www.logofoundation.org

Event Programming

by

Michael Tempel

Overview

During the first two weeks of July, 1994 I taught two workshops in Event Programming at the Omar Dengo Foundation in Costa Rica. This document includes articles about these workshops, workshop notes, and sample programs.

"Advancing Logo", by Seymour Papert, appeared in the Spring 1994 issue of *Logo Update*. Papert recalls discussions that led to the July workshops. "A New Approach" appeared in the Fall 1994 issue of *Logo Update*. In this article I summarize the planning and execution of the workshops and place them in the context of the ongoing work of the Programa Informática Educativa in Costa Rica.

"Workshop Notes" were the materials used in the first July workshop. They include Logo tool procedures, sample programs, and comentary. Of course, things did not go as planned. "The Second Workshop" describes the changes and simplifications we made to the tools and samples in preparation for the session in the second week of July.

Advancing Logo●

by Seymour Papert

●The following is based on an interaction with a group of teachers who will surely recognize themselves in it. I decided not to identify them because the limited space available here forces me into an oversimplified presentation that casts me in the role of "the expert" coming with "answers" to problems that other people ("just teachers") couldn't solve. I hate this role. In reality the discussions were far richer and more creatively interactive. A fuller narrative would cast me as a catalyst rather than as an expert.

In the last issue we were grappling with the theoretical problem of defining what kind of Logo is advanced Logo. Soon after writing my contribution I found myself in a meeting of Logo teachers who were grappling with a related but very practical problem. They felt that many of their students were not advancing in programming skills. For example even after several years of doing Logo, students still composed long and intricate strings of Logo instructions and resisted all suggestions of organizing their work as subprocedures. The teachers saw the situation as a "bug" in their own work and had tried, so far with limited success, a number of strategies to remedy it. Some of them, working on the assumption that the students did not understand the idea of subprocedure, had developed teaching units to explain some principles of structured programming. Others tried to influence the students by showing concretely how their code could be written differently. The teaching materials they developed were excellent and I am sure that these teachers would have exercised great skill in engaging students in discussion about the relative merits of different styles of programming. Why then did their strategies not succeed?

In the course of the discussion I came to appreciate more sharply than I had before the force of another factor that enters into this kind of situation. One might call it "the Piaget insight" which I formulated in the following way back in the days I when worked in Geneva. One of the major contributions Piaget made to the investigation of children's thinking is understanding that it is not right to label as "wrong" children's declarations that there are more red beads than blue beads or more water in the tall glass. If you truly succeed in putting yourself in their intellectual framework (no easy task!) you will understand that "the children are correctly answering the questions they are really answering . . . only their questions are not the same as yours even if both use the same words." This does not mean that children, like the rest of us, aren't sometimes just plain wrong. But the whole point of Piaget's empathic and interactive way of conducting these conversations is to distinguish between errors or slips or "mere misunderstandings" and what the children really believe.

The Piaget insight applies to programming. For example, in the early days of our work at the Hennigan School a remarkable fifth grader, Nicky Brown (now an MIT undergraduate), put me right when I tried to advise him to break up a program into subprocedures. He systematically refuted all my arguments. No, it would not help him understand his program more clearly: he demonstrated by his ease of debugging and modifying his work that his own way of understanding it worked perfectly well. Well yes, subprocedures might help others take over parts of his work, but this wasn't his main goal and besides he didn't really think that they were interested anyway. Finally it was clear from his discussion that he just didn't like the kind of structuring of his project that would be produced by the subprocedurization I suggested.

Remembering Nicky Brown - and many other talented young programmers - was a sharp reminder that there is something wrong with describing the problem raised by these teachers as if their students lacked "programming skill." They may have lacked one particular programming skill, but what their style of programming required was another kind of skill - and when one looks at how fluently many students debug their "spaghetti code," one is impressed by the level of that skill.

This diatribe against imposing structured programming on these children is not intended to avoid the issue that it would be immensely valuable for them to learn new programming ideas. Quite the contrary, it led me to a crisper formulation of an alternative strategy to achieve this not by trying to force new ways to write the children's old programs but by introducing them to new programming domains in which other ideas would come up more naturally.

My new insight consisted of looking more closely at the development of the practice of Logo in the schools from which these teachers came - and when it was finally formulated I see that it applies on a much larger scale perhaps to the way that Logo programming in schools has developed across the board. One of the most positive educational benefits that came with the introduction of Logo into the schools I was looking at was a shift towards project-based learning. An excellent progressive step. But every progress risks bringing a certain kind of conservatism in its wake. In this case the conservatism took the form of establishing a certain model of "project" and this model of project carried with it a match to a certain style of programming, in fact precisely the style from which the teachers were now trying to wean their students.

The style of project in question developed a wide presence with the introduction of LogoWriter. Schematically described, it consists of using Logo to make a presentation of the results of research conducted by students. In the best cases Logo serves as more than just presenting ideas that are already formulated; the struggle to represent material is part of thinking about it. Some examples of this kind of work have become well known in the Logo community. Early examples included a project on the life cycle of the fly developed by Marian Rosen's computer summer camp group, and projects by students in Joanne Ronkin's class at Hennigan on such topics as the human skeleton. From Costa Rica came a bold students' project on human reproduction and many on historical and ecological themes. There was no doubt in the minds of the teachers at the meeting I am reporting here that such projects were educationally valuable and a good use of Logo. What emerged only slowly during the meeting was an awareness that the projects favor a certain style: they are essentially narrative projects and favor a "narrative" style of programming that looks in the end more like a page full of text than like a structured piece of mathematics. Indeed I would go further and say that there has been a co-evolution of this kind of project and this style of programming.

The best way to diversify the style of Logo programming is to diversify the kinds of projects for which Logo is used. A clear example is provided by Lego-Logo. Programming a Lego turtle to follow a light requires programming based on repetitive runs of conditionals. But there is room for infinite diversity without hardware beyond the computer. Writing a video game in the style of

Pacman or Mario requires efficient testing for new conditions and thinking carefully about a user interface that will permit rapid response. Simulations involving probability lead into issues of representation. All of these put so much of a real premium on small self-contained tool-like procedures that there is no need to persuade students to use them. They are what comes naturally in these contexts. ▲

A New Approach
by Michael Tempel

It was near the end of a five-day Logo workshop, time for sharing projects on the big screen. There were car races, boat races, and fish races; a rabbit searching for carrots and a turtle trying to find its way home; a maze, a pacman, and a pong game.

I was facilitating this workshop for 25 "Tutors", the people who provide training and support for teachers in 160 Costa Rican elementary schools in the Programa Informática Educativa (PIE), a joint project of the Fundación Omar Dengo (FOD) and the Ministerio de Educación Pública (MEP). During the past year I have spent a good part of my time in Costa Rica as one of a group of international consultants working at the FOD under a grant from the Banco Interamericano Desarrollo(BID) [Interamerican Development Bank] . Our charge has been to develop Logo curriculum materials, and to define what that means.

The workshop was inspired by discussions Seymour Papert referred to in "Advancing Logo," his column in the spring, 1994 issue of *Logo Update*. The "narrative" style of Logo program that he describes - using Logo to present a report or tell a story - has been developed to a remarkable degree in the Costa Rican schools over the past six years. I attended a Children's Logo Conference last October where students presented a series of such projects to schoolmates, parents, and guests. One major work was a study of the presidents of Costa Rica. This 20 minute extravaganza included biographical and historical information, along with a full color portrait, drawn using the turtle, for each of Costa Rica's 39 presidents. Remarkable projects like this have been presented at national and regional Children's Logo Conferences over the years.

One is so overwhelmed by the beauty and quality of these projects that it is difficult to adopt a critical stance and look for areas of potential growth and improvement. Yet that is precisely what is happening within the PIE. In part there is a wish to encourage children to use a more procedural style of Logo programming. But also, with the PIE now six years old, there is a growing feeling of wanting to broaden and diversify the kinds of activities that students engage in - to develop a "new approach" to Logo programming and projects.

In April we began designing workshops for the Tutors and teachers to be followed by testing of materials and teaching strategies in the schools. The school year in Costa Rica goes from March through December with a two-week break in July. We planned to use this July break for workshops, and then pilot our new approach in selected schools during the second half of the school year.

As part of the workshop planning I wrote a game program and saved versions of it at various stages of development. Along with the game I wrote a collection of Logo tool procedures that could be used as if they were primitives. As I went along, I modified the Logo tool procedures and added new ones. This process helped me to become familiar with possibilities and problems and in the end, I had a collection of tools and sample programs for use in the July workshops. (See the "Logo Tool Box" on page 30.)

We began to use the term "event programming" to refer to a range of Logo projects that include interactive video games, animations, and simulations. An important element in this type of program is the uncertainty about its course and the unpredictability of its outcome. The flow of action is affected by pressing keys and there may be elements of randomness, as well.

I don't mean to suggest that with our "new approach" we were journeying into uncharted territory. Interactive programs and games have long been a part of the Logo culture, familiar in Costa Rica as well as in other centers of Logo activity. In fact, some of the Tutors used the workshop time to modify and improve projects they had begun previously. But in Costa Rica, and elsewhere, the narrative style of Logo project has tended to be dominant, at least among LogoWriter users. In our approach to event programming, we drew upon old ideas but added some new dimensions, especially in our way of using Logo tool procedures to support the style of programming we sought to encourage.

During several months in which I was involved in planning and teaching these workshops we had frequent discussions about the many educational issues involved in this new approach. How do we present the tools? Do we introduce them to learners in an orderly sequence, or on an "as needed" basis as projects are developed? What relevance do games have to school subjects and curriculum, and to important domains of human knowledge?

Papert shed light on this last issue in a video tape he prepared for use in the workshop. He showed several examples of Logo programs. In one, a bee flies randomly around the screen. When it encounters a red flower, it stops, but it ignores flowers of other colors. This beginning could be elaborated into a simulation of insect behavior and plant propagation. Later in the video, we see a person walking across the screen and attempting to jump over an obstacle. There are many ways to program the jump. While working on this problem the learner will be engaged in questions about the laws of motion.

Papert was involved in this workshop not only through his video presence, but much more directly, as well. He participated in the planning process during the months leading up to the workshop. During the workshop itself we had numerous email exchanges and a few live telephone calls using a speaker phone so that the entire group could participate in the conversations.

During the second week of July there was another workshop for the teachers who had been selected to pilot the new ideas and materials with their students. This time two of the Tutors, Efraim Lopez and Julia Rivera, joined me as instructors. As soon as the first workshop ended, the three of us looked closely at people's projects, sorted out what had happened, and developed plans for the following week. Based on our experience we simplified the tools and samples. During the second workshop the projects that people created were similar to those we saw during the first week, but they were more elaborate. This was partly because there were fewer problems with the tools, allowing people to move ahead more easily with their projects. And, since it was the final week of the World Cup competition, the projects included a soccer game.

The months of planning and the two weeks of workshops were an intense experience for me, and a very rewarding one. It has also been a time of exciting change and growth for the Logo project in Costa Rica. In May, a newly elected government took office. President José Figueres and Education Minister Eduardo Doryan are committed to increasing resources for education. This includes expansion of computers in education projects. The PIE, which now serves 30% of Costa Rica's elementary school children, will grow to reach 50% within the next four years and a new project is being initiated in the secondary schools.

But the FOD has other changes to adjust to. Clotilde Fonseca, Executive Director of the FOD, and Eleanora Badilla, Diretor of the PIE, have both taken postions in the new government. The expansion of the PIE is proceeding with the guidance of its new director, Andrea Anfossi. The BID project has come to an end and its director, Claudio Gutierrez, has returned to his position as Professor of Computer Science at the University of Delaware.

After I left Costa Rica in July there was yet another round of workshops for more teachers and for ten new Tutors who had just been hired to provide the additional support needed for the growing project. This story is far from over. To be continued . . .▲

Workshop Notes:
Interactive Logo Game Programs

By Michael Tempel
May 19, 1994

In preparation for the proposed July workshops I have developed a series of game programs in order to familiarize myself with the possibilities and problems of programming in this domain, and to develop tool procedures for this type of programming activity. I haven't given much attention to content or cosmetics, focusing rather, on the structure of the games.

In a workshop I would present the idea of an interactive target game and provide the initial tools. As people develop their games, new tools and techniques can be introduced. By going through the development of some programs myself, I hope to be able to anticipate some of the needs of people in the workshop. I expect that new possibilities and problems will emerge and we will create some new tools, and so be better prepared for future workshops with teachers and for work with children in the schools.

These programs are based on the flower eating turtle program we used in a workshop here at FOD last January, and on ideas discussed with Seymour over the past several weeks.

I started with a target game that used several Logo tool procedures to simplify the programming. The tools appear at the bottom of the flip side of each LogoWriter page. Some tools are used on all the pages. I added more as I went along incorporating new featured and modifications into the game.

The games are all structured in the same general way. At the top of the flip side is this procedure:

```
to game
setup
play
end
```

Setup sets up the game by placing the turtle, the target and other objects, if any, and in some cases setting an initial level of "energy" for the turtle.. On all the pages, **play** is

```
to play
forever [go]
end
```

Forever is a tool used on all the pages. It causes endless repetition of the list of instructions it gets as input, in **play** this list contains just the procedure **go**.

```
to forever :stuff
run :stuff
```

```
forever :stuff
end
```

A pair of tools work together to capture keystrokes and use them to activate instructions. **Get.key** remembers a key that is pressed. **If.key** takes two inputs. The first is the name of a key. The second is a list of instructions to activate if that key was the last key pressed. The first line of **if.key** prevents an error that would occur in the next line if **current.key** had no value. This situation will occur the first time a game is run before any key has been pressed. The instruction **make "current.key "nothing** prevents **:action**from being run over and over again. This would happen when the program repeatedly calls **if.key** many times before a new key is pressed.

```
to get.key
if key?
[name readchar "current.key]
end

to if.key :key :action
if no name? "current.key [stop]
if :key = :current.key
[run :action
make "current.key "nothing]
end
```

If.color is another tool used in all the games. It activates a list of instructions when the turtle is over a particular background color.

```
to if.color :color :action
if colorunder = :color
[run :action]
end
```

Here is the progression of games. The pages are named SAMPLE, SAMPLE2, etc.

SAMPLE
In this first version of the game the turtle stamps a target at a specific spot. The turtle is then restored to its original shape and color and placed at home. **Go** moves the turtle forward or back 20 steps, or turns it right or left 90 degrees, depending upon which key is pressed. **Go** also checks for the color of the target. You win when you reach the target. You can't lose, but you *can* lose interest.

```
to game
setup
play
end

to setup
rg
```

```
ct
target
place.turtle
end

to target
setsh 11
setc 2
pu
setpos [120 80]
pd stamp pu
end

to place.turtle
setsh 0
setc 1
home
end

to play
forever [go]
end

to go
get.key
if.key "r [rt 90]
if.key "l [lt 90]
if.key "f [fd 20]
if.key "b [bk 20]
if.color 2
[pr "|I win!|
stopall]
end
```

SAMPLE2
In the second game, a live turtle is used as the target rather than a stamp. I tried this with the idea of eventually using moveable and moving targets.

```
to setup
rg
ct
pu
ask 1 [target]
end

to target
setsh 24
```

```
setc 2
pu
setpos [120 80]
st
end
```

A new tool is needed to check if the wandering turtle is touching the target turtle.

```
to if.touching :who :action
if pos = ask :who [pos]
[run :action]
end
```

This change revealed a bug. In SAMPLE the turtle moved forward or back 20 steps with each keypress. If you press "a" several times from the start of the game **ycor** increases in the pattern 0, 20, 40, 80. The next "a" sends it beyond the top of the screen. It returns at the bottom with a ycor of -90. Then repeated presses of "a" move it in the pattern -70, -50, -30, -10, 10, 30, 50, 70, 90. Since the target is at [120 80] the turtle will be 10 steps high or 10 steps low, never "touching" according to **if.touching**. (If the turtle wraps again, it's back in sync with the target. A third wrap throws it off again, and a fourth wrap . . .)

This wasn't a problem in SAMPLE because the target was large enough so that **if.color** would still detect a hit even when the turtle was high or low by 10 steps. My quick fix was to change the step from 20 to 10.

SAMPLE4

I went back to using a stamped target and **if.color** to detect it. I added the idea of energy. The turtle starts with a certain amount of energy determined by **setenergy** in **setup**.

```
to setup
rg
ct
target
place.turtle
setenergy 20
end
```

She loses energy at each step forward or back. **Less.energy** is called in **go** to do this. If **energy** gets to 0 you lose.

```
to go
get.key
if.key "r [rt 90]
if.key "l [lt 90]
if.key "f [fd 20 less.energy]
if.key "b [bk 20 less.energy]
display :energy
if energy = 0
[pr "|I lose!|
stopall]
if.color 2
[pr "|I win!|
stopall]
end
```

Another new instruction in **go** is **display :energy** which prints the current energy level on the screen after erasing what was previously printed.

The new tools required are

```
to setenergy :how.much
make "energy :how.much
end

to energy
output :energy
end

to more.energy
setenergy energy + 1
end

to less.energy
setenergy energy - 1
end

to display :thing
ct
print :thing
end
```

It takes 10 steps to get to the target by the shortest route possible. You can lose, but you really have to go out of your way to do so. The game could be made more difficult by lowering the starting energy to reduce the margin of error. You could also change it so that energy is lost for turns as well as for forward and back steps. This would make routes that were previously equivalent very different.

SAMPLE5
I went back to the problem caused by wrapping in SAMPLE2 trying a different solution that I thought might be generally useful later. Instead of **if.touching** which demanded and exact hit, I tried a fuzzier **if.near** which checks for being near. How near is determined in **near**. To solve the immediate problem a distance of 11 or more is good.

```
to go
get.key
if.key "r [rt 90]
if.key "l [lt 90]
if.key "f [fd 20]
if.key "b [bk 20]
if.near 1
[pr "|I win!|
stopall]
end

to if.near :who :action
if near? :who [run :action]
end

to near? :wh
output
(distance ask :wh [pos]) < 11
end
```

SAMPLE7
Instead of a target we now have a house. The turtle is trying to get home. But the big change is the addition of a flower which gives the turtle more energy if she eats it. **Setup** now builds the house and plants the flower.

```
to setup
rg
ct
house
flower
place.turtle
setenergy 20
end

to house
setsh 20
setc 2
pu
setpos [120 80]
pd stamp pu
end
```

```
to flower
ask 2
[st pu
setsh 1
setrandompos]
end
```

The turtle is now also put at a random place rather than at home.

```
to place.turtle
setsh 0
setc 1
setrandompos
end
```

Flower and **place.turtle** use a new tool to plant the flower and place the turtle at random places.

```
to setrandompos
setheading 0
fd 10 * random 100
setheading 90
fd 10 * random 100
end
```

The positions are restricted so that **ycor** and **xcor** are always multiples of 10. I did this so that **if.touching** could work.

Here's the new version of **go**:

```
to go
get.key
if.key "r [rt 90]
if.key "l [lt 90]
if.key "f [fd 10 less.energy]
if.key "b [bk 10 less.energy]
display :energy
if energy = 0
[pr "|I lose!|
stopall]
if.touching 2
[setenergy energy + 10
ask 2 [setrandompos]]
if.color 2
[pr "|I win!|
stopall]
end
```

The new instruction in **go** is

```
if.touching 2
[setenergy energy + 10
ask 2 [setrandompos]]
```

It checks to see if the wandering turtle is on a flower. If so, she eats it and her energy goes up by 10 points. a new flower is planted somewhere else. Actually, she doesn't eat it. It just moves to a new position, but the illusion is okay.

In **setup**, the turtle may be placed at a position that is too far from the house for her to get home without eating. You may have to plan a route so as to get the flower. When the flower is eaten, a new one appears and the situation changes again. It is also possible for a game to be unwinnable.

SAMPLE8
The only change here is in **house**. Now the house is also placed randomly.

```
to house
setsh 20
setc 2
pu
setrandompos
pd stamp pu
end
```

SAMPLE10
One flower is not enough. Unfortunately there are only four turtles in LogoWriter. I didn't think three flowers was much better so I tried a different approach. Now the flowers are stamped so we can have as many as we want. When the turtle eats one, it is unstamped by erasing it with **pe**. Here's the new **flower**:

```
to flower
ask 1
[pu
setsh 1
setrandompos
pd stamp pu]
end
```

Setup is changed to plant 10 flowers

```
to setup
rg tr
ct
house
repeat 10 [flower]
place.turtle
```

```
setenergy 10
end
```

Go is changed to include a new procedure **eat**.

```
to go
get.key
if.key "r [rt 90]
if.key "l [lt 90]
if.key "f [fd 10 less.energy 1]
if.key "b [bk 10 less.energy 1]
display :energy
if.color 1
[eat
flower]
if.color 2
[pr "|I win!|
stopall]
if energy = 0
[pr "|I lose!|
stopall]
end

to eat
pe stamp pu
more.energy 5
end

In the instruction

if.color 1
[eat
flower]
```

in the procedure **go**, **eat** erases the flower and increases the turtle's energy by 5 points. **Flower** plants a new flower.

I also changed **more.energy** and **less.energy** so that they each take an input and change energy level accordingly. So in **go**, moving forward or back reduces energy level by one point:

```
if.key "f [fd 10 less.energy 1]
if.key "b [bk 10 less.energy 1]
```

In **eat**, **more.energy 5** increases energy by 5. The new tools are

```
to more.energy :how.much
setenergy energy + :how.much
end

to less.energy :how.much
setenergy energy - :how.much
end
```

SAMPLE11
I added obstacles. Bumping into one reduces energy. **Setup** now looks like this:

```
to setup
rg
ct
house
repeat 10 [flower]
repeat 10 [obstacle]
place.turtle
setenergy 10
end

to obstacle
ask 2
[pu
setsh 2
setc 3
setrandompos
pd stamp pu]
end
```

Go is changed to include a check for the obstacles ofcolor 3. **Crash** causes energy to be lost. It also backs the turtle off the obstacle. Before I added this back-up, the turtle stayed on the obstacle and was rapidly drained of all energy. There's a small bug, though. If you press "r" and back the turtle into an obstacle, she still moves back to get off it. I'd prefer she move forward in that case, but I let it go for now. I also put sound effects in **eat** and **crash**.

```
to go
display :energy
get.key
if.key "f [fd 10 less.energy 1]
if.key "b [bk 10 less.energy 1]
if.key "r [rt 90]
if.key "l [lt 90]
if.color 1
[eat
flower]
if.color 3 [crash]
if.color 2
[pr "|I win!|
stopall]
if energy < 0
[pr "|I lose!|
stopall]
end

to eat
tone 500 1
pe stamp pu
more.energy 10
end

to crash
tone 40 1
bk 10
less.energy 5
end
```

At first I used shape 11 for the obstacles, but it was too big in the vertical direction. The turtle could record two hits while passing over it. I made a smaller box in shape 2.

It is possible to accumulate so many points that you will always be able to get to the house. This makes the game less interesting. One could alter several factors to change the difficulty of the game. The initial conditions that may be changed are the numbers of flowers and obstacles, and the initial energy level of the turtle. The amount of energy gained when eating a flower and lost when hitting an obstacle may also be changed.

There are some little bugs I found and haven't bothered to fix, yet. A flower may be stamped on top of an obstacle. If this happens then the turtle may safely go on the obstacle and eat the flower. This is because the flower is centered in the obstacle and that's where **if.color** is checking. Not only is it safe, but eating that flower also puts a hole in the obstacle making it safe for the turtle to go over it in the future. (Actually, it's not a bug, it's a feature.)

SAMPLE12
Here's a big change. The turtle moves forward automatically. Keystrokes only turn left or right. You have to think quickly since the turtle is always in motion (and losing energy). Here's the new **go**.

```
to go
display :energy
fd 10
less.energy 1
wait 20
get.key
if.key "r [rt 90]
if.key "l [lt 90]
if.color 1
[eat
flower]
if.color 3 [crash]
if.color 2
[pr "|I win!|
stopall]
if energy < 0
[pr "|I lose!|
stopall]
end
```

Wait 20 keeps things from going to fast. You can change the input to **wait** for a faster or slower paced game. You could add procedures to allow the player to set the speed by typing "hard" or "easy".

Crash doesn't need **bk 10** anymore since the turtle moves forward off the obstacle after a hit.

```
to crash
tone 40 1
less.energy 5
end
```

SAMPLE13

Now there's a red rabbit on the screen. The rabbit is very dangerous. (He bears a long-standing grudge against the turtle that dates back to a foot race they had many years ago.) If the turtle bumps into the rabbit she loses 10 energy points.

The rabbit is turtle number 2 which is randomly positioned with **setrandompos**.

```
to rabbit
ask 2
[pu st
setsh 22
setc 4
setrandompos]
end
```

Rabbit is added to **setup**.

```
to setup
rg tr
ct
house
repeat 10 [flower]
repeat 10 [obstacle]
rabbit
place.turtle
setenergy 10
end
```

Collision is detected with **if.touching**. But, the rabbit may suddenly jump to a new poisition without warning. This requires a new tool:

```
to maybe :action
if 1 = random 10
[run :action]
end
```

Go now includes a line that may cause the rabbit to jump and a line to detect a collision which uses **if.touching**

```
to go
display :energy
fd 10
less.energy 1
quiz s [rabbit]
wait 20
get.key
if.key "r [rt 90]
if.key "l [lt 90]
if.color 1
[eat
flower]
if.color 3 [crash]
if.color 2
[pr "|I win!|
stopall]
if.touching 2
[ay!]
if energy < 0
[pr "|I lose!|
stopall]
end

to ay!
tone 60 2
tone 40 4
less.energy 10
end
```

That's it for now. I have some other ideas to try later:

- Some flowers, of a different color, will contain more energy.
- Some obstacles will cause a greater loss of energy than others.
- New obstacles will appear and disappear in a random way.
- When you accumulate a certain number of points, you go to another world with a new game (on another LogoWriter page.)

The Second Workshop

During the second week of July another workshop was held for a group of five teachers who would then work with other teachers on the "New Approach". The facilitators were Julia Rivera, Efraim Lopez, and myself.

The three of us spent the afternoon following the end of the first workshop planning for the next one. We sought to simplify the tools and samples and fix some bugs in the earlier materials.

Rather than include the tool procedures on the flip side of each sample page, we created a set of tools that could be used by all the samples, and saved it on a separate page. There was a **startup** procedure on the flip side of each sample page:

```
to startup
gettools "tools
end
```

The first sample places a stamped target in the upper right corner of the screen. The player directs the turtle to the target by pressing single keys to go forward, back, left or right. The target is detected with **colorunder**.

Sample1

```
to startup
gettools "tools
end

to game
setup
play
end

to setup
rg
ct
target
place.turtle
end

to target
setsh 11
setc 2
pu
setpos [120 80]
pd stamp pu
end

to place.turtle
setsh 0
setc 1
home
end

to play
forever [go]
end
```

```
to go
get.key
ifkey "r [rt 90]
ifkey "l [lt 90]
ifkey "f [fd 20]
ifkey "b [bk 20]
ifcolor 2
[print "|I win!|
stopall]
end
```

The second sample keeps track of energy, which decreases as the turtle is moved forward or back. Energy can run out before the target is reached, in which case, you lose.

Sample2

```
to startup
gettools "tools
end

to game
setup
play
end

to setup
rg
ct
target
place.turtle
setenergy 20
end

to target
setsh 11
setc 2
pu
setpos [120 80]
pd stamp pu
end

to place.turtle
setsh 0
setc 1
home
end
```

```
to play
forever [go]
end

to go
get.key
ifkey "r [rt 90]
ifkey "l [lt 90]
ifkey "f [fd 20 less.energy 1]
ifkey "b [bk 20 less.energy 1]
display :energy
if :energy = 0
[print "|I lose!|
stopall]
ifcolor 2
[print "|I win!|
stopall]
end
```

The third sample is the same as the second except that the target is at a random place on the screen.

Sample3

```
to startup
gettools "tools
end

to game
setup
play
end

to setup
rg
ct
target
place.turtle
setenergy 20
end
```

```
to target
setsh 11
setc 2
pu
setrandompos
pd stamp pu
end

to place.turtle
setsh 0
setc 1
home
end

to play
forever [go]
end

to go
get.key
ifkey "r [rt 90]
ifkey "l [lt 90]
ifkey "f [fd 10 less.energy 1]
ifkey "b [bk 10 less.energy 1]
display :energy
if :energy = 0
[print "|I lose!|
stopall]
ifcolor 2
[print "|I win!|
stopall]
end
```

In the fourth sample the target is a live turtle rather than a stamp. **Ifnear** is used to detect it rather than **ifcolor**.

Sample4

```
to startup
gettools "tools
end

to game
setup
play
end
```

```
to setup
rg ct
ask 1 [target]
place.turtle
end

to target
setsh 11
setc 2
pu
setpos [120 80]
st
end

to place.turtle
pu
setsh 0
setc 1
home
end

to play
forever [go]
end

to go
get.key
ifkey "r [rt 90]
ifkey "l [lt 90]
ifkey "f [fd 10]
ifkey "b [bk 10]
ifnear 1
[print "|I win!|
stopall]
end
```

The need for a new tool, **until.color**, emerged during the first workshop. In all of my original samples, the game ended when the target was detected. For example, in **go** below the game ends with **stopall** when the turtle goes over color 2.

```
to go
get.key
if.key "r [rt 90]
if.key "l [lt 90]
if.key "f [fd 20]
if.key "b [bk 20]
if.color 2
[pr "|I win!|
stopall]
end
```

But what if we wanted to have a two-part game? After hitting the first target, a second target might appear. Such a **game** program might look like this:

```
to game
setup
forever [go]
setup2
forever [go2]
end
```

But **stopall** stops all procedures and returns to "top level" or "command level". We would never get to **setup2**. What if we use **stop** instead of **stopall** in **go**:

```
to go
get.key
if.key "r [rt 90]
if.key "l [lt 90]
if.key "f [fd 20]
if.key "b [bk 20]
if.color 2
[pr "|I win!|
stop]
end
```

This won't work either because **stop** stops **if.color**, but **forever [go]** continues. We would want to stop **if.color**, **go**, and **forever**, allowing **game** to continue to **setup2**. This isn't possible in LogoWriter, although it is in other versions of Logo.

We got around this by creating two new tools. **until.color** and **until.near**. The fifth and sixth samples illustrate their use.

Sample5

```
to startup
gettools "tools
rg
pu
sety 85
setsh 11
setc 4
pd stamp pu
setx 140
setc 5
pd stamp pu
home
setsh 0
setc 1
end

to example
go
go2
end

to go
until.color 4 [fd 1 wait 2]
rt 90
end

to go2
until.color 5 [fd 1 wait 2]
tone 440 20
end
```

Sample6

```
to startup
gettools "tools
rg
pu
sety 85
setsh 11
setc 4
pd stamp pu
home
setsh 0
setc 1
ask 1
[st pu
setsh 11
setpos [140 85]
setc 5]
end

to example
go
go2
end

to go
until.color 4 [fd 1 wait 2]
rt 90
end

to go2
until.near 1 [fd 1 wait 2]
tone 440 20
end
```

Logo Tool Box

```
to get.key
ifelse key?
[name readchar "current.key]
[make "current.key "nothing]
end

to ifkey :key :action
if :key = :current.key
[run :action
make "current.key "nothing]
end

to ifcolor :color :action
each
[if colorunder = :color
[run :action]]
end

to ifnear :who :action
if near? :who [run :action]
end

to setsentitivity :what
make "sensitivity :what
end

to sensitivity
output :sensitivity
end

to near? :wh
if not name? "sensitivity
[make "sensitivity 11]
output
(distance ask :wh [pos])
< :sensitivity
end

to forever :stuff
run :stuff
forever :stuff
end
```

```
to setenergy :how.much
make "energy :how.much
end

to energy
output :energy
end

to more.energy :how.much
setenergy energy + :how.much
end

to less.energy :how.much
setenergy energy - :how.much
end

to display :thing
ct
print :thing
end

to setrandompos
seth 0
fd 10 * random 100
seth 90
fd 10 * random 100
end

to until.color :color :action
if colorunder = :color [stop]
run :action
until.color :color :action
end

to until.near :who :action
if near? :who [stop]
run :action
until.near :who :action
end

to maybe :action
if 1 = random 10
[run :action]
end
```