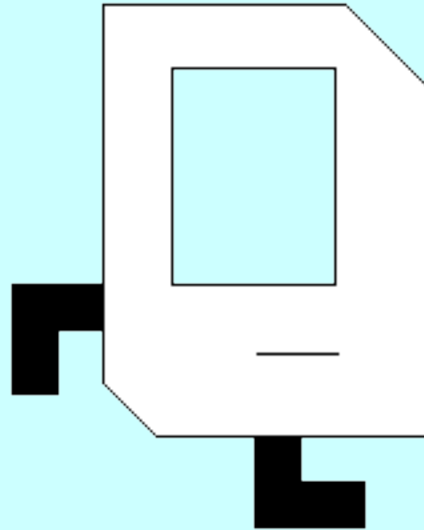
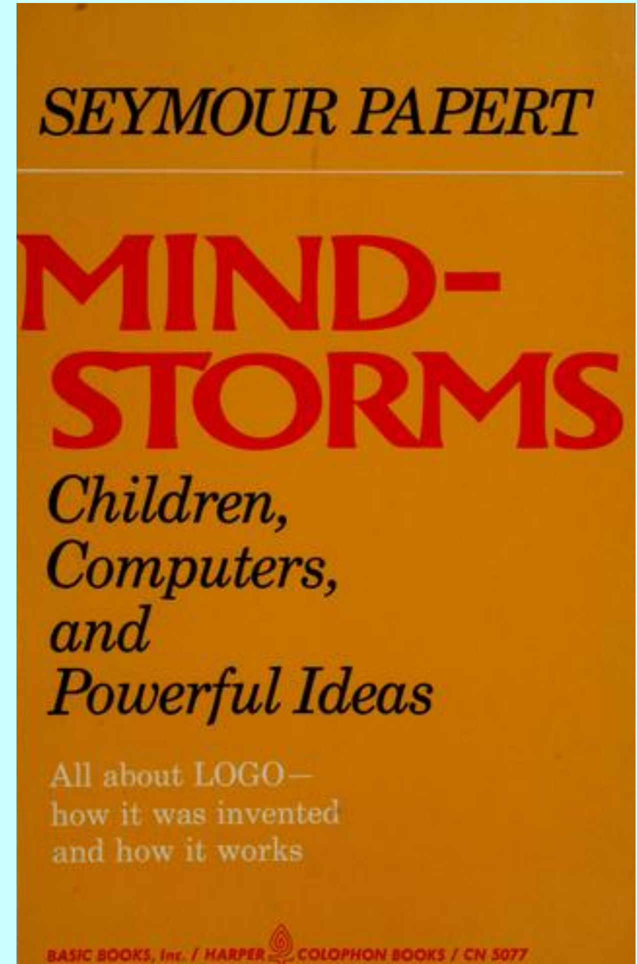
Introductory Programming: LOGO, Scratch, Karel the Robot, BASIC



Chris Gregg
Based on Slides from Eric Roberts
CS 208E
October 2, 2018

The Project LOGO Turtle

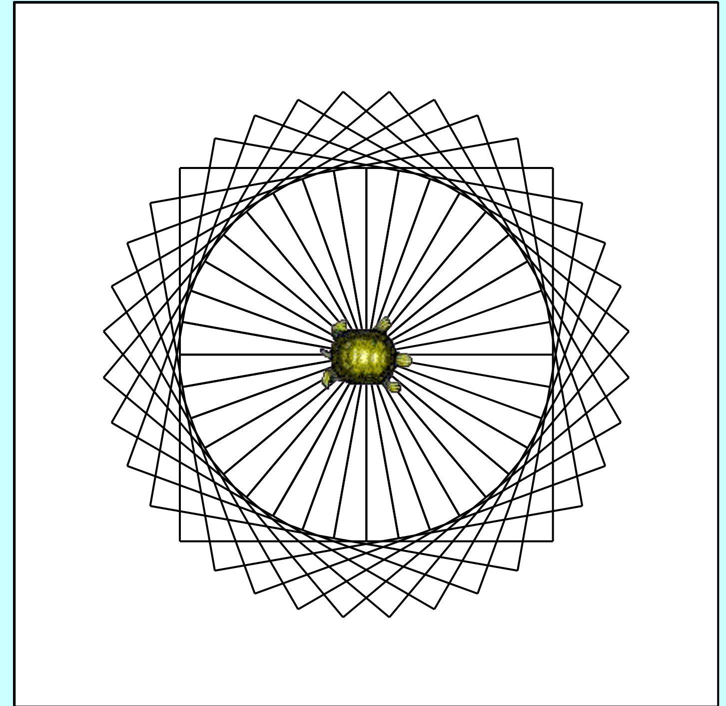
- In the 1960s, the late Seymour Papert and his colleagues at MIT developed the Project LOGO turtle and began using it to teach schoolchildren how to program.
- The LOGO turtle was one of the first examples of a ***microworld***, a simple, self-contained programming environment designed for teaching.
- Papert described his experiences and his theories about education in his book *Mindstorms,* which remains one of the most important books about computer science pedagogy.



Programming the LOGO Turtle

```
to square
  repeat 4
    forward 40
    left 90
  end
end


to flower
   repeat 36
      square
      left 10
   end
end
```

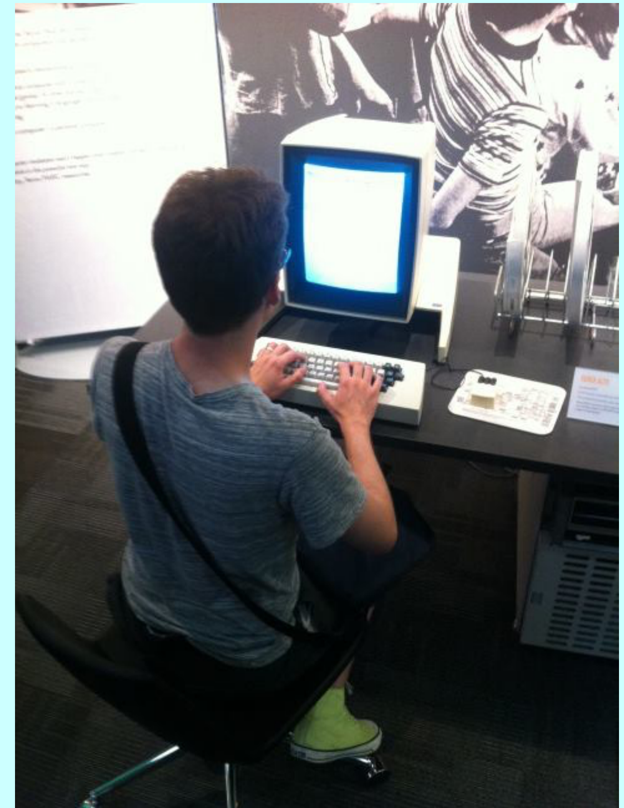


The Logo Turtle in Python

```
$ python
Python 2.7.13 (v2.7.13:a06454b1afa1, Dec 17 2016, 12:39:47)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> from turtle import *
>>> def sun():
...   color('red','yellow')
...   begin_fill()
...   while True:
...     forward(200)
...     left(170)
...     if abs(pos()) < 1:
...       break
...   end_fill()
...   done()
...
>>> sun()
```

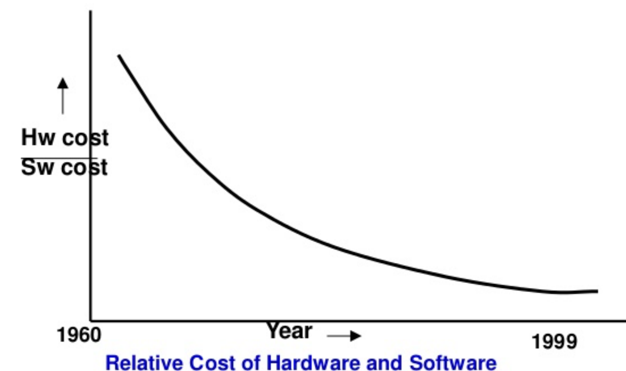
Hi!

- I'm Ben Allen, I'm a grad student in the MS in CS Education program
- I hold a PhD from the Modern Thought and Literature program, where my committee included members from Communication, History, and Rhetoric
- I'm interested in interdisciplinary research in computer science, particularly in the history of programming language design
- I've written articles defending widely loathed programming languages



“Software crisis”

- In the 1960s, the term “software crisis” was used to describe the major problem in computing – because of the ongoing shortage of trained programmers.
- This crisis never ended.
- Teaching good programming practice quickly has been a primary concern since the early days of computing – and one strategy is designing languages that are at least supposed to be quick to learn.

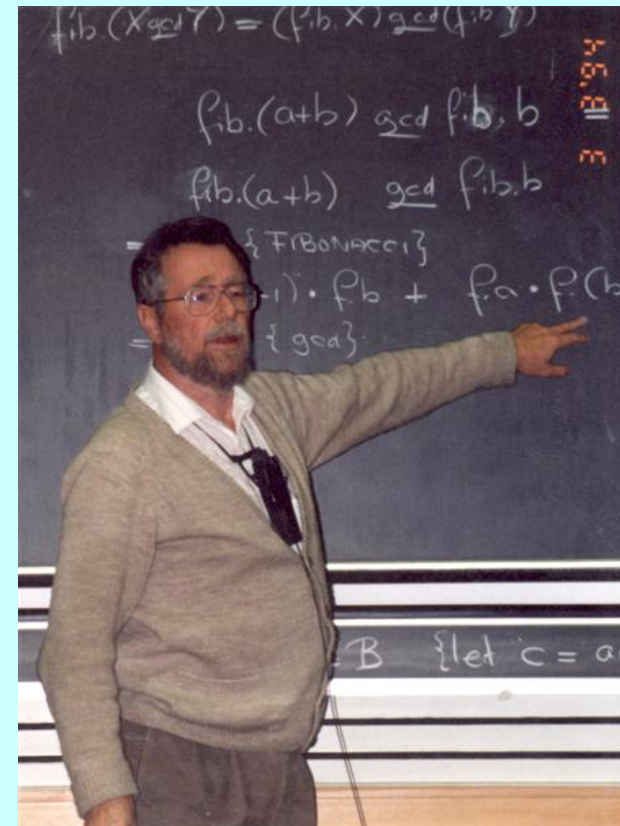


Bad ideas in computer history

```
11        PROCEDURE DIVISION.
12        100-MAIN.
13                PERFORM UNTIL MORE-DATA = 'N'
14                        DISPLAY 'Enter a sales amount ->'
15                        ACCEPT SALES-AMOUNT
16                        MULTIPLY .08 BY SALES-AMOUNT GIVING SALES-TAX
17                        COMPUTE SALES-TOTAL = (SALES-TAX + SALES-AMOUNT)
18                        DISPLAY 'The total sales is 'SALES-TOTAL
19                        DISPLAY 'Would you like to run it again (Y or N)?'
20                        ACCEPT MORE-DATA
21                END-PERFORM
22                STOP RUN.
```

How do we teach programming?

- How we teach programming depends on what we think programming is.
- The COBOL idea was that we should teach programming as English-language instructions. This didn't work out…
- On the right here is Edsger Dijkstra, maybe the most influential computer scientist *not* from Stanford.
- Dijkstra argued that we should teach programming as a *radical novelty.*
- Dijkstra on COBOL: "The use of COBOL cripples the mind; its teaching, therefore, should be regarded as a criminal offense.



Cruelty, or, Dijkstra Teaches Programming

On the cruelty of really teaching computing science

- In "on the cruelty of really teaching computing science," Dijkstra laid out a plan of instruction that he would later implement at UT Austin.
- For his intro to computing classes, he developed a language for which there was no compiler, and banned his students from using computers to test their programs.
- To Dijkstra, computing was best understood as a new branch of formal mathematics. With every assignment, students would have to prepare formal proofs of the correctness of their programs.
- This worked… okay…

"Lifelong Kindergarten"

Sophisticated Scratch

```
define Bubble Down
set bubble_down_value to index
if not 2 * index > length of list then
    if item 2 * index of queue < item index of list then
        set bubble_down_value to 2 * index
    if not 2 * index + 1 > length of list then
        if item 2 * index + 1 of list < item bubble_down_value of list then
            set bubble_down_value to 2 * index + 1
    if not bubble_down_value = index then
        Swap bubble_down_value index
        Bubble Down bubble_down_value

define Make Into Heap
```

How do students actually use scratch?

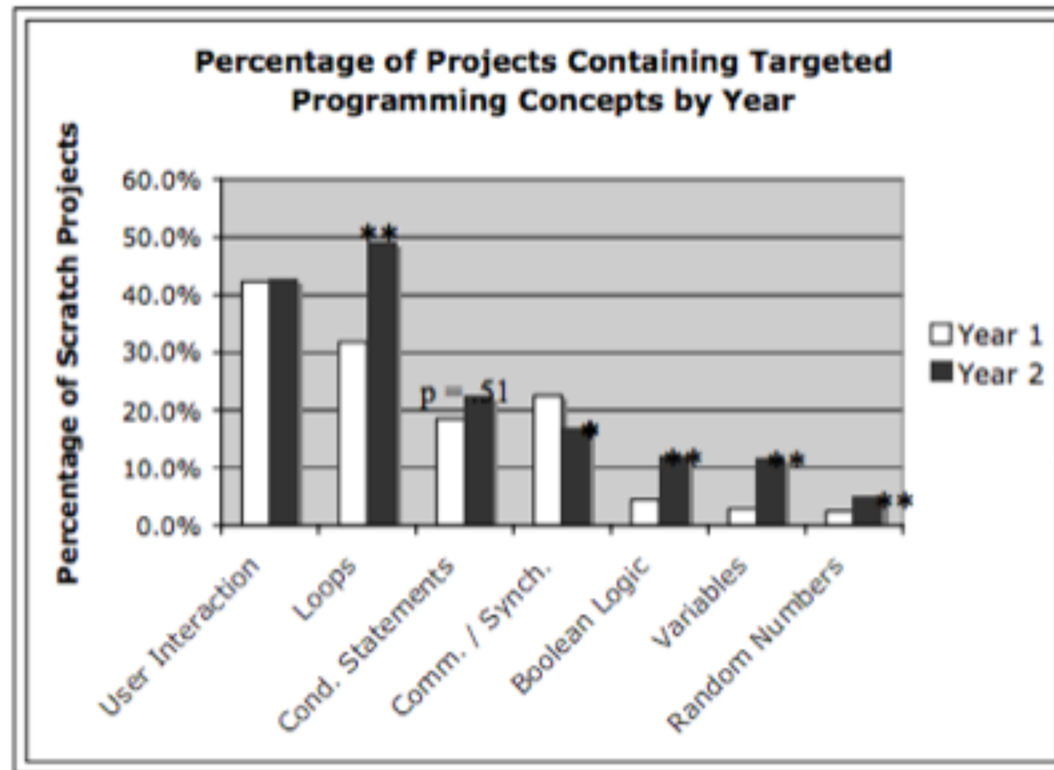
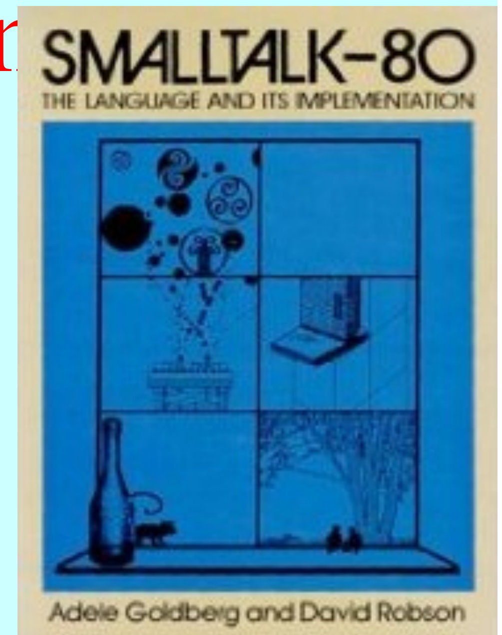
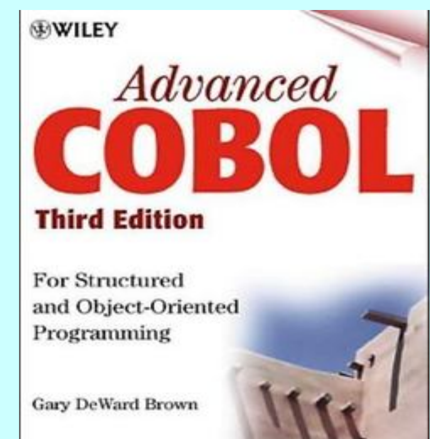


Figure 5. Graph demonstrating the change in the percentage of projects that used various programming concepts over time $^{**}p < .001$ $^{*}p < .05$

Educational programming and new programming paradigms



- Consider that the first two widely used object oriented languages (Smalltalk and Pascal) were originally designed as educational languages, with Smalltalk explicitly designed as a tool for constructivist learning
- Object orientation is more or less ubiquitous now: even fussy old COBOL supports it.
- Even if you're not particularly interested in CS education, paying attention to educational programming might still be worthwhile.



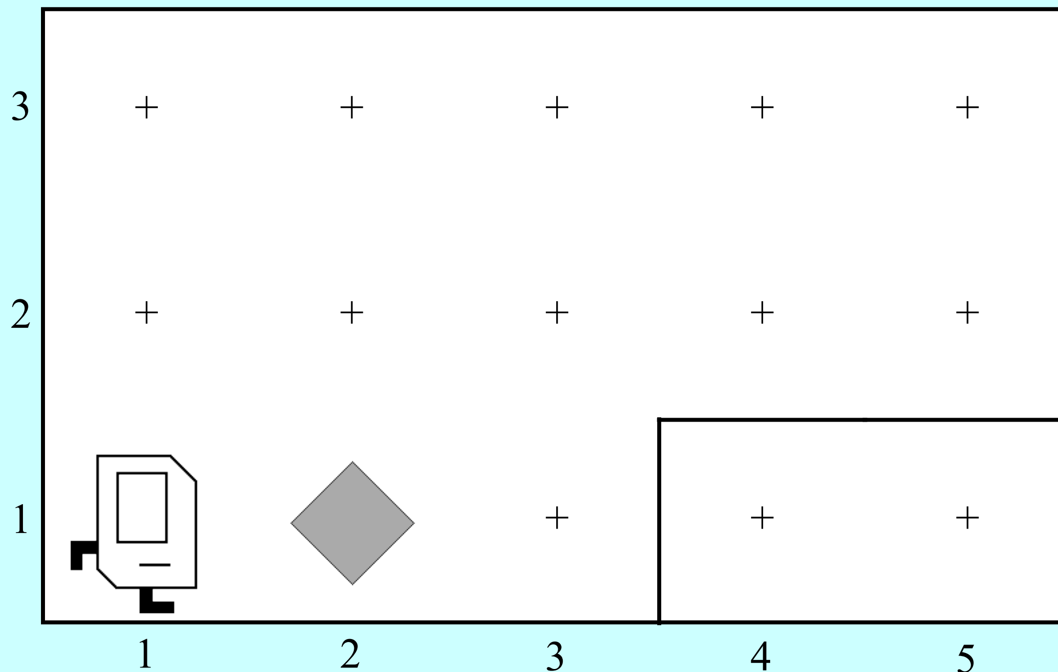
Rich Pattis and Karel the Robot

- Karel the Robot was developed by Rich Pattis in the 1970s when he was a graduate student at Stanford.
- In 1981, Pattis published *Karel the Robot: A Gentle Introduction to the Art of Programming,* which became a best-selling introductory text.
- Pattis chose the name *Karel* in honor of the Czech playwright Karel Čapek, who introduced the word *robot* in his 1921 play *R.U.R.*
- In 2006, Pattis received the annual award for Outstanding Contributions to Computer Science Education given by the ACM professional society.

Meet Karel the Robot

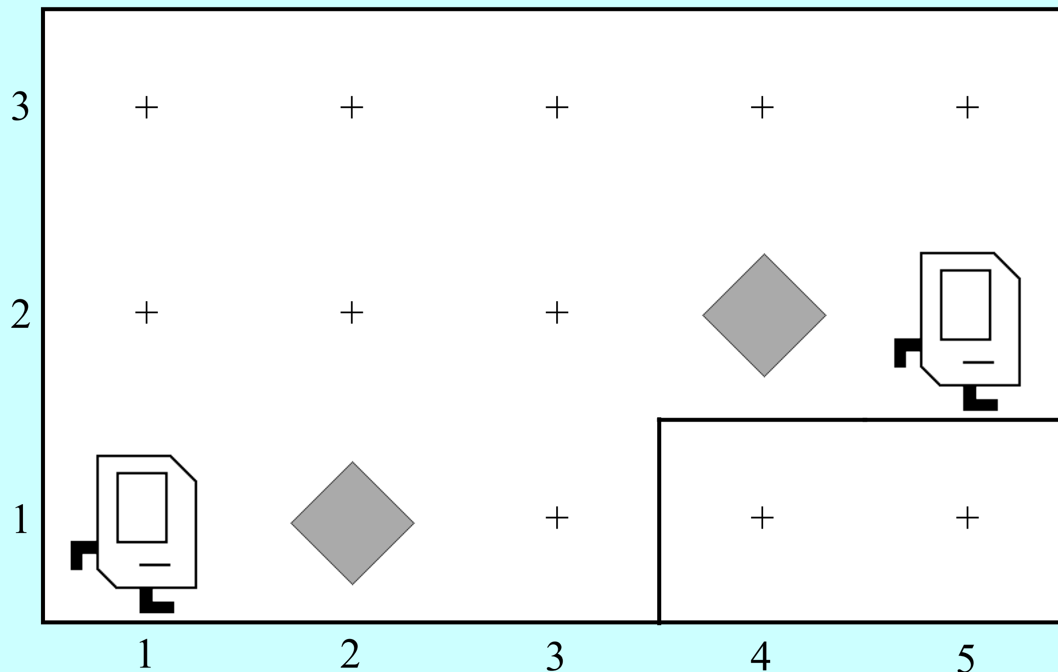
- Initially, Karel understands only four primitive commands:

`move()`	Move forward one square
`turnLeft()`	Turn 90 degrees to the left
`pickBeeper()`	Pick up a beeper from the current square
`putBeeper()`	Put down a beeper on the current square



Your First Challenge

- How would you program Karel to pick up the beeper and transport it to the top of the ledge? Karel should drop the beeper at the corner of 2nd Street and 4th Avenue and then continue one more corner to the east, ending up on 5th Avenue.



The `moveBeeperToLedge` Function

```
/*
 * File: MoveBeeperToLedge.k
 * -------------------------
 * This program moves a beeper up to a ledge.
 */

function moveBeeperToLedge() {
   move();
   pickBeeper();
   move();
   turnLeft();
   move();
   turnLeft();
   turnLeft();
   turnLeft();
   move();
   putBeeper();
   move();
}
```

Defining New Functions

- A Karel program consists of a collection of ***functions***, each of which is a sequence of statements that has been collected together and given a name. The pattern for defining a new function looks like this:

```
function name() {
    statements that implement the desired operation
}
```

- In patterns of this sort, the boldfaced words are fixed parts of the pattern; the italicized parts represent the parts you can change. Thus, every helper function will include the keyword **`function`** along with the parentheses and braces shown. You get to choose the name and the sequence of statements performs the desired operation.

The `turnRight` Function

- As a simple example, the following function definition allows Karel to turn right by executing three **`turnLeft`** operations:

```
function turnRight() {
   turnLeft();
   turnLeft();
   turnLeft();
}
```

- Once you have made this definition, you can use **`turnRight`** in your programs in exactly the same way you use **`turnLeft`**.
- In a sense, defining a new function is analogous to teaching Karel a new word. The name of the function becomes part of Karel's vocabulary and extends the set of operations the robot can perform.

Adding Functions to a Program

```
/*
 * File: MoveBeeperToLedge.k
 * -------------------------
 * This program moves a beeper up to a ledge using turnRight.
 */

function moveBeeperToLedge() {
   move();
   pickBeeper();
   move();
   turnLeft();
   move();
   turnRight();
   move();
   putBeeper();
   move();
}

function turnRight() {
   turnLeft();
   turnLeft();
   turnLeft();
}
```

Exercise: Defining Functions

- Define a function called **`turnAround`** that turns Karel around 180 degrees without moving.

```
function turnAround() {
   turnLeft();
   turnLeft();
}
```

- Define a function **`backup`** that moves Karel backward one square, leaving Karel facing in the same direction.

```
function backup() {
   turnAround();
   move();
   turnAround();
}
```

Control Statements

- In addition to allowing you to define new functions, Karel also includes three statement forms that allow you to change the order in which statements are executed. Such statements are called ***control statements***.
- The control statements available in Karel are:
  – The **`repeat`** statement, which is used to repeat a set of statements a predetermined number of times.
  – The **`while`** statement, which repeats a set of statements as long as some condition holds.
  – The **`if`** statement, which applies a conditional test to determine whether a set of statements should be executed at all.
  – The **`if-else`** statement, which uses a conditional test to choose between two possible actions.

The **`repeat`** Statement

- In Karel, the **`repeat`** statement has the following form:

```
repeat (count) {
    statements to be repeated
}
```

- Like most control statements, the **`repeat`** statement consists of two parts:
  - The ***header line***, which specifies the number of repetitions
  - The ***body***, which is the set of statements affected by the **`repeat`**
- Note that most of the header line appears in boldface, which means that it is a fixed part of the **`repeat`** statement pattern. The only thing you are allowed to change is the number of repetitions, which is indicated by the placeholder *count*.

Using the **repeat** Statement

- You can use **repeat** to redefine **turnRight** as follows:

```
function turnRight() {
   repeat (3) {
      turnLeft();
   }
}
```

- The following function creates a square of four beepers, leaving Karel in its original position:

```
function makeBeeperSquare() {
   repeat (4) {
      putBeeper();
      move();
      turnLeft();
   }
}
```

Conditions in Karel

- Karel can test the following conditions:

positive condition	*negative condition*
`frontIsClear()`	`frontIsBlocked()`
`leftIsClear()`	`leftIsBlocked()`
`rightIsClear()`	`rightIsBlocked()`
`beepersPresent()`	`noBeepersPresent()`
`beepersInBag()`	`noBeepersInBag()`
`facingNorth()`	`notFacingNorth()`
`facingEast()`	`notFacingEast()`
`facingSouth()`	`notFacingSouth()`
`facingWest()`	`notFacingWest()`

The `while` Statement

- The general form of the **`while`** statement looks like this:

```
while (condition) {
    statements to be repeated
}
```

- The simplest example of the **`while`** statement is the function **`moveToWall`**, which comes in handy in lots of programs:

```
function moveToWall() {
   while (frontIsClear()) {
      move();
   }
}
```

The `if` and `if-else` Statements

- The **`if`** statement in Karel comes in two forms:
  - A simple **`if`** statement for situations in which you may or may not want to perform an action:

```
if (condition) {
      statements to be executed if the condition is true
}
```

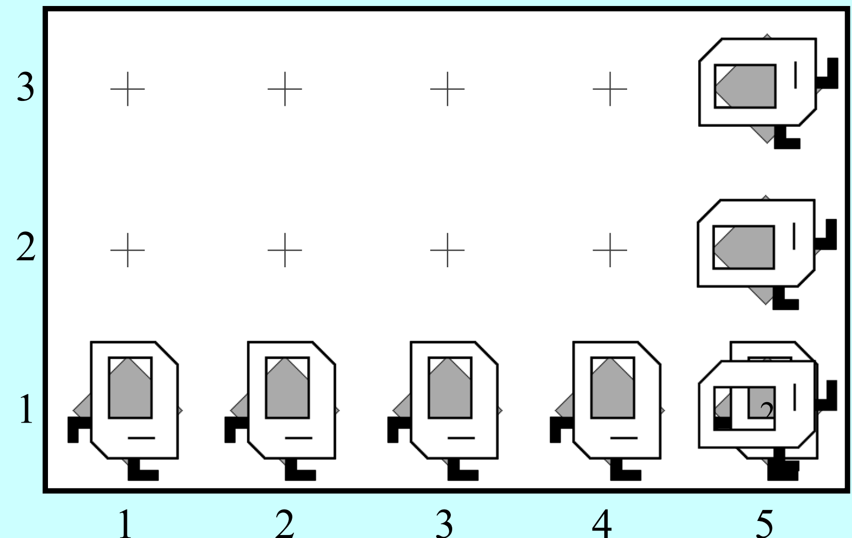
  - An **`if-else`** statement for situations in which you must choose between two different actions:

```
if (condition) {
      statements to be executed if the condition is true
} else {
      statements to be executed if the condition is false
}
```

Exercise: Creating a Beeper Line

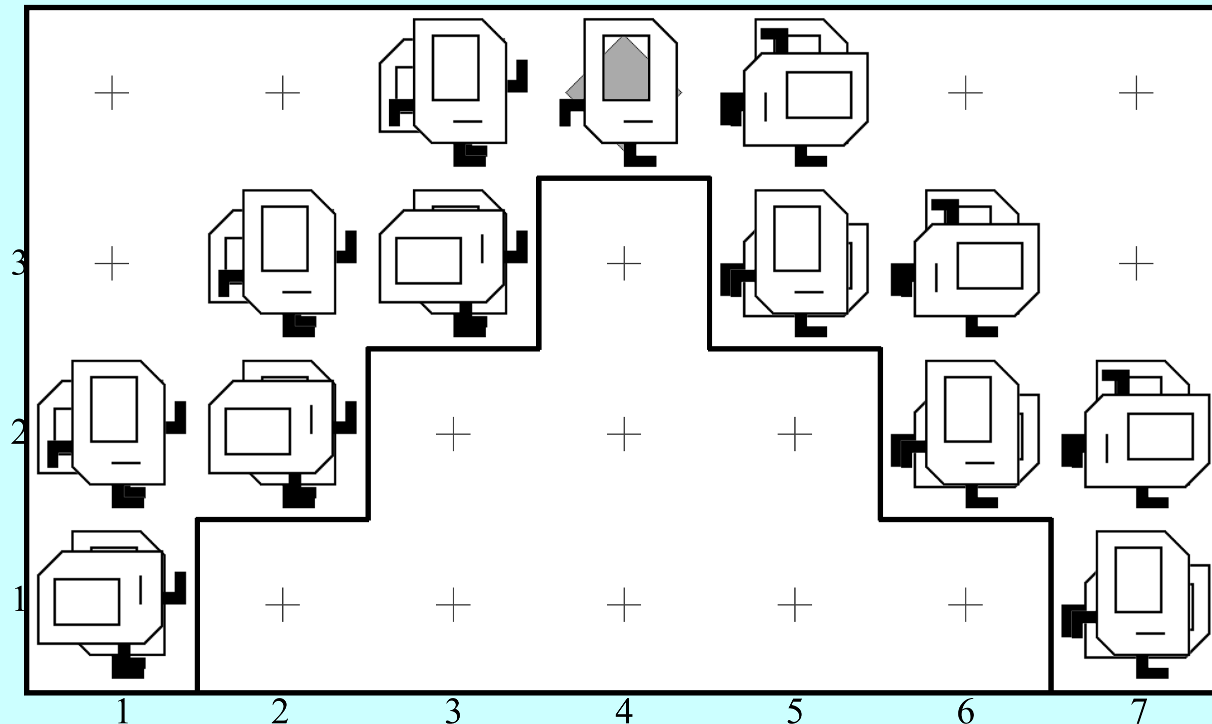
- Write a function **`putBeeperLine`** that adds one beeper to every intersection up to the next wall.
- Your function should operate correctly no matter how far Karel is from the wall or what direction Karel is facing.
- Consider, for example, the following function called **`test`**:

```
function test() {
   putBeeperLine();
   turnLeft();
   putBeeperLine();
}
```



Climbing Mountains

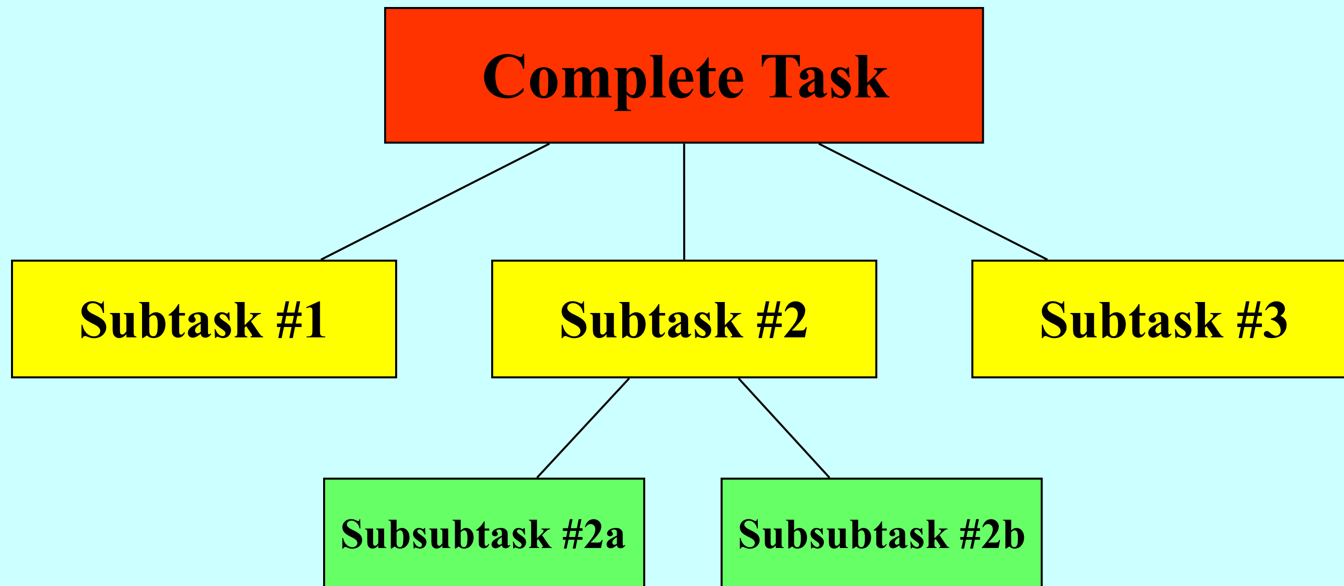
- Our next task explores the use of functions and control statements in the context of teaching Karel to climb stair-step mountains that look something like this:



- Our first program will work only in a particular world, but the goal is to have Karel be able to climb any stair-step mountain.

Stepwise Refinement

- The most effective way to solve a complex problem is to break it down into successively simpler subproblems.
- You start by breaking the whole task down into simpler parts.
- Some of those tasks may themselves need subdivision.
- This process is called ***stepwise refinement*** or ***decomposition***.



Criteria for Choosing a Decomposition

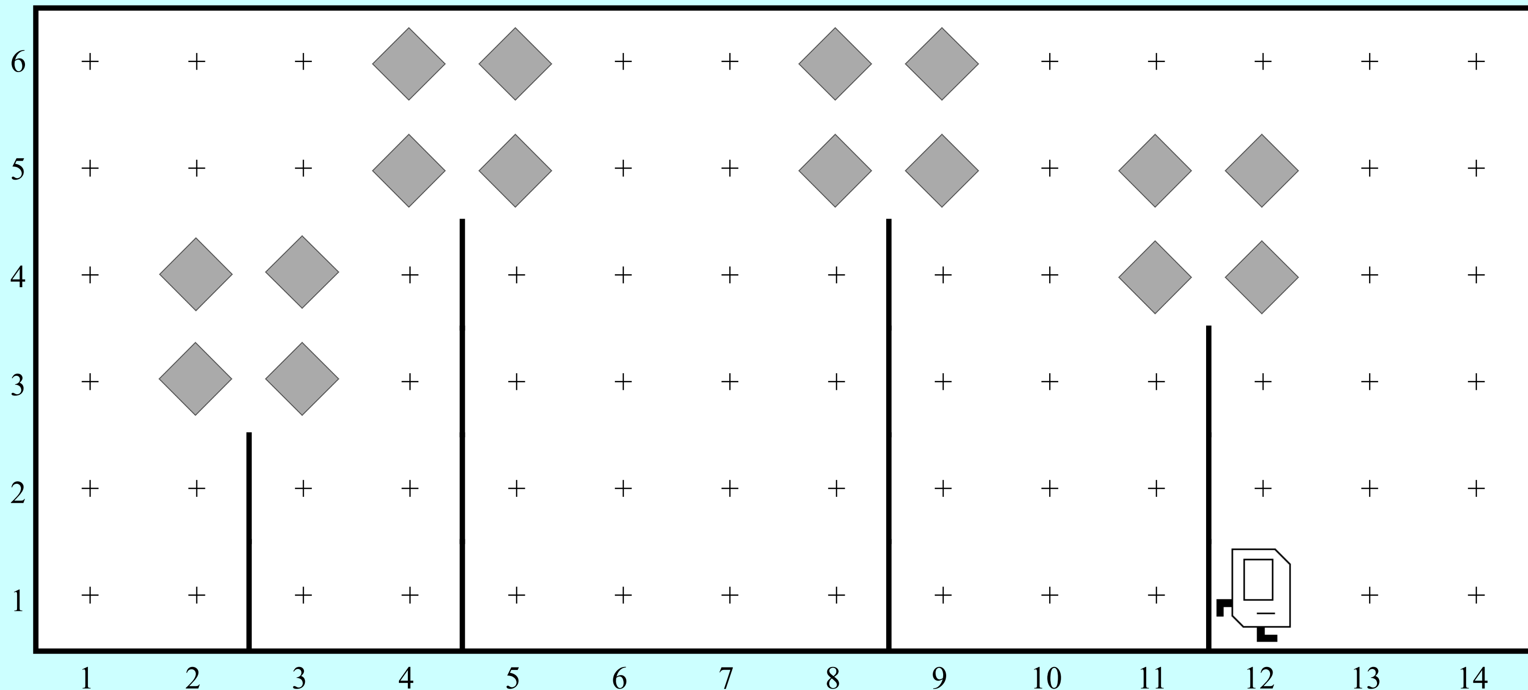
1. ***The proposed steps should be easy to explain.*** One indication that you have succeeded is being able to find simple names.

2. ***The steps should be as general as possible.*** Programming tools get reused all the time. If your functions perform general tasks, they are much easier to reuse.

3. ***The steps should make sense at the level of abstraction at which they are used.*** If you have a function that does the right job but whose name doesn't make sense in the context of the problem, it is probably worth defining a new function that calls the old one.

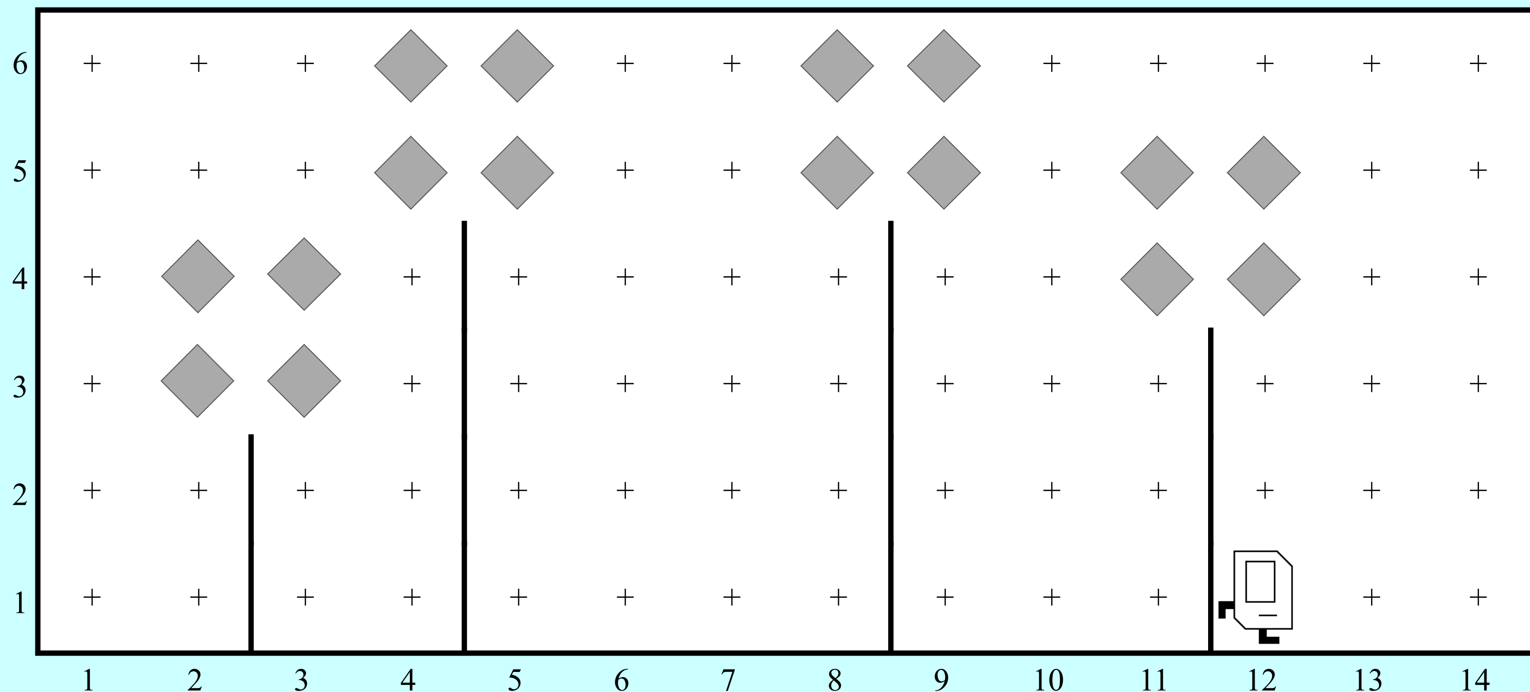
Exercise: Banishing Winter

- In this problem, Karel is supposed to usher in springtime by placing bundles of leaves at the top of each “tree” in the world.
- Given this initial world, the final state should look like this:



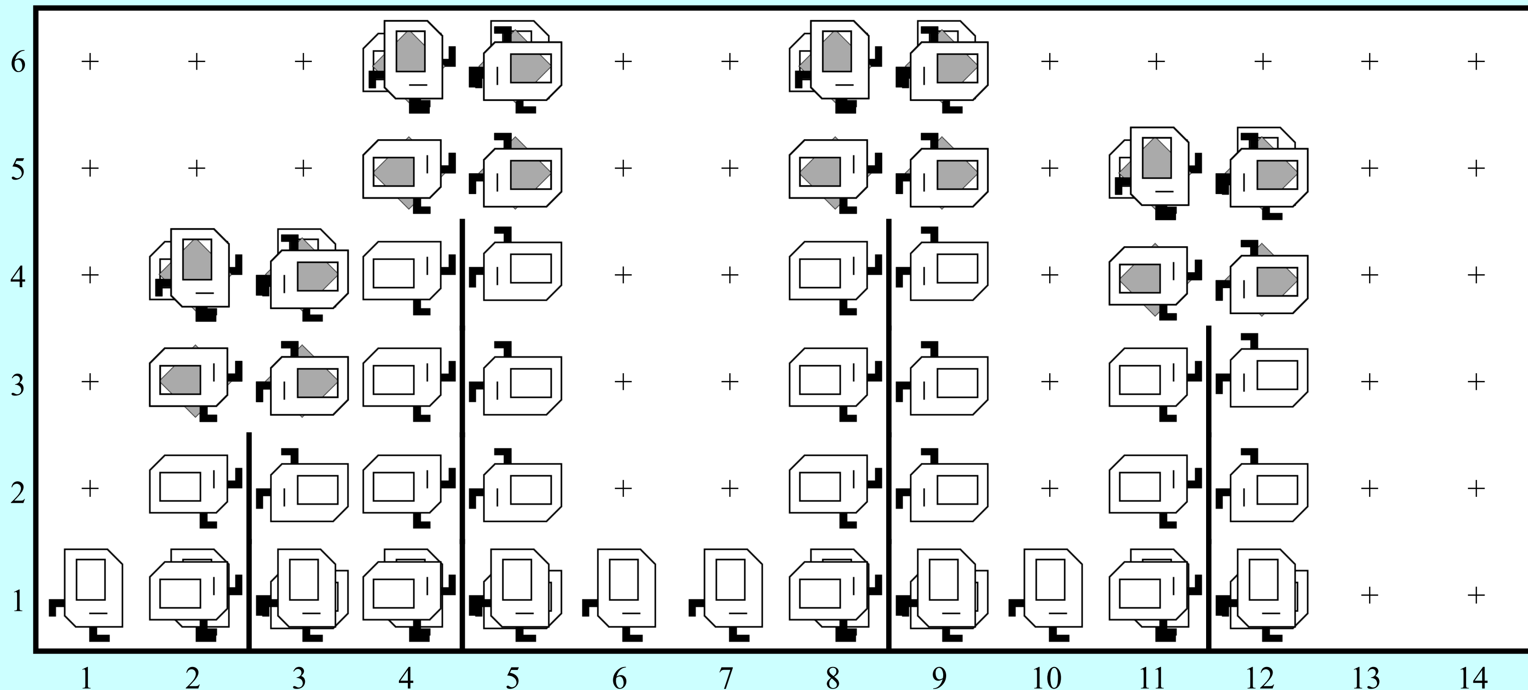
Understanding the Problem

- One of the first things you need to do given a problem of this sort is to make sure you understand all the details.
- In this problem, it is easiest to have Karel stop when it runs out of beepers. Why can't it just stop at the end of 1st Street?



The Top-Level Decomposition

- You can break this program down into two tasks that are executed repeatedly:
  - ☞ – Find the next tree.
  - ☞ – Decorate that tree with leaves.



BASIC

```
**** COMMODORE 64 BASIC V2 ****
64K RAM SYSTEM  38911 BASIC BYTES FREE
READY.
10 FOR I=0 TO 5
20 PRINT "HELLO, WORLD!"
30 NEXT I
RUN
HELLO, WORLD!
HELLO, WORLD!
HELLO, WORLD!
HELLO, WORLD!
HELLO, WORLD!
HELLO, WORLD!
READY.
```

The BASIC language, created in 1964 by John G. Kemeny and Thomas E. Kurtz at Dartmouth, was *the* programming language that was included with home computers during the 1970s and 1980s.

The language was usually included in ROM, so that when the computer booted up, users could immediately start programming.

BASIC

BASIC was designed so that students in non-scientific fields could learn to program.

Students learned BASIC in school, or by reading books that had listing of programs (often games) that they could type in relatively quickly.

BEGINNER PROGRAM

Commodore 64/Age Splitter

```
10 PRINT CHR$(147);
20 PRINT "TYPE YOUR ANSWER; THEN PRESS <RETURN>."
30 PRINT
40 PRINT "HOW MANY YEARS OLD ARE YOU";
50 INPUT AGE
60 PRINT CHR$(147);
70 PRINT "IF YOU ARE";AGE;"YEARS OLD,"
80 PRINT "YOU HAVE LIVED MORE THAN ..."
90 PRINT
100 PRINT AGE*12;"MONTHS, OR"
110 PRINT AGE*52;"WEEKS, OR"
120 PRINT AGE*365;"DAYS, OR"
130 PRINT AGE*365*24;"HOURS, OR"
140 PRINT AGE*365*24*60;"MINUTES, OR"
150 PRINT AGE*365*24*60*60;"SECONDS."
160 PRINT
170 PRINT "PRESS <P> TO PLAY AGAIN, OR <Q> TO QUIT."
180 GET K$
190 IF K$="P" THEN 10
200 IF K$<>"Q" THEN 180
210 END
```

IBM PCs/Age Splitter

```
10 KEY OFF
20 CLS
30 PRINT "TYPE YOUR ANSWER; THEN PRESS <ENTER>."
40 PRINT
50 PRINT "HOW MANY YEARS OLD ARE YOU";
60 INPUT AGE
70 CLS
80 PRINT "IF YOU ARE";AGE;"YEARS OLD,"
90 PRINT "YOU HAVE LIVED MORE THAN ..."
100 PRINT
110 PRINT AGE*12;"MONTHS, OR"
120 PRINT AGE*52;"WEEKS, OR"
130 PRINT AGE*365;"DAYS, OR"
140 PRINT AGE*365*24;"HOURS, OR"
150 PRINT AGE*365*24*60;"MINUTES, OR"
160 PRINT AGE*365*24*60*60;"SECONDS."
170 PRINT
180 PRINT "PRESS <P> TO PLAY AGAIN,"
190 PRINT "OR <Q> TO QUIT."
200 K$=INKEY$
210 IF K$="P" THEN 20
220 IF K$<>"Q" THEN 200
230 END
```

TRS-80 Color Computer/Age Splitter

```
10 CLS
20 PRINT "TYPE YOUR ANSWER;"
30 PRINT "THEN PRESS <ENTER>."
40 PRINT
50 PRINT "HOW MANY YEARS OLD ARE YOU";
60 INPUT AGE
70 CLS
80 PRINT "IF YOU ARE";AGE;"YEARS OLD,"
90 PRINT "YOU HAVE LIVED MORE THAN ..."
100 PRINT
110 PRINT AGE*12;"MONTHS, OR"
120 PRINT AGE*52;"WEEKS, OR"
130 PRINT AGE*365;"DAYS, OR"
140 PRINT AGE*365*24;"HOURS, OR"
150 PRINT AGE*365*24*60;"MINUTES, OR"
160 PRINT AGE*365*24*60*60;"SECONDS."
170 PRINT
180 PRINT "PRESS <P> TO PLAY AGAIN,"
190 PRINT "OR <Q> TO QUIT."
200 K$=INKEY$
210 IF K$="P" THEN 10
220 IF K$<>"Q" THEN 200
230 END
```

TRS-80 Model III/Age Splitter

```
10 CLS
20 PRINT "TYPE YOUR ANSWER; THEN PRESS <ENTER>."
30 PRINT
40 PRINT "HOW MANY YEARS OLD ARE YOU";
50 INPUT AGE
60 CLS
70 PRINT "IF YOU ARE";AGE;"YEARS OLD, YOU HAVE LIVED M
ORE THAN ..."
80 PRINT
90 PRINT AGE*12;"MONTHS, OR"
100 PRINT AGE*52;"WEEKS, OR"
110 PRINT AGE*365;"DAYS, OR"
120 PRINT AGE*365*24;"HOURS, OR"
130 PRINT AGE*365*24*60;"MINUTES, OR"
140 PRINT AGE*365*24*60*60;"SECONDS."
150 PRINT
160 PRINT "PRESS <P> TO PLAY AGAIN, OR <Q> TO QUIT."
170 K$=INKEY$
180 IF K$="P" THEN 10
190 IF K$<>"Q" THEN 170
200 END
```

Family Computing Magazine, July 1985

BASIC

While BASIC was relatively easy to learn, and while it introduced millions of kids to programming, it is not a particularly good language (at least the 1980s version — today, Visual Basic is decent). It is not

Famously, Edsgar Dijkstra said, in 1975, "It is practically impossible to teach good programming to students that have had a prior exposure to BASIC: as potential programmers they are mentally mutilated beyond hope of regeneration."

Students who learned BASIC on their own do, indeed, have some trouble graduating to a *structured* language such as C, Java, Javascript, etc., but it is probably not as dire as Dijkstra led on.

The End