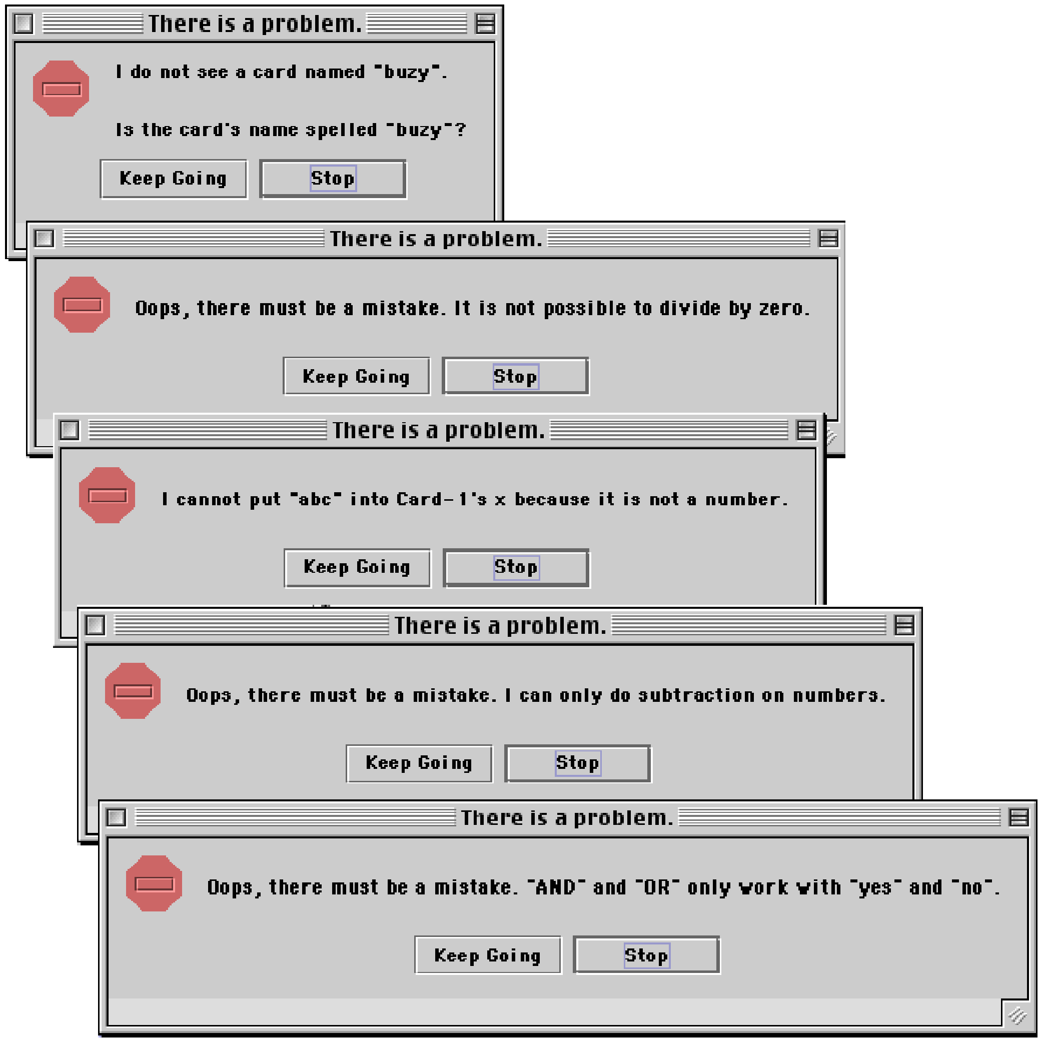
Is the property spelled "necter" on the card?

Keep Going   Stop

Figure 5-16. An example runtime error. Related cards are automatically flipped face-up. If the user chooses to keep going, the erroneous expression returns the string ERROR. The ERROR value is propagated to subsequent calculations that depend on the value, similar to error values in spreadsheets.

Figure 5-17 contains additional examples of error messages for:

1. Referring to a nonexistent card.
2. Division by zero.
3. Attempting to store a string into the x property, which must be numeric. The x, y and cardname properties are the only properties that have restrictions on what can be stored in them.
4. Attempting to perform subtraction on a non-numeric value.
5. Attempting to use the Boolean operators on non-Boolean values.


There is a problem.
I do not see a card named "buzy".
Is the card's name spelled "buzy"?

There is a problem.
Oops, there must be a mistake. It is not possible to divide by zero.

There is a problem.
I cannot put "abc" into Card-1's x because it is not a number.

There is a problem.
Oops, there must be a mistake. I can only do subtraction on numbers.

There is a problem.
Oops, there must be a mistake. "AND" and "OR" only work with "yes" and "no".

Figure 5-17. Examples of error messages for 1) referring to a nonexistent card, 2) division by zero, 3) attempting to store a string into the x property, which must be numeric, 4) attempting to perform subtraction on a non-numeric value, and 5) attempting to use the Boolean operators on non-Boolean values.

5.17 Implementation Details

HANDS is implemented in Java, using the JFC/Swing classes to implement the user-interface. All of the user interface elements are widgets from the Swing toolkit. The main HANDS application uses a JDesktopFrame, and all of the other windows are JInternalFrames. The system makes heavy use of the Java2 collections classes, which I back-ported into the pre-Java2 (Java 1.1) runtime that is available under Macintosh OS 9.

The parser was generated from a grammar description using JavaCC/JJTree [Webgain 2001]. One of the advantages of JavaCC is that it produces top-down recursive descent parsers, which can be used to begin parsing at any non-terminal. It has flexible lookahead capabilities that permit most of the grammar to efficiently parsed as LL(1), but at complex points the grammar I could specify greater lookahead amounts to resolve ambiguities. The lookahead can be specified syntactically, rather than specifying a fixed number of tokens. JJTree produces a parse tree, which is walked by the interpreter.

Simpler programs written in HANDS execute at barely adequate speeds on the five-year-old computers I used for development and testing. In the user study described in Chapter 6, speed was not an issue for the participants. However, programs with large numbers of objects or very complex computations run too slowly, and garbage collection or thread scheduling cause pauses in animations. I believe this slowness problem is not intrinsic to the model of computation or the event based paradigm, and could be addressed by looking for ways to improve efficiency (such as by compiling, reducing the burden on the garbage collector, caching numeric values to reduce conversions back and forth to strings, etc.).

The system is comprised of about 44 non-anonymous classes, about 80% of them hand-written and the rest generated by JavaCC, and it is about 20,000 lines of code, about 50% hand-written code and 50% generated by JavaCC.

5.17.1 HANDS Runtime Implementation

Currently, all of the HANDS runtime processing occurs in a Swing Timer thread, which is set to be called as frequently as possible. The event handler runs until the event queue is emptied or it has dispatched 20 events, then the animation engine runs for one time step, and then control is released by the Timer thread. The Swing user interface code takes over and performs screen updates and any other processing it needs to do, and another cycle is begun as soon as Swing gives control to the Timer thread again.

This strategy means that animation speeds are dependent on processor speed, the number of events generated by the program, and on the complexity of the code inside the event handlers. It is not possible to always just wait until the event queue is empty because some programs may always generate new events (card changed events) in the course of processing events.

The choice of 20 events dispatched per cycle was determined by experimenting with various values and selecting one that balances the tradeoff between good animation performance and the ability of the event processor to keep the event queue reasonably empty and to respond quickly to events. However, this choice is dependent on the particular HANDS program that is running. It works well for smaller programs, but is not optimal for programs with a very large number of animated objects. For example, each object that is moved by the animator generates a card changed event, so, without even considering other events such as collisions, if there are more than 20 moving objects it will not be possible for the event processor to dispatch all of the events in the queue before the animator runs again. One optimization that would partially relieve this problem would be to insert card changed events into the event queue only if there is an event handler actually watching for them.

If the animator runs before the event processor has been able to empty the event queue, a problem with collision detection arises. For example, suppose the program is written to stop an object or change its direction when it has a collision. If the collision event is not processed during the 20 events that are processed in a cycle, the animation engine will run again, moving the object further before the program has had an opportunity to respond.

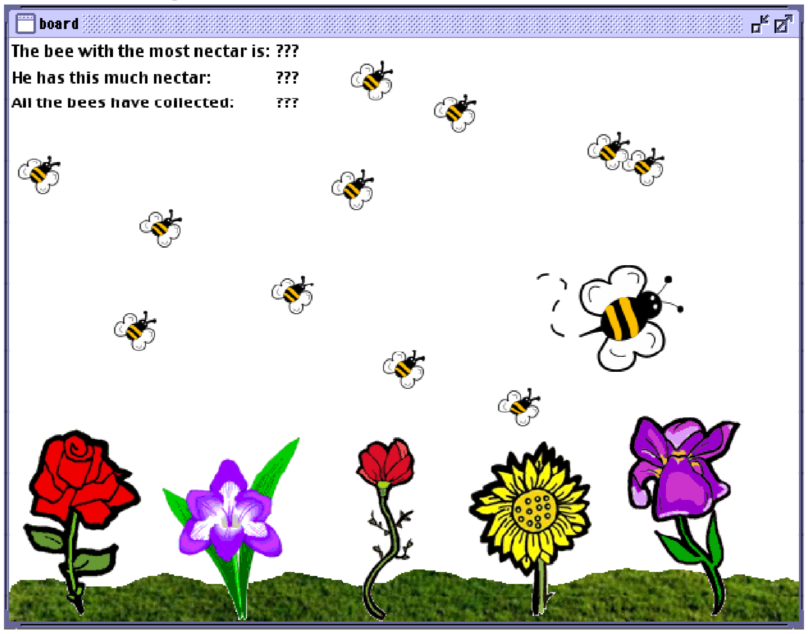
I experimented with running the animation engine in a separate thread from the event processing code, so that it would run on a more regular schedule, even if the event processing code was bogged down by very complex event handlers and a full event queue. This actually compounded the problem mentioned above regarding collision detection, because this removed the guarantee that 20 pending events would be handled before the next time the animation engine was run.

These problems arise due to a lack of computational resources. Using a faster processor or compiling HANDS code instead of interpreting it, would relieve this problem. With faster execution, the animation engine would run on a regular schedule and the event processor would have enough computational power to keep the event queue nearly empty.

5.17.2 Format for Saved Files

HANDS programs are saved as ordinary text files. These files are not intended to be edited by HANDS users, but programmers could use external text editors to edit these files, for example, to select parts of a program to copy into another file. In these files, cards are listed on a single line, beginning with the card name, followed by a space-separate list of property:value pairs, and ending with a semicolon. Event handler code is stored exactly the way it is seen in the Event Browser. In order for the parser to be able to read in syntactically incorrect event handlers, they stored inside special comment symbols($) which are used exclusively for this purpose.

5.18 Sample Program



```
The bee with the most nectar is: ???
He has this much nectar:         ???
All the bees have collected:     ???
```

Figure 5-18. In this example program, bees fly around collecting nectar from flowers.

This section describes the entire code for the program shown in Figure 5-18, where bees fly around collecting nectar from flowers. The large bee named `Buzzy` can be controlled by typing keys indicating which direction it should fly. Appendix B has additional examples of programs that have been build in HANDS.

- The cards for all of the bees:

```
Bumbles x:530 y:60 kind:bee back:bumbleb.gif nectar:5 speed:1 direction:261;
Bumbles2 x:465 y:35 kind:bee back:bumbleb.gif nectar:5 speed:1 direction:241;
Buzzy x:610 y:194 kind:bee back:bee.gif nectar:3 speed:3 direction:464;
Buzzy2 x:677 y:102 kind:bee back:bumbleb.gif nectar:6 speed:1 direction:267;
Fuzzy x:450 y:120 kind:bee back:bumbleb.gif nectar:3 speed:1 direction:279;
Fuzzy2 x:580 y:290 kind:bee back:bumbleb.gif nectar:7 speed:1 direction:296;
Honey x:280 y:230 kind:bee back:bumbleb.gif nectar:7 speed:1 direction:297;
Honey2 x:490 y:260 kind:bee back:bumbleb.gif nectar:1 speed:1 direction:253;
Killer x:650 y:90 kind:bee back:bumbleb.gif nectar:0 speed:1 direction:264;
Killer2 x:205 y:108 kind:bee back:bumbleb.gif nectar:0 speed:1 direction:265;
Stripes x:300 y:150 kind:bee back:bumbleb.gif nectar:8 speed:1 direction:273;
Stripes2 x:403 y:202 kind:bee back:bumbleb.gif nectar:3 speed:1 direction:292;
```

- The cards for all of the flowers:

```
Lily x:318 y:329 kind:flower back:lily.gif nectar:94;
Lily2 x:675 y:308 kind:flower back:lily2.gif nectar:96;
Rose x:205 y:319 kind:flower back:rose.gif nectar:98;
Rose2 x:465 y:328 kind:flower back:rose2.gif nectar:94;
Sunflower x:555 y:330 kind:flower back:sunflower.gif nectar:92;
```

- The card for the grass:

```
grass x:198 y:431 back:grass.gif kind:hwall;
```

- The on-screen text:

```
beeTotal x:406 y:60 back:"???";
beeTotalSign x:200 y:60 back:"All the bees have collected:";
bestBee x:406 y:20 back:"???";
bestBeeSign x:200 y:20 back:"The bee with the most nectar is:";
mostNectar x:406 y:40 back:"???";
mostNectarSign x:200 y:40 back:"He has this much nectar:";
```

- These event handlers allow the user to type keys to make Buzzy, the large bee, move down, left, right and up:

```
when D is typed
       set buzzy's direction to 270
end when
when L is typed
       set buzzy's direction to 180
end when
when R is typed
       set buzzy's direction to 0
end when
when U is typed
       set buzzy's direction to 90
end when
```

- This event handler transfers one unit of nectar from a flower to a bee that flies into it:

```
when any bee collides into any flower
       subtract 1 from the flower's nectar
       add 1 to the bee's nectar
       beep
end when
```

- This event handler updates the on-screen text when necessary:

```
when any flower changes
       set bestBee's back to cardwithgreatest nectar of all bees
       set mostNectar's back to nectar of (bestbee's back)
       set beeTotal's back to sum nectar of all bees
end when
```

- This event handler initializes each bee to fly in a random direction

```
when program starts
       with all bees calling each b
              set direction of b to random 0 to 359
       end with
end when
```

5.19 Importing Components

HANDS programs can be extended by importing one or more existing programs. The system integrates the new program by adding the cards to the table and adding the event handlers to the thought bubble. If a handler exists for a particular event in both programs, the system offers to merge the code automatically. This makes it very convenient to build and use a library of small autonomous objects, each as a small program with one card and the code to control its behavior.

For example, the bees in Figure 5-18 do not stop at the edges of the board. However, HANDS comes with a program called “Boundaries” that can be imported into any other program. This contains invisible cards that are positioned along the edges of the board, and code that responds to collisions by changing the direction of the colliding object to turn it around. If this program is imported into the program in Figure 5-18, the bees would bounce when they reach the edge of the board. A similar program can be imported to create an invisible “trap-door” that causes objects to be teleported to a different location on the board.

A compass card contains a list of directions, like north, south, east, west, up, down, left, right, etc. Importing this compass program would allow the programmer to use symbolic directions instead of numeric directions in their code, such as `compass’s north`.

Chapter 7 describes additional ideas I have about modularity and encapsulation of program components.

5.20 Summary

The unique set of features in HANDS is a direct result of the human-centered design process I used. The implementation demonstrates the feasibility of the HANDS model for representing computation. Chapter 6 presents a user study as well as less formal evaluations of HANDS.

CHAPTER 6 Evaluation

There are several ways that HANDS can be evaluated. One way would be to conduct a user study comparing the overall HANDS system with other programming systems for children such as Logo [Papert 1980] or Stagecast [Smith 1994]. This kind of study would assess the entire system, but if HANDS was found to be statistically better or worse than the other systems, the study would not yield information about which features contributed to these differences and in what proportion. It is also possible that some of the features enhance the usability of HANDS and others detract from usability, but these effects would cancel each other out in the study results. This kind of study also requires great care to perform rigorously, because there are so many differences between HANDS and the other environments that could confound the results.

On the other hand, an endless number of studies could be done to examine the effectiveness of individual features of HANDS. In these studies, one would try to isolate the features and test them without confounds. If particular features are shown to be statistically better than the features they replace, it would suggest that the features would be useful in other programming systems. However these results would not tell us how the individual features work together to enhance or hinder performance, or whether the system as a whole is more effective than other systems.

In the context of this thesis, I decided to conduct a study that falls somewhere in the middle. I chose to examine three key features of HANDS that are not found in most popular programming systems: queries, aggregate operations, and the high visibility of program data. In a sense the study skims off some the most likely parts of HANDS to have an impact, and tests them in a rigorous way.

I designed the study to isolate these key features and compare them with realistic alternative methods like the ones required in most programming systems, while controlling all other aspects of the programming experience. This evaluation can only answer questions about the collective impact of the three features, and not their individual contributions. It also says nothing definitive about the overall effectiveness of the HANDS system relative to other systems. However, if these key features are shown to improve performance in this study, it would suggest that they may also be effective collectively in other future and current programming systems. The results of this study can also help us to form hypotheses about both the individual contributions of features and the overall effectiveness of HANDS, to guide further evaluation in the future.

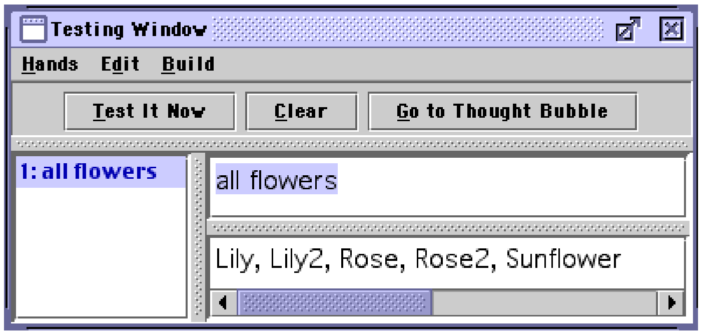
This chapter describes the user study of three key features of HANDS, and concludes with some additional, less formal, evaluations.

6.1 User Study

The study examines the effectiveness of three features of HANDS: *queries, aggregate operations, and data visibility.* For this study, a comparison system was constructed by taking the HANDS source code and disabling these features. All other aspects of the system were identical between the two conditions. The comparison system is still fully capable, because HANDS contains alternative features that can be used to solve any programming problem. These alternative features are realistic, because they are the features that must be used in most other programming systems. In essence, these alternatives are: to create and maintain data structures, to use iteration to operate on groups of objects one at a time, and to use debuggers or inspectors to view program data.

6.1.1 Queries and the Alternative

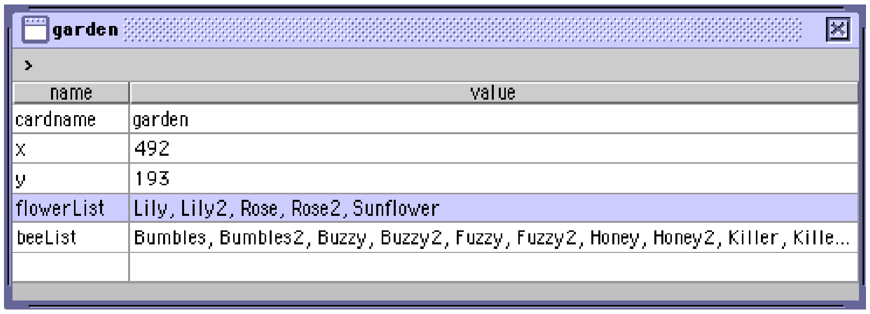
The HANDS query feature allows the programmer to assemble lists of objects on demand, by asking for all the objects with data matching certain criteria. For example, in Figure 6-1 the query `all flowers` searches for all of the cards that contain the string "flower" or "flowers" in any slot and returns a list of their names (e.g., `Lily, Lily2, Rose, Rose2, Sunflower`). HANDS supports more complex queries, but only this simple keyword form of query was used in this study.



```
all flowers
Lily, Lily2, Rose, Rose2, Sunflower
```

Figure 6-1. In HANDS, programmers can use content-based queries to create lists of cards.

Most programming systems do not have a query feature. In those systems, the programmer must create and maintain data structures that provide access to the desired information. This is also necessary in the limited version of HANDS. For example, the programmer could create a card that holds a list of all of the flowers, as shown in Figure 6-2. This list has to be updated each time a flower is added or removed from the program.



name	value
cardname	garden
x	492
y	193
flowerList	Lily, Lily2, Rose, Rose2, Sunflower
beeList	Bumbles, Bumbles2, Buzzy, Buzzy2, Fuzzy, Fuzzy2, Honey, Honey2, Killer, Kille...

Figure 6-2. Other programming systems require the programmer to create and maintain data structures to keep track of the program's data. This garden card has lists of all of the flowers and bees in the system. When one of these objects is added or deleted, the list must be updated.

6.1.2 Aggregate Operators and the Alternative

In HANDS, all operations can be performed on a whole list of objects, including query results, with a single command. For example, the code in Figure 6-3 will set the nectar properties of all of the flowers to zero, no matter how many flowers there are. Continuing with the above example in Figure 6-1, the Lily, Lily2, Rose, Rose2, and Sunflower cards would all have their nectar properties set to zero.

```
set the nectar of all flowers to 0
```

Figure 6-3. In HANDS, all operations can applied to lists of objects.

Most other programming systems do not support aggregate operations. In those systems, the programmer must iterate over the list of objects, operating on them one at a time. This is also the case in the limited version of HANDS, where the example shown in Figure 6-3 can be accomplished by the code shown in Figure 6-4.

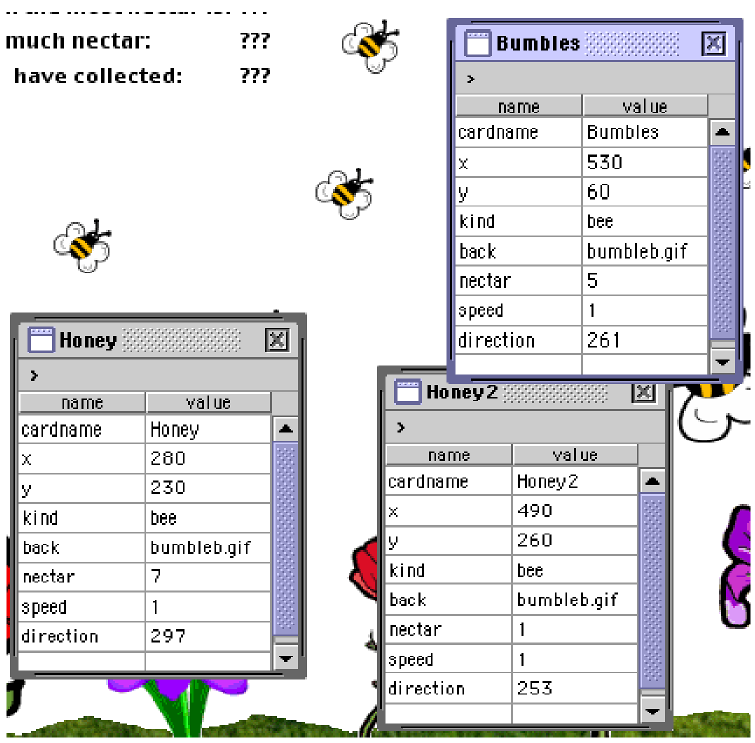
```
with garden’s flowerList calling each the flower
     set the nectar of the flower to 0
end with
```

Figure 6-4. Without aggregate operations, iteration must be used to operate on groups of objects one at a time.

6.1.3 Visibility of Data and the Alternative

All data in HANDS is stored on cards, in name-value pairs called properties (Figure 6-5). Cards are always visible, even when the program is not running. They can be created and edited by direct manipulation as well as by actions taken by the program itself. The properties of multiple cards can be viewed simultaneously.

Traditional programming systems often do not provide these features for data. Variables might exist only temporarily while certain parts of the program are running. Data may not be visible to the programmer unless a debugging tool is used. In some systems, objects can only be created by executing code, and they do not exist when the program is not running.



much nectar: ???
have collected: ???

Bumbles

name	value
cardname	Bumbles
x	530
y	60
kind	bee
back	bumbleb.gif
nectar	5
speed	1
direction	261

Honey

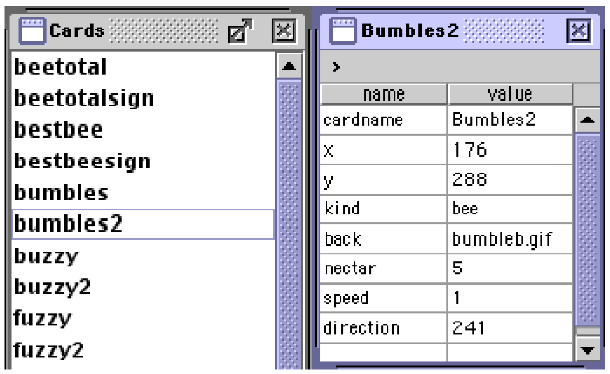
name	value
cardname	Honey
x	280
y	230
kind	bee
back	bumbleb.gif
nectar	7
speed	1
direction	297

Honey2

name	value
cardname	Honey2
x	490
y	260
kind	bee
back	bumbleb.gif
nectar	1
speed	1
direction	253

Figure 6-5. In HANDS, all data is stored on cards, which are visible and persistent. The properties of multiple cards can be viewed at the same time.

The limited version of HANDS has only minor restrictions compared with these other systems. In the limited version, cards are only visible if they represent on-screen objects, although all cards can be inspected, one at a time, using the Cards window (Figure 6-6). This mechanism that is similar to the property inspector in Visual Basic.



Cards

beetotal
beetotalsign
bestbee
bestbeesign
bumbles
bumbles2
buzzy
buzzy2
fuzzy
fuzzy2

Bumbles2

name	value
cardname	Bumbles2
x	176
y	288
kind	bee
back	bumbleb.gif
nectar	5
speed	1
direction	241

Figure 6-6. Many programming systems do not have visible representations of all data, and require the use of a debugger or inspector to view data.

6.2 The Study

6.2.1 Participants

Volunteers were recruited from the fifth-grade class at a public elementary school in Pittsburgh. The students in this school are diverse in race, socio-economic status, and academic achievement. The 23 volunteers ranged in age from 9 to 11 years. There were 12 girls and 11 boys. All were native speakers of English, and none had computer programming experience. The participants came to the Carnegie Mellon campus on one of two Saturday mornings for a three-hour session, and were paid $20 for their participation. On one Saturday, 12 participants used the full-featured HANDS system (*Full*), and on the other Saturday 11 participants used the limited system (*Limited*).

6.2.2 Materials

Appendix G contains copies of all of the materials used in this study. This includes the tutorials and tasks described below, along with solutions to the tasks for each of the two conditions.

In the *Full* condition, a 13-page tutorial was used to teach the participants the basics of the HANDS system. The tutorial began with an empty program, and the participants built a program with several flowers and a bee that flies around collecting nectar from them. The bee is controlled by keyboard commands, and the program displays some basic statistics about which flower has the least nectar and the amount of nectar the flowers have.

The tutorial for the *Limited* condition was derived from the full-featured tutorial. Those portions utilizing a feature that was missing in the limited system were replaced with material teaching the easiest way to use the system's remaining features to achieve the same result. This modification increased the size of the tutorial by one page, to 14 pages.

After completion of the tutorial, participants were given a two-page set of five tasks, plus an optional bonus problem. Each of these tasks was selected because its solution would make use of at least one of the features that are missing in the *Limited* system. All participants started the tasks by loading a partially implemented program. This program was similar to the one they had been working on, but it had more bees and some pre-defined cards to help solve the tasks. Once again, these materials were constructed to be as similar as possible in the two conditions, differing only where necessary due to the limitations of the reduced-feature version of HANDS.

6.2.3 Procedure

The participants worked individually, at their own pace. When they finished the tutorial, they immediately started on the tasks. They were permitted to continue referring to the tutorial while solving the tasks, and the task descriptions had references to relevant pages of the tutorial. Participants could stop working before the three-hour session was over if they finished the tasks, or if they wanted to quit for any other reason. At the end of the session, the participants filled out a brief questionnaire, providing information about their prior computer experience and indicating how much they enjoyed the activity.

During the sessions, the experimenters answered the participants questions and helped with any problems that the participants encountered, unless the assistance would reveal part of a task solution. In such a case, the experimenters simply referred the participants to material in the tutorial that might be helpful.

6.2.4 Results

Overall, the children enjoyed the activity. The average rating for enjoyment was 4.3 on a scale of 1 to 5. The threshold I used for testing significance was $p<.05$. There was a marginally significant difference in enjoyment between the two groups, although the trend was in favor of the Full condition (4.5 to 4.0). In the Full condition, the children tended to rate the level of difficulty to be lower than in the Limited condition (2.8 to 3.5), but this difference was also only marginally significant.

There was no significant difference in performance between boys and girls. All of the children were able to accomplish some programming by following the explicit instructions in the tutorial, and most of them completed the tutorial: in the *Full* condition, 75% (9 of 12) of the participants completed the tutorial and began to work on the tasks; and this ratio was 82% (9 of 11) in the *Limited* condition. This difference is not significant, and the remainder of this analysis examines only the participants who achieved this level of success.

On average, the participants in the *Full* condition spent 121 minutes working on the tutorial, while the participants in the *Limited* condition spent 139 minutes. This difference is not significant, but it does mean the participants in the *Full* condition had more time available to complete the tasks. Indeed, on average the participants in the *Full* condition spent more time on the tasks, 36 minutes compared to 30 minutes in the *Limited* condition. However this difference is also not significant. These times spent working on the tutorials plus tasks do not add up to the full 180 minute session (3 hours) because the participants took breaks or stopped working early.

In the *Full* condition, seven participants solved at least one problem correctly, while in the *Limited* condition only one participant achieved this. This difference in the number of students completing at least one task is significant (p<.05).

Participants received one point for each task problem they completed correctly. No partial credit was given. With the bonus problem, the maximum score was 6. Of the nine participants in the *Full* condition, the scores ranged from 0 to 6, with an average of 2.1. Cumulatively, the participants in the *Full* condition received 19 points. Of the nine participants in the *Limited* condition, the scores ranged from 0 to 1, with an average of 0.1. Cumulatively, the participants in the *Limited* condition received only 1 point. This difference in the number of points scored is significant (p<.05). These results are summarized in Table 6-1.

Table 6-1. Summary of results from this study. Participants using the *Full* system performed significantly better on the tasks than participants in the *Limited* condition.

	Full	*Limited*
Total number of participants	12	11
Participants completing the tutorial	9	9
Participants solving at least one task correctly (p<.05)	7	1
Cumulative number of tasks solved (p<.05)	19	1

One question that arises is whether the extra time that the *Limited* participants spent working on the tutorial, and the corresponding decrease in the amount of time spent working on the tasks, can account for the difference in performance. This is unlikely. The participants in the *Full* condition were, on average, able to solve 2.1 tasks in 36 minutes, which is a rate of about 17 minutes per solution. The participants using the *Limited* system had an average of 30 minutes to work on the tasks, so if they were indeed capable of achieving the same problem solving rate as the other participants, they should have had adequate time to solve at least one problem per person on average. They did not come close to achieving this.

6.2.5 Informal Observations

The experimenters took notes about any interesting things they observed the participants doing. Sometimes the participants asked for help with their problems, and sometimes they were able to figure out the solution without help, and the experimenter simply observed over their shoulders. Two of the most common observations are listed here.

- The participants made many spelling errors. After seeing a lot of spelling problems during pilot testing for this study, the error messages for when the system attempts to access non-existent cards or properties were changed to explicitly suggest the possibility of spelling problem (for example, see Figure 5-16). However, there are situations where the system cannot determine that there is an error. For example, if the property name is misspelled when using the `set` command, the system automatically creates a new property instead of reporting an error. While this behavior is often convenient, more spelling errors would be detected if this were an error and an explicit command was required to create a new property.
- The participants assumed the system's capabilities and vocabulary is much larger than it is. After successfully learning several statements in the HANDS language, the participants often tried typing in their own natural language commands that are not part of the language. This occurred even though the tutorial was careful to point out that Handy is not very intelligent and has limited vocabulary. This problem is to be expected when the syntax of the language is verbose and like natural language. It is interesting to note that several participants in the *Limited* condition were observed typing commands that use query and aggregate features, even though those features were not available and were not mentioned in the tutorial. For example, one participant spontaneously typed, "set all bee's speed to 0," which would have actually worked correctly in the *Full* system.

6.2.6 Summary of Study

The superior performance of participants in the *Full* condition can be attributed to the presence of queries, aggregate operations, and data visibility in the system they used. This suggests that these features could improve the usability of programming systems in general. However, the study does not tease apart the contributions of the individual features. However, it is my conjecture that the largest portion of the impact came from the combined power of queries and aggregates, and that the visibility differences made little contribution to the difference in performance.

This study also does not provide any evidence whether the HANDS system as a whole is better than other programming systems. However, in a three hour session, children who had never before programmed were able use a tutorial to learn how to program in HANDS, and then go on to solve additional programming problems. This, in itself, is a success.

6.3 Example Programs

In order to assess the suitability of HANDS for building larger programs, an undergraduate computer science student working on a one-semester independent study project, used HANDS to build a game and a simulation. In addition, I implemented programs to solve the Towers of Hanoi problem, and to compute prime numbers. These programs are summarized here, and more details are available in Appendix B.

6.3.1 Breakout Game

The game is a version of the game Breakout, where there are rows of bricks at the top of the screen and a user-controlled paddle at the bottom (Figure 6-7). A ball bounces around the screen and eliminates bricks when it hits them. The object of the game is to remove all of the bricks without allowing the ball to fall below the paddle. A two-level version of this game was implemented with 12 rules containing 178 lines of program code, and 62 cards. 53 of the cards represent bricks. Each additional level added to the game would require about 25 more cards and 15 more lines of code.

6.3.2 Simulation of the Ideal Gas Law

The second program is a simulation of the ideal gas law, which specifies the relationships among pressure, volume, and temperature according to the formula: $PV=nRT$ (Figure 6-8). This program displays a chamber with small molecules bouncing around inside. The user can manipulate the pressure, volume, or temperature of this chamber, and observe its effect on the other variables as well as seeing changes in the speeds of the molecules. This simulation was implemented with 18 rules containing 180 lines of code, and 36 cards. 12 of the

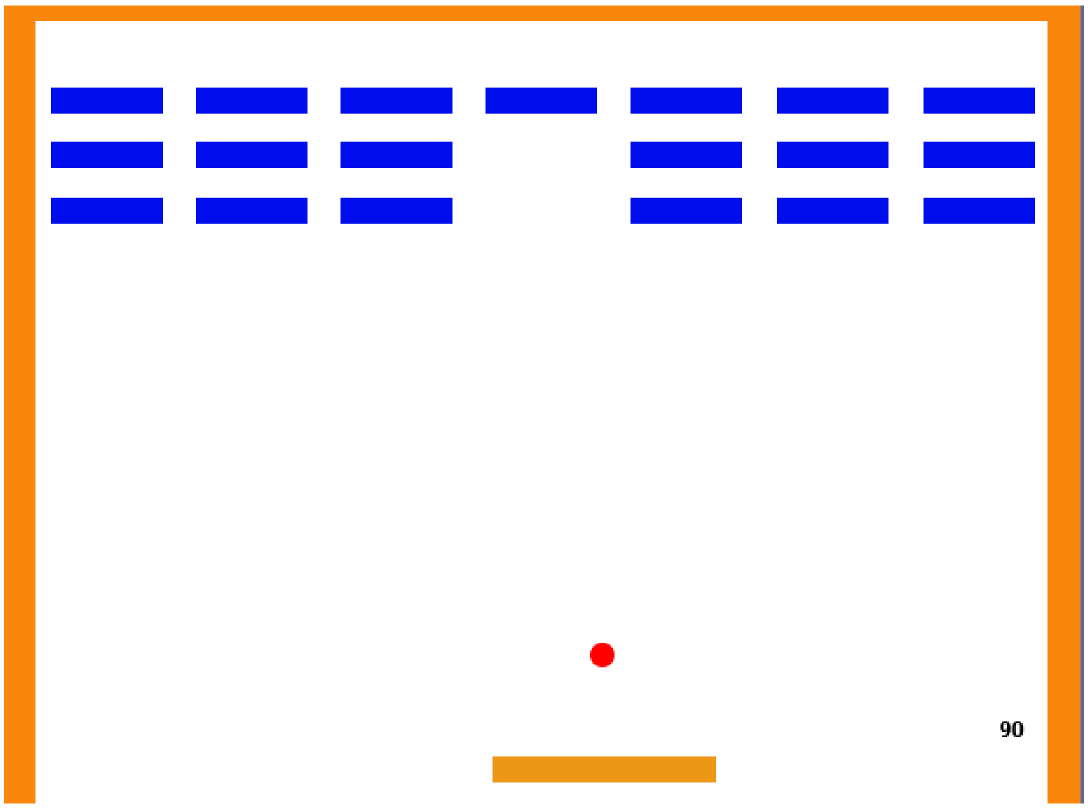


Figure 6-7. A version of the game Breakout, implemented in HANDS.

rules and 12 of the cards implemented checkboxes and scrollbars for controlling the simulation, which would probably be supplied in a toolkit in other systems.

6.3.3 Towers of Hanoi

The Towers of Hanoi solution shown in Figure 6-9 was implemented using 6 rules containing 53 lines of code, and 10 cards. One of these cards is off-screen, where the goals and subgoals are stored as the problem is solved. This program is a general solution that works for any number of rings. Adding an additional ring to the pictured three-ring solution can be done by importing a program file that has two cards and no code (see Section 5.19 on page 145). One of the cards represents the fourth ring, and the other card replaces the goal card with an updated goal card indicating the presence of the fourth ring. I was pleased how easy this was to implement in HANDS, especially since I had never solved this problem without using recursion, and HANDS does not support recursion. For comparison, a recur-

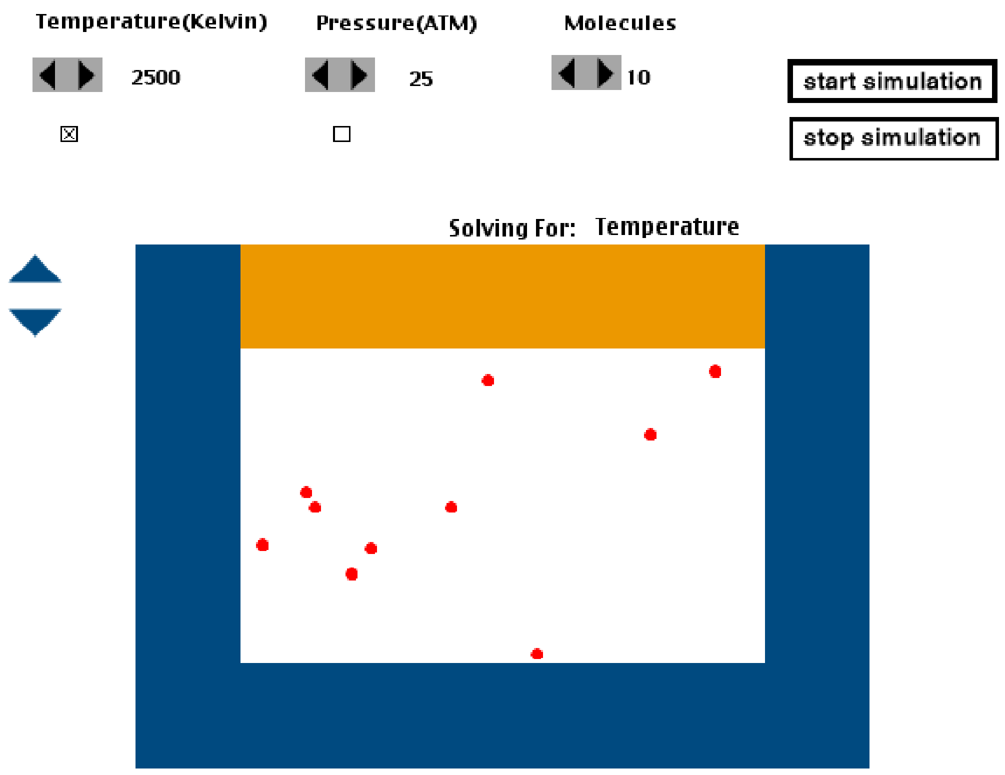


Figure 6-8. Simulation of the ideal gas law, as implemented in HANDS.

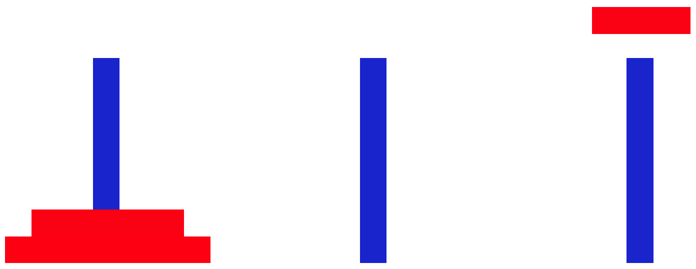
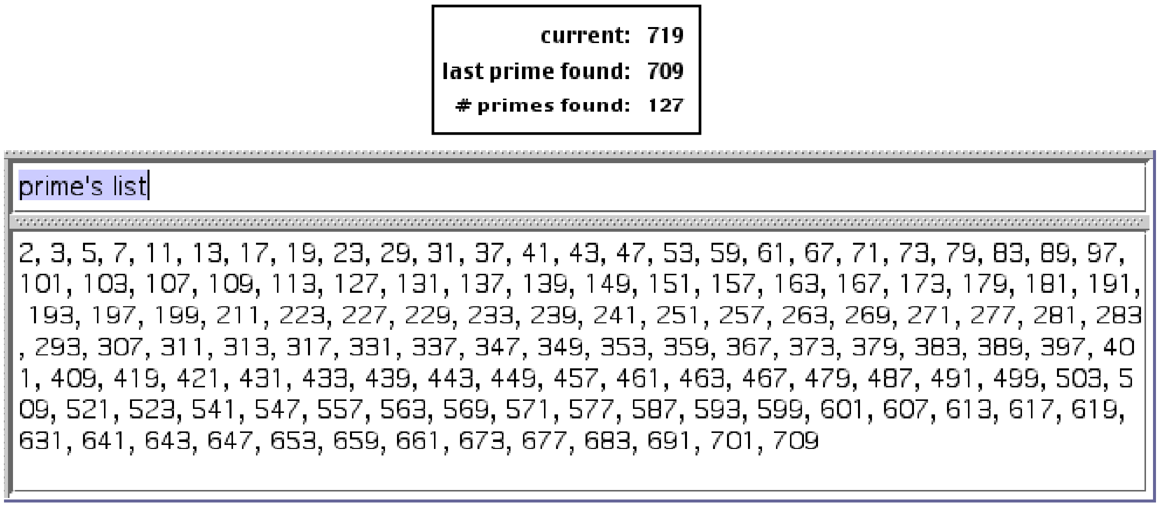


Figure 6-9. A solution to the Towers of Hanoi problem, as implemented in HANDS.

sive solution to this problem in Logo required 117 lines of code (see Figure 2-1 on page 20).

6.3.4 Computing Prime Numbers

The program shown at the top of Figure 6-10 uses a sieve technique to compute prime numbers. It has only one rule with 8 lines of code, and 6 cards. This program takes about 5 seconds to compute the first 100 primes on a 400 MHz Powerbook G3. I wrote this program to show that HANDS is useful for general computation, not just for games and animations.



```
current:  719
last prime found:  709
# primes found:  127
```

```
prime's list
```

```
2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97,
101, 103, 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173, 179, 181, 191,
 193, 197, 199, 211, 223, 227, 229, 233, 239, 241, 251, 257, 263, 269, 271, 277, 281, 283
, 293, 307, 311, 313, 317, 331, 337, 347, 349, 353, 359, 367, 373, 379, 383, 389, 397, 40
1, 409, 419, 421, 431, 433, 439, 443, 449, 457, 461, 463, 467, 479, 487, 491, 499, 503, 5
09, 521, 523, 541, 547, 557, 563, 569, 571, 577, 587, 593, 599, 601, 607, 613, 617, 619,
631, 641, 643, 647, 653, 659, 661, 673, 677, 683, 691, 701, 709
```

Figure 6-10. The top of this figure shows the screen of a program that computes prime numbers using a sieve technique, as implemented in HANDS. The bottom of this figure shows the primes that have been computed so far, which are stored in the `list` property of the card `prime`.

6.4 Comparison with Another System

A high school student with some programming experience compared HANDS with Stagecast [Smith 1994] in an informal evaluation for his science project. He implemented PacMan in both systems, and concluded that HANDS was easier to learn and use, required fewer lines of code, and enabled him to implement more features than Stagecast. Because Stagecast is not a textual language, he used this heuristic to count lines of code within each rule: each rule was counted as one line of code, and each `then` clause, each `and if` clause, and each `appearance` check was counted as one additional line of code. His implementation of PacMan in HANDS is shown in Figure 6-11, and his statistics are shown in Table 6-2.

Table 6-2. Statistics from the comparison of HANDS with Stagecast.

	HANDS	Stagecast
Minutes learning system with tutorial	40	60
Hours spent implementing Pacman[a]	15	9
Types of objects	9	6
Number of event handlers	15	64
Number of “lines” of code	183	253

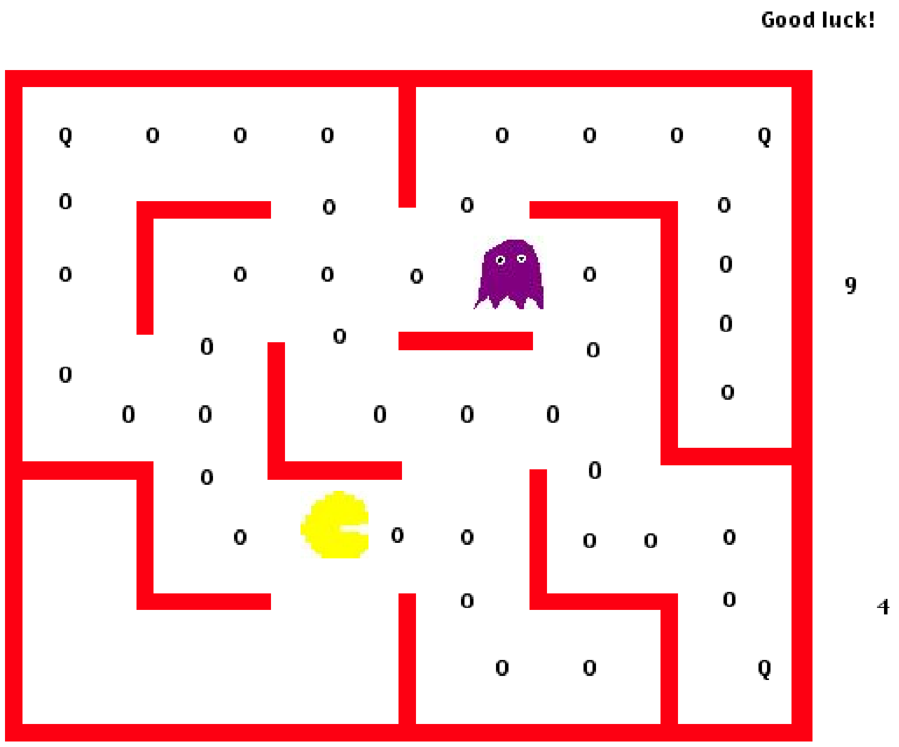


Figure 6-11. A high school student created this implementation of PacMan as part of his science project, where he compared HANDS to Stagecast.

a. The student implemented Pacman first in HANDS, and while he was doing this he learned strategies and algorithms for implementing various features that he then reused when he implemented Pacman in Stagecast. Therefore, the time difference can be discounted by this learning effect.

He commented that some of the advantages of HANDS were: “simple syntax, logical method of programming, information about the game can easily be displayed on screen, can move cards wherever you want, cards can easily be picked up or put down, and HANDS is more flexible.” His list of disadvantages of HANDS were: “not very good collision algorithm, doesn’t come with drawing editor, and it is tedious to copy and arrange cards.”

That last point arises because face-down cards on the board cannot be dragged without flipping them face-up first. But while the cards are face-up, the image in their back property is not shown. This makes it very inconvenient to precisely position these objects. The cards must flipped face-up, moved off the board, and flipped face-down. Now the face-down card can be dragged back onto the board, so that the picture on the back can be viewed while it is positioned. This problem can be eliminated by implementing the capability to drag cards that are face-down on the board.

6.5 Some Weaknesses of HANDS

The game and simulation programs are much larger than most of the other examples, and they exposed some weakness of the HANDS system:

- These programs would have been easier to implement if the collision detection mechanism was more advanced. For example, when objects collide, the collision event does not include any additional information beyond the names of the objects involved. The system could report the actual point of contact, to assist the programmer in determining how to react to the collision. Other issues with the collision detection mechanism are discussed in Section 5.14.4.1 on page 127.
- The mechanism for accurate timing mechanism would be very useful, both for controlling the speed of the game and for taking measurements such as collision rates in the simulation. Ideas for extending the HANDS metaphor with a timer mechanism are discussed in Chapter 7.
- For programmers at this skill level and programs of this size, the abstraction capabilities described Chapter 7 would be quite useful for organizing and modularizing the code.
- Execution performance is often sluggish, and often the animations are not very smooth, perhaps due to garbage collection or thread scheduling. These performance issues are discussed in Section 5.17 on page 141.

Nonetheless, it was reasonably easy to implement these programs, and they demonstrate the wide range of programs that can be built in HANDS.

6.6 Range of Capabilities

In HANDS, it is very easy to make interactive graphical programs such as games and simulations. The difficulties that arise with larger programs are discussed above and are addressed in Chapter 7. It was also quite easy to make small programs of the type that are used in computer science courses (computing prime numbers, Towers of Hanoi, etc.). In the case of the primes program, aggregates make the program simpler than it is in many other languages. The primes program ran with surprising speed, however for very intense numeric calculations, the performance of HANDS will likely have to be improved, for example by compiling instead of interpreting programs. HANDS, as it is today, is not really suitable for creating business programs such as word processors, spreadsheets, web browsers, or anything that requires a lot of user interface widgets. This class of programs is outside the original domain that HANDS targets, but this limitation can be addressed by adding domain specific features such as user-interface widgets (buttons, scrollbars, menus, etc.), text objects, better support for mouse manipulations (drag and drop, double-clicks, etc.), keyboard commands, and so forth. Assuming these kind of extensions can be made for any particular domain, there are no known classes of programs that would be impossible to build in HANDS.

6.7 Programming Strategies

There are several strategies a programmer can use to improve the aesthetics and efficiency of HANDS programs.

- Graphics files used in the `back` property should have tight bounding boxes, because collision detection and mouse-click detection use this bounding box.
- Graphics files used in the `back` property that have lots of white space should use the GIF transparency feature. This prevents the invisible parts of an object from blocking objects that are behind.
- Cards can be renamed to take advantage of the alphabetic layering feature. Cards with names later in the alphabet are drawn over cards with names earlier in the alphabet.
- If objects have lower speeds, they move more smoothly, and are less likely to penetrate or jump over objects they are colliding with. The optimal speed value for balancing smooth motion and collision accuracy with speed of motion is 1.
- Computation should not be done in the `anything happens` event handler if it can be avoided. This event handler is called when every event is dispatched, and in many cases this would cause the code to be executed more frequently than necessary.

- If the exact same calculations are being repeated in several event handlers, try putting the calculations in a single event handler that is triggered by a particular card changing. If none of the cards in the system fit this description, an off-board card can be used for this purpose. In the various places where recalculation must take place, the trigger card can be changed by setting any of its properties.
- Small invisible objects are very useful for making visible objects do something, such as change direction, when they reach a particular location. The invisible object can be positioned at the location, and the behavior can be programmed into the event handler for the collision between the two objects.

6.8 Evaluation of Earlier Design Ideas

The HANDS design evolved from earlier preliminary designs that included various features that I eventually decided to change, due to problems discovered in early pilot testing. One was to use allow multiple agents to be present around the table, working in cooperation or competition to complete tasks (Figure 6-12). In this design, the table is a shared data space, accessible to all agents, but cards in one agent’s hand would not be visible to the other agents, providing data privacy. It also enables the programmer to modularize the code by grouping related event handlers into separate agents. In addition, imported code could be kept separate from existing code by automatically installing it into a new agent, instead of merging it into Handy’s thought bubble. The multi-agent extension is implemented in the underlying HANDS system. In this implementation, the agents always execute in a fixed order, although it may be desirable to give the programmer better control over this. The multi-agent feature was thought to be too confusing for beginners so it was not exposed in the user-interface.

In even earlier designs, I proposed a *card game metaphor*, where the agents were players in a card game. In addition to data cards, the system would also have *rule cards* holding program code, which could be on the table or in a player’s hand (see Figure 6-13). A player would execute code on the rule cards in his hand. An agent could disable rules by putting them face-down on the table in a pile in front of the agent, and re-enable the rules by picking them back up. When an event occurs, execution order might be determined by the ordering of players around the table, which could vary over time. All of the players might

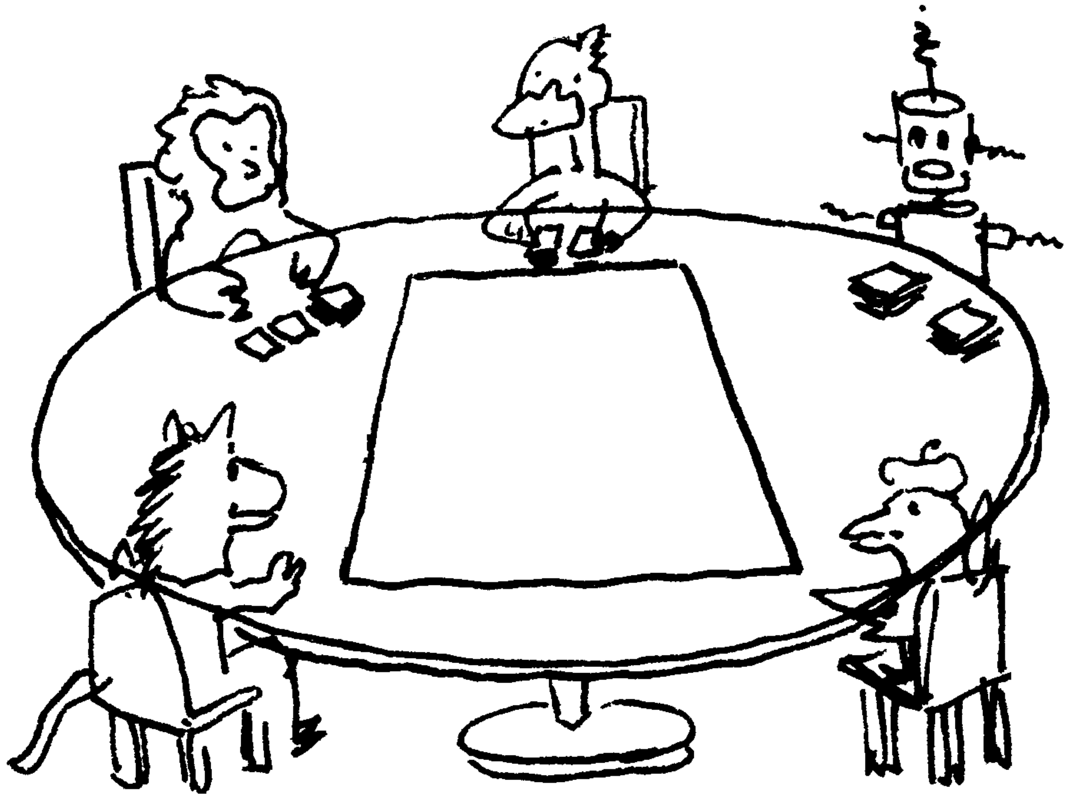
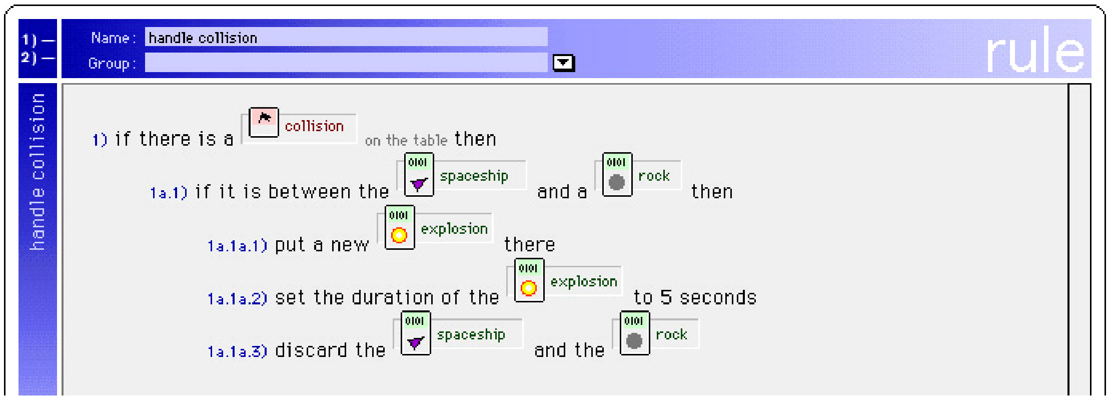


Figure 6-12. The underlying HANDS system can support multiple agents working in cooperation or competition, but this feature was removed from the user interface because it was too complicated for beginners. Sketch by Joonhwan Lee.

have an opportunity to respond to an event unless one of the players removes it from the table, rendering it invisible to the other players. Within a particular player, execution order could be determined by the ordering of the rules in the player's hand. Shuffling could be used to put non-deterministic orderings on the execution of rule cards. Rule cards might even be given to other players, as a mechanism for assigning work to other processes or threads.

The purpose of this metaphor was to enable beginners to figure out how the system works by relating it to a card game. However, as I developed this metaphor many problems arose, including inconsistencies with real card games and complexities that do not exist in real card games. For example, there were many different roles that might be confused by beginners: the programmer, the end user, the players, the characters in the program, etc. In real card games the players usually abide by a uniform set of rules, not by their own personal sets of rules. There were also issues about what would happen when rule cards were handed from one agent to another or placed face-up on the table, and whether it would be too difficult to find all of the code if it was scattered throughout the system. Resolving these issues would have resulted in increased complexity that outweighed the benefits of the card game metaphor, so it was discarded in favor of the much simpler model now used in HANDS, where all of the code is in the agent's head.



Name: handle collision
Group:

```
1) if there is a collision on the table then
     1a.1) if it is between the spaceship and a rock then
          1a.1a.1) put a new explosion there
          1a.1a.2) set the duration of the explosion to 5 seconds
          1a.1a.3) discard the spaceship and the rock
```

Figure 6-13. In early designs of HANDS, I experimented with the idea of storing code on *rule cards*. Sketch by John Chang.

6.9 Some Criticisms of HANDS

Some computer science researchers and educators are troubled because the HANDS system is so different than other programming systems. They are concerned that children learning HANDS are not learning essential computer science concepts and techniques they will need in the future. There are two parts to my reply to this criticism.

First, my objective in designing this system was to *enable* a broad range of children to express their ideas and to explore the grand possibilities of computer programming. This is something that most children are eager to try, and it is very important that their early experiences are fulfilling ones. Unfortunately, the programming systems used by most professional programmers and computer science students are very difficult to learn and use, and are inappropriate for the "casual jottings of ordinary people [diSessa 1986, p. 859]." When children try to use these tools, the vast majority will become frustrated, and may be permanently turned off programming. If, on the other hand, the early experiences are positive ones, and children can accomplish their objectives, there is a good chance their eyes will be opened to the possibilities and some of them may be more likely to pursue more formal computer science knowledge. Children using HANDS will learn some of the difficult skills that are fundamentally important in programming, rather than struggling with difficulties that are caused by the limitations of current languages. For example, they will learn how to precisely specify tasks, which is a universal requirement in computer programming. I also expect that, just like computer scientists, children will find it easier to learn subsequent programming languages after they have already successfully learned one.

Second, this research has produced some useful ideas that may impact how computer science is taught and what topics are considered essential. Perhaps by the time some of the children using HANDS start to learn computer science there will not be so many differences between HANDS and the systems they learn. For example, other researchers are also questioning whether the computational metaphors established by Turing and von Neumann are the correct ways for modern computer scientists to think about computation [Stein 1999]. In addition, perhaps more mainstream computer languages will begin to support queries and aggregate operations, which seem universally useful if they can be implemented efficiently. As computer science evolves, the set of concepts that are considered to be essential will surely change. Having children struggle to learn C++ or Java today may not prepare them for future careers in computer science any more than learning assembly language or Fortran helped students of previous decades.

Another criticism of HANDS is that there is no evidence that its model of computation will scale to large problems or other domains such as creating programs for office tasks. This is a valid point. While I have listed some promising ideas in Chapter 7 for expanding the range, the ideas remain untested. No showstopper problems have been encountered so far, and it will be worthwhile to explore this issue in future work.

Finally, many people point out that the natural-language-like syntax of HANDS is problematic. Indeed, I observed children making errors and incorrect assumptions about the language because it is not obvious what the limitations are (see Section 6.2.5 on page 155). The authors of MOOSE Crossing studied this issue, and concluded that this kind of error is easy for beginners to recover from [Bruckman 1999]. It is important to remember that beginners will have great difficulty with the syntax of any programming language. Programming editors can guide people on the syntax and limits of the language, and certainly the HANDS environment could be enhanced in this way. I subscribe fully to the principle to *speak the user’s language* and for the young non-programmers who will use HANDS, natural language is the only fully developed language they know. They are learning mathematical notations, but many programming language notations aren’t even consistent with mathematics (e.g. a = a + 1). As people learn to use more concise formal notations, I believe it is good to support these notations in programming languages, but it is not necessary for beginners to learn a new formal notation in order to write their first programs. Furthermore, reading code is an essential component of programming and debugging, and so improving the readability of programs is bound to improve usability.

6.10 Summary of Evaluation

The formal evaluation of HANDS showed that three key features had a significant impact on the ability of fifth-grade non-programmers to learn to program and accomplish tasks in a 3-hour session. Less formal evaluation by an older student with some programming experience concluded that HANDS is easier to learn and use, requires fewer lines of code, and enables implementation of more features than a commercial programming environment for children. In addition, expert programmers have implemented a wide variety of programs in HANDS, demonstrating its breadth. The criticisms and weaknesses that were identified do not point to flaws in the design and architecture of HANDS, but rather mainly derive from the limited time available for implementing HANDS to date. They can all be addressed by making the code run faster, improving the user interface, and improving and extending the domain specific support for collisions, timing, and animation.

CHAPTER 7 *Future Work*

This chapter discusses some of the ways this thesis work could be extended. They are categorized into three sections: further evaluation work, direct extensions to HANDS, and applications of the research results to other areas.

7.1 Further Evaluation and User Testing

The formal evaluation of HANDS discussed in Chapter 6 does not provide conclusive evidence about how the HANDS programming system stands against the alternative programming systems children might use for learning to program and building their first programs. Such a study could use an approach similar to the one described in Chapter 6, but instead of using a version of HANDS for the comparison system, one or more of the existing programming systems for children, such as Stagecast, Logo, or Boxer, would be selected. Participants would work with one system or the other, first learning it and then solving programming tasks. Much care would have to be taken in selecting tasks and preparing the materials, to eliminate bias and minimize confounding factors.

The study in Chapter 6 measures the collective impact of queries, aggregate operations, and the high visibility of program data. It would be useful to determine the individual contributions of these features, and whether there are dependencies that make a particular feature more useful in the presence of another feature. Similarly, it would be useful to look at other important features of the HANDS system, to assess their contributions to usability. For example, the computational model portrayed by HANDS, with cards on a table and the agent Handy manipulating them, could be tested by comparing it with a system that uses a more traditional model of computation. This could be done by again producing a modified version of HANDS that presents a more traditional model of computation, while keeping other factors such as the language fixed. The verbose style of the language could be tested by creating a version of HANDS with a more terse language that has identical structure and semantics. It would be harder to modify HANDS for testing the event-based paradigm, because so much of the system is dependent on its event-based structure.

Finally, HANDS surely could be improved by observational user testing, which would uncover common problems that people have with the system. These observations could then be used to improve the HANDS design.

7.2 Ideas for Extending HANDS

There are many things that can be done to improve and extend the HANDS system. Some of these are engineering improvements, such as improving the collision detection algorithm, while others are research ideas. Many extensions were discussed in prior chapters, but a few more are presented here.

7.2.1 Modularity and Encapsulation

One problem with the event-oriented approach in HANDS is that it is impossible to factor code into a function or subroutine that can be called during the course of an event handler. I have worked on a design for extending the HANDS model of computation with a special kind of card that represents procedural abstraction. Parameters could be passed to the abstraction by setting properties on this card. A special property, perhaps named `result`, would represent the result of executing the abstraction. When the `result` property is read by an event handler, the processing of the event handler would be suspended while the subroutine is run. The subroutine would perform its calculations, and store the result into the `result` property. When the subroutine finished, control would return to the suspended event handler, and it would receive the result as a consequence of having read the `result` property.

The subroutine would be programmed as an event handler responding to a new kind of event: `<identifier> is run`.

For example, consider how a dice-rolling subroutine might be created. A card `dice` might have a `sides` property, indicating the number of faces of the dice. A caller could set this property to 6, for example, then attempt to read `the dice’s result`. This would suspend the caller and call the event handler for `when dice is run`. This event handler could animate the dice by inserting a series of pictures into the `back` property, and use the random function to produce a random number between 1 and the value in `dice’s sides`, storing the result into `dice’s result`. When the dice event handler finishes, control would return to the caller, and the value read from `the dice’s result` would be the same value that was set by the subroutine.

Unfortunately, this mechanism would not support recursion unless the subroutine has a private copy of the dice card’s properties that cannot be corrupted by other invocations of the same subroutine. Further work is required to address this problem in a way that is easy to understand and within the card model. One idea is to have the subroutine run on a completely separate table with a copy of the one card that represents the subroutine and its parameters. Additional private cards could be present on this other table, but none of the other cards from the original table (global data) would be accessible to the subroutine. This would open the possibility of having multiple event handlers involved in the subroutine’s computation, and raises the possibility of multithreading, with its attendant issues of interprocess communication, synchronization, deadlock, and starvation. Work would also have to be done to provide a simple concrete representation for these subprograms that allows the programmer to collect together the needed cards and event handlers. This would be an interesting and fertile area to explore.

7.2.2 Multiple Agents

My original idea for the HANDS model of computation had multiple agents sitting at the table (see Section 6.8 on page 163). In fact, this capability is built-in to the underlying

HANDS system but is not exposed in the user interface because it was believed to be too confusing for children. However, having multiple agents would enable many interesting ideas to be explored in HANDS.

The cards in the hands of the agents would represent private data which is not accessible to other agents. These agents could be used to modularize the code, so that related event handlers and data could be kept separate from unrelated ones. When new capabilities are imported into a program (see Section 5.19 on page 145), they could be automatically stored into a new agent to help keep the program organized. Agents could be used to represent thread or processes running on local or remote processors, and cards could be used for private communication between these processes. The table itself would be the global data store, like in blackboard architectures in AI [Carver 1994] and tuple spaces in Linda [Carriero 1989]. Agents could be turned on and off to enable and disable program features or to implement modes.

One issue to examine is the current policy that all cards must have a unique name throughout the whole system. Private cards in the hand of one agent would be invisible to other agents, so it may be possible to relax this constraint so that all cards must unique names only from each agent’s point of view. That is, two agents could hold private cards with the same name. An agent could act as a namespace, but the system would have to have a consistent way to handle cards that are put down onto the table (which makes them globally visible) or passed to other agents.

7.2.3 Graphics Primitives

HANDS only supports import of graphical pictures, and should be extended so that the programmer can draw graphics (such as lines, circles, etc.) onto the screen. One idea, to stay within the card model, is to allow the programmer to create a blank card of a certain size and position, and then draw the graphics primitives into the back slot of this card.

7.2.4 Improvements to Collision Detection and Animation

Engineering improvements to the HANDS runtime engine could solve many of the problems described in Section 5.17.1 on page 141. For example, the collision detection and animation mechanism could work more closely together to use the actual object boundaries instead of bounding boxes; handle multi-object collisions in a more sophisticated manner than the current multiple pairwise collisions; to enable programmers to handle collisions without penetration; and to determine the exact points of collisions. A further improvement would be to extend the animation and collision engines to handle 3-dimensional graphics.

7.2.5 Timers

Timers are an important missing feature in the current HANDS system. I have some preliminary ideas of how to use cards to provide timing mechanisms. Like the speed and direction properties which are monitored by the animation mechanism, other properties of cards could be monitored by a timing mechanism. For example, the timer might recognize a card with a `countdown` property, decrement the value of this property every second (or millisecond), and generate a `<identifier> expired` event when the `countdown` property reached zero. Similarly, an `expires` property might cause a card to be automatically deleted when the value reaches zero.

7.2.6 Match Forms

Match forms (Chapter 4) are designed to be incorporated into the HANDS system. Although match forms can express arbitrarily complex queries in disjunctive normal form, this is sometimes less concise than unrestricted Boolean expressions would allow. This could be relieved somewhat by allowing an entire form to be negated (“objects that do not match ...”). The match forms in HANDS would also have property names alongside the values, like cards do, so the programmer can easily restrict the match of a value to a specific property.

7.2.7 Widget Library

A large fraction of the code in the simulation sample program written in HANDS (Figure 6-8 on page 158) was dedicated to managing widgets. A future version of HANDS could provide a library of widgets, such as scroll bars, buttons, text panes, etc. The interfaces to these widgets would be through card properties, but their behaviors could be automatic, reducing the amount of code that a programmer would have to write to make these widgets work.

7.2.8 Dealing with Large Numbers of Cards

The visibility of cards is an advantage for smaller programs, but can become a problem in programs with a very large number of cards. The HANDS model needs a way to collect groups of related cards, such as piles, decks, or paperclips. In HANDS, there is already a new card pile and discard pile (accessible from code but not visible in the user interface). Other piles could be used to provide special features, like the shuffle feature in GAMUT [McDaniel 1999], which is used to randomly select one card to be visible at the top of the pile. Piles might be displayed in various ways, such as all stacked up where only the top card is showing, or spread out like in solitaire card games. Card piles might be used to represent all of the cards in a subroutine (Section 7.2.1 on page 170) or all of an agent's cards (Section 7.2.2 on page 171). Piles of cards might be used to group graphical objects, so that they move around together on the screen and can be manipulated as if they were a single object. Open issues are whether these collections should have names, and whether the existing features of HANDS, such as lists, will allow these groups to be manipulated without adding a new collection data type.

7.2.9 Editing and Debugging Support

Much work can be done to support editing and debugging programs. Some examples are:

- the system could support a drag-and-drop syntax-directed editor, as seen in Squeak's eToys interface [Steinmetz 2001] and other systems.
- the system could include a spelling checker that watches for possible spelling errors, of language syntax as well as identifiers. For example, if there is a `nectar` property on a card, and the programmer writes code to set the `necter` property, the system might ask the user if he/she really intends to create a new property so similar to the existing `nectar` property. This relates to the old *do what I mean* (DWIM) mechanism of Interlisp [Teitelman 1981], but modern spell checking technologies and user interfaces can be used to make it more successful than DWIM was.
- a file dialog could come up so that the programmer doesn't have to enter the image filename into the back property. If the entry in the back property looks like a file name (i.e., it ends in "`.gif`" or "`.jpg`") but the file is not found or doesn't contain an image, the

system might explain the problem to the user instead of simply displaying the filename as a string on the back of the card.

7.2.10 HANDS as a Complete Package for Teachers and Students

In addition to adding many of the features listed above, and improving the general robustness of the system, there would be more work involved in turning HANDS into a "complete" package that could be used by many students and teachers. The HANDS system would have to be supplemented with complete tutorials and study materials, and a curriculum for use in schools. A big library of pictures and sample code should be provided, including libraries for creating a full suite of programs, so that HANDS will be a fertile environment for long-term study. In order to extend the kinds of problems students could work on beyond games and simulations, additional libraries and domain-specific features could be added for creating business programs and for hooking into the user-interface components that are available in the Swing toolkit or the OS-native windowing systems such as Windows or Macintosh.

7.3 Applications of Results to Other Areas

There are several ways that this research could impact other areas of computer science.

7.3.1 Model of Computation

The HANDS model of computation is a new way of describing and thinking about programs. It would be very interesting to push this computational model, to see if it is a generally useful way to represent large programs and whether it offers any benefits over existing models of computation used by experienced programmers. Several extensions to the model are proposed above in Section 7.2 on page 170, and surely more will become necessary as the model is tested.

7.3.2 Export Features to Other Languages

The query and aggregate operations in HANDS, and the way they work in combination, are powerful and useful to beginners. I believe they would also be very useful for other types of end-user programming tasks, such as multimedia and web authoring, as well as for experienced programmers. It would be interesting to explore how these features, especially queries, might be efficiently implemented in professional programming languages, and how their availability might affect programmers' productivity and bug rates. If these features are shown to be successful in a professional programming setting, it is more likely that the designers of future programming languages will incorporate them.

7.3.3 Influence Design Process for Future Languages and Domains

The process used in designing the HANDS system, and the knowledge gathered along the way is now available for the designers of future programming languages. As mentioned in Chapter 1, the design of the system is dependent on the target group of people who will be using the system, including their cultural background and their place in history. The process can and should be applied over and over to building systems for various audiences over time. Hopefully, this will eventually cause all new programming languages to become more usable.

7.3.4 Applications of Match Forms

In Chapter 4, Match Forms were shown to be better than textual query languages in a study of non-programmers. Already, this research has influenced the interface for the search engine at the HCI Bibliography (www.hcibib.org), and early analysis shows it to have improved usability (Section 4.9 on page 84). Match Forms could be deployed in many places on the web and in other online databases, to improve the usability of search engines.

CHAPTER 8 Conclusion

This thesis is a case study of a new, human-centered approach to the design of programming languages. It tracks the design and evaluation of a new programming system for children, describing how HCI techniques and evidence from the literature, as well as new studies investigating unaddressed questions, impacted the design and selection of features for the system.

8.1 Contributions

This section summarizes the contributions of this thesis.

8.1.1 Design Process

This thesis describes and exemplifies a new design process for programming systems, where first a target audience and domain are identified, and then the target audience is studied to examine the ways they solve problems and the problems they encounter when trying to program. This information is used to design the new system, and the system is then evaluated for usability. Any problems uncovered are fed back into redesign.

Specifically, this thesis targets ten year old children creating interactive games and simulations. In addition to studying the literature about children and beginner programmers, I performed three new studies to gain a better understanding of this group. I then designed a new system, refined the design based on early testing, and then evaluated the system in a user study.

8.1.2 HANDS

This thesis resulted in a new programming system for children, called HANDS, with a unique set of features due to its user-centered design. In addition to the HANDS programming system as a standalone artifact, HANDS embodies features that may be broadly useful in other languages.

HANDS incorporates a new model of computation, or way of thinking about programs, that is concrete and based on familiar concepts, unlike the traditional Turing machine or von Neumann machine models. In HANDS, all data is stored on cards, which are visible on a table. An agent named Handy manipulates the cards in response to events.

HANDS also incorporates a general-purpose programming language that offers database-style access to the program's data, and in which all operators can be uniformly applied to singletons and lists. Three of the features in HANDS, queries, aggregate operations, and the high visibility of data, were demonstrated to be more usable than the features found in typical programming systems.

8.1.3 Tabular Method for Expressing Boolean Queries

This thesis describes Match Forms, a new tabular method I invented for expressing Boolean queries. Match Forms were compared to textual expressions and shown to improve beginners' performance.

The match form research was applied to creating a new search interface for the HCI Bibliography (www.hcibib.org), and preliminary analysis comparing it with the old search interface shows that it has reduced users' error rates.

8.1.4 User Studies

Several empirical user studies were conducted as part of this thesis. Two studies examined how non-programmers express problem solutions, and provided empirical data that be used to help designers generate and select programming system features that provide a close mapping between those problem solutions and their expression in program code. An additional study provided empirical evidence characterizing the kinds of errors made by inexperienced users of textual Boolean expressions.

A user study of HANDS demonstrated the effectiveness of queries, aggregate operations, and high-visibility of data, in comparison to the typical features sets of programming systems. This study also provides evidence that children who had never programmed before were able to learn to program in a three hour session with HANDS.

8.1.5 Survey of Prior Work

The thesis also includes a broad survey of the prior work on beginner programmers, organized in a form that can be used by other programming system designers. This survey appears in Appendix C.

8.2 Closing Remarks

This thesis set out to demonstrate that programming systems can be made significantly more usable by applying a human-centered design process. This goal has been achieved. The thesis statement set forth in the Chapter 1 has been validated: “this user-centered design process, incorporating principles from human-computer interaction, psychology of programming, and empirical studies, will result in a unique programming system that is easier to learn and use than more conventional programming systems.” The combined contributions of the HANDS system, the facts discovered about non-programmers, and the design process described in this thesis promise to have a broad impact on improving the usability of computer programming in the future.

CHAPTER 9

References

Anderson, J.R. and Jeffries, R. (1985). "Novice LISP Errors: Undetected Losses of Information from Working Memory." Human-Computer Interaction 1: 107-131.

Anick, P.G., Brennan, J.D., Flynn, R.A., Hanssen, D.R., Alvey, B. and Robbins, J.M. (1990). A Direct Manipulation Interface for Boolean Information Retrieval via Natural Language Query. Proceedings of the Thirteenth Annual International ACM SIGIR Conference on Research and Development in Information Retrieval. Brussels, Belgium: 135-150.

Baraff, D. (1989). "Analytical Methods for Dynamic Simulation of Non-Penetrating Rigid Bodies." Computer Graphics 23(3): 223-232.

Biermann, A.W., Ballard, B.W. and Sigmon, A.H. (1983). "An Experimental Study of Natural Language Programming." International Journal of Man-Machine Studies 18(1): 71-87.

Blackwell, A.F. (1996). Metacognitive Theories of Visual Programming: What Do We Think We Are Doing? Proceedings of the VL'96 IEEE Symposium on Visual Languages. Boulder, CO, IEEE Computer Society Press: 240-246.

Blackwell, A.F. and Green, T.R.G. (2000). A Cognitive Dimensions Questionnaire Optimised for Users. Proceedings of the 12th Annual Meeting of the Psychology of Programmers Interest Group. A. F. Blackwell and E. Bilotta. Corigliano Calabro, Italy, Edizioni Memoria: 137-154.

Bonar, J. and Soloway, E. (1989). Preprogramming Knowledge: A Major Source of Misconceptions in Novice Programmers. Studying the Novice Programmer. E. Soloway and J. C. Spohrer. Hillsdale, NJ, Lawrence Erlbaum Associates: 325-353.

Bonar, J.G. and Cunningham, R. (1988). Bridge: Tutoring the Programming Process. Intelligent Tutoring Systems: Lessons Learned. J. Psotka, L. D. Massey and S. A. Mutter. Hillsdale, NJ, Lawrence Erlbaum Associates: 409-434.

Bourne, L.E. (1966). Human Conceptual Behavior. Boston, Allyn & Bacon.

Bruckman, A. and Edwards, E. (1999). Should We Leverage Natural-Language Knowledge? An Analysis of User Errors in a Natural-Language-Style Programming Language. Proceedings of the 1999 Conference on Human Factors in Computing Systems. Pittsburgh, PA, ACM Press: 207-214.

Brusilovsky, P., Calabrese, E., Hvorecky, J., Kouchnirenko, A. and Miller, P. (1997). “Mini-languages: A Way to Learn Programming Principles.” Education and Information Technologies 2(1): 65-83.

Carriero, N. and Gelernter, D. (1989). “Linda in Context.” Communications of the ACM 32(4): 444-458.

Carver, N. and Lesser, V. (1994). “The Evolution of Blackboard Control Architectures.” Expert Systems with Applications 7(1): 1-30.

Conway, D.M. (1998). An Algorithmic Approach to English Pluralization. Proceedings of the Second Annual Perl Conference. C. Salzenberg. San Jose, CA, O'Reilly.

Conway, M., Audia, S., Burnette, T., Cosgrove, D., Christiansen, K., Deline, R., Durbin, J., Gossweiler, R., Koga, S., Long, C., Mallory, B., Miale, S., Monkaitis, K., Patten, J., Pierce, J., Shochet, J., Staack, D., Stearns, B., Stoakley, R., Sturgill, C., Viega, J., White, J., Williams, G. and Pausch, R. (2000). Alice: Lessons Learned from Building a 3D System for Novices. Proceedings of CHI2000 Conference on Human Factors in Computing Systems. T. Turner and G. Szwillis. The Hague, Netherlands, ACM Press: 486-493.

Conway, M.J. (1997). Alice: Easy-to-Learn 3D Scripting for Novices. Ph.D. Thesis. University of Virginia. School of Engineering and Applied Science, 242 pages.

Cordy, J.R. (1992). Hints on the Design of User Interface Language Features – Lessons from the Design of Turing. Languages for Developing User Interfaces. B. A. Myers. Boston, Jones and Bartlett: 329-340.

Cypher, A. and Smith, D.C. (1995). KidSim: End User Programming of Simulations. Proceedings of CHI'95 Conference on Human Factors in Computing Systems. Denver, ACM: 27-34.

Davies, S.P. (1993). Externalising Information During Coding Activities: Effects of Expertise, Environment and Task. Empirical Studies of Programmers: Fifth Workshop. C. R. Cook, J. C. Scholtz and J. C. Spohrer. Palo Alto, CA, Ablex Publishing Corporation: 42-61.

Détienne, F. (1990). Difficulties in Designing with an Object-Oriented Programming Language: An Empirical Study. Proceedings of INTERACT '90 Conference on Computer-Human Factors. Cambridge, England: 971-976.

Détienne, F. (2001). Software Design: Cognitive Aspects. London, Springer.

DiGiano, C., Kahn, K., Cypher, A. and Smith, D.C. (2001). "Integrating Learning Supports into the Design of Visual Programming Systems." Journal of Visual Languages & Computing 12(5): 501-524.

DiGiano, C.J. (1996). Self-Disclosing Design Tools: An Incremental Approach Toward End-User Programming. Boulder, CO, University of Colorado**:** Department of Computer Science Technical Report CU-CS-822-96, 154 pages.

diSessa, A.A. and Abelson, H. (1986). "Boxer: A Reconstructible Computational Medium." Communications of the ACM 29(9): 859-868.

du Boulay, B. (1989a). Some Difficulties of Learning to Program. Studying the Novice Programmer. E. Soloway and J. C. Spohrer. Hillsdale, NJ, Lawrence Erlbaum Associates: 283-299.

du Boulay, B., O'Shea, T. and Monk, J. (1989b). The Black Box Inside the Glass Box: Presenting Computing Concepts to Novices. Studying the Novice Programmer. E. Soloway and J. C. Spohrer. Hillsdale, NJ, Lawrence Erlbaum Associates: 431-446.

Earhart, C., Ed. (1999). Stagecast Creator Teacher's Guide. Palo Alto, CA, Stagecast Software, http://www.stagecast.com.

Essens, P.J.M.D., McCann, C.A. and Hartevelt, M.A. (1992). An Experimental Study of the Interpretation of Logical Operators in Database Querying. Cognitive Ergonomics: Contributions from Experimental Psychology. G. C. v. d. Veer, S. Bagnara and G. A. M. Kempen. Amsterdam, North-Holland, Elsevier Science Publishers: 201-225.

Finzer, W.F. and Gould, L. (1993). Rehearsal World: Programming by Rehearsal. Watch What I Do: Programming by Demonstration. A. Cypher, MIT Press.

Galotti, K.M. and Ganong, W.F., III (1985). "What Non-Programmers Know About Programming: Natural Language Procedure Specification." International Journal of Man-Machine Studies 22: 1-10.

Glass, R.L. (1995). "OO Claims – Naturalness, Seamlessness Seem Doubtful." Software Practitioner 5(2).

Goodman, D. (1987). The Complete HyperCard Handbook. New York, Bantam Books.

Gould, L. and Finzer, W. (1984). "Programming by Rehearsal." BYTE Magazine 9(6).

Green, T.R.G. (1990). The Nature of Programming. Psychology of Programming. J.-M. Hoc, T. R. G. Green, R. Samurçay and D. J. Gilmore. London, Academic Press: 21-44.

Green, T.R.G. and Petre, M. (1992). When Visual Programs are Harder to Read than Textual Programs. Human-Computer Interaction: Tasks and Organisation, Proceedings of ECCE-6 (6th European Conference on Cognitive Ergonomics). G. C. van der Veer, M. J. Tauber, S. Bagnarola and M. Antavolits. Rome, CUD.

Green, T.R.G. and Petre, M. (1996). "Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework." Journal of Visual Languages and Computing 7(2): 131-174.

Greene, S.L., Devlin, S.J., Cannata, P.E. and Gomez, L.M. (1990). "No IFs, ANDs, or ORs: A Study of Database Querying." International Journal of Man-Machine Studies 32(3): 303-326.

Grice, H.P. (1975). Logic and Conversation. Syntax and Semantics III: Speech Acts. P. Cole and J. Morgan. New York, Academic Press.

Gross, P. (1999). Director 7 and Lingo Authorized, Peachpit Press.

Hays, J.G. and Burnett, M.M. (1995). A Guided Tour of Forms/3, Oregon State University**:** Dept. of Computer Science Technical Report 95-60-6.

Hildreth, C. (1988). Intelligent Interfaces and Retrieval methods for Subject Search in Bibliographic Retrieval Systems. Research, Education, Analysis & Design. Springfield, IL.

Hix, D. and Hartson, H.R. (1993). Developing User Interfaces: Ensuring Usability Through Product and Process. New York, New York, John Wiley & Sons, Inc.

Hoc, J.-M. (1989). Do We Really Have Conditional Statements in Our Brains? Studying the Novice Programmer. E. Soloway and J. C. Spohrer. Hillsdale, NJ, Lawrence Erlbaum Associates: 179-90.

Hoc, J.-M., Green, T.R.G., Samurçay, R. and Gilmore, D.J., Eds. (1990a). Psychology of Programming. Computers and People Series. London, Academic Press.

Hoc, J.-M. and Nguyen-Xuan, A. (1990b). Language Semantics, Mental Models and Analogy. Psychology of Programming. J.-M. Hoc, T. R. G. Green, R. Samurçay and D. J. Gilmore. London, Academic Press: 139-156.

Hutchins, E.L., Hollan, J.D. and Norman, D.A. (1986). Direct Manipulation Interfaces. Hillsdale, NJ, Lawrence Erlbaum Associates.

Ingalls, D.H.H. (1981). Design Principles Behind Smalltalk. BYTE Magazine, August 1981.

Joers, J. (1999). Stagecast Creator Creator's Guide. Palo Alto, CA, Stagecast Software, http://www.stagecast.com/.

Jones, S. (1998). Graphical Query Specification and Dynamic Result Previews for a Digital Library. Proceedings of the ACM Symposium on User Interface Software and Technology: 143-151.

Kahn, K. (1996). “ToonTalk: An Animated Programming Environment for Children.” Journal of Visual Languages and Computing 7(2): 197-217.

Kahn, K. (1999). From Prolog and Zelda to ToonTalk. Proceedings of the 1999 International Conference on Logic Programming. D. De Schreye, MIT Press.

Kohl, A. and Rupietta, W. (1987). The Natural Language Metaphor: An Approach to Avoid Misleading Expectations. Proceedings of IFIP INTERACT'87: Human-Computer Interaction: 555-560.

Lewis, C. and Olson, G.M. (1987). Can Principles of Cognition Lower the Barriers to Programming? Empirical Studies of Programmers: Second Workshop. G. M. Olson, S. Sheppard and E. Soloway. Norwood, NJ, Ablex: 248-263.

Martin, F.G. and Resnick, M. (1993). LEGO/Logo and Electronic Bricks: Creating a Scienceland for Children. Advanced Educational Technologies for Mathematics and Science. D. L. Ferguson. Berlin, Springer-Verlag.

Mayer, R.E. (1989). The Psychology of How Novices Learn Computer Programming. Studying the Novice Programmer. E. Soloway and J. C. Spohrer. Hillsdale, NJ, Lawrence Erlbaum Associates: 129-159.

McDaniel, R. (1999). Building Whole Applications Using Only Programming-by-Demonstration. Ph.D. Thesis. Carnegie Mellon University. Computer Science Department. Pittsburgh, PA, 271 pages.

McIver, L.K. (2001). Syntactic and Semantic Issues in Introductory Programming Education. Ph.D. Thesis. Monash University. School of Computer Science and Software Engineering. Australia, 200 pages.

McQuire, A. and Eastman, C.M. (1995). Ambiguity of Negation in Natural Language Queries. Proceedings of the Eighteenth Annual International ACM SIGIR Conference on Research and Development in Information Retrieval: 373.

Michard, A. (1982). “Graphical Presentation of Boolean Expressions in a Database Query Language: Design Notes and an Ergonomic Evaluation.” Behaviour and Information Technology 1(3): 279-288.

Miller, L.A. (1974). “Programming by Non-Programmers.” International Journal of Man-Machine Studies 6(2): 237-260.

Miller, L.A. (1981). “Natural Language Programming: Styles, Strategies, and Contrasts.” IBM Systems Journal 20(2): 184-215.

Miller, P., Pane, J., Meter, G. and Vorthmann, S. (1994). “Evolution of Novice Programming Environments: The Structure Editors of Carnegie Mellon University.” Interactive Learning Environments 4(2): 140-158.

Modugno, F. (1995). Extending End-User Programming in a Visual Shell with Programming by Demonstration and Graphical Language Techniques. Ph.D. Thesis. Carnegie Mellon University. Computer Science Department. Pittsburgh, PA, 334 pages.

Modugno, F., Corbett, A.T. and Myers, B.A. (1996). Evaluating Program Representation in a Visual Shell. Empirical Studies of Programmers: Sixth Workshop. W. D. Gray and D. A. Boehm-Davis. Norwood, NJ, Ablex Publishing Corporation: 131-146.

Mulholland, P. and Watt, S.N.K. (2000). “Learning by Building: A Visual Modelling Language for Psychology Students.” Journal of Visual Languages and Computing 11(5): 481-504.

Myers, B.A. (1992). “Demonstrational Interfaces: A Step Beyond Direct Manipulation.” IEEE Computer 25(8): 61-73.

Nardi, B.A. (1993). A Small Matter of Programming: Perspectives on End User Computing. Cambridge, MA, The MIT Press.

Newell, A. and Card, S.K. (1985). “The Prospects for Psychological Science in Human-Computer Interaction.” Human-Computer Interaction 1(3): 209-242.

Nielsen, J. (1994). Heuristic Evaluation. Usability Inspection Methods. J. Nielsen and R. L. Mack. New York, John Wiley & Sons: 25-62.

Pane, J.F. and Myers, B.A. (1996). Usability Issues in the Design of Novice Programming Systems. Pittsburgh, PA, Carnegie Mellon University**:** School of Computer Science Technical Report CMU-CS-96-132, 85 pages.

Pane, J.F. and Myers, B.A. (2000). Tabular and Textual Methods for Selecting Objects from a Group. Proceedings of VL 2000: IEEE International Symposium on Visual Languages. Seattle, WA, IEEE Computer Society: 157-164.

Pane, J.F., Ratanamahatana, C.A. and Myers, B.A. (2001). “Studying the Language and Structure in Non-Programmers’ Solutions to Programming Problems.” International Journal of Human-Computer Studies 54(2): 237-264.

Papert, S. (1980). Mindstorms: Children, Computers, and Powerful Ideas. New York, Basic Books.

Pattis, R.E., Roberts, J. and Stehlik, M. (1995). Karel the Robot: A Gentle Introduction to the Art of Programming. New York, John Wiley & Sons.

Pea, R. (1986). "Language-Independent Conceptual "Bugs" in Novice Programming." Journal of Educational Computing Research 2(1).

Pictorius (1996). Prograph CPX User Guide. Halifax, Nova Scotia, Pictorius Incorporated, http://www.pictorius.com/prograph.html.

Repenning, A. (2000). AgentSheets®: an Interactive Simulation Environment with End-User Programmable Agents. Interaction 2000, Tokyo, Japan.

Repenning, A. and Sumner, T. (1995). "Agentsheets: A Medium for Creating Domain-Oriented Visual Languages." Computer 28: 17-25.

Resnick, M. (1994). Turtles, Termites, and Traffic Jams: Explorations in Massively Parallel Microworlds. Boston, The MIT Press.

Sammet, J.E. (1981). The Early History of COBOL. History of Programming Languages. R. Wexelblat. New York, Academic Press.

Sherwood, B.A. (1988). The cT Language. Champaigne, IL, Stipes Publishing Company.

Shneiderman, B. (1983). "Direct Manipulation: A Step Beyond Programming Languages." IEEE Computer 16(8): 57-69.

Smith, D.C. (2000). "Building Personal Tools by Programming." Communications of the ACM 43(8): 92-95.

Smith, D.C., Cypher, A. and Spohrer, J. (1994). "KidSim: Programming Agents Without a Programming Language." Communications of the ACM 37(7): 54-67.

Soloway, E., Bonar, J. and Ehrlich, K. (1989a). Cognitive Strategies and Looping Constructs: An Empirical Study. Studying the Novice Programmer. E. Soloway and J. C. Spohrer. Hillsdale, NJ, Lawrence Erlbaum Associates: 191-207.

Soloway, E. and Spohrer, J.C., Eds. (1989b). Studying the Novice Programmer. Hillsdale, NJ, Lawrence Erlbaum Associates.

Spohrer, J.G. and Soloway, E. (1986). Analyzing the High Frequency Bugs in Novice Programs. Empirical Studies of Programmers. E. Soloway and S. Iyengar. Washington, DC, Ablex Publishing Corporation: 230-251.

Stein, L.A. (1999). “Challenging the Computational Metaphor: Implications for How We Think.” Cybernetics and Systems 30(6): 473-507.

Steinmetz, J. (2001). Computers and Squeak as Environments for Learning. Squeak: Open Personal Computing and Multimedia. M. Guzdial and K. Rose, Prentice Hall: 453-482.

Tanaka, J. (1999). The Perfect Search. Newsweek. 134: 71, September 27 1999.

Teitelman, W. and Masinter, L. (1981). “The Interlisp Programming Environment.” Computer 14(4): 25-34.

Thimbleby, H., Cockburn, A. and Jones, S. (1992). HyperCard: An Object-Oriented Disappointment. Building Interactive Systems: Architectures and Tools. P. Gray and R. Took. New York, Springer-Verlag: 35-55.

Thomas, J. and Gould, J. (1975). A Psychological Study of Query by Example. National Computer Conference. Anaheim, CA, AFIPS. 44: 439-445.

Turtle, H. (1994). Natural Language vs. Boolean Query Evaluation: A Comparison of Retrieval Performance. Proceedings of the Seventeenth Annual International ACM SIGIR Conference on Research and Development in Information Retrieval: 212-220.

Wason, P.C. (1959). “The Processing of Positive and Negative Information.” Quarterly Journal of Experimental Psychology 11.

Webgain (2001). JavaCC - The Java Parser Generator, http://www.webgain.com/products/metamata/java_doc.html.

Weinberg, G.M. (1971). The Psychology of Computer Programming. New York, Van Nostrand Reinhold Company.

Wilcox, E. and Burnett, M. Programming a Single Digit LED in Forms/3 http://www.cs.orst.edu/~burnett/Forms3/LED.html.

Young, D. and Shneiderman, B. (1993). "A Graphical Filter/Flow Representation of Boolean Queries: A Prototype Implementation and Evaluation." Journal of American Society for Information Science 44(6): 327-339.