IS12 - Introduction to Programming

Lecture 6: Advanced Programming with Karel

Peter Brusilovsky
http://www2.sis.pitt.edu/~peterb/0012-051/

`if` inside `while`

- Using `if` inside `while` will help us to solve a large class of complex problems
- `While` determines the repeating pattern, `if` takes care about the situations where different things have to be done in different iterations
- Do not use `if` inside `while` of vice versa for redundant checking (Pattis:5.5)

Finding a Beeper in a Cell

```
beginning-of-execution
   while not-next-to-a-beeper do
   begin
        if front-is-clear then
               move;
        else
               turnleft;
   end;
   turnoff;
end-of-execution
```

- What is true at the beginning of every iteration?
- What is different for each subsequent iteration that makes it closer to the solution?

Odd Harvest Problem

- New situation

```
beginning-of-execution
   move;
   while next-to-a-beeper do begin
        harvest-1-row;
        go-to-next-row;
        if next-to-a-beeper then begin
               harvest-1-row;
               position-for-next;
        end;
   end;
   position-for-next;
   move;
   turnoff;
end-of-execution
```

- Old situation

```
beginning-of-execution
   move;
   while next-to-a-beeper do
   begin
        harvest-1-row;
        go-to-next-row;
        harvest-1-row;
        position-for-next;
   end;
   position-for-next;
   move;
   turnoff;
end-of-execution
```

Nested Loops in Rich Harvest

- Implicitly embedded while loops

```
define-new-instruction harvest-
   1-row as begin
      while next-to-a-beeper
      do begin
         pick-all-beepers;
         move;
      end;
      step-back;
end;
define-new-instruction pick-
   all-beepers as
   while next-to-a-beeper do
        pickbeeper;
```

- Explicitly embedded while loops

```
define-new-instruction harvest-1-
   row as begin
    while next-to-a-beeper do
   begin
        while next-to-a-beeper do
              pickbeeper;
        move;
     end;
     step-back;
end;
```

Recursion

- How to do problems that require repetitions?
  - Iteration (iterate, while)
  - Recursion
- Recursion happens when a command calls itself (directly or indirectly)
- Recursion programs may be less intuitive from the first sight, but often they are easier to write

Recursion 1

- Recursive solution

```
beginning-of-program
  define-new-instruction
   find-beeper as
     if not-next-to-a-beeper
        then begin
           move;
           find-beeper;
      end;
  beginning-of-execution
        find-beeper;
        turnoff;
  end-of-execution
end-of-program
```

- Iterative solution

```
beginning-of-program
   define-new-instruction find-
   beeper as
        while not-next-to-a-
   beeper do
                move;
   beginning-of-execution
        find-beeper;
        turnoff;
   end-of-execution
end-of-program
```

Anatomy of Recursion

- Recursive function usually contains an if statement that analyses several cases (at least 2) and take actions
- A case could be a *base case* if the problem in this case is solved directly
- A case is a *recursive case* if we need to make a recursive call
- A recursive call need to address a *simplified problem*

Lost Beeper Mine (Recursion 2)

- Find a mine that is as far from a beeper to the north as a beeper itself from us to the east

```
define-new-instruction find-mine as
   if next-to-a-beeper then
        turnleft
   else begin
        move;
        find-mine;
        move;
   end;
```

Base Case

Recursive Case

Move Beepers (Recursion 3)

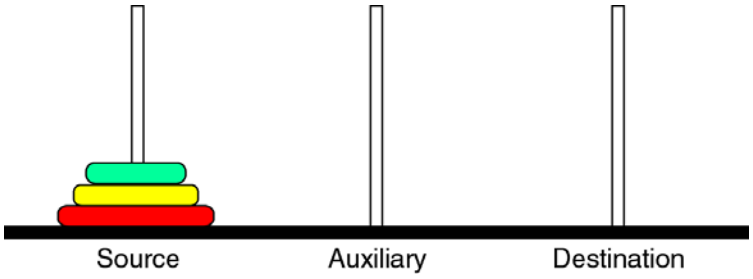
- Move a stack of beepers 3 blocks north

```
define-new-instruction move-beepers as
   if not-next-to-a-beeper then begin
        move;
        move;
        move;
   end
   else begin
        pickbeeper;
        move-beepers;
        putbeeper;
   end;
```

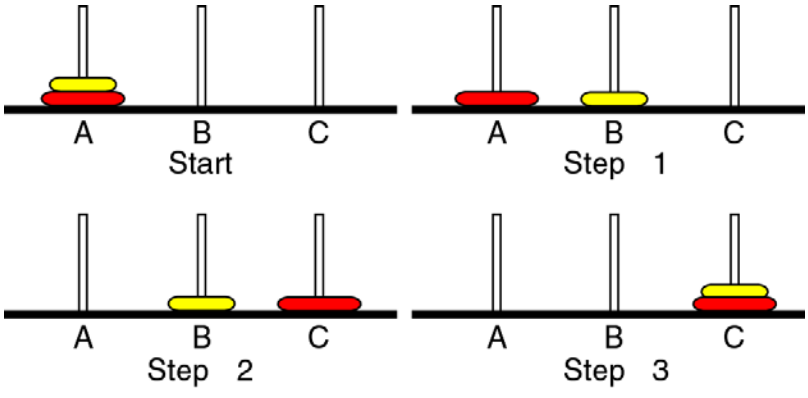
Base Case

Recursive Case

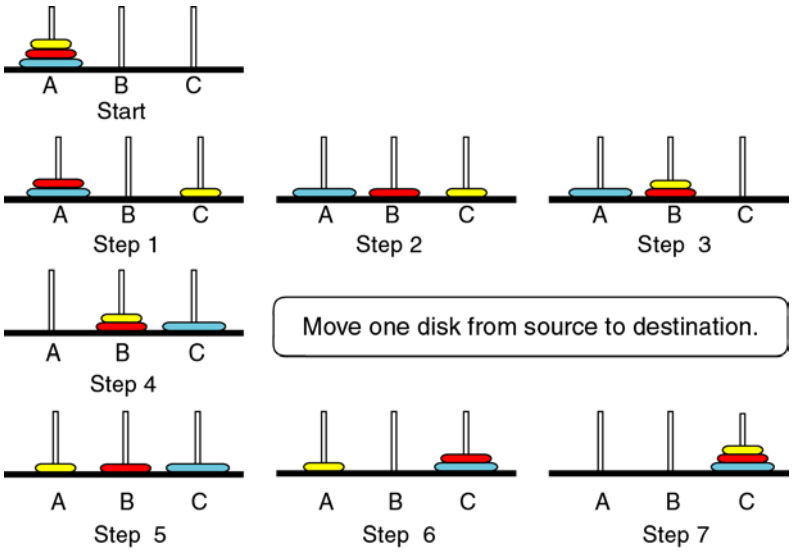
The Towers of Hanoi



Solution for two disks



Solution for three disks



Recursive algorithm

- Algorithm:
  - Move n-1 disks from source to auxiliary
  - Move one disk from source to destination
  - Move n-1 disks from auxiliary to destination
- Function towers(disks, sour, dest, aux)
  - towers(n-1, sour, aux, dest)
  - move one disk from sour to dest
  - towers(n-1, aux, dest, sour)

Calls:	Output:
Towers (3, A, C, B)	
Towers (2, A, B, C)	
Towers (1, A, C, B)	
	Step 1: Move from A to C
	Step 2: Move from A to B
Towers (1, C, B, A)	
	Step 3: Move from C to B
	Step 4: Move from A to C
Towers (2, B, C, A)	
Towers (1, B, A, C)	
	Step 5: Move from B to A
	Step 6: Move from B to C
Towers (1, A, C, B)	
	Step 7: Move from A to C

Before next lecture:

- Reading assignment
  - Pattis: Chapter 5, Sections 5.7-5.8; Chapter 6, Sections 6.1
- Run Classroom Examples
- Attempt exercises 16 and beyond from Section 5.9.
  - Try to re-write programs recursively