

Constructive Artificial Intelligence

Recursion, Examples and Exercises

Daniel Polani

School of Computer Science
University of Hertfordshire

October 30, 2009

All rights reserved. Permission is granted to copy and distribute these slides in full or in part for purposes of research, education as well as private use, provided that author, affiliation and this notice is retained.

Use as part of home- and coursework is only allowed with express permission by the responsible tutor and, in this case, is to be appropriately referenced.

Exercise 1

Assumptions

Consider the tower of Hanoi problem with three poles and n disks.

Task

Write a function `hanoi(.)` that moves a tower of size n from a given start pole to a target pole.

Exercise 2

Assumptions

Consider the tower of Hanoi problem with three poles and n disks.

Task

Write a function `hanoi(.)` that moves a tower of size n from a given start pole to a target pole.

Example

```
def hanoi(n, start, goal, ignore):  
    if n == 1:  
        print start, "->", goal  
        return  
  
    hanoi(n-1, start, ignore, goal)  
    print start, "->", goal  
    hanoi(n-1, ignore, goal, start)  
  
hanoi(4, 1, 2, 3)
```

Stateful “Functions” (Generators and Yields)

Functions:

- information processing, then return result
- a new call, new variables, new results

Generators:

- information processing, then yield result
- new call to generator retains old state and continues with new results

Example

```
def colours():  
    yield "red"  
    yield "green"  
    yield "black"  
  
for c in colours():  
    print c
```

Example

```
def colours():  
    for c in ["red", "green", "black"]:  
        yield c  
  
for c in colours():  
    print c
```

Exercise 3 (Permutation)

Assumptions

Consider a list [...] of elements.

Task

Write a generator `perm(.)` that produces all permutations of above list.

Exercise 4 (Permutation)

Assumptions

Consider a list [...] of elements.

Task

Write a generator `perm(.)` that produces all permutations of above list.

Example (Wrong! — Why?)

```
def insertion(e, s):
    for i in range(len(s)+1):
        s[i:i] = [e]
        yield s

def perm(s):
    if s == []:
        yield []
    else:
        e, s1 = s[0], s[1:]
        for s1p in perm(s1):
            for p in insertion(e,s1p):
                yield p

for p in perm([1,2,3,4]): print p
```

Exercise 5 (Permutation)

Assumptions

Consider a list [...] of elements.

Task

Write a generator `perm(.)` that produces all permutations of above list.

Example (Right! — Why?)

```
def insertion(e, s):
    for i in range(len(s)+1):
        yield s[:i] + [e] + s[i:]

def perm(s):
    if s == []:
        yield []
    else:
        e, s1 = s[0], s[1:]
        for s1p in perm(s1):
            for p in insertion(e,s1p):
                yield p

for p in perm([1,2,3,4]): print p
```

Warning: Applying Functions to Lists

Functions/generators can modify lists destructively!

Example

```
def modifier(s):  
    s[2:4] = ["foo", "bar"]  
  
s = [1,2,3,4,5,6]  
  
print "before:", s # [1,2,3,4,5,6]  
modifier(s)  
print "after:", s # [1,2,'foo','bar',5,6]
```

Warning: Applying Functions to Lists

Functions/generators can modify lists destructively!

Example

```
def modifier(s):  
    s[2:4] = ["foo", "bar"]  
  
s = [1,2,3,4,5,6]  
  
print "before:", s # [1,2,3,4,5,6]  
modifier(s)  
print "after:", s # [1,2,'foo','bar',5,6]
```

Example

```
def modifier(s):  
    s = s[:]  
    s[2:4] = ["foo", "bar"]  
  
s = [1,2,3,4,5,6]  
  
print "before:", s # [1,2,3,4,5,6]  
modifier(s)  
print "after:", s # [1,2,3,4,5,6]
```

Sheep and Cow Problem

Solution Part 1

E = 0

C = 1

S = 2

```
start_stable = [C, C, C, C, E, S, S, S, S]
```

```
goal_stable = [S, S, S, S, E, C, C, C, C]
```

Sheep and Cow Problem

Solution Part 2

```
def successors(stable):
    # find empty spot
    empty = stable.index(E)

    # generate list of unfiltered candidate positions
    candidates = [empty-2, empty-1, empty+1, empty+2]

    # keep only those which are inside the stable
    candidates = [c for c in candidates if c >= 0 and c < len(stable)]

    # Cows can always move right, Sheep always left, and from two fields
    # apart, they have to jump over an opposite animal

    candidates = [c for c in candidates if
        stable[c:c+2] == [C, E] or # cow can move right
        stable[c-1:c+1] == [E, S] or # sheep can move left
        stable[c:c+3] == [C, S, E] or # cow jumps over sheep
        stable[c-2:c+1] == [E, C, S]] # sheep jumps over cow

    # make sure that all these entries are occupied
    # (not necessary for operation, just better style)
    assert not [c for c in candidates if stable[c] == E]

    for c in candidates:
        new_stable = stable[:]          # make a copy

        # move the candidate into empty pos
        new_stable[c], new_stable[empty] = new_stable[empty], new_stable[c]

        yield new_stable                # remember where we were
```

Sheep and Cow Problem

Solution Part 3

```
def solution(stable):
    if stable == goal_stable:
        return [stable]

    # else, depth first
    for new_stable in successors(stable):
        # print new_stable
        sol = solution(new_stable)
        if sol:
            return [stable] + sol

for s in solution(start_stable):
    print s
```

Exercise 6 (Recursion: Ackermann Function)

Assumptions

Define the *Ackermann Function* as

$$A(m, n) := \begin{cases} n + 1 & \text{if } m = 0 \\ A(m - 1, 1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m - 1, A(m, n - 1)) & \text{if } m > 0 \text{ and } n > 0. \end{cases}$$

Task

- 1 Code the function in Python and calculate the values of $A(m, n)$ for various values of m and n .
- 2 Check its behaviour and why it's doing what it's doing.
- 3 Demonstrate that the base cases and the recursion step are well defined.