

Natural Programming

NATURAL PROGRAMMING

- Want to find out how people "naturally" specify software.

PAC-MAN (JOHN PANE AND BRAD MYERS)

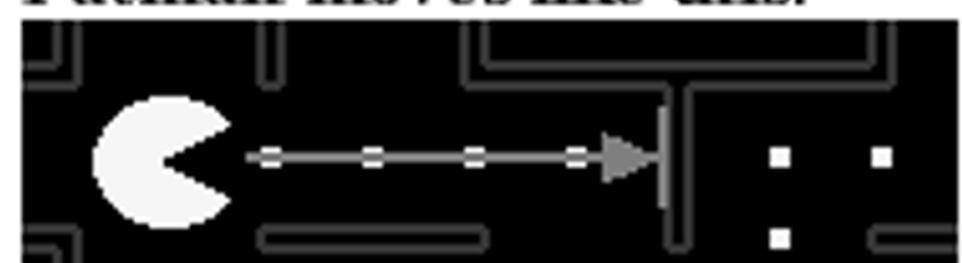
Usually Pacman moves like this.



Now let's say we add a wall.



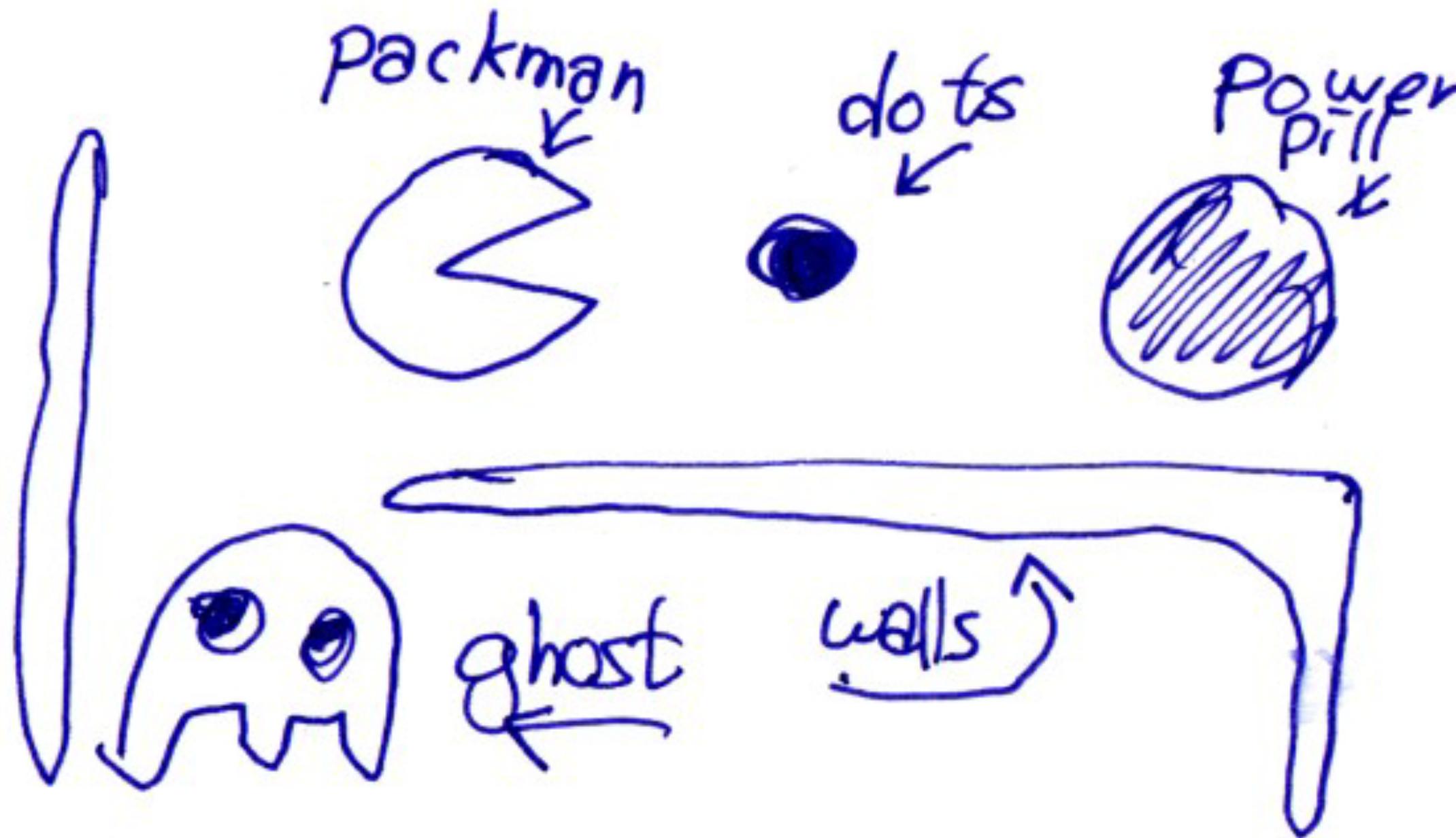
Pacman moves like this.



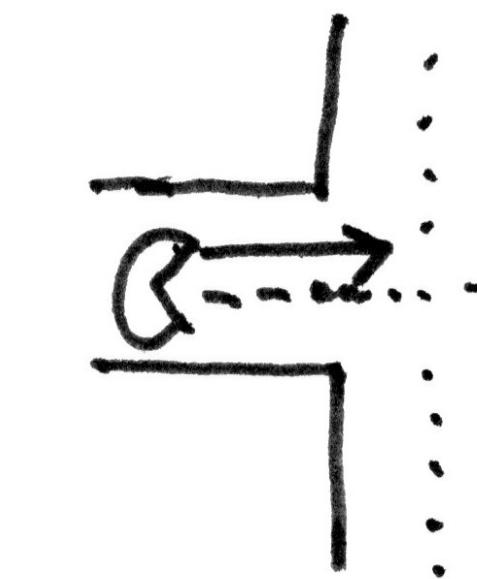
Not like this.



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



When "k" is hit, packman goes down



when he hits the wall ^{PacMan will} stop until the player hits another key.

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

When the left arrow key is pressed Pac man moves left.

(1004)

When the up arrow key is pressed Pac man moves up.

" " down " " " " " " down

" " right " " " " " " right

When he goes into a wall he backs up and goes a different way.

When Pacman hits the ghost or the monster, he ~~has~~ lose a life and start again.

When Pac man goes into a ghost he will lose a life. When Pac man eats a special dot he is able to eat the ghost or monster. There is about 30 seconds when they can be eaten.

When the dot is eaten, the ghost and monsters turn a blue color and the player gets more points. Usually it will go up 50 point.

The fruit will appear every minute and disappear in 30 seconds.

After Pacman eats the last dot, a new level ~~starts~~ starts. The next level is harder, and the dots appear as the last dot is eaten.

When a person has a high score, everybody moves a place down ~~and~~ except for the people above it. The last person then doesn't have a place. Example: The new person got in 5th place, the people from 5th - 10th move down a place. The people from 1-4 stay in their place.

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

RANDOMNESS

3.3.9.5 Randomness

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

Prevalance: 1.4 occurrences per participant.

- 47% - using precision, where no element of uncertainty is expressed.

Example: Put the new fruit in every 30 seconds.

- 20% - using words other than *random* to express the uncertainty.

Example: The fruit will go away after a while.

- 18% - using precision with hedging to express uncertainty.

Example: After around 3 or 4 more seconds the fruit disappears.

- 15% - other (often the action was tied to another event).

Example: Put a fruit on the screen when PacMan is running out of power.

- 0% - used the word *random*.

EXAMPLE

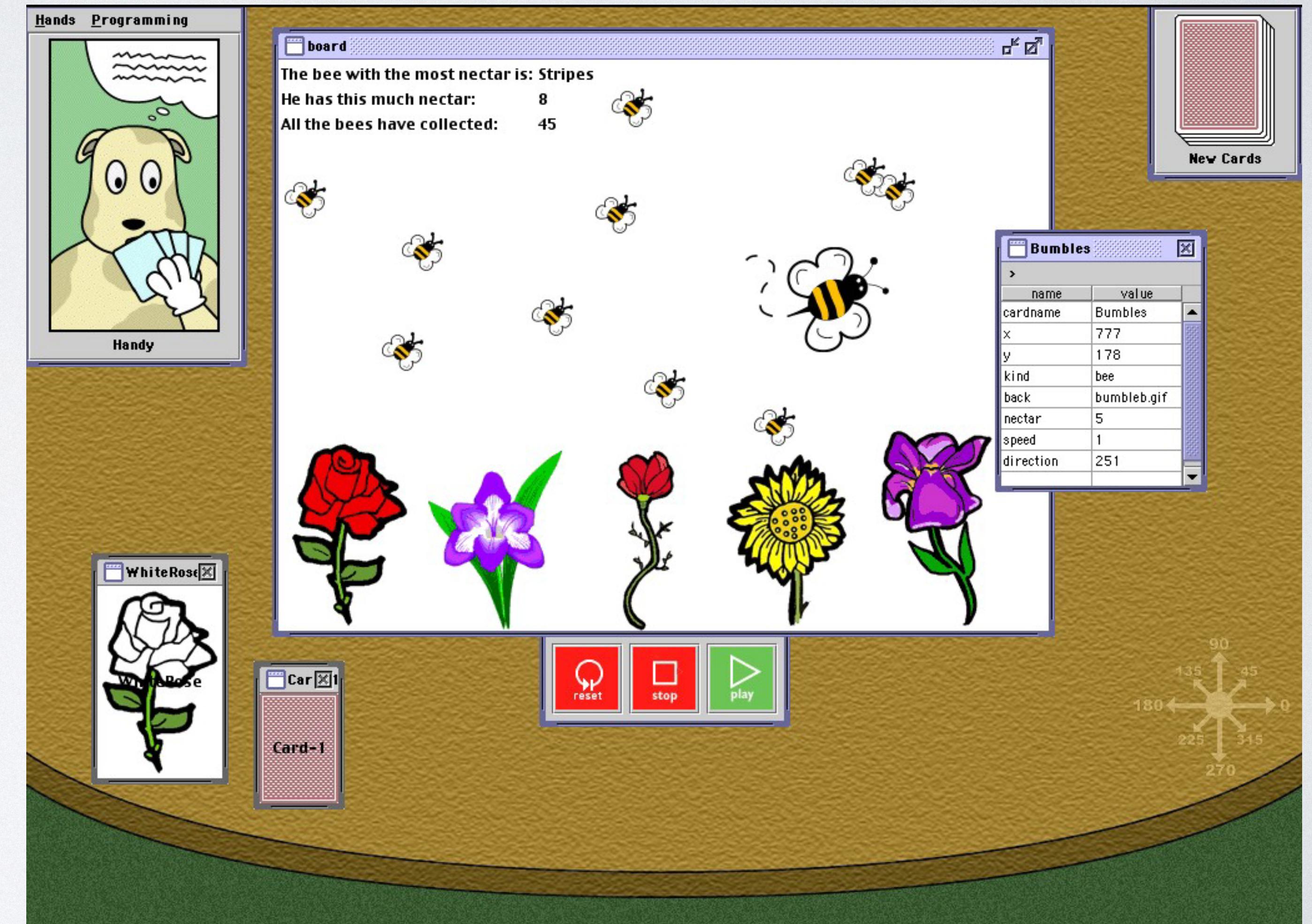
- Suppose we want to design a system that allows users to create and participate in lotteries. The rules of the lottery are as follows:
 - Someone creates a lottery with a secret "winning number" between 0 and 100.
 - Anyone can purchase a lottery ticket for a set amount of money. A lottery ticket also has a number (picked by the buyer) between 0 and 100. If the lottery ticket's number is equal to the lottery's number, that is a winning ticket.
 - The buyer of a ticket can check whether a ticket is the winning ticket and redeem a winning lottery ticket for a set amount of money. The buyer of a ticket can redeem the ticket at any point after they buy it.
 - It is only possible to redeem a lottery ticket from the lottery where the ticket was purchased. For instance, if you buy a lottery ticket from lottery1, you cannot redeem your ticket from lottery2, even if that ticket's number matches the winning number of lottery2.

EXAMPLE (CONT.)

- Design, using pseudocode, a program to handle this lottery system. Do not worry about writing code that resembles any particular language, and feel free to make up any language features you may want. We want to see the kind of code you would want to be able to write. The goal here is to see how people naturally go about solving a problem like this.
- It is critical to the system that a lottery ticket can be redeemed only from the lottery where it was purchased. You can assume that creator and players of the lottery have accounts of some sort from which money can be withdrawn and deposited.

HANDS

- The HANDS system portrays the components of a program on a round table. All data is stored on cards, which can be drawn from the pile at the top right and dragged into position. At the lower left, two cards are shown face-down on the table. One has a generic card back and the other has been given a picture by the programmer. In the center of the table is a board, where the cards are displayed in a special way where only the contents of the back are displayed. Each picture, string, and number on the board is a card. At the right, one of the cards has been flipped face-up, where its properties can be viewed and edited. The programmer inserts code into Handy's thought bubble, by clicking on Handy's picture in the upper left corner. When the play button is pressed, Handy begins responding to events by manipulating cards according to the instructions in the thought bubble. The stop button halts the program, and the reset button will restore all cards to their state at the time the play button was last pressed. For reference, a compass is embossed on the table at the lower right.
- Credit: J.F. Pane, B.A. Myers, and L.B. Miller, "Using HCI Techniques to Design a More Usable Programming System," Proceedings of IEEE 2002 Symposia on Human Centric Computing Languages and Environments (HCC 2002), Arlington, VA, September 3-6, 2002, pp. 198-206.



HANDS

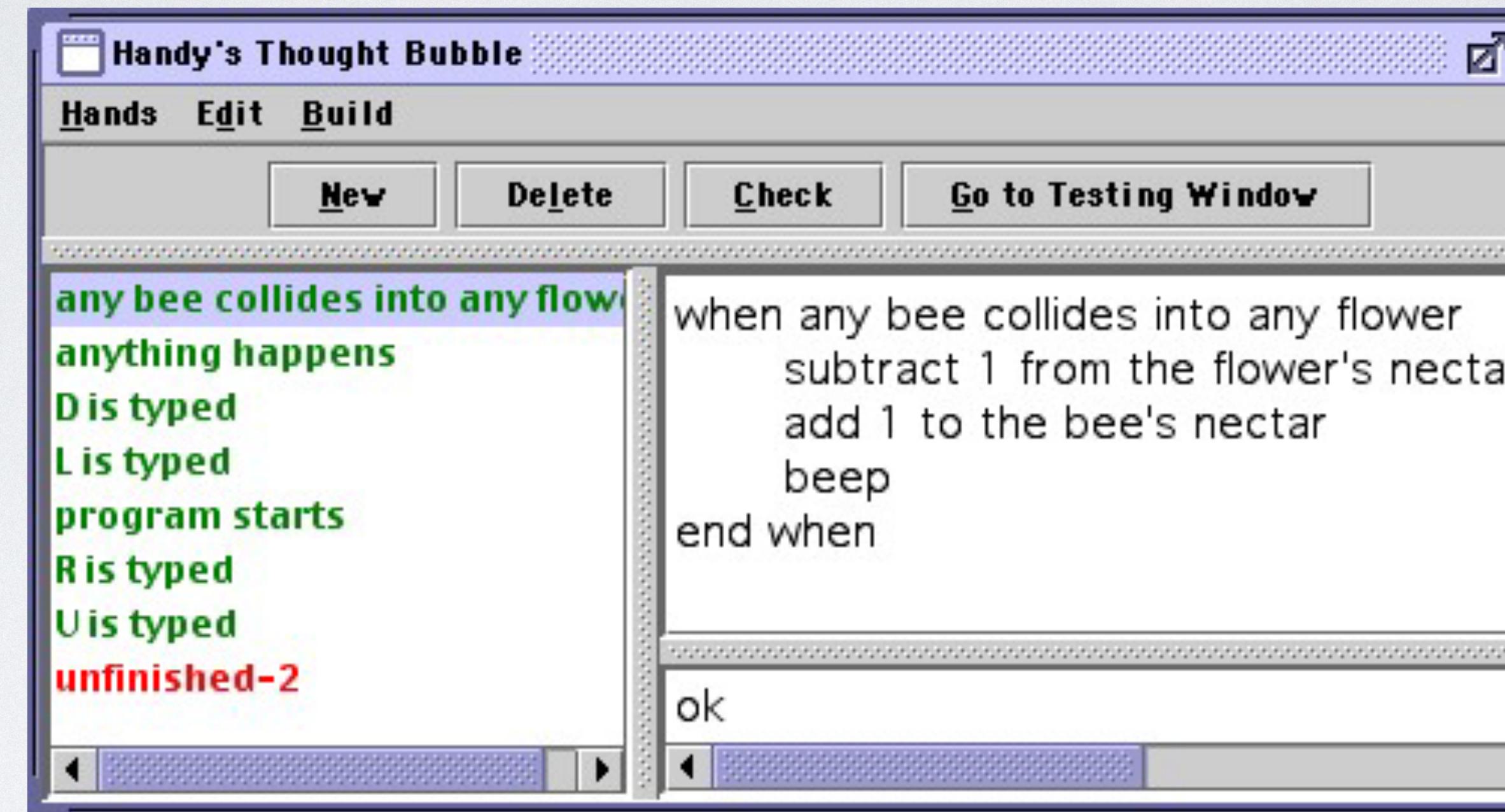


Figure 2. HAND**S** is an event-based system. The left pane lists seven complete (syntactically correct) event handlers, and one that is marked in red because it is not finished (unfinished-2). The upper right pane shows the code for *when any bee collides into any flower*. The lower right pane displays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

HANDS Design

- Based on natural programming studies
- Principle: *speak the user's language*
- $x = x + 10$ doesn't make sense in math
- instead: **add 100 to score**
- Allow "the" anywhere
- See also HyperTalk / AppleScript

MORE SYNTAX

```
if (...) ...
```

```
if (...) ...
```

```
else ...
```

```
if (...) ... end if
```

```
if (...) ...
```

```
else ... end if
```

which `if` does this `else` go with?

Now it's clear.

Is there empirical data
on this? If not, do we
need some?

PROPERTIES

- nectar of flower
- flower's nectar
- cardname

OPERATIONS ON CARDS

- set nectar of flower to 100
- set flower's nectar to flower's nectar + 25
- add 25 to flower's nectar

AGGREGATE OPERATIONS

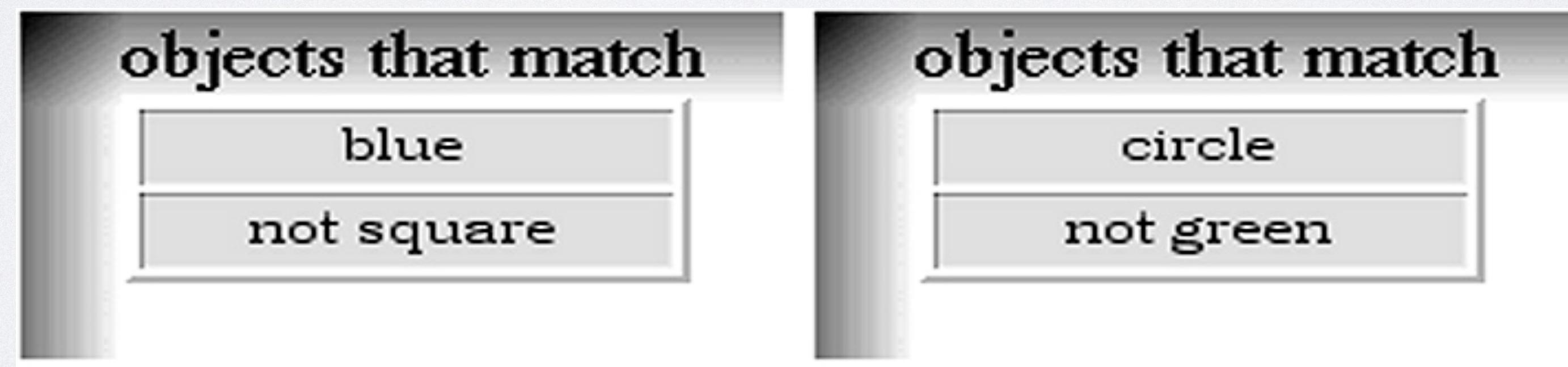
- All operations take lists!
- $1 + (1, 2, 3)$ evaluates to $2, 3, 4$
- $(1, 2, 3) + (2, 3, 4)$ evaluates to $3, 5, 7$

QUERIES

- Users don't maintain and traverse data structures
- Instead they use queries
 - **all flowers** evaluates to orchid, rose, tulip
 - **all bees** evaluates to bumble
 - **all snakes** evaluates to the empty list
 - **all (flower and (nectar < 100))** evaluates to orchid

BOOLEANS

- "The common uses of the words AND and OR in natural language lead to errors when these words are used to name the Boolean operators in queries; and the intended scope of the NOT operator is ambiguous."
-



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

LOOPS AND CONDITIONALS

with all flowers calling each the flower

set the flower's nectar to random from 50 to 100

end with

if f=tulip then

set f's nectar to 0

otherwise set f's nectar to 100 end if

USER STUDY

- 23 fifth graders, no programming experience
- HANDS vs. limited HANDS (no queries/aggregate operations, less data visibility)
- Tutorial; "Those portions utilizing a feature that was missing in the limited system were replaced with material teaching the easiest way to use the system's remaining features to achieve the same result."
- Of children who finished tutorial, HHANDS participants performed better than "limited HHANDS". $p < .05$

Table 6-1. Summary of results from this study. Participants using the *Full* system performed significantly better on the tasks than participants in the *Limited* condition.

	<i>Full</i>	<i>Limited</i>
Total number of participants	12	11
Participants completing the tutorial	9	9
Participants solving at least one task correctly ($p < .05$)	7	1
Cumulative number of tasks solved ($p < .05$)	19	1

LIMITATIONS OF NATURAL PROGRAMMING

- If you ask a Java programmer, won't they just write Java code?
- Yes, depends on prior experience
 - Find people without experience
 - Put people in a novel situation
 - Constrain choices

EXAMPLE

- Gave seven participants a description of a voter registration system.
- "Please implement this system in pseudocode using any language features you want."
 - Two invented syntax denoting states and transitions.
 - Many of the rest were unsafe, e.g. separate lists for unregistered and registered voters.
- "Here's a state diagram that models the system. Modify your pseudocode to use states and transitions."
 - Two participants: explicit state blocks with variables nested inside

```
contract Citizen {
```

```
    Citizen() {
        state = Unregistered;
    }
```

```
    state Unregistered {
        transaction submitApplication
```

```
    }
    ...
}
```

```
var states = {
    (0, "Processing"),
    (1, "CanVote"),
    (2, "CannotVote")}

var state = states[0]

function processed(a):
    # run this once "a" has been processed
    if a.accepted:
        state = states[1]
    else:
        state = states[2]
```

DISCUSSION

- For which research questions is natural programming a useful methodology?
- How might we mitigate the limitations of natural programming?