Lowering the Barriers to Programming: a survey of programming environments and languages for novice programmers

Caitlin Kelleher and Randy Pausch

May 12, 2003
CMU-CS-03-137

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Since the early 1960's, researchers have built a number of programming languages and environments with the intention of making programming accessible to a larger number of people. This paper presents a taxonomy of languages and environments designed to make programming more accessible to novice programmers of all ages. The systems are organized by their primary goal, either to teach programming or to use programming to empower their users, and then by the authors' approach to making learning to program easier for novice programmers. The paper explains all categories in the taxonomy, provides a brief description of the systems in each category, and suggests some avenues for future work in novice programming environments and languages.

This work was partially supported by an NSF graduate fellowship and grants from DARPA, NSF, NASA, and ONR. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of DARPA, NSF, NASA, or ONR.

Keywords: Human-Computer Interaction, Design, programming environments, programming languages, novice, Computer Science Education, learning

1. INTRODUCTION

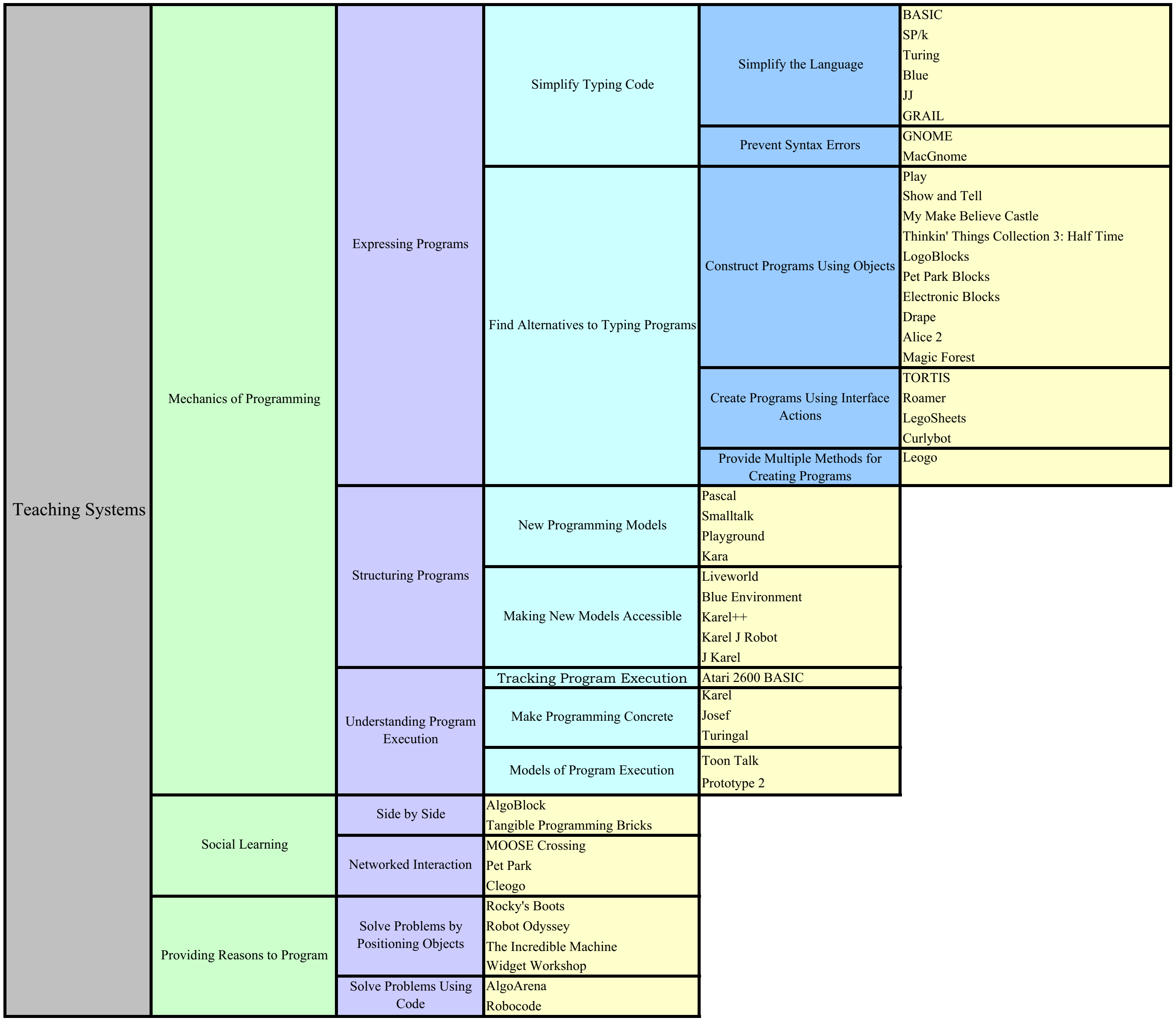
Learning to program can be very difficult for beginning students of all ages. In addition to the challenges of learning to form structured solutions to problems and understanding how programs are executed, beginning programmers also have to learn a rigid syntax and commands that may have seemingly arbitrary or perhaps confusing names. Tackling these challenges all simultaneously can be overwhelming and often discouraging for beginning programmers. Since the early 1960's, researchers have built a number of programming languages and environments with the intention of making programming accessible to a larger number of people. This paper presents a taxonomy of these languages and environments and discusses the challenges they address.

For the purposes of this paper, we define programming as the act of assembling a set of symbols representing computational actions. Using these symbols, users can express their intentions to the computer and, given a set of symbols, a user who understands the symbols can predict the behavior of the computer. This definition excludes many of the programming through demonstration systems in which the computer develops sequences of instructions and when to execute those instructions internally by observing the user in such a way that the user cannot accurately predict the actions of a program in all circumstances.

In this paper, we describe the high level organization of the taxonomy, present the taxonomy and briefly describe all of the categories and systems within those categories. We then present a table of the most influential systems and a table comparing the survey systems based on what programming constructs they support and their approaches to making programming more accessible to novice programmers. Finally, we summarize the approaches and discuss some possible avenues for future work in this area.

2. TAXONOMY

In creating a programming environment for novices, one of the first questions that must be answered is why novices need to program. There are a variety of possible motivations for learning to program: to pursue programming as a career path, to learn how to solve problems in a structured and logical way, to build software customized for personal use, to explore ideas in other subject areas, etc. The systems in this taxonomy (see Figure 1) fall into two large groups: systems that attempt to teach programming for its own sake and those that attempt to support the use of programming in pursuit of another goal.



- Teaching Systems
  - Mechanics of Programming
    - Expressing Programs
      - Simplify Typing Code
        - Simplify the Language
          - BASIC
          - SP/k
          - Turing
          - Blue
          - JJ
          - GRAIL
        - Prevent Syntax Errors
          - GNOME
          - MacGnome
      - Find Alternatives to Typing Programs
        - Construct Programs Using Objects
          - Play
          - Show and Tell
          - My Make Believe Castle
          - Thinkin' Things Collection 3: Half Time
          - LogoBlocks
          - Pet Park Blocks
          - Electronic Blocks
          - Drape
          - Alice 2
          - Magic Forest
        - Create Programs Using Interface Actions
          - TORTIS
          - Roamer
          - LegoSheets
          - Curlybot
        - Provide Multiple Methods for Creating Programs
          - Leogo
    - Structuring Programs
      - New Programming Models
        - Pascal
        - Smalltalk
        - Playground
        - Kara
      - Making New Models Accessible
        - Liveworld
        - Blue Environment
        - Karel++
        - Karel J Robot
        - J Karel
    - Understanding Program Execution
      - Tracking Program Execution
        - Atari 2600 BASIC
      - Make Programming Concrete
        - Karel
        - Josef
        - Turingal
      - Models of Program Execution
        - Toon Talk
        - Prototype 2
  - Social Learning
    - Side by Side
      - AlgoBlock
      - Tangible Programming Bricks
    - Networked Interaction
      - MOOSE Crossing
      - Pet Park
      - Cleogo
  - Providing Reasons to Program
    - Solve Problems by Positioning Objects
      - Rocky's Boots
      - Robot Odyssey
      - The Incredible Machine
      - Widget Workshop
    - Solve Problems Using Code
      - AlgoArena
      - Robocode

Figure 1: Novice Programming Systems and Languages Taxonomy

Empowering Systems	Mechanics of Programming	Code Is Too Difficult	Demonstrate Actions in the Interface	Pygmalion Programming by Rehearsal Mondrian
			Demonstrate Conditions and Actions	AgentSheets ChemTrains Stagecast
			Specify Actions	Pinball Construction Set Alternate Reality Kit Klik N Play
		Improve Programming Languages	Make the Language More Understandable	COBOL Logo Alice 98 HANDS
			Improve Interaction with Language	Body Electric Fabrik Tangible Programming with Trains Squeak eToys Alice 99 AutoHAN Physical Programming Flogo
			Integration with Environment	Boxer Hypercard cT Chart N Art
	Activities Enhanced by Programming	Entertainment	Bongo Mindrover	
		Education	SOLO Gravitas Starlogo Hank	

Figure 1: Novice Programming Systems and Languages Taxonomy

Because these two goals place very different constraints on systems, the taxonomy is organized first by the system goals, either teaching or using programming, and, second, by the primary aspect of programming that the system attempts to simplify. Each system appears in the taxonomy only once. However, many of the systems in the taxonomy have built on the ideas of earlier systems. So, a system that was influenced by natural language programming may not be classified with other natural language systems if the natural language influence was not the primary concern in building the system.

3. TEACHING SYSTEMS

These systems were designed with the goal of helping people learn to program. Most of the systems in this category are (or include) simple programming tools that provide novice programmers exposure to some of the fundamental aspects of the programming process. After gaining experience with a teaching system, students are expected to move to more general-purpose languages such as Java, C, or C++. A few systems attempt to provide support in learning a more general language from the start. Because students interacting with teaching systems are expected to transition to general purpose languages, many teaching systems have similarities to general-purpose languages. Knowing that a student will eventually have to do for loops in a Java-style, the designers of teaching languages are less likely to introduce a different style of looping. Because general-purpose languages are not always designed with beginners in mind, the systems in this category are juggling two possibly conflicting goals: making it easier for beginners to get started programming and giving students a background that makes it easy for them to transition from the teaching system to a general-purpose language.

The teaching systems focus on several areas that can be difficult for novice programmers. The majority of the systems in this category address the mechanics of programming: both expressing intentions to the computer and understanding the actions of the computer. Other systems attempt to place programming in a context that is accessible and motivating to a wider audience of people, either by providing concrete reasons for programming or by supporting novice programmers working together and learning from one another.

3.1 Mechanics of Programming

The systems in this category are designed around the hypothesis that the primary barrier in learning to program lies in the mechanics of writing programs. To successfully write a program, users must understand several topics: how to express instructions to the computer (e.g. syntax), how to organize these instructions (e.g. programming style), and how the computer executes these statements. Systems in this category attempt to make it easier for beginners to learn one of these three skills.

3.1.1 Expressing Programs

In most general-purpose languages, users create programs by typing syntactic sentences into a text editor. Beginning programmers often have trouble translating their intentions into syntactically correct statements that the computer can understand. The systems in this category explore two possible avenues for making this process easier for beginning programmers: improve the language such that beginners have an easier time learning or find alternate ways for beginners to communicate their instructions to the computer.

3.1.1.1 SIMPLIFY TYPING CODE

Many general-purpose languages have been influenced by the need for sufficient power to tackle arbitrary programming tasks and a desire to make the programming language easier to implement, making the resulting languages unnecessarily difficult for beginning programmers. The systems in this category examine three approaches to making languages more approachable for beginning programmers: simplifying the language, tailoring the language for a specific, small domain of programming problems, and preventing syntax errors.

1.1.1.1.1. Simplify the Language

General-purpose languages typically include a large variety of syntactic elements that can be particularly difficult for beginners because these syntactic elements don't have an obvious meaning. The languages in this category use a few simple observations to decrease the number of potentially confusing syntactic elements beginning users encounter while trying to maintain as much similarity as possible to general-purpose languages. General-purpose languages often contain unnecessary syntax, use commands whose names are unfamiliar or have different meanings in spoken English, have inconsistent uses for syntactic elements, or include features inappropriate for beginning programmers. Using these observations, it is possible to make a language syntactically easier for beginners to handle without fundamentally changing the common control structures found in general-purpose languages. Consequently, when a student moves from one of these languages to a general-purpose language, they should be able to transfer their knowledge from the teaching language.

BASIC: J.G. Kemeny and T. Kurtz, Dartmouth College, 1963 [Kurtz, 1981]
Basic was designed to teach Dartmouth's non-science students about computing through programming. FORTRAN and ALGOL, the commonly used languages at the time, were both large and complex. Kemeny and Kurtz believed that the students would "balk at the seemingly pointless detail" (Kurtz, 1981). After considering using subsets of FORTRAN or ALGOL, Kemeny and Kurtz agreed they would have to create their own language. The BASIC (Beginners All-purpose Symbolic Instruction Code) language was designed to support a small set of instructions and remove unnecessary syntax. The environment was designed to have rapid turn-around time and sacrifice computer time for user time (in 1963, the computer science community was arguing against high level languages because the compilation time was seemingly wasted computation).

Statements in BASIC consist of three parts: a line number (e.g. 110), an operator (e.g. LET), and an operand (e.g. S = S + 1). All commands begin with an English word to make the language easier for the novice; the designers believed that LET S= S + I would be easier for students to understand than S = S + I. Figure 2 (below) shows a simple summation loop in both FORTRAN and BASIC. While the statements have a similar structure, the BASIC program uses language more suitable for a novice, removes elements like labels (e.g. 30) that require a more detailed understanding of the program counter, and does not depend on spacing for syntactic meaning.

FORTRAN:	BASIC:
`      do 30 i = 1, 10`	`100 FOR I = 1 TO 10`
`         m = m + I`	`110 LET S = S + I`
`30   continue`	`120 NEXT I`

Figure 2. A *for* loop to compute the sum of the numbers from 1 to 10 written in FORTRAN and BASIC.

SP/k: R.C Holt et al, University of Toronto, 1977 [Holt, 1977]
SP/k is a subset of PL/1 chosen for teaching introductory programming. The features of the SP/k language were chosen to remove redundant constructs, inconsistencies in the language that go against students' intuitions (in PL/1 the expression 25 + 1/3 evaluates to 5.3333), constructs that are easily misused such as pointers, and constructs like concurrent programming that are suited for advanced programmers. The difficulty of compiling constructs was also considered. The result of pruning was a simpler language for introductory programming that both students and teachers generally preferred over FORTRAN. The authors also provided an order for introducing programming constructs

as a sequence of subsets of SP/k. SP/1 introduces expressions and output. By SP/8, students have learned all of SP/k. By introducing things gradually, students can master a small piece of the language at a time, allowing them to devote more time to problem solving than memorizing the features of the language.

Turing: R.C. Holt and J.R. Cordy, University of Toronto, 1988 [Holt and Cordy, 1988]
The Turing language was developed as a general-purpose and instructional language for the Computer Science Department at the University of Toronto. Consequently, while the designers intended that Turing be used in teaching programming, the goals for the language included many programming power and implementation concerns. The Turing language contains all the features of Pascal (see section 3.1.2.1) and adds dynamic arrays, modules, and varying length strings. In addition, Turing simplifies the syntax by removing the requirement for headers declaring the name of the program and semi-colons at the end of each statement.

Blue Language: M. Kolling and J. Rosenberg, University of Monash, 1996 [Kolling and Rosenburg, 1996]
Blue is an object-oriented language designed to be taught as a first language. After using Blue for a year, students are expected to move to an industrial language, such as C++. The designers of the language used four criteria in creating Blue: there should be only one way to do everything; the language should cleanly reflect the theoretical model; the language should be readable so students can learn by reading examples; and the language should explicitly support software engineering mechanisms like pre and post conditions. The Blue language is a pure object-oriented language that supports single inheritance, garbage collection, and strong static typing. Classes are defined in single files with a structure that clearly reflects which routines others can call and which routines are internal to the class by placing routines in separate *internal* and *interface* areas within the file. Routine definitions include explicit pre and post conditions. Blue provides a single loop structure that consists of a set of statements followed by a list of conditions that should cause the loop to exit which can be used to create loops that function like traditional for and while loops. Each loop exit condition can include statements to execute if the loop exits on that particular condition. The designers of the language also created an environment for beginning programmers that will be discussed separately.

JJ: J. Motil and D. Epstein, California State University and California Institute of Technology, 1998 [Motil and Epstein, 1998]

Full featured, general-purpose languages force beginning students to focus on the syntax rather than the problem they are trying to solve in writing a program. JJ (Junior Java) is a language designed to remove much of the syntactic complexity to allow students to focus on the concepts of programming. It removes much of the punctuation such as braces and semi-colons and has only one way to do anything; there is one integer type, one way to create a comment, etc. The language also provides an easy migration to Java after the first half of the semester. Students can either do this by hand or the environment can convert their JJ code to Java automatically. Figure 3 shows an example of computing weekly pay in JJ and the equivalent code in Java. Due to lack of adoption, the designers of JJ have moved towards concentrating on better compilation error messages and allowing students to program over the web.

Computing weekly pay in JJ:

```
If (hours <= 40) then
   Set pay = 10 * hours
Else
   Set pay =
      400 + 15*(hours - 40)
EndIf

Output "The pay is "
Outputln pay
```

The same code in Java:

```
if  (hours <= 40) {
   pay = 10 * hours;
} else {
   pay =
      400 + 15 * (hours - 40);
} // EndIf

System.out.print ("The pay is " );
System.out.println( pay );
```

Figure 3. A short segment of code to compute a worker's weekly pay shown in both JJ and Java. Note the line by line correspondence.

GRAIL: L. McIver, Monash University, 1999 [McIver, 1999, 2001]

GRAIL was developed in response to the hypothesis that "it is the unfamiliarity of 'hieroglyphics' (i.e. the language syntax) and the sheer complexity of the full theory that are the primary stumbling blocks for the novice" (McIver, 2001). Three guiding principles governed the design of GRAIL: maintain a consistent syntax; use terms that novice programmers are likely to be familiar with and avoid standard programming terms that have different meanings in English; and include only constructs that are fairly simple and have a "single, obvious syntax" (McIver, 2001). These guidelines led to an imperative language with many small differences from commonly used teaching languages such as Pascal (see section 3.1.2.1). The list of changes is too long to reproduce here, but we list a few to give the reader a feel for the kinds of changes made for the GRAIL language. Rather than using * for multiplication, GRAIL uses x because it is a symbol that novice programmers will understand from mathematics classes. Values are assigned using an arrow indicating where the answer will be placed since a = b is ambiguous. McIver removed pointers because they are difficult to use correctly; using pointers it is very easy for beginners to create problems they cannot easily understand or explain. The full details of the GRAIL language can be found in McIver's thesis.

1.1.1.1.2. Prevent Syntax Errors

One of the largest and most frustrating challenges for novice programmers is syntax. The Cornell Program Synthesizer [Teitelbaum and Reps, 1981], which was a prototype system that removed the ability for students to make syntax errors by presenting the set of allowed commands at each point in the program code, inspired the systems in this category. The prototype system was limited to 24 lines and included a limited subset of PL/1. The presentation of allowable commands made it impossible to compose a syntactically invalid program. The systems in this category were an attempt to make a more versatile structure editor that was useful to novice programmers. They are not languages, but environments that prevented novices from making syntax errors with existing languages such as Pascal and Fortran.

GNOME: P. Miller et al, Carnegie Mellon University, 1984 [Miller et al, 1994]
The GNOME environments were created for Karel the Robot, Pascal, Fortran, and Lisp and used an abstract syntax tree to detect syntax errors as they occurred. GNOME displayed programs hierarchically, encouraging students to think about programs as hierarchical collections of procedures. Students navigated through their programs using arrow keys that corresponded to movements in the abstract syntax tree; GNOME displayed program segments in the familiar textual form. When the programmer attempted to move the edit cursor, GNOME analyzed the program, reported any syntax errors, and prevented the programmer from moving on until the program was syntactically correct. The programmer could also request an analysis of the program at any time. While this environment prevented syntax errors, it actually required students to think more about syntax than they previously had: they needed to have a mental model of the syntax tree to navigate through the system; the abstract syntax representation sometimes differed from the textual representation (particularly with mathematical equations); and the requirement for syntactic correctness sometimes prevented students from making desired changes in the program because the fastest route to a correct program required intermediate stages that were not syntactically correct.

MacGnome: P. Miller et al, Carnegie Mellon University, 1986 [Miller et al, 1994]
The MacGnome project attempted to cleanly integrate structure-editing capabilities of GNOME with the text-editing model present in traditional programming editors. The GNOME project demonstrated that students have difficulty navigating in the abstract syntax tree; to alleviate this problem, MacGnome allowed students to navigate using point and click with a mouse. In GNOME, students often had trouble modifying code because of the requirement to maintain syntactic correctness. Rather than requiring syntactic correctness at all times, the MacGnome project editors converted the syntax tree into a textual representation to allow editing without syntactic constraints. Once the user finished editing, it converted the modified code back to tree representation using an incremental parser. By allowing students to edit code textually, the MacGnome environment could not prevent syntax errors. However, MacGnome detected and reported all syntax errors as soon as the code was parsed, allowing students to correct them before moving to other sections of the program. The novice programming environments produced as a result of the MacGnome project are called Genies.

3.1.1.2 FIND ALTERNATIVES TO TYPING PROGRAMS

Despite the attempts to make programming languages simpler and more understandable, many novices still struggle with syntax: remembering the names of commands, the order of parameters, whether or not they are supposed to use parentheses or braces, etc. Another large set of systems are designed around the belief that to enable novices to understand what programming really is, we need to bypass the syntax problems altogether. The systems in this category represent three major approaches to bypassing syntax: creating objects that represent code that can be moved around and combined in different ways, using actions of the user within the interface to define programs, and providing multiple methods for creating programs.

1.1.1.1.3. Construct Programs Using Objects

The systems in this group use graphical or physical objects to represent elements of a program such as commands, control structures, or variables. These objects can be moved around and combined in different ways to form programs. Novice programmers need only to recognize the names of commands and the syntax of the statements is encoded in the shapes of the objects, preventing them from creating syntactically incorrect statements.

Play: S. Tanimoto and M. Runyan, University of Washington, 1986 [Tanimoto and Runyan, 1986]
Play is a system designed to allow preliterate children to create graphical plays using an iconic language. Stories consist of a linear sequence of actions that is displayed at the top of the screen, above the story's stage, as a sequence of icons similar to a comic strip. The character, what the character should do, and one additional piece of information, typically a direction to move, all selected from menus, specify each action in the story. Play also provides a character editor where children can draw additional images of their characters and compose those images to create new animations. Play does not allow children to use more complicated control structures such as loops and conditionals or define procedures.

Show and Tell: T. Kimura et al, Washington University and Bell Labs, 1990 [Kimura et al, 1990]
Show and Tell is a data flow based visual language designed for children. A program in Show and Tell consists of a series of connected boxes. A box can represent a value or an operation on values. The program includes boxes that represent basic arithmetic functions, system input and output, and some special purpose boxes that play sounds or act as timers, etc. Children can build procedures by drawing their own icon for a box and defining what should happen in the procedure using other boxes. Procedures can call themselves. Because boxes are not permitted to form cycles or loops, users cannot construct for and while loops. However, Show and Tell provides an iteration box that provides bounded iteration, in other words, the function will continue repeating until a boundary value is reached. If two connecting boxes contain different values (e.g. 2 and 3), they and their parent box are marked "inconsistent" and become invisible to the other boxes. By checking for consistency and inconsistency in particular boxes, children can represent simple Boolean conditions.

My Make Believe Castle: Logo Computer Systems Incorporated, 1995 [LCSI, 1995]
My Make Believe Castle is a play program for children ages 4-7 that contains activities designed to help develop children's problem solving, critical thinking, sequential planning, and memory. The castle consists of a number of rooms, each containing an activity. In the courtyard of the castle, characters such as the dragon, prince, princess, and horse move around. When the user clicks on them with a particular tool, they will dance, slip on banana peels, do somersaults, etc. After children have played in the courtyard space, they can be introduced to a very simple, rule-based programming system. Editors for each character allow children to specify which action a character should take when it meets another specific character. A typical rule might be "Nicky dances when it meets

the horse" (see Figure 4). Rules are specified graphically; children select the action using icons and the character that should trigger the action by selecting a picture of that character.

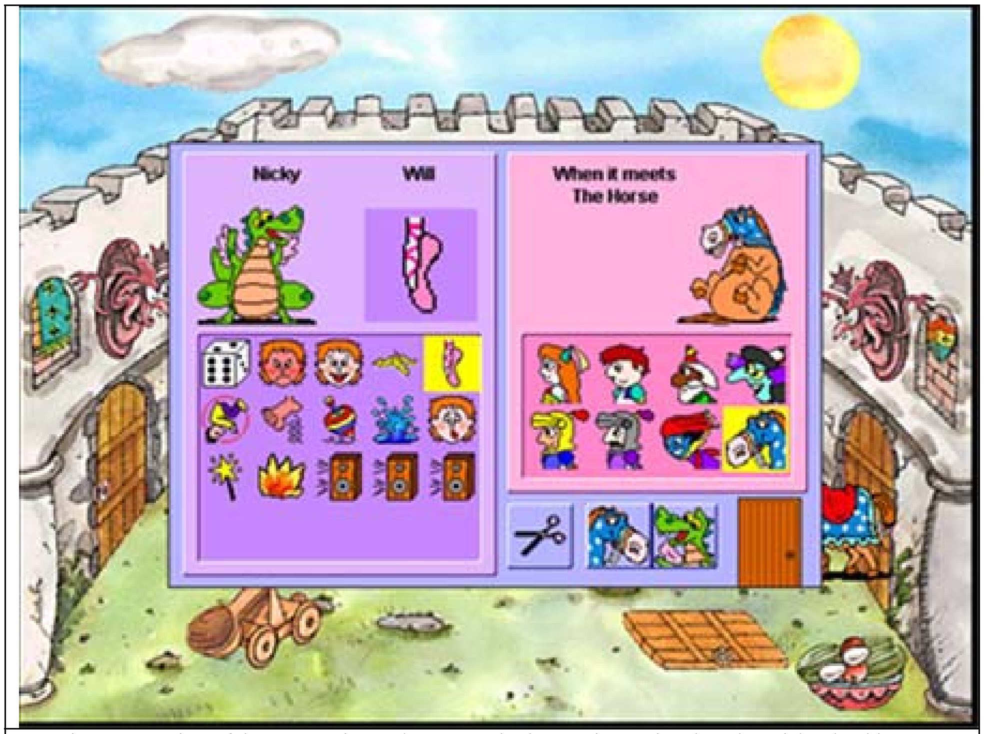


Figure 4. A view of the My Magic Castle courtyard. The user is creating the rule "Nicky should dance when it meets the horse."

Thinkin' Things Collection 3- Half Time: Edmark Corporation, 1995 [Edmark, 1995]

Half Time is one of the activities in the computer game Thinkin' Things Collection 3. The activity revolves around creating a half time show (see Figure 5). Users can select characters from the top left and drag them onto the field; each half time show can have a total of thirty characters across three types (such as tuba, percussion, and trumpet players). At the bottom of the screen, there is a line for each of the three types of characters in which users can drop instructions for them to perform. The available instructions are similar to those of the Logo (see section 4.1.2.1) turtle: move forward, turn left and right, turn randomly, pause, pen down and up, etc. Programs are created by dragging the icons for instructions (shown below the football field) into the lines for a particular type of character. Counted loops are supported, but no other block statements are available.

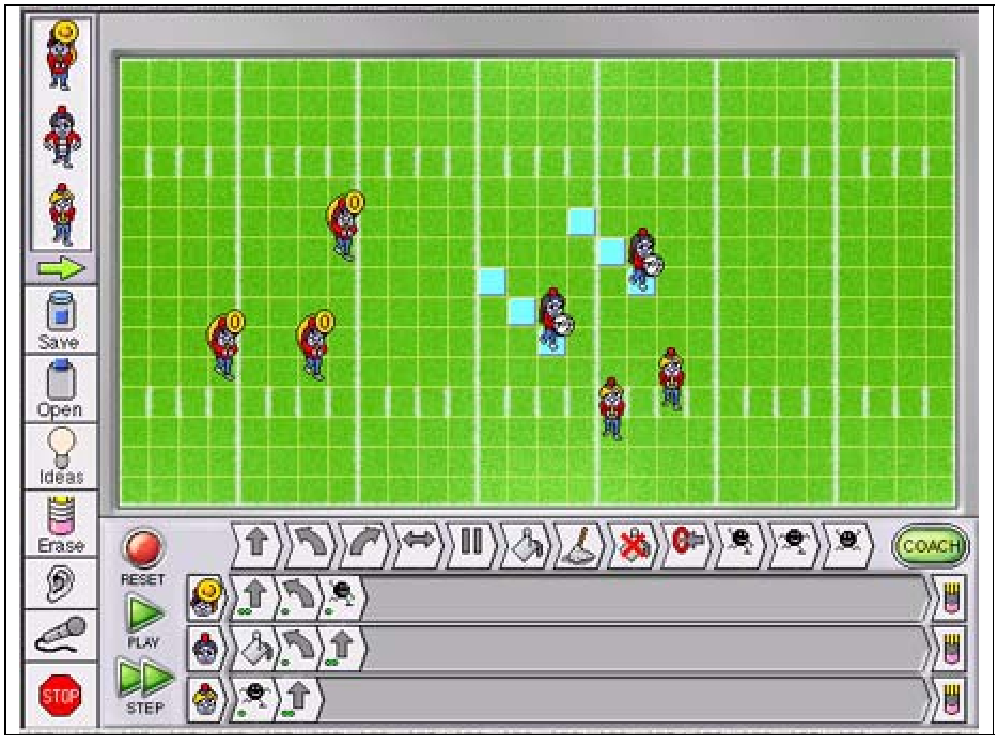


Figure 5. A screenshot of Half Time from Thinkin Things Collection 3

LogoBlocks: A Begel, MIT Media Lab, 1996 [Begel, 1996, MIT Media Lab]
LogoBlocks is a graphical programming language designed for the Programmable Brick, a precursor to the commercial Lego Mindstorms system [Lego], developed by the MIT Media Lab (see Figure 6). In LogoBlocks, labeled graphical shapes represent commands in BrickLogo, an extension of Logo (see section 4.1.2.1) that provides commands for the Programmable Brick. These graphical blocks can be dragged off a tool palette on the side of the screen to a main work area where they can be placed next to other blocks to form programs. Like many visual programming environments, changes to programs may require the user to move existing statements to make room for new ones. The parts in the palette can take several forms, for example a block marked 'A' specifies the motor A as the recipient of commands following it, but, by clicking on the 'A' block, the user can turn it into a 'B' or an 'AB' block. Commands and conditionals also have multiple forms; the blocks in the tool palette represent kinds of objects rather than all available objects. Commands and conditionals requiring arguments have shapes with cutouts for placing the arguments so that it is clear both that the command requires an argument, and the type of the argument which is specified by the shapes of blocks that will fit into the cutout. LogoBlocks includes support for procedures; users can attach commands to purple procedure blocks and name their procedures.

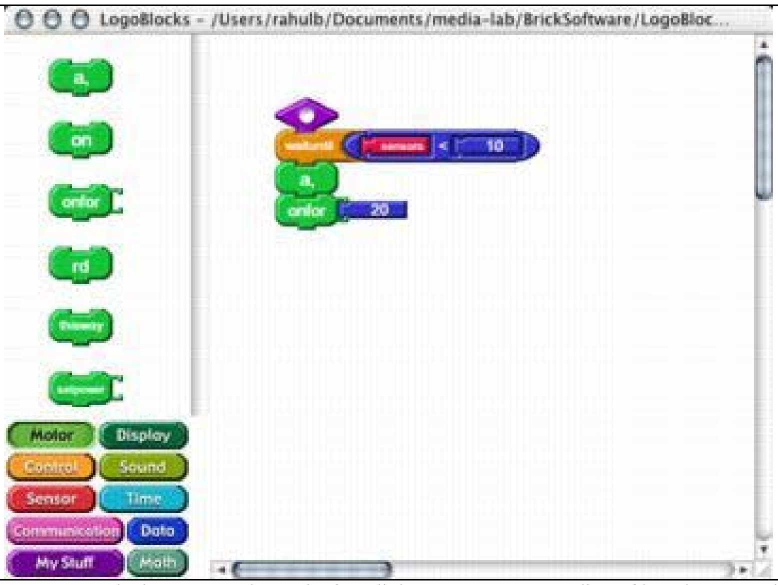


Figure 6. A LogoBlocks program that waits for a light sensor to get a reading of less than 10 and then turns motor A on for 20 seconds.

Pet Park Blocks: A. Cheng, MIT Media Lab, 1998 [Cheng, 1998]

Pet Park Blocks is a graphical programming language, inspired by LogoBlocks, which was developed for the Pet Park collaborative environment (described later). Animations are represented by notched squares that fit together. Conditionals are represented by squares with half oval cutouts where conditions can be added. Like LogoBlocks, programming constructs are kept in a palette from which users can drag them onto an active area. Pet Park Blocks provides a button that allows users to see their Blocks program as a textual program. This allows users to gradually transition to text-based programming.

Drape: M. Overmars, Universiteit Utrecht, 2000 [Overmars]

Drape is a programming environment that allows users to draw pictures (see Figure 8). There is a collection of pictorial icons on the left side of the interface that represent different commands similar to the Logo (see section 4.1.2.1) turtle commands: pen up, pen down, move in different directions, move in shapes, etc. The icons can be dragged to the lines at the bottom of the screen that represent the program; commands are executed from left to right. There are extra lines associated with their own icons that can serve as procedure calls. The system does have support for some predefined blocks such as repeat 10 times (shown as x10) However, to apply the repeat 10 to more than a single object, the sequence needs to be enclosed in brackets, which introduces the possibility for syntax errors in the form of mismatched braces.

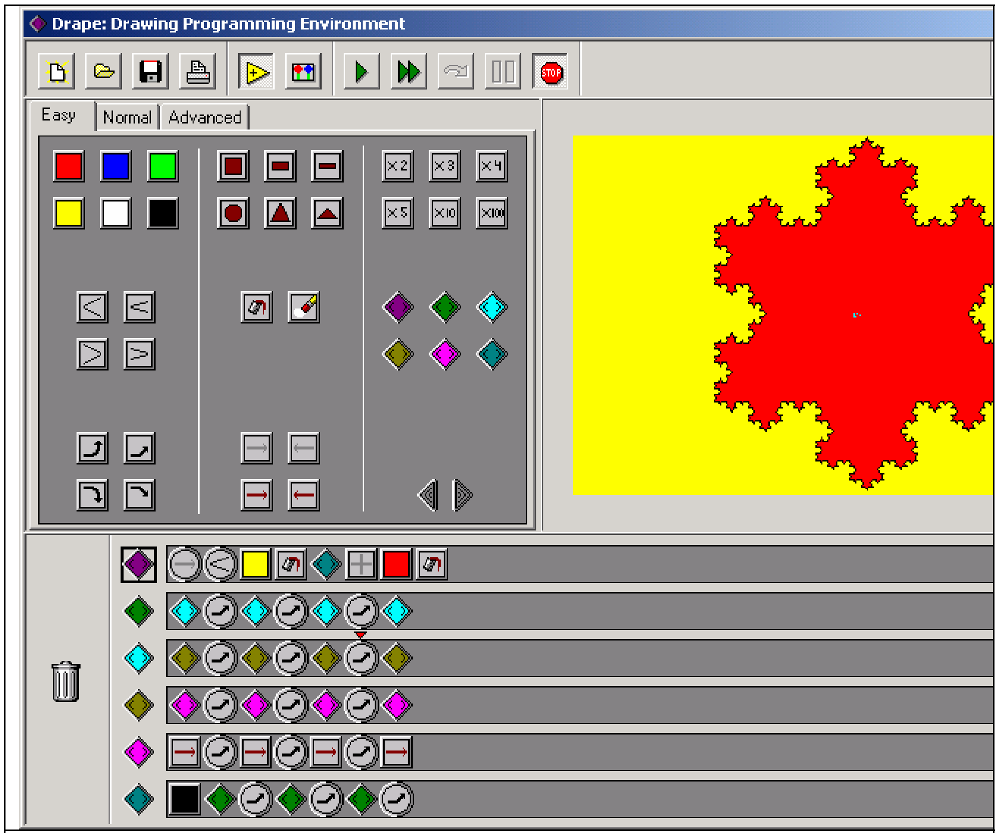


Figure 7. DRAPE Drawing and Programming Environment allows children to draw pictures.

Electronic Blocks: P. Wyeth and H. Purchase, University of Queensland, 2000 [Wyeth, 2000]

Unlike the graphical objects used to construct programs in other systems, Electronic Blocks are physical Lego blocks designed to allow young children (ages 3-8) to create Lego forms with interesting behaviors (see Figure 7). Preschool children can build block towers that flash when they talk or cars that move when a flashlight shines on them. Three types of blocks are provided: sensor blocks that can detect light, sound, and touch; logic blocks that can compute AND, NOT, TOGGLE, and DELAY; and action blocks that can produce light, sound, and motion. The syntax of Electronic Blocks is very simple; the only requirements are that each stack includes a sensor block and an action block and that the action block be at the bottom of that stack. Action blocks are smooth on the bottom so they cannot be placed on top of other block types.

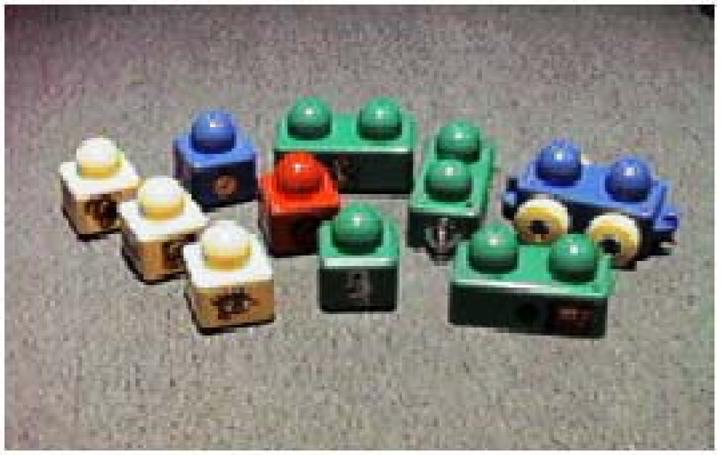
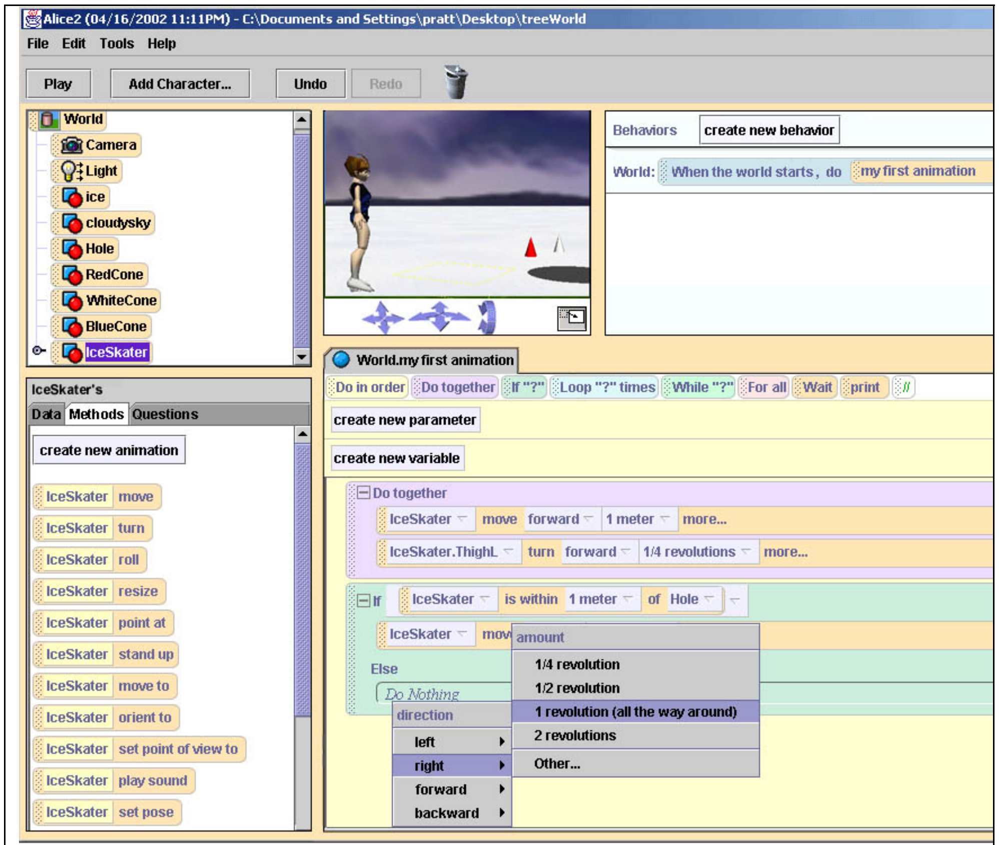


Figure 8. Electronic Blocks: the three sensing blocks are pictured on the left, the logic blocks in the middle, and the action blocks on the right.

Alice 2: Carnegie Mellon University, 2002 [www.alice.org]
Alice is a programming system for building 3D virtual worlds, typically short animated movies or games (see Figure 10). In Alice users construct programs by dragging and dropping graphical command tiles and selecting parameters from drop-down menus. Figure 1 shows an Alice screen as a user creates a simple animation. To add to the current animation, the user drags a graphical tile labeled with the name of the animation from the selected object's animations, in this case the IceSkater's animations, displayed in the lower left panel. When the user drops the tile, the system automatically cascades to menus that allow the user to select valid parameters for the chosen animation. In Figure x, the user has just dragged *IceSkater turn* from the panel and has chosen to have *IceSkater* turn right one full turn. Students can also add standard programming control structures such as if-statements and loops by dragging *if* and *loop* tiles from the top bar. Unlike many no-typing programming systems, Alice allows students to gain experience with all of the standard constructs taught in introductory programming classes in an environment that prevents them from making syntax errors



```
Do together
    IceSkater move forward 1 meter more...
    IceSkater.ThighL turn forward 1/4 revolutions more...
If IceSkater is within 1 meter of Hole
    IceSkater mov
Else
    Do Nothing
```

Figure 9. Building *my first animation* in Alice. In *my first animation*, *IceSkater* moves forward while she raises her leg. Then, if *IceSkater* is close to a hole in the ice, she falls through it.

Magic Forest: Logotron, 2002 [Logotron]

Magic Forest allows children ages four and up to play with, change, and create *Activities* that consist of 2D sprites that can move around, change appearance, and react to simple events. Each sprite can be given a set of *Rules* (represented by a scroll containing stones), a combination of an event and a list of things that should happen, in order, after that event occurs. Both events and actions are represented by graphical stones that can be identified by their icons, making it possible for children to learn how to use Magic Forest without needing to know how to read. Magic Forest supports a variety of events, such as mouse based events, events based on the relative positions of objects, and message passing events. Actions might change the direction or speed of an object, the appearance of an object, send a message, play sounds, or update the score. To add a new rule to a sprite, a child selects an event from a scrolling list of available event stones, clicks on it to pick it up, and then drops it onto a scroll associated with that sprite. The child can then attach action tiles to the end of the event. As in Logoblocks, some tiles can have multiple forms;

a single tile can be used to increase the speed, heading, or size of an object. Children can click on a tile to change which form it takes (increase speed, heading, or size).

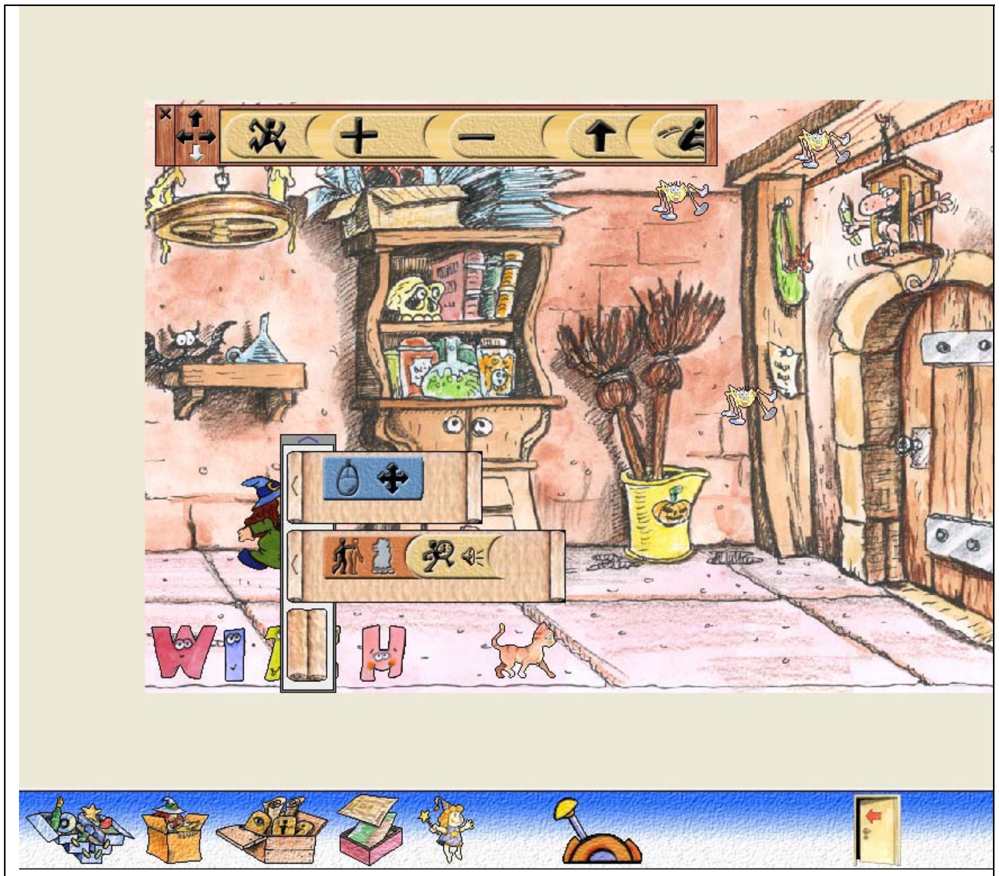


Figure 10. Magic Forest allows children to control the actions and appearances of 2D characters. This activity has five characters: a witch, a cat, and three spiders. The witch has two rules controlling her behavior. The top one (blue tile on a scroll) allows the user to move the witch around the scene. The second says that when the witch touches another object, she should make a sound (e.g. laugh). The witch also has an empty scroll to which the user can add new behaviors by selecting events and actions from the brown window at the top of the screen and placing them together on her scroll.

1.1.1.1.4. Create Programs Using Interface Actions

The systems in this category attempt to make programming more accessible using physical objects or the simulation of physical objects. User interface actions such as button presses or motion through space are mapped to commands in a programming language. Since most of these interfaces rely on the motion of or buttons on physical objects, the interfaces either tend to be fairly limited in the number and types of commands possible or require the user to perform interface actions (such as pressing buttons) in a specific sequence, introducing the possibility for sequences of actions that do not correspond to valid program instructions.

TORTIS: R. Perlman, MIT Artificial Intelligence Lab, 1976 [Perlman, 1976]
TORTIS provides two different physical interfaces for young children to control a robotic turtle inspired by the Logo turtle (see section 4.1.2.1); since the robotic turtle is very slow, a simulated version is also provided for more advanced students. The first interface is called the Button Box and provides a set of four boxes for controlling the turtle that can be given to a child gradually. The first box provides buttons that move and turn the turtle, pick up or put down the pen, turn a light on and off, and sound a horn. The second box adds numbers such that a child can repeat a command multiple times by pressing a number followed by a command. The third box adds a program area where children can get the turtle to "remember" commands and then play back remembered commands. The fourth and final box creates four procedures (named by colors) that can call each other. The button box system did not allow students to edit programs after creating them, making the gradual modification of programs difficult. The second interface, a Slot Machine with cards that represented commands, attempted to solve this problem. Children created programs by placing cards in the slot machine and having the turtle execute the cards in order. The Slot Machine supported the easy modification of programs since children could simply add cards, remove unwanted cards, or reorder cards if the program did not do what they wanted.

Roamer: D. Catlin, Valiant Technologies, 1989 [Catlin]
Roamer is a programmable, mobile robot that has capabilities similar to those of the Logo turtle: the Roamer can move forward and back, turn left and right, wait, and make sounds. Programs are entered using a set of buttons, icons for the commands and a number pad to indicate how far to move or turn and what sound to play. Buttons are also provided for creating procedures and repeating statements. The Roamer can remember up to 59 instructions in either the main program (the GO program) or numbered procedures that can be called from the GO program or each other. An expansion set allows users to add on sensors, two-state outputs, and a stepper motor, allowing a greater variety of programs.

LegoSheets: Gindling et al, University of Colorado, 1995 [Gindling, 1995]
LegoSheets attempts to provide a gentle introduction to programming for the MIT Programmable Brick by beginning with manual control of the elements of the brick and

gradually progressing to writing programs. Users are presented with a simulated version of the Programmable Brick in which the parts can be manipulated; users can change the speed of a motor connected to the simulated brick by typing in a value or using arrow buttons to increase or decrease the value. Once users are comfortable with manipulating the values of motors and observing the values of sensors in response to different types of actions, they can double click on the representation of a motor or sensor and bring up a rule editor for that object. The rule editor provides buttons to add conditionals or initial values to control the behavior of the brick. Conditionals are provided in a template form where users only have to type the names of objects they want to use and arithmetic operations. There are also buttons for increasing and decreasing the priority of the current rule.

Curlybot: P. Frei et al, MIT Media Lab, 2000 [Frei et al, 2000]
Curlybot is an educational toy for children aged four years and older. It consists of a two-wheeled vehicle with electronics that allow it to record its motions. The Curlybot has a single button and a single LED. The LED is used to indicate whether it is in record mode (red) or playback mode (green). When a child wants to record a motion, he or she pushes the button, demonstrates the motion, and then pushes the button again, which stops recording and starts replaying the motion. The motion is repeated until the button is pushed again, turning Curlybot off. While Curlybot cannot provide the complexity of a full programming language, it does allow children to gain intuition about repeated motions. The designers describe how sensors could be added to Curlybot to allow children access to if and while statements, but these additions have not been implemented.

3.1.1.3 PROVIDE MULTIPLE METHODS FOR CREATING PROGRAMS

Entering programs as text can be much harder than alternatives such as direct manipulation or form filling but often gives the student more power. In a system that provides multiple methods for specifying programs and represents the resulting program in all program formats, students can use an easier method of program specification to help in learning a more complex, more powerful one. The system in this category provides multiple methods, including standard text, for specifying programs so that students can leverage the simpler methods to learn to program in a standard, textual format.

Leogo: A. Cockburn and A. Bryant, University of Canterbury, 1997 [Cockburn and Bryant, 1997]

Leogo is a system that produces drawings similar to the Logo turtle (see section 4.1.2.1). However, rather than concentrating on one method for creating programs, it provides three: a typed syntax similar to Logo, a direct manipulation interface in which the turtle is dragged around and his actions are recorded, and an iconic language which contains templates for defining structures and using common turtle commands. Motions are expressed in all code styles simultaneously; when the turtle is dragged forward 15 units, the text window shows forward 15, and the iconic window shows forward 15 in icons so it is possible to learn some of the iconic and typed languages using direct manipulation.

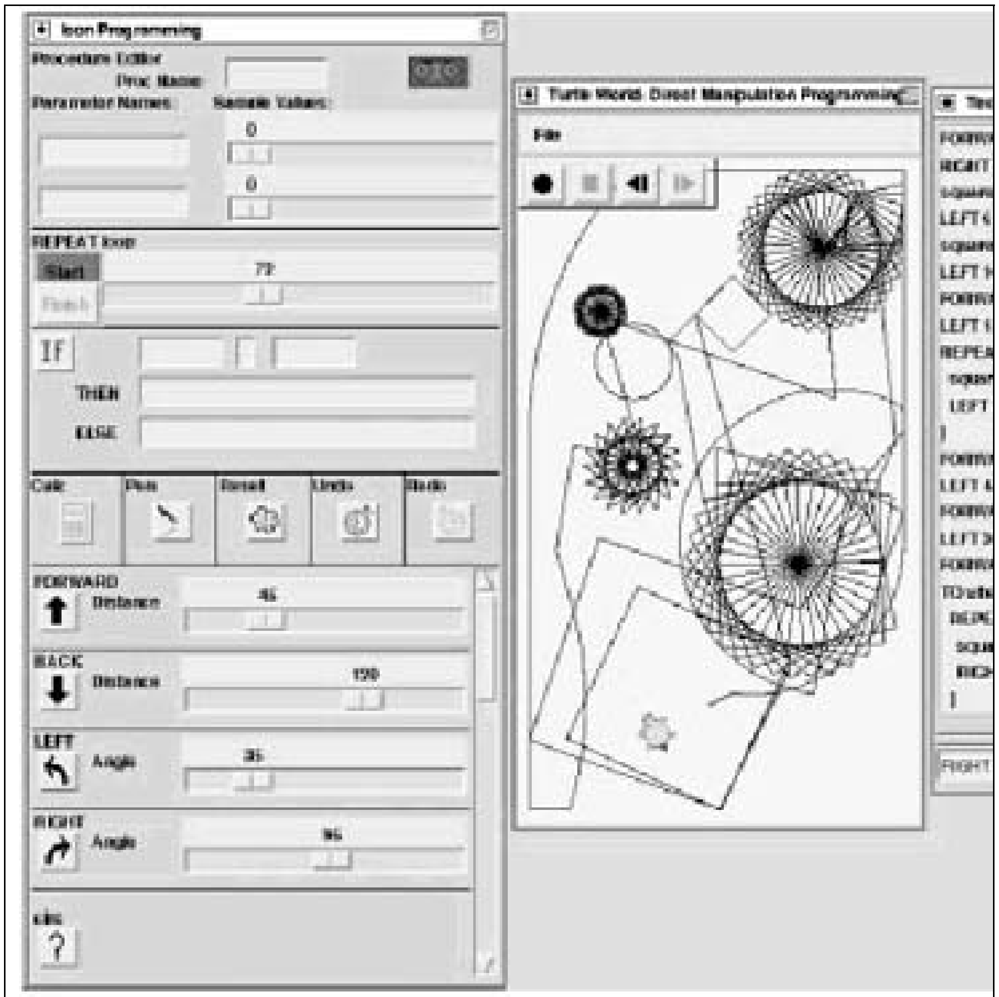


Figure 11. The Leogo interface showing iconic, direct manipulation, and textual programming.

3.1.2 Structuring Programs

These systems concentrate on the structure of code and how it is organized rather than on the syntax of short segments of code. This section includes systems that have tried "new" paradigms for programming. There are two groups here – ones that are changing the paradigm and ones that are trying to make changed paradigms more understandable

3.1.2.1 NEW PROGRAMMING MODELS

Instead of focusing on the syntax of specifying small sections of programs, these systems focus on how instructions are combined and organized to form more complex programs.

Pascal: N. Wirth, Institut fur Computersysteme, 1970 [Wirth, 1970]
The first version of Pascal was created in 1970 for use in teaching programming, particularly systems programming. At the time, the other available languages were FORTRAN, COBOL, and Algol, none of which supported the Structured Programming proposed by Dijkstra [Dijkstra, 1969]. Pascal was introduced in beginning programming classes in 1971 to enable professors to teach Structured Programming to their students in their first course. Although Pascal was designed with teaching in mind, the improvements in the language can be seen as general improvements in programming languages. Algol, one of the primary influences, had ambiguities in the ways nested ifs could be interpreted; Pascal removed these. In addition, Pascal added new basic types and the ability to define special purpose types through struct statements.

Smalltalk: A. Kay and A. Goldberg, Xerox PARC, 1971 [Kay, 1971]
The first version of Smalltalk was created in 1971 at Xerox PARC as the language for the KiddyKomputer, Alan Kay's original name for a portable computer designed for use by a child. Where BASIC attempted to provide a simpler programming language by reducing the number of commands and removing unnecessary syntax, the Learning Research Group (LRG) at PARC concentrated on the model of programming. The group wanted to create a programming language with a simple model of execution and a method of programming that could accommodate a wide variety of programming styles. Smalltalk was based around three ideas: (1) everything is an object, (2) objects have memory in the form of other objects, (3) and objects can communicate with each other through messages.

Playground: J. Fenton and K. Beck, Apple Computer, 1989 [Fenton and Beck, 1989]
Playground is an object oriented programming environment designed to allow children to create their own graphical objects and give them behavior. The programming model was

based on a biological metaphor in which all objects are independent “organisms”; the model was influenced both by Minsky’s Society of Mind [Minsky, 1986] and by classical ethology (the study and description of animal behavior). Each object has its own sensors, effectors, and processing elements so it can act independently. Programming in Playground is rule-based; rules describe both the action and the circumstances under which it should occur. Students specify rules for each object using a natural-language-influenced scripting language. One of the suggested projects for the system is a virtual aquarium with different species of fish and plankton that feed on each other. A fish might have a rule that caused it to eat an algae cell if it saw one and was hungry. A larger fish might eat a smaller fish.

Kara: R. Reichert, W. Hartmann, J. Nievergelt, M. Braendle, T. Schlatter ETH Zurich, 2001 [Hartmann, 2001]
Kara is a graphical programming language based on Karel the Robot that uses finite state machines to organize procedures. Kara can move, turn, pick up and place clovers, and detect tree stumps and clovers; these commands and questions are represented graphically. In each state, the user can ask questions of Kara’s current position and, based on the answers to these questions, supply a sequential list of instructions and the name of the next state in the machine. The finite state machine diagram of the program is provided to show the structure of the program and to allow the user to select a pre-existing state to edit. The use of the simple finite machine model for programming allows the Kara environment to be completely graphical; no typing is necessary, which is an advantage for beginning programmers. In addition, to aid the transition from introductory programming in Kara to “real programming” the authors have supplied JavaKara, an environment that provides a transition to Java, MultiKara, an environment that introduces concurrent programming, and TuringKara, an environment that allows students to experiment with Turing machines in a two dimensional plane.

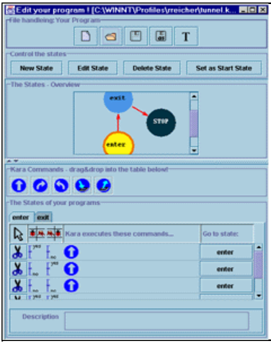

Edit your program ! [C:\WINNT\Profiles\rreicher\tunnel.k...
File handling: Your Program
Control the states
New State
Edit State
Delete State
Set as Start State
The States - Overview
exit
STOP
enter
Kara Commands - drag&drop into the table below!
The States of your programs
enter
exit
Kara executes these commands...
Go to state:
enter
enter
enter
Description

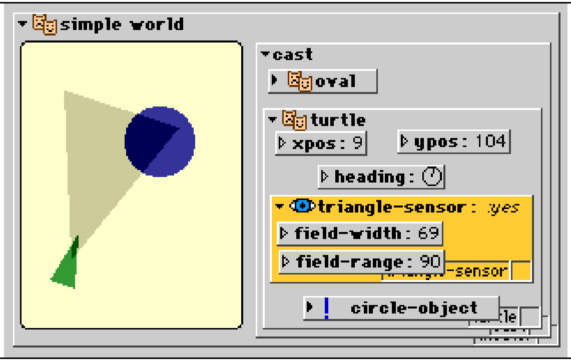
Figure 12. A screenshot of Kara showing a finite state machine with three states: enter, exit, and stop. Below the state machine are Kara’s instructions based on whether there are tree stumps beside her. Each line contains instructions for a given scenario. For example, if there is a stump on Kara’s right and not on her left, she should move forward and go to state enter.

3.1.2.2 MAKING NEW MODELS ACCESSIBLE

Some programming styles, such as object-oriented programming, can be difficult for beginners to understand but can be helpful either in organizing larger programs or representing particular types of behaviors. Rather than requiring novice programmers to learn multiple styles of programming, the systems in this category attempt to make these more complex, but ultimately helpful, styles of programming accessible to novice programmers.

Liveworld: M. Travers, MIT Media Lab, 1994 [Travers, 1994]

Liveworld is an object oriented programming environment built to improve on Playground (see section 3.1.2.1). In Playground, creating and interacting with graphical elements is very simple, but interacting with the rules and attributes that govern the behavior of the objects is much more difficult. Liveworld attempts to create a graphical interface for the rules and attributes of objects so they are more accessible to novice programmers. The interface is similar to a hierarchical browser; parts of objects can be opened, revealing the details of those objects. The user can dive down and change the Lisp code controlling the behavior of objects or simply use the objects, depending upon how much detail the user of the system wants to see. This allows novice programmers to use more complicated objects as black boxes, which would have been difficult in Playground.



```
if (> (ask self :distance-senser)
      (ask self :last-distance))
(ask self: turn-left (arand 0 180))
(ask self: turn-left (arand 0 10)))
```

Figure 13. (a) A simple world in Liveworld containing two objects, an oval and a turtle. The turtle is open so that the user can see its details. (b) An example of Lisp code used in Liveworld to turn a turtle.

Blue Environment: M. Kolling and J. Rosenberg, University of Sydney, 1996 [Kolling, 1996]

There are a number of steps involved in creating an executable program: writing, editing, compiling, testing, debugging. While there are a variety of Integrated Development Environments (IDEs) available (e.g. Visual C++, JBuilder, etc), most of these were created to support a procedural style of programming. To make it easier for students to learn object-oriented programming in their first course, environments should be designed to support object-oriented programming. The Blue environment supports object-oriented programming by explicitly representing the relationship between the objects in a graphical tree. Users can click on a particular class to view the code for that class. In addition, a class-testing bench allows users to create an instance of any class and call its public methods. This allows users to test individual objects outside of the context of the running program, better supporting an object-based design. Compiling and debugging are also supported in the environment, similar to other commercially available IDEs.

Karel++: J. Bergin et al, Pace University, 1997 [Bergin et al, 1997]
Karel J Robot: J Bergin et al, Pace University, 2000 [Bergin et al, 2000]
J. Karel: B. Becker, University of Waterloo, 2001 [Becker, 2001]
Karel J Robot, J.Karel, and Karel++ are versions of Karel the Robot that concentrate on preparing students for object-oriented programming rather than procedural programming. Karel J Robot and J Karel use Java-style syntax; Karel++ uses C++ style syntax. Rather than creating procedures to teach Karel to turn right, students subclass a basic robot to create a right-turning robot. These systems all leverage off the success of the original Karel the Robot to attempt to introduce object-oriented programming early such that thinking and programming in an object-oriented manner will seem more natural to students.

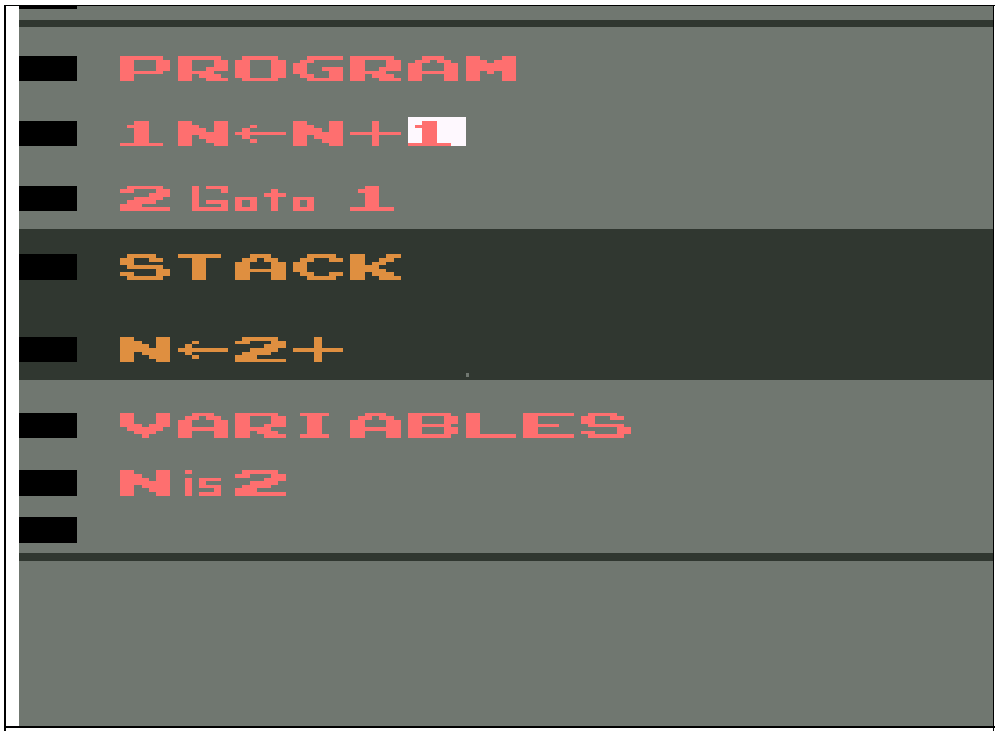
3.1.3 Understanding Program Execution

A syntactically correct program may not perform the actions that the student author intended. For beginning programmers, understanding how programs are executed and how to find mistakes in their programs can be difficult. The systems in this category try to help students understand what happens during the execution of programs, either by placing programming into a concrete setting or by providing a physically based model of how programs are executed in more general-purpose languages.

3.1.3.1 TRACKING PROGRAM EXECUTION

Atari 2600 BASIC: W. Robbinett, Atari, 1979 [Robbinett, 1979]
The Atari BASIC Cartridge allowed children to write short programs in a variant of the BASIC language and watch them as they executed. Atari BASIC divided the screen into six regions: the Program region, which displayed the child's program; the Stack region, which displayed expressions as they were evaluated; the Variables region, which displayed each variable and its current value; the Output region, which displayed all

program output; the Graphics region, a 2D graphical region with sprites; and the Status region, which displayed the current execution speed of the interpreter and the amount of remaining memory. Atari BASIC contained simple support for observing what was happening as the program executed, similar to the supports found in many debuggers. As a child's program ran, several parts of the display changed to reflect the current state of the program: a program cursor showed the current line of code being executed; the stack updated as expressions were added or evaluated; the values of variables changed as appropriate; sprites might move in the graphics region; and the program might play a sound.



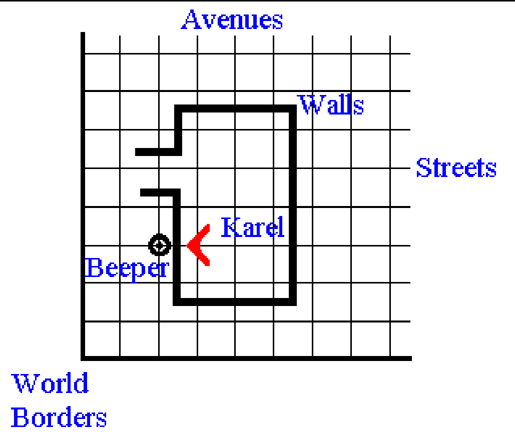
```
PROGRAM
1 N←N+1
2 Goto 1
STACK

N←2+
VARIABLES
N is 2
```

Figure 14. A simple program in Atari 2600 BASIC. The areas of the screen update to show the current position and state of the program.

3.1.3.2 MAKE PROGRAMMING CONCRETE: ACTORS IN MICROWORLDS

Most introductory programs in general-purpose languages are fairly abstract; the computer performs arithmetic operations on numbers and stores the results in invisible registers, making it difficult for students to understand and correct problems in their programs. The micro-world, inspired by the Logo turtle (see section 4.1.2.1), attempts to make programming more concrete by introducing students to programming constructs through controlling the behavior of an actor in a simple, physically based world. The actors usually perform only a few actions, resulting in small languages that students can master more quickly than general-purpose languages. Micro-world based systems also typically include simulators that allow students to watch the progress of their programs. Using micro-worlds, students can quickly gain familiarity with many of the control structures like if-statements and loops, allowing them to devote more time and energy to mastering the syntax and new commands when they move on to general-purpose languages.

Karel: R. Pattis, Carnegie Mellon University, 1981 [Pattis, 1981]
Karel the Robot is one of the most widely-used mini-languages, originally designed for use at the beginning of a programming course, before the introduction of a more general-purpose language. Karel is a robot that inhabits a simple grid world (see Figure 15) with streets running east-west and avenues running north-south. Karel's world can also contain immovable walls and beepers. Karel can move, turn, turn himself off, and sense walls half a block from him and beepers on the same corner as him. A Karel simulator allows students to watch the progress of their programs step by step. Unlike many of the systems discussed in this paper, Karel is supported by a short textbook, making it easier for teachers to incorporate Karel in their classes.



```
BEGINNING-OF-PROGRAM
  DEFINE-NEW-INSTRUCTION
turnright AS
    ITERATE 3 TIMES
      turnleft;

  BEGINNING-OF-EXECUTION
    turnright;
    ITERATE 2 TIMES
      move;
    turnleft;
    ITERATE 2 TIMES
      move;
    turnleft;
    ITERATE 2 TIMES
      move;
    turnleft;
    move;
    pickbeeper;
    turnoff;
  END-OF-EXECUTION
END-OF-PROGRAM
```

Figure 15. Left, a simple Karel world with Karel in a room and a beeper outside the door. On the right, a program that will move Karel to the beeper's location and have him pick up the beeper.

Students can create procedures using DEFINE-NEW-INSTRUCTION (Figure 15), but variables and data structures are not supported in the language. The syntax was designed to be similar to Pascal (see section 3.1.2.1) to ease the transition from Karel to Pascal after the first few weeks of an introductory programming course. There are a number of other robot-based micro-worlds that are described in a survey of mini-languages (Brusilovsky et al, 1997).

Josef the Robot: I. Tomek, Acadia University, 1983 [Tomek, 1983]
Like Karel, Josef is intended to introduce programming to beginners using a robot, Josef, in a simulated world. Josef lives in Wolfville, which is represented by an ASCII map; users can replace the map of Wolfville with one of their own choosing. He knows how to turn left and right, and move forward. The user can also set the speed at which Josef moves. However, unlike Karel, Josef can say and listen for text strings, enabling input - output programs. Additionally, he can drop text markers (e.g. the string "cat") similar to Karel's beepers anywhere in his world. Unlike Karel, Josef was intended to be used for a full semester of programming for non Computer Science majors. To support a full semester of use, it includes many more programming constructs than Karel, such as parameters, variables, and recursion.

Turingal: P. Brusilovsky, University of Pittsburgh, 1991 [Brusilovsky, 1991]
Turingal is micro-world based language in which the actor is a Turing machine and the world is the infinite tape designed to give students exposure to the standard programming constructs as well as the classic Turing machine. The instructions in the language allow the actor to move left and right along the infinite tape as well as read and write symbols on the tape. Like Karel, the basic instructions are easy to visualize. The Turingal language supports conditional, loop and case statements and procedures so that students can gain experience with them in a visual setting. The language uses Pascal syntax (see section 3.1.2.1) to ease the transition from Turingal to Pascal. In support of a computer literacy course for Russian high school students, Brusilovsky also created Tortoise, a micro-world based on Turingal which uses a two-dimensional field of symbols to make it more attractive to younger students (Brusilovsky et al, 1997).

3.1.3.3 MODELS OF PROGRAM EXECUTION

Rather than creating a language that has a simple, physical interpretation, the systems in this category provide physically based metaphors for explaining actions in a more general-purpose language. These metaphors can help students both to imagine the execution of their programs and perhaps more clearly understand why their programs do not perform as expected.

ToonTalk: K. Kahn, Animated Programs, 1996 [Kahn, 1996]
ToonTalk has a physical metaphor for program execution that is similar to that of Prototype 2. In ToonTalk, cities and the creatures and objects that exist in cities represent programs. Most of the computation takes place inside of houses; trainable robots live inside the houses. Communication between houses is accomplished with birds that carry objects back to their nests. Unlike Prototype 2, the ToonTalk environment places the user within the city (program). Using interaction techniques commonly found in videogames, users can navigate around the space, pick up tools, and use tools to affect other objects. By entering the thought bubbles of robots and showing them what they should do using standard ToonTalk tools, users construct programs.

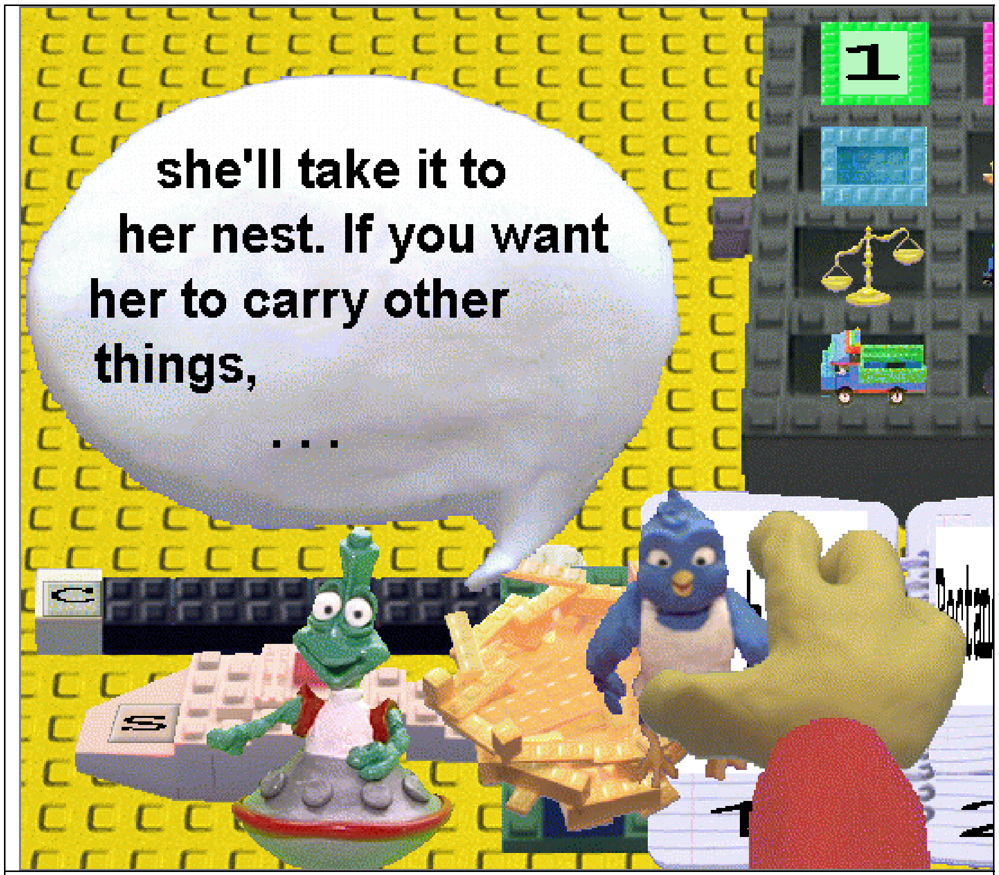

she'll take it to her nest. If you want her to carry other things, . . .

Figure 16. A view of ToonTalk from inside a house. Marty the Martian provides information about objects and what they can do.

Prototype 2: D. Gilligan, Victoria University, 1998 [Gilligan, 1998]
Prototype 2 personifies the flow of control in a computer using a clerk following instructions. The clerk can interact with calculators, I/O devices, worksheet machines, and his clipboard in executing a program. Calculators represent the computer's math processor, I/O devices represent communication with the computer user, the clipboard represents the program stack, and the worksheet machines produce stacks of worksheets that represent the instructions in user-defined subroutines. Rather than imagining the internals of a computer, a novice programmer can imagine the clerk walking around a room interacting with calculators, I/O devices, worksheet machines, and his clipboard, and executing the instructions specified on his clipboard. This model was used in the creation of a programming by demonstration-based system in which the user plays the part of the clerk and demonstrates the actions the clerk should take. The system records these actions. While Prototype 2 uses an anthropomorphic metaphor, the system does not include a graphical representation of the clerk and the objects in his world; instead it is a standard graphical user interface with sections of the interface that represent each of the objects in the clerk's world (e.g. the calculator, I/O devices, etc.) that the novice programmer can use to demonstrate how the clerk should behave.

3.2 Social Learning

Some of the most effective learning is done in a social context where more than one person is working with a problem. Since programming is known to be hard and children often learn more effectively in groups, perhaps it may help the learning process to provide a social context in which learning can occur. The systems in this category investigate different methods for allowing students to work together: co-located and over a network connection.

3.2.1 Side By Side

Most computer interfaces are designed for single users. Consequently, when groups of children use a standard mouse, monitor, and keyboard setup in learning, one child tends to dominate the process. The systems in this category use tangible interfaces to allow multiple students in informal groups to work together in solving programming problems. Because of the difficulty of representing the wide variety of programming constructs in a tangible form, these systems concentrate on small subsets of programming.

AlgoBlock: H. Suzuki and H. Kato, NEC Information Technology Research Laboratories, 1995 [Suzuki and Kato, 1995]

The authors of AlgoBlock wanted to create an active learning community among children learning to program in which children can share notes and techniques, and learn from each other. They created AlgoBlock, a set of blocks, each of which corresponds to a simple command in Logo (see section 4.1.2.1). The blocks can be connected together to form programs that control the movements of a submarine in a maze. The blocks are tangible and large enough that they can be arranged on a desk that several students can work around. This allows students to work with the blocks in a social context, learn from each other, and communicate what they are learning. The tangible nature of the blocks made it easy for children to take turns manipulating the blocks and communicate about which pieces should be placed where. The AlgoBlock project demonstrates that, in a suitable environment, children will work together in building programs. However, the blocks supported a limited set of programming constructs; the children were not able to explore concepts like procedures, parameters, or control structures.

Tangible Programming Bricks: T. McNerney, MIT Media Lab, 2000 [McNerney, 2000]

Tangible Programming Bricks are Lego blocks that can be stacked together to form programs. The designer's intent in creating these was to provide a simple interface to appliances and toys and to create a programming environment that would allow children to collaboratively explore ideas. While the work concentrated on the hardware implementation of the Lego blocks, the designer created three prototype environments using Lego blocks that represent commands. To allow a greater variety of commands, users could insert a small card (e.g. microchip) into a block. Each block could accept a single card, allowing users to communicate with other blocks via IR transmission, supply parameters to commands, sense the environment, or display variables. The three prototype languages allowed children to teach toy cars to dance, kitchen users to program microwaves, and toy trains to react to signals along the side of the tracks in unique ways. By stacking blocks together with accompanying cards, if necessary, users could construct simple programs.

3.2.2 Networked Interaction

Rather than trying to move away from the common single user, single computer paradigm, the systems in this category attempt to allow students using different machines to work together over the network. While the systems designed for students working side-by-side can assume all children can see the state of the current program and what other children are doing, programming systems designed for network use need to explicitly support the exchange of this kind of information.

MOOSE Crossing: A. Bruckman, MIT Media Lab, 1997 [Bruckman, 1997]
MOOSE Crossing is a networked programming environment built for children. It is an adapted text-based MUD (multi-user dungeon) in which children can use an object-oriented scripting language to create spaces and characters that inhabit a textual world (see Figure 17). Children often create spaces and characters similar to those found in text adventure games such as castles complete with secret passages that other children can explore. Once their projects are completed, any child in the MOOSE Crossing environment can interact with them. In addition, the environment allows children to view the scripts controlling any object or character in the environment and chat with children that are currently logged onto MOOSE Crossing. In general, children work alone on projects but one child will often use another child's project as an example. Children may also ask another user for help or advice. The MOOSE Crossing community has provided a source of help, role models, and positive feedback for users of the system as they create their own projects.

```
on pet this
   tell player "You pet Rover."
   if player member_of my friends
       emote "wags his tail."
end
```

Figure 17. A MOOSE Crossing script that allows MOOSE users to pet Rover. When a user pets Rover, they are told "You pet Rover." If they are one of Rover's friends, then Rover wags his tail.

Pet Park: A. DeBonte, MIT Media Lab, 1998 [DeBonte, 1998]
Pet Park is an exploration of the ideas of MOOSE Crossing in a 2D graphical domain rather than a textual one. Children can choose one of 5 dogs to be their pet. Each dog comes with a few animations, such as wagtail, jump, walk, laugh as well as basic ones like wait, turnLeft, say, etc. Users can combine these simple commands to create their own animations using a textual scripting environment or a set of graphical blocks representing each command. As in MOOSE Crossing, Pet Park is a networked programming environment in which children can talk, ask each other for help, and show off their creations. While in MOOSE Crossing, children create spaces by describing them with text; in Pet Park, creating a space requires graphical objects. In response, the system

provides a variety of furniture, objects, and rooms. Furniture and rooms can be programmed to react to simple events such as avatars coming near them.

Cleogo: A. Cockburn, University of Canterbury, 1998 [Cockburn, 1998]
Cleogo is a networked version of Leogo (described earlier) that allows children to see and interact with the same Leogo workspace. Rather than concentrating on building a community of programmers, Cleogo creates a shared environment, the current program being edited, and allows multiple children to see and manipulate that environment. Cleogo does not attempt to provide children with a way to communicate with each other about their project. Instead, it assumes that they are either in the same room or can talk to each other using the phone or some equivalent.

3.3 Providing Reasons to Program

Beginning programmers often do not know exactly what they want to build, what is possible in the programming system they are using, or how difficult certain kinds of projects will be to complete. The systems in this category provide starting points in learning to program. By providing specific activities for beginning programmers, these systems can introduce programming constructs gradually which may help to prevent beginning students from getting overwhelmed. In addition, these systems often use themes they believe children will find appealing.

3.3.1 Solve Problems by Positioning Objects

In these systems, students position objects to solve a series of puzzles. As students get more advanced, the puzzles become more difficult. The gradual progression of difficulty allows the designers of the system to introduce constructs and problems and provides students with a series of realizable and interesting goals.

Rocky's Boots / Robot Odyssey: W. Robbinett, The Learning Company, 1982 [Robbinett, 1982]
Rocky's Boots was one of the first educational software products for personal computers to successfully use an interactive graphical simulation as a learning environment. The game allows children to connect logic gates (AND, OR, NOT and flip-flop) together to create circuits using a joystick (see Figure 18). When the circuits are active, users can watch the wires turn from white to orange as the electricity passes through them. The game provides a series of puzzles in which the player is supposed to separate the shapes matching a certain criteria from those that do not using logic gates, sensors that can detect certain kinds of shapes, and a boot that, when activated by a true value, kicks the current

shape out of the line and off to one side. Robot Odyssey follows the same basic pattern; the player connects gates together to solve problems. However, Robot Odyssey includes a larger selection of objects like the shape-kicking boot that perform physical actions when they are activated, creating a wider set of possibilities for the behaviors of circuits.

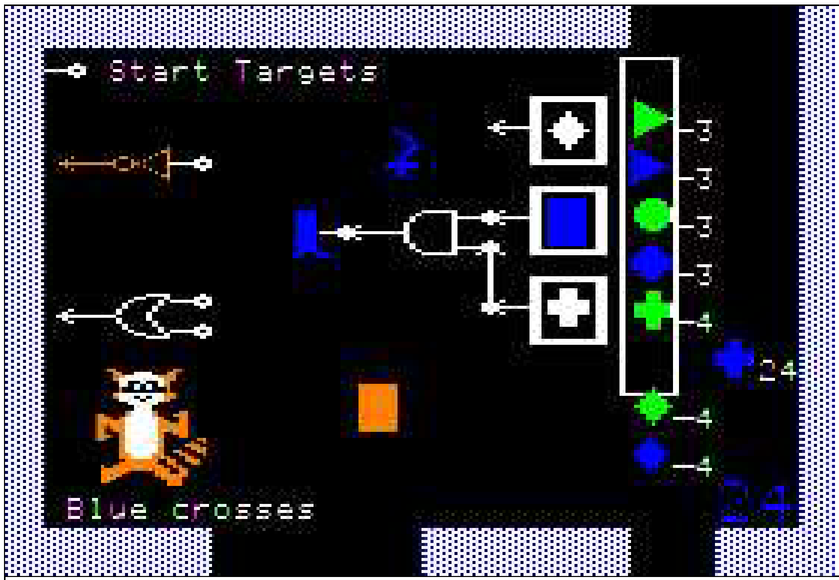


Figure 18. A puzzle from Rocky's Boots in which the player is asked to create a circuit that separates blue crosses from the other shapes. When the circuit is switched on, shapes move up the right side of the screen. When they enter the white rectangle, the shape sensors to the right of the rectangle can detect them. The player is asked to attach a sequence of logic gates to the sensor that will activate the boot (center) when a blue cross enters the box. The boot, when activated, will kick the shape out of the rectangle.

The Incredible Machine: Sierra Entertainment, 1993. [Sierra, 1993]
In the Incredible Machine, the player is given a series of Rube Goldberg style challenges (see Figure 19). For example, the player may be asked to construct a way to get a ball to fall into a basket. Each challenge includes a short description and all the parts necessary to create the machine described. Players can select parts and position them in the world and then start the simulation to test their machine. When the simulation is running, the parts respond as they would in the physical world. If users run into trouble, they can ask for hints. More advanced users can use a free play mode to create their own machines.

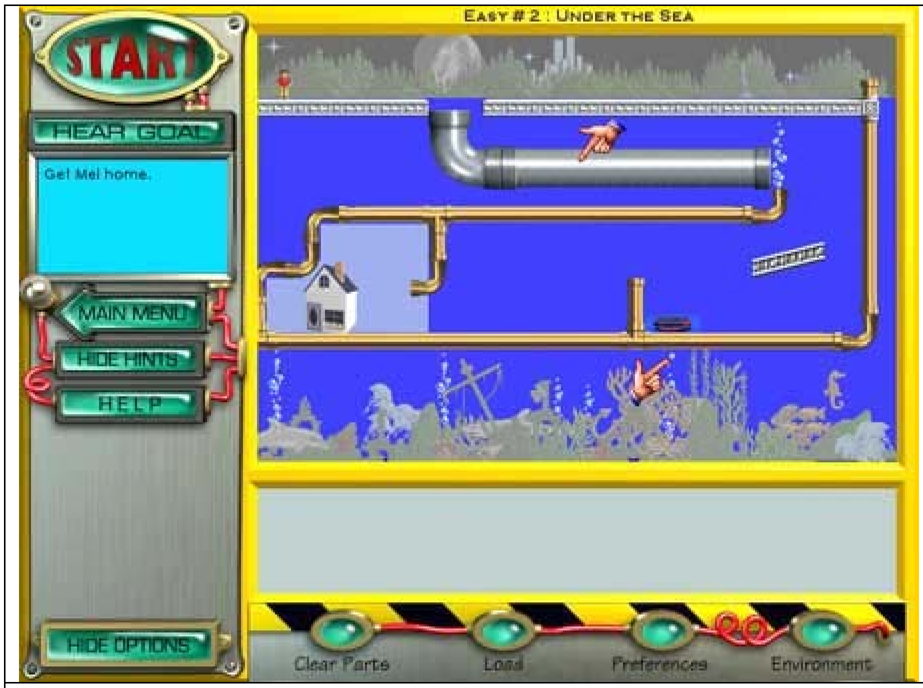


Figure 19. An easy challenge in The Incredible Machine: the player needs to help Mel (top left) get back to his house. The puzzle has been solved by positioning the grey pipe, ramp, and a trampoline so that Mel will go through the pipe, slide down the ramp, and bounce off the trampoline and over the barrier to get home.

Widget Workshop: Maxis, 1995 [Maxis, 1995]
Widget Workshop provides a series of puzzles that players attempt to solve by connecting different components together using graphical wires. Each puzzle poses a specific question (e.g. what colors of light do you add together to get white) and provides a context in which to experiment with that question (e.g. red, green, and blue lights controlled by switches that connect to a "light box" where they are combined). Widget Workshop also provides a free play mode in which users can create their own widgets by connecting pre-made parts together.

3.3.2 Solve Problems Using Code

Motivation can be a key element in learning; if students want to learn to do something, obstacles will not deter them as much. These systems concentrate on providing a reason that a novice would want to program by creating an environment in which the novice programmer gets to do something fun.

AlgoArena: H. Kato and A. Ide, NEC Information Technology Research Laboratories, 1995 [Kato and Ide, 1995]
In AlgoArena, players write programs to control the behavior of sumo wrestlers fighting tournaments. The programs are written in a language based on Logo (see section 4.1.2.1). When a player has completed a program, the player can log onto a website and have his or her wrestler fight against another student's wrestler. Over time, by analyzing the circumstances in which the player's sumo wrestler loses tournaments, the player is expected to learn more complex ways to control the wrestler, perhaps querying the position and posture of their opponent before deciding which moves to execute.

Robocode: M. Nelson, IBM Advanced Technology, 2001 [Nelson, 2001]
Robocode is designed to help novices learn Java through programming a robotic battletank for a "fight to the finish". The tutorial teaches novices to subclass an existing battletank robot and extend the robot's capabilities using standard Java and a set of classes written for the Robocode environment. Upon completion of a robot, users can upload their creation to a number of websites or join a robotic battle league. The designer of the system believes that the ability to program robotic battles will provide enough motivation to get a novice programmer over the hurdles of beginning to program.

4. EMPOWER PEOPLE

The systems in this category are built with the belief that the important aspect of programming is that it allows people to build things that are tailored to their own needs. Consequently, the designers of these systems are not concerned with how well users can translate knowledge from these systems to a standard programming language. Instead, they focus on trying to create languages and methods of programming that allow people to build as much as possible.

4.1 Mechanics of Programming

The systems in this category are designed around the hypothesis that the primary barrier for people attempting to use programming as a tool is the mechanical difficulties of creating programs. Systems in this category examine ways of improving programming languages and alternative ways for creating.

4.1.1 Code is Too Difficult

Many researchers have examined the problem of making languages more understandable and usable for novices. While progress has been made making programming languages more understandable, there still are many barriers for novices trying to build their own programs. These systems examine creating programs either through demonstrating correct behavior or selecting actions through the interface.

4.1.1.1 DEMONSTRATE ACTIONS IN THE INTERFACE

The systems in this category examine ways that users can program a system by showing the system what to do through manipulating the interface, without relying on a programming language.

Pygmalion: D. Smith, Stanford University, 1975 [Smith, 1993]
Pygmalion was the first programming by demonstration system. Unlike many of the systems that came after it which concentrated on graphical objects, Pygmalion attempted to get people to write more abstract programs, such as a program to compute the factorial of a number. However, rather than building factorial by typing statements in a programming language, Pygmalion relied on editing an artifact. To create a factorial program, the user creates an icon with two sub-icons, one for the input and one for the output, and draws a symbol to represent factorial. The user can then enter remember mode, in which all of the actions made by the user are remembered by the system. Consequently, the user can program the computer by working out an example of how to compute factorial. However, the user must anticipate the handling of the value one and test whether or not the current value, say three, is equal to one, something that novices may not be well prepared to do. If the user does not demonstrate his or her current actions as the case for the current value not being equal to one, Pygmalion will not know that one should be handled differently and, consequently, will not prompt the user to demonstrate how one should be handled.

Programming by Rehearsal: W. Finzer and L. Gould, Xerox PARC, 1984 [Finzer and Gould, 1984]
Programming by Rehearsal was built to help non-programmers create educational software. It is designed around a theater metaphor in which components of the interface are performers that interact with one another on a stage by sending and responding to cues. A user of the system would begin creating a piece of software by auditioning performers to use as building blocks, selecting their cues via a pop-up menu and observing their responses to those cues. The user would then copy the chosen performers onto the stage, placing and sizing them appropriately. The rehearsal portion of development consists of showing the performers what actions they should take in response to user input or cues sent by other performers. Objects that accept user input, such as buttons, have cue sheets that allow users to fill in their responses to those user

inputs. Users can press a closed eye icon to tell the system to observe what their actions. Then, by selecting cues from the menus of other performers, they can show the system how to react to those cues. By pressing the eye icon again, users indicate they have finished. The system comes with 18 basic performers users can audition and use in their own creations. Additionally, the system allows users to create new performers by combining existing performers and teaching them new cues.

Mondrian: H. Lieberman, MIT, 1992 [Lieberman, 1993]
Mondrian is a programming by demonstration system for drawing and graphical editing in which commands are shown with "domino" icons that depict the before and after states for that command. To execute a command, users select the command icon and select the object or area to which the command should be applied. The user can create new commands in a storyboarding style by showing how to do each step in the new command. These steps are displayed at the bottom of the screen in comic book format with a short caption describing each step. Drawing a rectangle on the screen would show a box with the new screen state captioned by "rectangle". If the user then moves the rectangle, a "move" domino would appear beside the "rectangle" domino in the definition of the new command. New commands created by the user are displayed in the same domino style as the commands built into the system. In addition, the system provides speech synthesis capabilities to give an English description of what a command does.

4.1.1.2 DEMONSTRATE CONDITIONS AND ACTIONS

Like the previous category, the systems in this category try to avoid forcing users to express their intentions in code. However, instead of demonstrating programs by performing actions in the user interface, as the systems in the previous category did, the systems in this category allow users to depict the conditions in which they want the program to perform an action and the results of that action.

AgentSheets: A. Repenning, University of Colorado, 1991 [Repenning, 1993]
AgentSheets is an environment for building simulations consisting of graphical agents (represented by icons) in a grid-based world. In early versions of the system users specified the behaviors of their simulations by graphical rewrite rules in which the user selected conditions (configurations of icons in the world or relative to each other) and showed the system what should happen under these conditions by moving the agents to their new positions in the world. However, graphical rewrite rules on their own are insufficient for creating more realistic simulations and complex games. To support a

broader range of simulations, AgentSheets now uses Tactile Programming in which users still specify a list of conditions and a list of actions to take if all of those conditions are true. Conditions can check information such as the appearance of agents, read data from other agents or web pages. Actions might change the appearance of agents, destroy agents, create new agents, or open web pages.

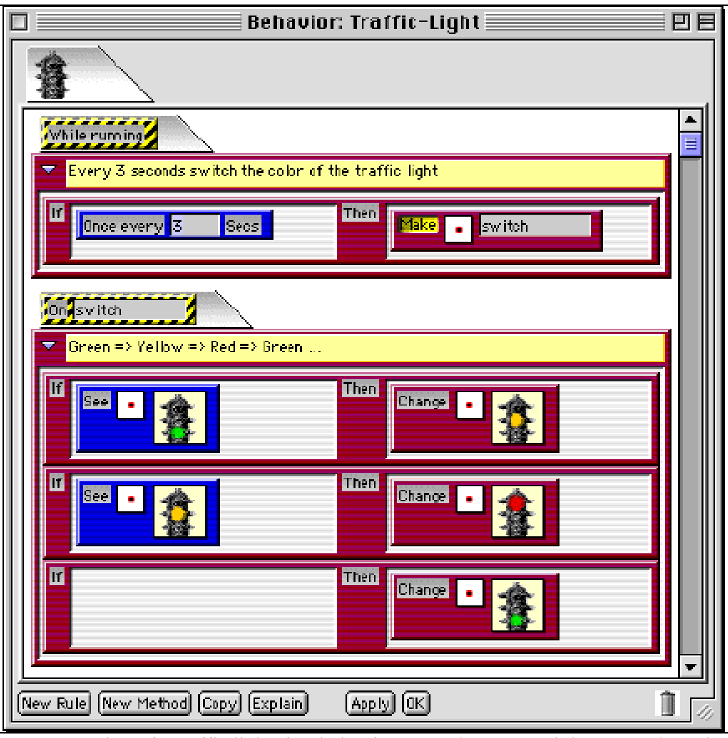


Figure 20. A screenshot of a traffic light simulation in AgentSheets containing two rules. The first rule runs continuously: every three seconds it triggers the second rule. The second rule looks at the current color of the traffic light and changes it to the next one in the sequence green, yellow, red.

ChemTrains: B. Bell and C. Lewis, US West Advanced Technologies, University of Colorado, 1993 [Bell and Lewis, 1993]

ChemTrains is a pictorial rule-based language that attempts to make it easy for people to create a wide variety of "behaving pictures". ChemTrains is similar to Stagecast (see below) in that users show both the conditions and results of a rule through pictures. In ChemTrains the pictures used to specify conditions and results are interpreted as patterns of connections rather than collections of pixels. For example, in simulating an AND gate, if there is any box with a zero connected to the AND gate (from any direction and any distance away), the output of that gate should become zero. A similar statement in Stagecast would only work if the zero connected to the AND gate was always in the same relative position to the AND gate. As in Stagecast, the order of the ChemTrains rules dictates how they are applied; only the first matched rule is applied in each time slot. Additionally, the ChemTrains pattern matcher can use variables; in ChemTrains, variables are specially marked pictorial elements that can match any element of the simulation display. The addition of variables allows users to create a wider range of simulations.

Stagecast: D. Smith, A. Cypher, and J. Spohrer, Apple Computer, 1995 [Smith, 1997]
Stagecast, a commercial version of KidSim (see below), is an environment for creating simulations. Children are presented with a grid-based world in which they can create their own actors. Users define rules for the simulation by selecting a before condition from the grid world and then demonstrating how that condition should change (see Figure 21). When the simulation is started, when a section of the grid matches a condition of one of the rules, the rule is applied. Stagecast applies only the first rule (in top to bottom order) that matches a section of the grid.

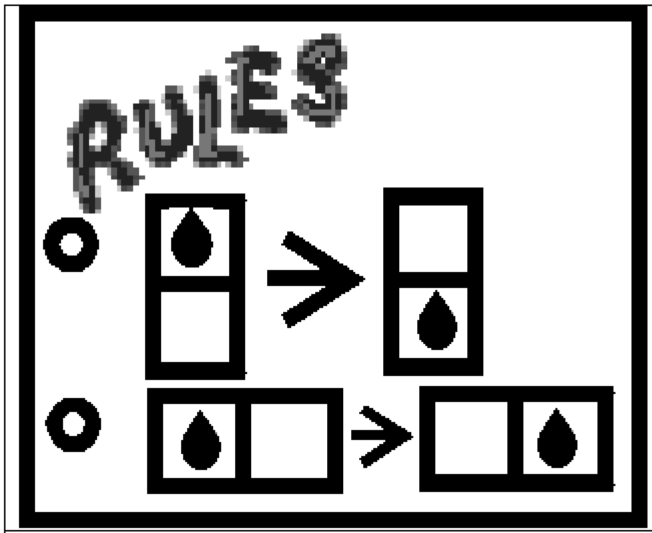


Figure 21. This drawing shows an example of how users create rules in Stagecast. On the left side are the conditions in which each rule should be applied. On the right, the results of each rule are shown. In this drawing, if there is a raindrop with an empty space between below it, the raindrop should move down. Likewise, if there is a raindrop with an empty space on its right, it should move right.

4.1.1.3 SPECIFY ACTIONS

In these systems, the user creates programs by using the interface to specify the desired behavior. The user does not see any code, but unlike in programming by demonstration systems, the user does not show the computer what to do, he or she selects the program's actions.

Pinball Construction Set: B. Budge, Exidy Software, 1983 [Budge, 1983]
The Pinball Construction Set was written in 1983 to allow users to design and build their own pinball machine simulations (see Figure 22). It provided a construction space, a set of pinball parts, and bitmap editing capabilities to allow users to build themed pinball machine simulations. Physical laws and behaviors were written into each part; each part provided could be seen as acting on balls that collide with it in defined ways. In this system, users can program by placing pinball parts in well-defined relationships; this method of programming is similar that employed by The Incredible Machine (see section 3.3.1). For example, users may want to specify that when a ball hits a certain target, it is diverted onto a ramp, and its path affected by a magnet.

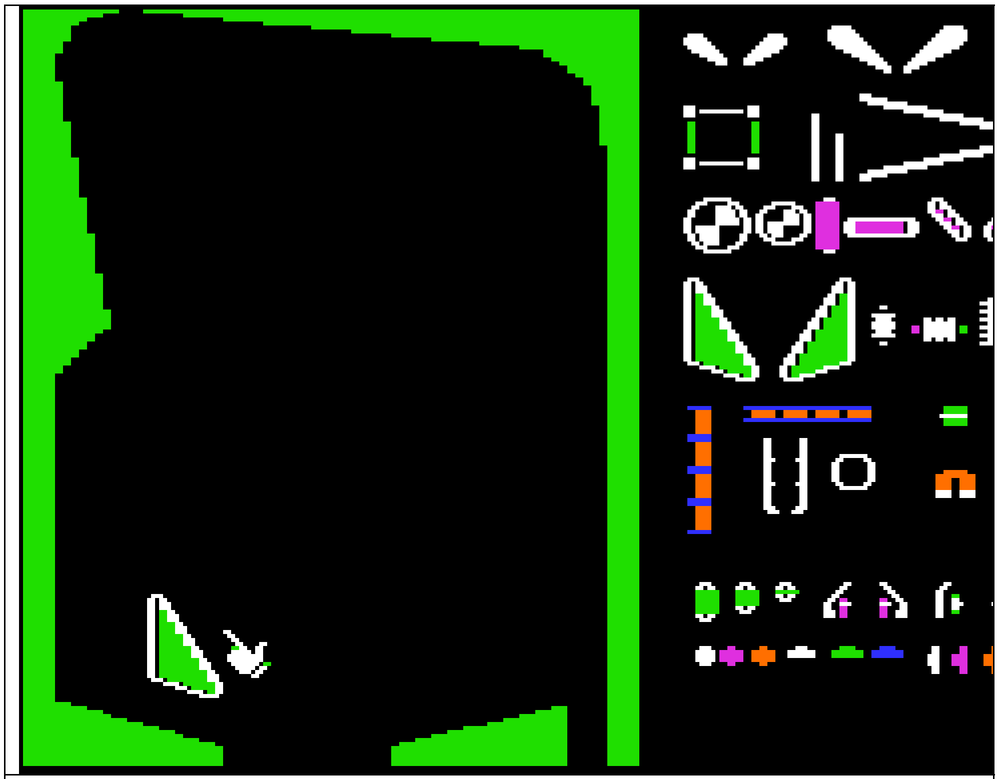


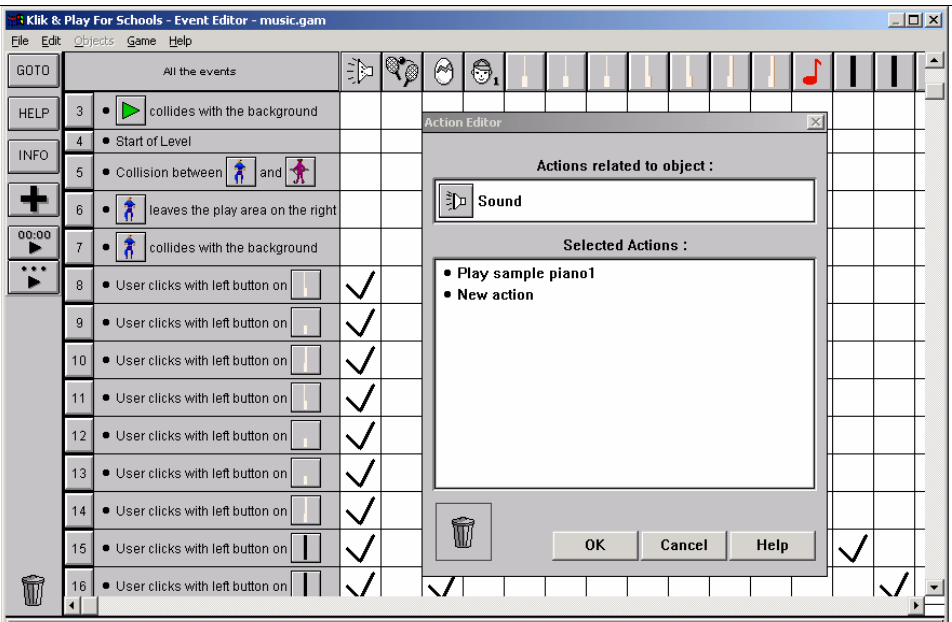
Figure 22. A screenshot of the Pinball Construction Set. On the right is an empty pinball game; on the left are a variety of parts that users can put into their pinball games.

Alternate Reality Kit: R. Smith, Xerox PARC, 1987 [Smith, 1987]
The Alternate Reality Kit (ARK) is an environment in which users can build interactive simulations. Users interact with objects built on a physical-world metaphor; each object has an image, position, velocity, and can be influenced by forces. Users can pick up objects, move them, drop them, or throw them using mouse gestures. Users can query or change the state of objects by sending messages, represented by buttons, to those objects.

To connect a button with a particular object, the user drops the button onto that object. If the object understands the message the button represents, the button "sticks" to the object, otherwise it falls through. Buttons that require a parameter have a little "plug" where users can hook up a value for the parameter.

Klik N Play: F. Lionet and Y. Lamoureux, Europress, 1994 [Lionet and Lamoureux, 1994]

Klik N Play is designed to allow the user to create simple level-based games. The application has three modes: a storyboard editor, which allows the user to see all levels as thumbnails, a level editor, and an event editor. The level editor allows the user to select the background, add predefined objects to the level, and provides users with the ability to create their own objects and animations for those objects. Users create animations frame by frame with a bitmap editor and use controls to set the speed and motion of objects. The event editor uses a table format and allows the user to specify actions for a variety of predefined events (see Figure 23). Klik N Play's events are based on collisions between objects, mouse and keyboard input, time, the state of players, and the states of variables and objects in the level.



All the events

3 • collides with the background
4 • Start of Level
5 • Collision between and
6 • leaves the play area on the right
7 • collides with the background
8 • User clicks with left button on
9 • User clicks with left button on
10 • User clicks with left button on
11 • User clicks with left button on
12 • User clicks with left button on
13 • User clicks with left button on
14 • User clicks with left button on
15 • User clicks with left button on
16 • User clicks with left button on

Actions related to object :

Sound

Selected Actions :

• Play sample piano1
• New action

Figure 23. A view of the event editor in Klik N Play while the user builds a graphical piano program. The user is currently specifying that when the "User clicks with left button on white piano key," the game should play "sample piano1." The events are organized in table form based on their effects: all sound events are in the first column, events on the user's objects, piano keys in this screenshot, begin at column 5.

4.1.2 Improve Programming with Languages

The designers of many of the teaching languages are concerned with how well students can transfer the knowledge they gain in the teaching language to more general-purpose languages. Consequently, the designers of teaching languages have been hesitant to deviate very far from these general-purpose languages. However, the systems in this category endeavor to empower their users to create interesting programs; whether the users of these systems can transfer their programming knowledge to more general purpose languages is not important. Consequently, the designers of these systems can make changes to standard programming languages that the authors of teaching languages might hesitate to make.

4.1.2.1 MAKE THE LANGUAGE MORE UNDERSTANDABLE

These systems include languages that were developed with a focus on the language and words novices use to describe situations. Most previous languages have been developed with a focus on consistency between languages or on mathematical simplicity. These languages instead focus on choosing words that the users of the system understand and can use effectively without having to translate their words in their everyday vocabularies into the words that the computer language uses for the same concept.

COBOL: C. Phillips et al, Department of Defense, 1960 [Sammet, 1981]
COBOL is the COmmon Business Oriented Language, designed to support the creation of business applications. It was intended to be usable by novice programmers and readable by management; spoken English influenced many of the programming constructs (see Figure 24). The designers also added "noise" words to increase the readability of the language: *ADD X TO Y* rather than *ADD X,Y.*

```
  IF X = Y <...>
  IF GREATER <...>
OTHERWISE <...>
```

Figure 24. A conditional statement in COBOL. Conditionals can use implied subjects and objects as seen in the second and third lines of the conditional statement.

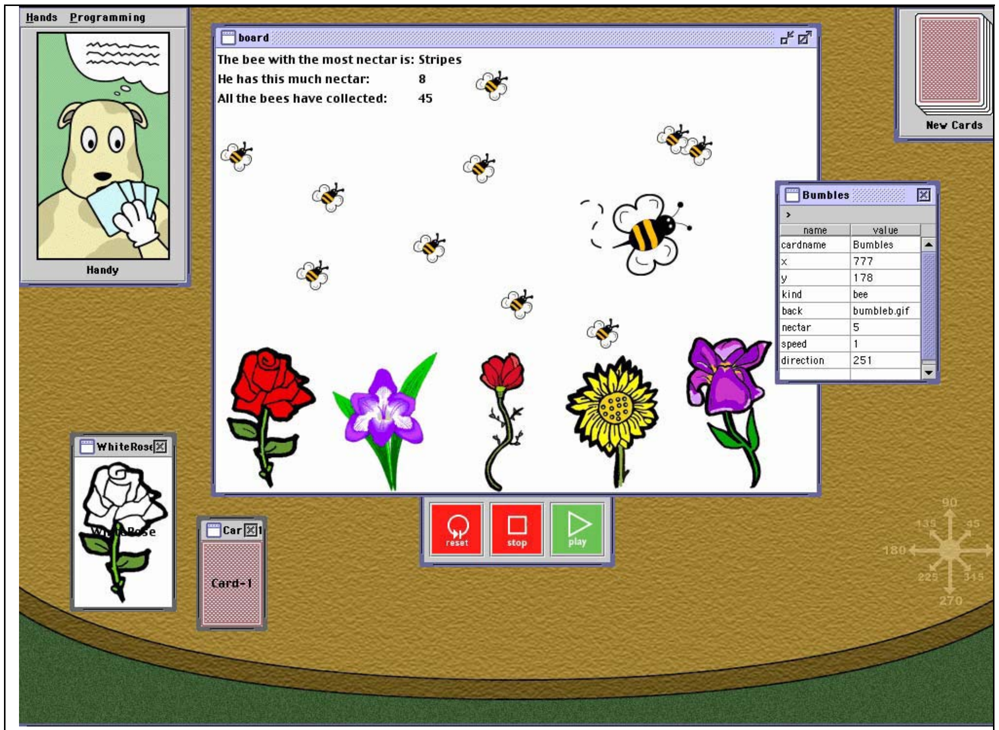
Logo: Seymour Papert, MIT, 1967 [Papert, 1980]
The Logo programming language is a dialect of Lisp with much of the punctuation removed to make the syntax accessible to children. It was intended to allow children to explore a wide variety of topics, from mathematics and science to language and music. The most well known part of Logo is the Logo turtle, which began as a robotic turtle that could draw on the ground. It was later replaced by a simulated actor in a two dimensional

graphical world that can move, turn, and leave trails. The turtle's directions are object-centric; if a child tells the turtle to "forward 10", the turtle will move in his own forward direction rather than a direction defined by the screen. Many children have been introduced to programming through making the turtle draw simple pictures. However, the Logo language includes a wider variety of possibilities. Classes of children have written music programs, programs that translate English to French, and many others. The Logo language is an interpreted language with descriptive error messages. For example, if a student typed "foward 10" instead of "forward 10" the system would respond with "I don't know how to foward."

Alice98: M. Conway et al, Carnegie Mellon University, 1997 [Conway, 1997]
Alice98 is a programmable 3D authoring tool, designed to make authoring interactive 3D graphical worlds accessible to college-level, non-science majors. The authoring tool consists of a scene layout editor in which the user can create their opening scene, and a script tab in which the user can specify the behavior of the world. The programming language in Alice is Python, with a few changes suggested by user testing: it is not case sensitive and ½ evaluates to 0.5 rather than 0. However, Alice provides domain-specific commands for manipulation of objects in 3D. The structure and naming of these domain-specific commands were influenced greatly by user testing. As in Logo, commands utilize object-centric notation: forward, backward, up, down, left and right are used to describe direction. This description is equivalent to XYZ notation, but is much easier for novices to understand. Similarly, the names of commands are drawn from the language that users would choose to describe those actions; for example, translate became move, scale became resize, and rate became speed. Alice commands can also be accessed with varying degrees of detail. At the simplest, *bunny.move* only needs a direction. The user can also specify how far bunny should move, how long the animation should take, what speed he should move at, whether he should move in someone else's coordinate system, and different interpolation styles. This allows novices to begin by learning a very simple command for moving the bunny and, as they gain more experience, learn to express greater control over how the bunny moves through additional options. Alice98 also animates all commands so that the user can understand what has happened. Because Alice98 animates all changes to the state of the program, the user can more easily understand the behavior of their programs.

HANDS: J. Pane, Carnegie Mellon University, 2001 [Pane, 2001]

The HANDS system was designed to allow children in 5th grade and older to create games and simulations similar to the ones with which they play. The design of the system was informed by studies of the language that children with no programming experience use in expressing solutions to programming problems. The environment provides a concrete model of computation, represented by an agent, HANDY the dog, who manipulates a deck of cards. All information used in a program is stored on two-sided cards. The front of each card contains object-related data; the back displays a picture of the object. The user can place cards on the surface of the table, which represents the end-users' view of the program. The HANDS language was designed based on the ways that non-programmers describe solutions to programming problems. It includes queries and aggregate operations that reduce the need for data structures and iteration through lists of items. Children using the HANDS system perform better than children using a version of the HANDS system that does not include queries and aggregate operations.



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 25. All data in HANDS is stored in cards, which the user can draw from a pile shown on the top right of the screen. Two cards are shown, face down, on the lower left. One card on the right has been flipped to face up so that the user can see and edit it's properties. When cards are on the board (in the center of the screen), only the image on their backs are visible. Users of HANDS can add code into Handy's thought bubble by clicking on his picture in the upper left corner.

4.1.2.2 IMPROVE INTERACTION WITH THE LANGUAGE

In addition to changing the language and the words used to describe programming commands and constructs, another area for improvement is in the ways that people interact with language. This includes the interaction involved in entering programs and the process involved in running programs users have written.

1.1.1.1.5. Manipulation of Language

Traditionally, users program systems by typing program statements into a text editor. For novice programmers, typing programs and the strict syntax of most programming languages can be particularly difficult and frustrating. The systems in this category examine different methods for creating programs in ways that are easier for novice programmers to understand and less prone to errors. The systems use a variety of techniques from dataflow metaphors, to menu selection, to physical proximity to allow users to express their intentions without having to type traditional programming statements.

Body Electric: J. Lanier, VPL [Blanchard et al, 1990]

Body Electric was designed as an authoring tool for a two-person virtual reality system. Programs in Body Electric are data driven; raw data from sensors (such as positional sensors on people) can be passed to the representation of the virtual world through modules that are capable of transforming the data or generating events. These modules are represented in the authoring environment as boxes connected by arrows in a flow diagram. Users can create programs that modify and react to sensor data by sending the sensor data through a sequence of modules. Programs are always live, allowing the author to immediately see the results of changes. This allows worlds to be quickly prototyped, tested, and modified.

Fabrik: Ingalls et al, Apple Computer, 1988 [Ingalls et al, 1988]

Fabrik is a computational construction kit in which pieces of functionality (procedures) appear as boxes with connectors. These boxes can be wired together to create a variety of programs (see Figure 26). The user is supplied with a parts bin that includes simple computational elements, such as string and integer manipulation, as well as interface elements such as buttons, images, and lists. By dragging boxes into a working area and connecting them together, the user can create programs. These programs are always live so they can be tested as they are being built. During development, user interface elements and computational elements share screen space. However, once a program is finished, the user can choose to view only the interface elements. In addition, finished programs can be used as elements in subsequent programs, so the user can extend the capabilities of the construction kit.

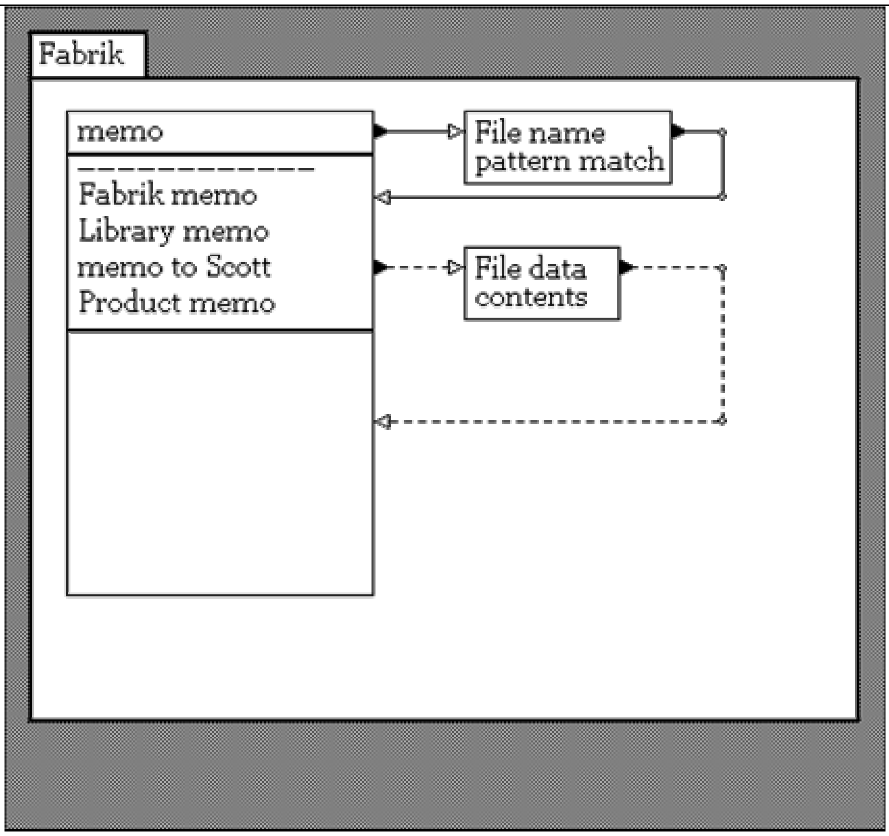
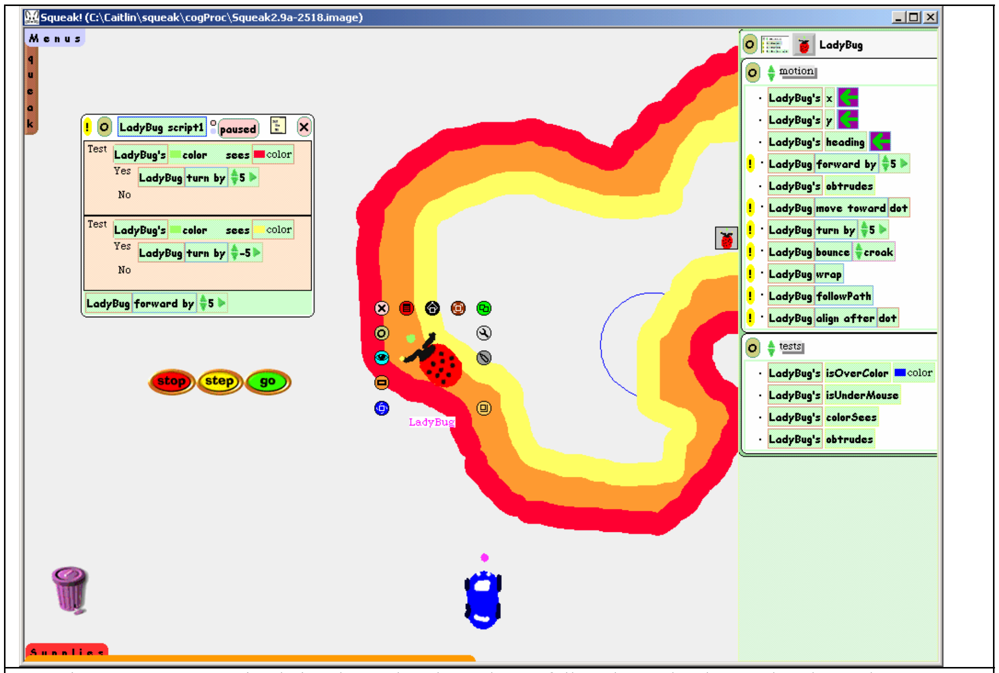


Figure 26. A Fabrik program to create a simple text file editor. In the top left text field, the user can enter a search string for file names. The user's string is passed to a file name pattern matcher and then to a GUI list element. The user can then select the file they want to edit. When a file is selected, the name of the file is passed to a module to retrieve its contents and the contents are passed into a text field for the user to edit.

Tangible Programming with Trains: F. Martin et al, MIT Media Lab, 1996 [Martin et al, 1996]

Tangible Programming with Trains is a train set and collection of active train toys that influence the behavior of the train. The Tangible Programming with Trains system was designed to allow children to explore "pre-programming concepts – causality, interaction, logic, and emergence" (Martin et al). For example, a stop sign that causes the train to stop or a sign that asks the train to turn on its lights. The active train toys and the train can communicate via IR signals such that when the train is close to one of these toys, the train will change its behavior appropriately. Children can place these objects around the path of the train such that it will stop at a station or turn its lights on when it goes through a tunnel.

Squeak Etoys: A. Kay et al, Disney, 1997 [Kay]
Squeak Etoys are designed to allow children to learn ideas by "building and playing around with them" (Kay) either through interacting with simulations others have built or creating their own simulations (see Figure 27). The Etoys environment provides students with a variety of pre-made objects, from simple shapes to trashcans, and a simple drawing tool with which students can create their own objects. All objects have viewers that contain object-specific information as well as tiles that the student can drag out of the viewer to build programs that control the behavior of the object. Programs can change the position, orientation, size, and appearance of objects as well as play sounds. Users can create simple if-statements in their program, but no other standard control structures are included in the Etoys system. Users can trigger object behaviors based on a variety of mouse events, or the behaviors can be started, stepped and stopped with a set of pre-made buttons users can add to their simulations.



```
LadyBug script1   paused
Test  LadyBug's color  sees 🟥 color
  Yes   LadyBug turn by 5
  No
Test  LadyBug's color  sees 🟨 color
  Yes   LadyBug turn by -5
  No
LadyBug forward by 5
```

```
LadyBug
motion
· LadyBug's x
· LadyBug's y
· LadyBug's heading
! · LadyBug forward by 5
· LadyBug's obtrudes
! · LadyBug move toward dot
! · LadyBug turn by 5
! · LadyBug bounce croak
! · LadyBug wrap
! · LadyBug followPath
! · LadyBug align after dot
tests
· LadyBug's isOverColor 🟦 color
· LadyBug's isUnderMouse
· LadyBug's colorSees
· LadyBug's obtrudes
```

Figure 27. An Etoys simulation that makes the LadyBug follow the track. The user has dragged statements from the LadyBug's viewer (right) into a script (left) so that the LadyBug continually moves forward, turning right when she is over red and left when she is over yellow. The script is currently paused, but if the user pressed the "go" button, the LadyBug would start following the track.

Alice99: Carnegie Mellon University, 1999 [www.alice.org]
The developers of Alice98 (see section 4.1.2.1) noticed that typing was difficult for many users. This system is a follow-on system to Alice98 that focuses on exploring ways to reduce the amount of text users have to type. In Alice98, users create both animations and events by typing statements in a programming language. In Alice99, users create basic animation using drag and drop: the user selects the character of interest from the tree of characters on the left of the screen and drags that character into the animations window. When the user drops the character in the animations window, a series of menus appears showing the actions the character can take, such as move, turn, resize, etc, and the options for each of those choices; a character can move forward, backward, left, right, etc. The drag and drop system in Alice99 does not provide support for many of the traditional programming constructs present in the Alice98 system; to create more complex programs, users must still type. The animation editor can create only fully specified, linear animations. The scripting system was left in place to allow advanced users to build complex worlds. Alice99 also introduced an event editor that allowed users to specify events in a table form in which they selected the event and the animation they wanted to trigger in response to that event.

AutoHAN: A. Blackwell and R. Hague, University of Cambridge, 2001[Blackwell and Hague, 2001]
The AutoHAN project grew out of the desire to provide a single programming interface for the many home appliances that are being shipped with customization or programming features. The goal of the project is to provide a language and interface that home users can use to program their appliances to do simple tasks such as recording a particular TV show, switching on an outside light when the doorbell rings, or starting the coffee pot when the alarm goes off in the morning. This language must be usable by people who can operate remote controls. The AutoHAN project elected to create a variety of physical "media" cubes for this purpose. At their simplest, they operate as single button remote controls that can be associated with a wide variety of appliances. For example, a play cube can be associated with a CD player by holding it close to the CD player. Once the association has been created, the user can press the cube's button to play a CD. The user can later associate that same play cube with a VCR and use it to play a movie. Additionally, the cubes can be composed together to form programs, such as starting the coffee pot when the alarm goes off. These programs can be stored by the AutoHAN system for later use. The designers proposed two languages for the media cubes: one based on ontological abstraction, the other based on linguistic abstraction. The

ontological language includes event cubes which reference changes of state in the home, channel cubes which grant access to different channels of information, and aggregate cubes which allow cubes to be grouped together to form a set (a set of events to react to, for example). The linguistic language includes cubes that are linked to particular words in English, for example, stop, go, and play. Cubes that support more abstract data roles such as variables and lists are also included.

Physical Programming: J. Montemayor, University of Maryland, 2001 [Montemayor, 2002]
The Physical Programming work describes a method for children ages 4-6 to build interactive story spaces using StoryRoom Kits that provide sensors and actuators that can be used to augment everyday objects, such as chairs or teddy bears. The StoryRoom kits allow children to create stories in which objects in the real world represent characters or elements in the story the children are telling. Seeking stories in which one character is asking a series of other characters where to find an object, character, or piece of information work very well in this context. The Physical Programming method was prototyped using Wizard of Oz techniques and the following tools: a foam hand to indicate touch, a light for lighting up objects to draw attention to them, a sound box which had a different sound associated with each side of the box, and a magic wand for users to indicate when they were programming and when they wanted to tell a story using their augmented story room. To create a program, a child associates sensors, actuators, and props using the magic wand. For example, to have the teddy bear say something when it is touched, the child would tap the hand and the teddy bear to indicate that the bear should respond when touched, and one side of the sound box to indicate which sound should be played when the teddy bear is touched. When the wand is put away, the StoryRoom goes into "story" mode and the rules the child created are active.

Flogo: C. Hancock, MIT Media Lab, 2001 [Hancock, 2001]
Flogo is a visual dataflow language designed to enable children to build more complex robotic behaviors with their lego robotics kits. The designers of the system believe that visualizing the temporal structure of a program is helpful in understanding how it works (or why it does not work). The visual dataflow model is well suited to showing the temporal structure of a program. Consequently, Flogo programs use a visual dataflow model. Sensor outputs can be connected in the box and wires style to arithmetic operations, Boolean tests, and motor controls. Flogo programs are always live; a change in the inputs to the sensors will be immediately reflected in the representation of the

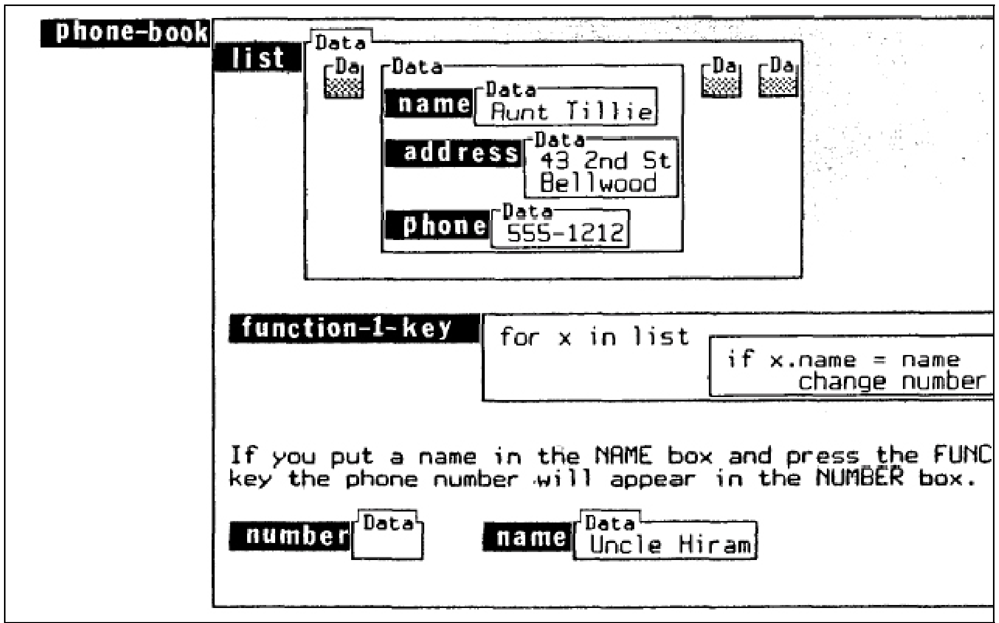
program, making Flogo a tinkering-friendly language even when the program a child is working on is incomplete.

1.1.1.1.6. Integration with Environment

To write a program in most general-purpose languages, a user must type their program into a text editor, compile the program, fix any syntax errors, build the program, and then run it. For a novice programmer, this is a lot of steps and the time and effort involved in making changes to a program can discourage experimentation. The systems in this category integrate the environment in which users write programs with the environment in which users run programs. Many of these systems also allow users to test the effects of individual program statements so that they can experiment while building programs.

Boxer: A. diSessa and H. Abelson, University of California at Berkeley, 1986 [diSessa and Abelson, 1986]

Like Hypercard, Boxer is one of the first environments designed to allow non-expert programmers to program. It presents a hierarchical world composed of boxes that can contain other boxes (see Figure 28). Rather than separating the act of programming, programming is integrated into an environment that a typical person might use, primarily for text editing and graphical layout. Boxer programs contain three types of boxes: standard boxes which can contain text or program code, data boxes which contain string literals for use in programs, and graphics boxes which contain graphical displays. The composition of the boxes has meaning; it indicates that sub-procedures are parts of procedures and records are part of databases. In general, sub-boxes are only accessible from inside a box. The boxes provide the novice programmer with a simple mechanism for abstracting program and data elements. Boxes also allow the novice to view program elements as black boxes that they can use in their programs without fully understanding. As users gain experience, they can return to these black boxes and open them to discover how they work.



name: Aunt Tillie
address: 43 2nd St
Bellwood
phone: 555-1212

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

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

number:

name: Uncle Hiram

Figure 28. A phone number look up program written in Boxer. If a user enters a name in the "name" box and presses the Function-1 key, Boxer will search through the entries in "list", another box shown at the top of the screen, and display the phone number associated with that name.

Hypercard: Bill Atkinson, Apple Computer, 1987 [Goodman, 1987]
Hypercard is described by its creator Bill Atkinson as "an authoring tool and a sort of cassette player for information." The application itself allows users to create stacks of cards, somewhat like a Rolodex program, that contain images, text, and buttons. At their simplest, buttons can trigger visual changes, make sounds, or show a new card. A scripting language called Hypertalk is provided to allow users to build more functionality into the stacks they author. Spoken English heavily influenced the Hypertalk language itself; the language provides constructs such as the first card and the last card, descriptors that are easily understandable to most users. In designing the system, Atkinson concentrated on the user's first experience with the tool. He focused on supporting the user's immediate success using Hypercard and tried to reveal features gradually. A beginning user could learn to create cards and used text-editing tools before moving on to graphics editing. The user could learn about using the message box as a calculator before moving onto placing values in fields. By the time the user was ready to write a full script, he or she would already be familiar with how to access information in different parts of the interface.

cT: B. Sherwood and J. Sherwood, Carnegie Mellon, 1988 [Sherwood and Sherwood, 1988]
This system attempts to simplify the process of creating graphics-oriented programs by providing higher-level primitives. Programs are created in an integrated environment where users can see the results of their programs immediately. The cT environment also provides a method for users to specify shapes using mouse clicks on the screen. Finished programs can be executed as separate programs.

Chart N Art: C. Digiano, University of Colorado, 1996 [Digiano, 1996]
Chart N Art is a graphical editor similar to MacDraw that reveals a programming language. As designers manipulate the interface to create drawings and charts, the equivalent programming statements are printed in a scrolling history area at the bottom. These statements can be copied from the history area into an interaction pane, edited, and executed. The interface provides operations on sets of objects as well as single objects, allowing designers to learn how to specify sets of objects to manipulate using the scripting language. The goal of the interface is to allow designers to automate the creation of custom designed charts, giving them more control than graphing and charting packages, but removing the necessity to draw every aspect of the chart by hand.

4.2 Activities Enhanced by Programming
The systems in this group look at programming as a way to enhance activities, either by allowing greater control or creating opportunities to explore particular domains. Rather than trying to create full general-purpose programming environments, the designers of these systems have tailored the functionality in the programming languages to specific domains.

4.2.1 Entertainment
These systems use programming to support entertaining activities. These systems use programming models inspired by earlier systems to make programming more realizable to novices and provide activities that the designers believe users will find enjoyable.

Bongo: A. Begel, MIT Media Lab, 1997 [Begel, 1997]
Bongo enables children to create their own video games and share them with others through the web. Bongo builds upon Starlogo (see section 4.2.2), and adds primitives for playing sounds, changing shapes, and detecting collisions between characters on the screen; it customizes Starlogo for use in the domain of games programming. High-level movement of objects in the system can be done using drag and drop, but procedures are

created with text-based programming. Bongo supplies a command center that allows users to test out code and observe its results.

Mindrover: Cognitoy, 2001 [Cognitoy, 2001]
Mindrover is a commercial game in which the user is a researcher on Europa, one of the moons of Jupiter. In the researcher's free time, he or she programs robotic rovers to race around hallways and battle other rovers. The game allows users to program their rovers using a drag and drop programming system, inspired by a data-flow visual programming model and The Incredible Machine (see section 3.3.1). Users select pre-built components (such as thrusters and steering wheels) and sensors, place them in a limited number of slots on their rovers, and wire the components and sensors together to give their vehicles certain behaviors. The programming model is similar to the box and wires approach seen in Fabrik, Flogo, and Body Electric. Wires contain information about when signals are sent from sensors to components and the actions triggered by those signals. Boolean gates are provided to allow users to create more complex behaviors.

4.2.2 Education

These systems use programming to allow users to explore different domains of knowledge and how they are affected by different factors. They are intended to allow users to explore and experiment with specific domains of knowledge; the programming languages are tailored for these specific domains.

SOLO: M. Eisenstadt, The Open University, 1983 [Eisenstadt, 1983]
SOLO is a Logo-inspired (see section 4.1.2.1), interpreted textual programming language designed for cognitive psychology modeling. The typical psychology student has little computer experience, no programming experience, occasional access to a computer, and often works on projects in groups. The SOLO language provides psychology students with a simple way to model cognitive processes through accessing and manipulating a simple database of triples. Each triple represents a relationship: for example, "Fido ***isa*** dog". The language provides 10 commands that allow students to store triples, remove triples, test for relationships via pattern matching, define procedures, iterate through triples, and view and edit procedures. Students are able to quickly create simple models of human memory and reasoning, similar to those discussed in introductory psychology classes, and use these programs to reason about how cognition works.

Gravitas: R. Sellman, The Open University, 1992 [Sellman, 1992]
Gravitas is an object-oriented discovery learning environment that allows students to experiment with Newtonian Gravitation. The environment includes both a graphical interface controlled by the mouse and a textual Logo-based (see section 4.1.2.1) programming interface. Students can control the x and y position, x and y velocity, x and y accelerations, and the mass of the spherical objects in the world. Students typically start with the graphical interface to Gravitas, and, as they gain more experience progress to typing Logo commands.

Starlogo: M. Resnick, MIT Media Lab, 1996 [Resnick, 1996]
Starlogo is a programmable modeling environment designed to allow students to explore decentralized systems, such as ant colonies and traffic patterns. Users can write simple rules that control thousands of objects and observe the patterns that arise as a result of these rules. The Starlogo programming language is based on Logo (see section 4.1.2.1). However, instead of controlling a single turtle, users control thousands of turtles. The Starlogo turtles have improved senses: they can detect each other, nearby turtles, and scents in the world. Each pixel in the world has additional capabilities. Rather than containing a single piece of information (color), each pixel is modeled as a turtle that cannot move; it can contain an arbitrary amount of information. Pixels in the world can affect the state of other pixels, causing growth or dispersal of scent, for example.

Hank: Mulholland and Watt, The Open University, 1998 [Mulholland and Watt, 1998]
Hank is a visual programming language designed for cognitive psychology students to use in the construction of cognitive models of human behavior. The typical psychology student has little computer experience, no programming experience, occasional access to a computer, and often works on projects in groups. Consequently, the Hank language was designed with five goals in mind: support the creation of cognitive models; consider the requirements of the non-programmer; support group work; clearly show the execution path; and support paper-based use of the language. Based on findings that spreadsheets tend to allow a number of interested people to understand how the spreadsheet is being developed, Hank is a spreadsheet-based language. The architecture of Hank is similar to the information processing architectures taught to psychology students. There are three components: a database where information can be stored and represented (i.e. long term memory), a workspace where information can be worked upon (i.e. short term memory), and an executive component that carries out processing, input, and output. Data is represented with fact cards that typically represent relationships between entries, similar

to a typical spreadsheet. Programs are expressed on instruction cards using queries for entries on cards and arrows to indicate what to do when entries are found or not. The execution model is explained using a dog named Fido who performs programs according to a few simple rules. The authors designed Fido to be similar to the Logo turtle (see section 4.1.2.1), in the sense that he gives students a physical being to imagine executing their programs, increasing the likelihood that they will be able to accurately simulate their programs on paper. In addition, the environment provides a comic strip representation of the execution of each program; by double clicking on a cell in the comic strip, at student can view the related part of the program.

5. ADDITIONAL SYSTEM INFORMATION

We placed systems in our taxonomy based on the primary problem that particular system was trying to address. However, many of the systems described in this paper have incorporated ideas drawn from earlier systems. In this section, we try to pinpoint some of the most influential systems, identify which approaches to making programming more accessible each system has incorporated, and provide information about which programming constructs are included.

5.1 System Influences

Figure 29 attempts to provide some insight into which systems have most influenced the design of later programming systems for novice programmers using the number of citations. The system with the most citations (from papers referenced by this survey) appears first. Underneath the system name is the list of all references to it.

5.2 System Attributes

The systems presented in this paper vary in a number of dimensions. Figure 30 is intended to allow readers to quickly compare some aspects of the systems discussed in this survey: programming style, supported programming constructs, how programs are constructed and represented, and some of the ways in which the designers of these systems have tried to make programming more accessible to novice programmers. Each system appears in the taxonomy once but many have built on the lessons of systems that have come before. This table attempts to show the major design influences, including those that were not the primary contribution of the system.

Logo 1967

29 — AgentSheets
Alice 98
Bongo
Boxer
Cleogo
Curlybot
Drape
Electronic Blocks
GRAIL
HANDS
Hank
Josef
Kara
Karel
LegoSheets
Leogo
LogoBlocks
Magic Forest
Mindstorms
MOOSE Crossing
Pet Park
Pet Park Blocks
Physical Programming
Playground
Smalltalk
StarLogo
Tangible Programming Bricks
TORTIS
Turingal

Stagecast 1995

13 — AgentSheets
Bongo
Cleogo
Electronic Blocks
HANDS
Kara
Leogo
LogoBlocks
Pet Park Blocks
Physical Programming
Prototype 2
Tangible Programming Bricks
Toontalk

AgentSheets 1991

9 — Bongo
Chemtrains
HANDS
LegoSheets
LogoBlocks
Pet Park Blocks
Prototype 2
Stagecast
Tangible Programming Bricks

Smalltalk 1971

7 — Alice 98
Alternate Reality Kit
Blue
HANDS
GRAIL
MOOSE Crossing
Playground

Karel 1981

6 — GNOME
GRAIL
HANDS
Kara
MacGNOME
Turingal

LogoBlocks 1996

6 — Bongo
Flogo
Mindstorms
Pet Park Blocks
Physical Programming
Tangible Programming Bricks

Toontalk 1996

6 — Cleogo
Electronic Blocks
HANDS
Kara
Leogo
Tangible Programming Bricks

Algoblock 1995

5 — AutoHAN
Cleogo
Leogo
Tangible Programming Bricks
Toontalk

Pascal 1970

5 — GRAIL
Karel
SP/k
Turing
Turingal

Programming by Rehearsal 1984

5 — Alternate Reality Kit
Fabrik
HANDS
Liveworld
Prototype 2

Pygmalion 1975

5 — Leogo
Physical Programming
Prototype 2
Toontalk
TORTIS

Alternate Reality Kit 1987

4 — Alice 98
Liveworld
Playground
Prototype 2

Boxer 1986

4 — HANDS
Liveworld
MOOSE Crossing
Prototype 2

Hypercard 1987

4 — GRAIL
HANDS
Leogo
MOOSE Crossing

MOOSE Crossing 1997

3 — Bongo
Pet Park
Pet Park Blocks

BASIC 1961

3 — Atari 2600 basic
GRAIL
MOOSE Crossing

Blue 1996

3 — Blue environment
GRAIL
JJ

Figure 29 System Influences.

		Simplify the language						Prevent syntax errors		Construct Programs Using Objects										Create Programs Using Actions				Provide Multiple Methods for	New Programming Models			
		1963 BASIC	1977 SP/k	1988 Turing	1996 Blue	1998 JJ	2001 GRAIL	1984 GNOME	1986 MacGnome	1986 Play	1990 Show and Tell	1995 My Make Believe Castle	1995 Thinkin' Things Collection 3: Half Time	1996 LogoBlocks	1998 Pet Park Blocks	2000 Drape	2000 Electronic Blocks	2002 Alice2	2002 Magic Forrest	1976 TORTIS	1989 Roamer	1995 LegoSheets	2000 Curlybot	1997 Leego	1970 Pascal	1971 Smalltalk	1989 Playground	2001 Kara
Style of Programming	procedural	x	x	x		x	x				x		x	x		x			x	x	x		x	x	x			
	functional																											
	object-based																	x									x	
	object-oriented				x	x									x											x		
	event-based											x		x	x		x		x			x				x	x	
	state-machine based																											x
Programming Constructs	conditional	x	x	x	x	x	x							x	x		x	x		x	x	x		x	x	x	x	x
	count loop						x						x	x		x		x			x							
	for loops	x	x	x		x																		x	x	x		
	while loops	x	x	x	x	x												x							x	x		
	variables	x	x	x	x	x	x				x							x							x	x	x	
	parameters	x	x	x	x	x	x				x							x						x	x	x		
	procedures/methods	x	x	x	x	x	x				x					x		x			x			x	x	x	x	x
	user-defined data types	x	x	x	x		x																		x	x		
	pre and post conditions				x	x																						
Representation of Code	text	x	x	x	x	x	x	x				x		x	x			x		x	x	x		x	x	x	x	
	pictures										x	x	x		x	x			x		x	x						x
	flow chart										x																	
	animation																						x					
	forms																							x				
	finite state machine																											x
	physical objects																x			x								
Construction of Programs	typing code	x	x	x	x	x	x	x	x												x			x	x	x	x	
	assembling graphical objects										x		x	x		x		x	x									x
	positioning graphical objects																											
	demonstrating actions																			x			x	x				
	selecting/form filling											x										x		x				
	physical manipulation																x			x								
Support to Understand Programs	back stories																											
	debugging																											
	physical metaphor												x			x			x	x	x							x
	liveness										x									x		x						
	generated examples																							x				
Preventing Syntax Errors	physical shape affordance												x	x	x		x	x	x									
	selection from valid options																	x						x				
	syntax directed editing							x	x																			
	dropping only in valid location													x				x	x									
	better syntax error messages					x																						
Designing Accessible Languages	limit the domain										x	x	x	x	x	x	x		x	x	x	x	x	x			x	x
	select user-centered keywords																	x						x			x	
	remove unnecessary punctuation			x		x												x										
	use natural language																										x	
	remove redundancy		x		x	x	x																					
Support Commu-nication	side by side																x			x	x		x					
	networked- shared manipulation																											
	networked - shared results														x													
Choice of Task	fun & motivating										x	x	x	x	x	x	x	x	x	x	x	x	x				x	
	useful	x	x																						x	x		
	educational	x	x	x	x	x	x	x	x		x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x

Figure 30: System Attributes

		Making New Programming Models Accessible					Tracking Execution	Make Programming Concrete			Models of Program Execution		Side by Side		Networked			Solving Problems by Positioning Objects				Solve Problems Using Code		Demonstrate Actions in the Interface			Demonstrate Conditions and Actions		
		1994 Liveworld	1996 Blue Environment	1997 Karel++	2001 Karel J Robot	2001 J Karel	1979 Atari 2600 BASIC	1981 Karel	1983 Josef the Robot	1991 Turingal	1996 ToonTalk	1998 Prototype 2	1995 AlgoBlock	2000 Tangible Programming Bricks	1997 MOOSE Crossing	1998 Pet Park	1998 Cleogo	1982 Rocky's Boots	1982 Robot Odyssey	1993 The Incredible Machine	1995 Widget Workshop	1995 AlgoArena	2001 Robocode	1975 Pygmalion	1984 Programing by Rehersal	1992 Mondrian	1991 AgentSheets	1993 ChemTrains	1995 Stagecast
Style of Programming	procedural						x	x	x	x	x	x	x	x			x	x	x	x	x	x		x	x	x			
	functional																												
	object-based																												
	object-oriented	x	x	x	x	x									x	x							x						
	event-based	x													x	x					x				x		x	x	x
	state-machine based																												
Programming Constructs	conditional	x		x	x	x	x	x	x	x	x	x			x	x	x	x	x	x		x	x	x			x	x	x
	count loop			x	x			x	x					x		x						x							
	for loops	x				x	x			x					x		x						x	x					
	while loops	x		x	x	x	x	x	x	x		x			x							x	x	x					
	variables	x				x	x		x		x			x	x	x						x	x	x					
	parameters	x				x	x		x		x	x		x	x		x					x	x	x					
	procedures/methods	x		x	x	x	x	x	x	x	x	x			x		x					x	x	x	x	x	x	x	x
	user-defined data types	x				x	x																x						
	pre and post conditions																												
Representation of Code	text	x	x	x	x	x		x	x	x		x			x	x	x					x	x		x				
	pictures		x															x	x	x	x			x		x	x	x	x
	flow chart																	x	x	x	x			x					
	animation										x																		
	forms																x								x				
	finite state machine																												
	physical objects												x	x															
Construction of Programs	typing code	x	x	x	x	x		x	x	x					x	x	x					x	x						
	assembling graphical objects																				x			x		x			
	positioning graphical objects																	x	x	x							x	x	x
	demonstrating actions										x	x					x							x	x		x	x	x
	selecting/form filling											x					x					x							
	physical manipulation												x	x															
Support to Understand Programs	back stories			x	x	x		x	x		x	x																	
	debugging																												x
	physical metaphor							x	x		x		x							x	x	x							
	liveness		x																					x					
	generated examples																x												
Preventing Syntax Errors	physical shape affordance																	x	x	x									
	selection from valid options																x												
	syntax directed editing																												
	dropping only in valid location																												
	better syntax error messages																												
Designing Accessible Languages	limit the domain			x	x	x		x		x			x				x				x	x							
	select user-centered keywords			x	x	x		x									x												
	remove unnecessary punctuation																												
	use natural language																									x			
	remove redundancy																												
Support Commu-nication	side by side												x																
	networked- shared manipulation																x												
	networked - shared results														x	x													
Choice of Task	fun & motivating										x		x		x	x		x	x	x	x	x	x						x
	useful	x	x											x										x	x	x			
	educational	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x				x	x	x

Figure 30: System Attributes

		Specify Actions			Make the Language More Undestandable				Improve Interaction with Language							Integration with Environment					Entertainment		Education			
		1983 Pinball Construction Set	1987 Alternate Reality Kit	1994 Klik N Play	1960 COBOL	1967 Logo	1997 Alice 98	2001 HANDS	1985 Body Electric	1988 Fabrik	1996 Tangible Programming with Trains	1997 Squeak e-Toys	1998 Alice 99	2001 AutoHAN	2001 Physical Programming	2001 Flogo	1986 Boxer	1987 Hypercard	1988 cT	1996 Chart N Art	1997 Bongo	2001 Mindrover	1983 SOLO	1992 Gravitas	1996 Starlogo	1998 Hank
Style of Programming	procedural				x	x			x	x						x	x	x	x	x		x	x		x	x
	functional																									
	object-based							x				x	x													
	object-oriented						x														x			x		
	event-based	x	x	x			x	x	x	x	x	x	x	x	x	x	x	x			x	x			x	
	state-machine based																									
Programming Constructs	conditional	x		x	x	x	x	x		x		x				x	x	x	x	x	x	x	x	x	x	x
	count loop																	x		x				x		
	for loops				x	x	x	x									x	x	x	x	x				x	
	while loops				x	x	x										x	x		x	x			x	x	
	variables			x	x	x	x			x		x					x	x	x	x	x		x	x	x	x
	parameters		x		x	x	x			x				x			x	x	x	x	x		x	x	x	x
	procedures/methods				x	x	x	x		x		x		x			x	x	x	x	x		x	x	x	x
	user-defined data types				x		x	x																		
	pre and post conditions																									
Representation of Code	text		x		x	x	x	x				x	x				x	x	x	x	x		x	x	x	
	pictures	x	x																			x				
	flow chart								x	x						x										x
	animation																									
	forms			x																				x		
	finite state machine																									
	physical objects										x			x	x											
Construction of Programs	typing code				x	x	x	x									x	x	x	x	x		x	x	x	x
	assembling graphical objects								x	x		x	x			x						x				x
	positioning graphical objects	x	x																							
	demonstrating actions																									
	selecting/form filling			x				x					x						x							x
	physical manipulation										x			x	x											
Support to Understand Programs	back stories							x																		
	debugging																									
	physical metaphor	x	x			x	x	x				x												x		
	liveness		x					x	x	x					x	x	x						x			
	generated examples																			x			x			
Preventing Syntax Errors	physical shape affordance													x												
	selection from valid options			x								x	x													
	syntax directed editing							x															x			
	dropping only in valid location		x									x														
	better syntax error messages																									
Designing Accessible Languages	limit the domain	x	x	x		x						x	x	x									x	x		
	select user-centered keywords						x	x				x														
	remove unnecessary punctuation					x																				
	use natural language				x			x				x														
	remove redundancy							x																		
Support Commu-nication	side by side													x	x											
	networked- shared manipulation																									
	networked - shared results																									
Choice of Task	fun & motivating	x	x	x		x	x	x				x	x		x						x	x				
	useful				x		x	x	x	x	x		x	x	x		x	x	x	x	x	x	x		x	x
	educational	x	x	x		x	x					x			x	x							x	x	x	x

Figure 30: System Attributes

6. SUMMARY AND FUTURE DIRECTIONS

The systems presented in this paper have tried to make programming accessible in three main ways: simplifying the mechanics of programming, providing support for learners, and providing students with motivation to learn to program. The majority of the systems have focused on the mechanics of programming. Clearly, beginners need to feel that they can make progress in learning to program. However, pure difficulty is not the only reason that people hesitate to learn to program. There are a variety of sociological factors (including students not seeing the relevance of programming or perceiving computer science as being a socially isolating career path) that can prevent people from learning to program. Creating environments that address some of these sociological barriers to programming by supporting learners or providing interesting reasons to program have the potential to attract a more diverse group of people to computer science. If the population of people creating software is more closely matched to the population using software, the software designed and released will probably better match users' needs.

6.1 Mechanical Barriers to Programming

Most of the programming systems built for children and novice adults have focused on making the mechanics of programming more manageable. Systems have removed unnecessary syntax, designed languages that are closer to spoken English, introduced programming in visible contexts (such as the Logo turtle) in which students can see the immediate results of their commands, and explored alternatives to typing programs. Using these ideas, it is possible to create a system that will allow a wider audience of people to begin programming. While these systems do not take all of the challenges out of programming, they can allow students to focus on the logic and structures involved in programming rather than worrying as much about the mechanics of writing programs. However, even with these improvements to a beginner's first programming experience, there are a number of questions that remain.

Many of the teaching languages have been heavily influenced by the prevalent general-purpose languages of their time. Designers of these systems chose to make the programming constructs and syntax very similar to those of the general-purpose languages to ease the transition from teaching languages to general-purpose languages. While it seems obvious that students need to understand the parallels between the programming constructs in teaching and general-purpose languages, it is not clear how closely and in what ways teaching languages must resemble general-purpose languages.

We can now more easily introduce beginners to programming; perhaps it is time to begin studying the intermediate programmer, someone who has been introduced to programming through a system designed for beginners and wants to apply that experience to learning a general language. What are the hardest aspects of that transition and how are those aspects affected by the teaching system? What are the trade-offs between presenting issues of syntax and program expression earlier or later in the process?

6.2 Sociological Barriers to Programming

In some ways, sociological barriers can be harder to address than mechanical ones because they are harder to identify and some cannot be addressed through programming systems. However, by studying particular groups of people who choose not to learn to program, identifying the reasons behind their decisions, and trying to address those reasons in our programming systems and textbooks, we may be able to attract a broader audience of people to programming and Computer Science. The systems in the taxonomy have identified and are beginning to address two kinds of sociological barriers to programming: the lack of a social context for programming and the lack of compelling contexts in which to learn programming.

6.2.1 Social Support

It can be easier and more fun to learn with a group of people. MOOSE Crossing and, later, Pet Park added support for social interaction so that students using these systems can share projects, provide examples for each other, and chat. Future communities might provide support for students helping each other learn the interface and programming constructs, support students working on projects together, or try to capture and strengthen the positive feedback that members of the community give to each other through looking at and using each other's work.

6.2.2 Reasons to Program

Several systems have tried to provide motivating contexts such as building robots, fighting battles, and constructing machines in which to learn programming. While these systems have been very effective for a segment of the population, they do not have broad appeal. What programming activities can we provide that will interest girls or artistic or musical students? Future systems might provide contexts for programming that are relevant to under-represented groups in computer science.

7. ACKNOWLEDGEMENTS

We would like to thank the members of the Stage 3 Research group for their support and assistance, the many who provided suggestions for systems that should be included, and, especially, Peter Scupelli who spent more than a few hours trying to make our diagrams interpretable. Caitlin Kelleher was supported by an NSF graduate fellowship.

8. REFERENCES

1. ATKINSON, B. 1987. Hypercard, Apple Computer.
2. BECKER, B. 2001 *Robots: Learning to Program with Java.*
3. BEGEL, A. 1996. LogoBlocks: A Graphical Programming Language for Interacting with the World. Electrical Engineering and Computer Science Department, MIT, Boston, MA.
4. BEGEL, A. 1997. Bongo: A Kids' Programming Environment for Creating Video Games on the Web. Electrical Engineering and Computer Science Department, MIT, Boston, MA.
5. BELL, B. AND LEWIS, C. 1993. ChemTrains: A Language for Creating Behaving Pictures. *Visual Languages.*
6. BERGIN, J., STEHLIK, M., ROBERTS, J., AND PATTIS, R., 2001. *Karel J. Robot: A Gentle Introduction to the Art of Object-Oriented Programming in Java.* Available on the web at http://csis.pace.edu/~bergin/KarelJava2ed/Karel++JavaEdition.html.
7. BERGIN, J., STEHLIK, M., ROBERTS, J. AND PATTIS, R. 1997. *Karel++: A Gentle Introduction to the Art of Object-Oriented Programming*, John Wiley and Sons, New Jersey.
8. BLACKWELL, A. F. AND HAGUE, R. 2001. AutoHAN: An Architecture for Programming the Home. In *Proceedings of IEEE Symposia on Human-Centric Computing Languages and Environments 2001*, Stresa, Italy, IEEE Computer Society.
9. BLANCHARD, C., BURGESS, S., ET AL. 1990. Reality Built For Two: A Virtual Reality Tool. In *Proceedings of the Symposium on Interactive 3D Graphics*.
10. BRUCKMAN, A. 1997. MOOSE Crossing: Construction Community, and Learning in a Networked Virtual World for Kids. MIT Media Lab, Boston, MA, 1997.
11. BRUSILOVSKY, P. 1991. Turingal - the language for teaching the principles of programming. In *Proceedings of Third European Logo Conference 1991*, Parma, Italy.
12. BRUSILOVSKY, P., CALABRESE, E., ET AL. 1997. Mini-Languages: A Way to Learn Programming Principles. *Education and Information Technologies* 2,1: 65-83.
13. BUDGE, B. 1983. Pinball Construction Set, Exidy Software.
14. CATLIN, D. 1989. Roamer, Valiant Technologies.
15. CHENG, A. 1998. A Graphical Programming Interface for a Children's Constructionist Learning Environment. Electrical Engineering and Computer Science Department, MIT Boston, MA.
16. COCKBURN, A. AND BRYANT, A. 1997. Leogo: An Equal Opportunity User Interface for Programming. *Journal of Visual Languages and Computing* 8,5-6: 601-619.
17. COCKBURN, A. AND BRYANT, A. 1998. Cleogo: collaborative and multi-metaphor programming for kids. In *Proceedings of the 3rd Asia Pacific Conference on Computer Human Interaction* .
18. COGNITOY, 2001. *Mindrover.*
19. CONWAY, M. 1997. Alice: Easy-to-Learn 3D Scripting for Novices. School of Engineering and Applied Science, University of Virginia. Charlottesville, VA.
20. DEBONTE, A. M. 1998. Pet Park: A Virtual Learning World for Kids. Electrical Engineering and Computer Science. MIT, Boston, MA.
21. DIGIANO, C. 1996. Self-Disclosing Design Tools: An Incremental Approach Toward End-User Programming. Department of Computer Science. University of Colorado at Boulder, Boulder, Colorado.
22. DIJKSTRA, E.W. 1969. Structured Programming. In *Software Engineering Techniques: Report on a conference sponsored by the NATO Science Committee*, Rome, Italy.
23. DISESSA, A. AND ABELSON, H. 1986. Boxer: A Reconstructible Computational Medium In *Communications of the ACM* 29,9: 859-868.
24. EDMARK CORPORATION, 1995. *Thinkin' Things 3.*
25. EISENSTADT, M. 1983. A User-Friendly Software Environment for the Novice Programmer. In *Communications of the ACM* 26,12 : 1058-1063.
26. FENTON, J. AND BECK, K. 1989. Playground: an object-oriented simulation system with agent rules for children of all ages. In *Proceedings on Object-oriented Programming Systems, Languages and Aapplications.*

27. FINZER, W. AND GOULD, L. 1984. Programming by Rehearsal. In *Xerox Palo Alto Research Center Research Report*. Palo Alto, CA.
28. FREI, P., SU, V., MIKHAK, B. AND ISHII, H. 2000. Curlybot: Designing a New Class of Computational Toys. In *Proceedings of CHI 2000*.
29. GILLIGAN, D. 1998. An Exploration of Programming by Demonstration in the Domain of Novice Programming. Department of Computer Science, Victoria University, Wellington, Victoria**:** 176.
30. GINDLING, J., IOANNIDOU, A., ET AL. 1995. LEGOsheets: A Rule-Based Programming, Simulation and Manipulation Environment for the LEGO Programming Brick. In *Proceedings of Visual Languages 1995*, Darmstadt, Germany.
31. GOODMAN, D. 1987. *The Complete HyperCard Handbook.* Birmingham, Bantam Computer Books.
32. HANCOCK, C. 2001. Children's Understanding of Process in the Construction of Robot Behaviors. In *Proceedings of Symposium on Varieties of Programming Experiences*.
33. HARTMANN, W., NIEVERGELT, J., AND REICHERT, R. 2001. Kara, finite state machines, and the case for programming as part of general education. In *Proceedings of Symposia on Human Centric Computing 2001*, Stresa, Italy.
34. HOLT, R. C., WORTMAN, D.B., BARNARD, D.T., AND CORDY, J.R. 1977. SP/k: A System for Teaching Computer Programming. In *Communications of the ACM* 20**,**5: 301-309.
35. HOLT, R. AND CORDY, J. 1988. The Turing Programming Language. In *Communications of the ACM* 31**,**12: 1410-1423.
36. INGALLS, D., WALLACE, S. ET AL. 1988. Fabrik: A Visual Programming Environment. In *Proceedings of Object Oriented Programming Systems, Languages, and Applications 1988*.
37. KAHN, K. 1996. Drawings on napkins, video-game animation, and other ways to program computers. In *Communications of the ACM* 43**,**3: 104-106.
38. KATO, H. AND IDE, A. 1995. Using a Game for Social Setting in a Learning Environment: AlgoArena -- A Tool for Learning Software Design. In *Proceedings of Computer Supported Collaborative Learning 1995*.
39. KAY, A. 1993. The Early History of Smalltalk. *ACM SIG PLAN Notes* 28**,**3: 69-96.
40. KAY, A. Etoys and Simstories in Squeak. Available at http://www.squeakland.org/author/etoys.html
41. KIMURA, T., CHOI, J. AND MACK, J. 1990. Show and Tell: A Visual Programming Language. In *Visual Programming Environments: Paradigms and Systems*, E.P. Glinert, ed, IEEE Computer Science Press: 397-404.
42. KOLLING, M. AND ROSENBERG, J. 1996. Blue - A language for teaching object-oriented programming, In *Proceedings of Computer Science Education 1996*, Philadelphia, PA.
43. KOLLING, M. AND ROSENBERG, J. 1996. An Object-Oriented Program Development Environment for the First Programming Course, In *Proceedings of Computer Science Education 1996*, Philadelphia, PA.
44. KURTZ, T. 1981. BASIC. In *History of Programming Languages.* R. Wexelblat, ed. New York, Academic Press: 515-537.
45. LEGO. Lego Mindstorms robotics invention system. Available at http://mindstorms.lego.com
46. LIEBERMAN, H. 1993. Mondrian: A Teachable Graphical Editor. In *Watch What I Do: Programming by Demonstration*. A. Cypher, ed. MIT Press, Cambridge, MA.
47. LOGO COMPUTER SYSTEMS INC. 1995. *My Make Believe Castle*.
48. LIONET, F. AND LAMOUREUX, Y. 1994. Klik and Play, Maxis.
49. MARTIN, F., COLOBONG, G.L., AND RESNICK, M. 1999. Tangible Programming with Trains. Available at http://el.www.media.mit.edu/projects/trains.
50. MAXIS, 1995. *Widget Workshop.*
51. MCIVER, L. K. 1999. Grail: A Zeroth Programming Language. In *Proceedings of Conference on Computers in Education*.
52. MCIVER, L. K. 2001. Syntactic and Semantic Issues in Introductory Programming Education. School of Computer Science and Software Engineering, Monash University**:** 200.
53. MCNERNEY, T. S. 2000. Tangible Programming Bricks: An Approach to making programming accessible to everyone. Media Lab, MIT. Cambridge, MA**:** 86.
54. MILLER, P., PANE, J., METER, G., AND VORTHMANN, S. 1994. Evolution of Novice Programming Environments: The Structure Editors of Carnegie Mellon University. In *Interactive Learning Environments* 4**,**2: 140-158.
55. MINSKY, M. 1986. *The society of mind.* Simon and Schuster, New York.
56. MIT MEDIA LAB, *Programmable Bricks Documentation*, on the web at http://llk.media.mit.edu/projects/cricket/doc/index.shtml,
57. MONTEMAYOR, J. 2001. Physical Programming: Software You Can Touch. In *Proceedings of Conference on Human Factors in Computing Systems 2001*, Seattle, WA.
58. MOTIL, J. AND EPSTEIN, D. 1998. JJ: a Language Designed for Beginners (Less Is More). Available at http://www.publicstaticvoidmain.com.
59. MULHOLLAND, P. AND WATT, S. 1998. Hank: A Friendly Cognitive Modelling Language for Psychology Students. In *Proceedings of IEEE Symposium on Visual Languages*, Nova Scotia.

60. NELSON, M. 2001. Robocode. IBM Advanced Technologies. Available at robocode.alphaworks.ibm.com/home/home.html
61. OVERMARS, M. Drape - Drawing Programming Environment. Available at http://www.cs.uu.nl/people/markov/kids/.
62. PANE, J.F. 2002. A Programming System for Children that is Designed for Usability Computer Science Department, Carnegie Mellon University. Pittsburgh, PA: CMU-CS-02-127.
63. PAPERT, S. 1980. *Mindstorms: Children, Computers, and Powerful Ideas*. New York, Basic Books.
64. PATTIS, R. E. 1981. *Karel the Robot: A Gentle Introduction to the Art of Programming with Pascal*. New York, John Wiley and Sons.
65. REPENNING, A. 1993. Agentsheets: a tool for building domain-oriented visual programming environments. In *Proceedings of the conference on Human factors in computing systems*: 142-143.
66. REPENNING, A., A. IOANNIDOU, AND J. ZOLA. 2000. AgentSheets: End-User Programmable Simulations. *Journal of Artificial Societies and Social Simulation* 3,3.
67. RESNICK, M. 1996. StarLogo: an environment for decentralized modeling and decentralized thinking. In *Proceedings of the 1996 Conference Companion on Human Factors in Computing Systems*.
68. ROBINETT, W. AND GRIMM, L. 1982. Rocky's Boots / Robot Odyssey, The Learning Company.
69. ROBINETT, W. 1979. Atari 2600 Basic Cartridge. Atari Company.
70. SAMMET, J. 1981. The Early History of Cobol. *In History of Programming Languages*. R. Wexelblat, ed. New York, Academic Press.
71. SELLMAN, R. 1992. Gravitas: An object-oriented discovery learning environment for Newtonian gravitation. In *Proceedings of East-West International Conference on Human-Computer Interaction*: 31-41.
72. SHERWOOD, B. A. 1988. *The cT Language*. Champaigne, IL, Stipes Publishing Company.
73. SMITH, R. 1987. Experiences with the Alternate Reality Kit: An Example of the Tension Between Literalism and Magic. In *Proceedings of Human Factors in Computing Systems*: 61-67.
74. SIERRA GAMES. 1993. *The Incredible Machine*
75. SMITH, D. C. 1993. Pygmalion. In *Watch What I Do: Programming by Demonstration*. A. Cypher, ed. MIT Press, Cambridge, MA.
76. SMITH, D. C., CYPHER, A. AND SPOHRER, J. 1997. KidSim programming agents without a programming language. In *Proceedings of the 6th European conference on Software engineering*.
77. SUZUKI, H. AND KATO, H. 1995. Interaction-Level Support for Collaborative Learning: AlgoBlock -- An Open Programming Language. In *Proceedings of Computer Supported Collaborative Learning.*
78. TANIMOTO, S. AND RUNYAN, M. 1986. Play: An Iconic Programming System for Children. In *Visual Languages*. S. K. Chang, T. Ichikawa and P. A. Ligomenides, Plenum Publishing Corporation**:** 191-205 1986.
79. TEITELBAUM, T. AND REPS, T. 1981. The Cornell Program Synthesizer: A syntax-directed programming environment. In *Communications of the ACM*: 24, 9: 563-573.
80. TOMEK, I. 1983. *The First Book of Josef: an introduction to computer programming*. Englewood Cliffs, New Jersey, Prentice-Hall.
81. TRAVERS, M. 1994. Recursive interfaces for reactive objects. In of *Proceedings on Human Factors in Computing Systems 1994*.
82. WIRTH, N. 1993. Recollections about the Development of Pascal. In *ACM SIGPLAN Notices* 28**,**3: 333-342.
83. WYETH, P. AND PURCHASE, H.C. 2000. Programming without a computer: a new interface for children under eight. In *Proceedings of First Australasian User Interface Conference 2000*.