SICP section 4.3.2

January 05, 2008 at 17:38 Tags SICP

Exercise 4.38

The modification is done by commenting out this line:

```
(require (not (= (abs (- smith fletcher)) 1)))
```

And the interaction is:

```
;;; Amb-Eval input:
(multiple-dwelling)

;;; Starting a new problem
;;; Amb-Eval value:
((BAKER 1) (COOPER 2) (FLETCHER 4) (MILLER 3) (SMITH 5))

;;; Amb-Eval input:
try-again

;;; Amb-Eval value:
((BAKER 1) (COOPER 2) (FLETCHER 4) (MILLER 5) (SMITH 3))

;;; Amb-Eval input:
try-again

;;; Amb-Eval value:
((BAKER 1) (COOPER 4) (FLETCHER 2) (MILLER 5) (SMITH 3))

;;; Amb-Eval input:
try-again

;;; Amb-Eval value:
((BAKER 3) (COOPER 2) (FLETCHER 4) (MILLER 5) (SMITH 1))

;;; Amb-Eval input:
try-again

;;; Amb-Eval value:
((BAKER 3) (COOPER 4) (FLETCHER 2) (MILLER 5) (SMITH 1))

;;; Amb-Eval input:
try-again

;;; there are no more values of
(MULTIPLE-DWELLING)
```

There are 5 solutions without this constraint.

Exercise 4.39

The order of the restrictions does not affect the answer, because eventually the answer must comply to all of the restrictions anyway.

The runtime, however, is affected. For example, this version:

```
'(define (md2)
  (let ((baker (amb 1 2 3 4 5))
        (cooper (amb 1 2 3 4 5))
        (fletcher (amb 1 2 3 4 5))
        (miller (amb 1 2 3 4 5))
        (smith (amb 1 2 3 4 5)))
    (require (not (= cooper 1)))
    (require (not (= fletcher 1)))
    (require (not (= fletcher 5)))
    (require (not (= baker 5)))
    (require (> miller cooper))
    (require (not (= (abs (- smith fletcher)) 1)))
    (require (not (= (abs (- fletcher cooper)) 1)))
    (require
     (distinct? (list baker cooper fletcher miller smith)))
    (list (list 'baker baker)
          (list 'cooper cooper)
          (list 'fletcher fletcher)
          (list 'miller miller)
          (list 'smith smith)))))
```

Runs more than twice as fast as the one given in the book. Note that the requirement of distinctness was moved to the end. `distinct?` is a function that runs in quadratic time, and so moving it to the end saves us time because it is called less. When located first, it is continuously called until satisfied, on many possible 5-tuples. In the end, many possibilities were thrown out by the earlier restrictions and `distinct?` takes less time to run.

Exercise 4.40

The first improvement that can be made is to remove the “static” restrictions (such as `(not (= cooper 1))`), and not include those floors in the `amb` assignments in the first place:

```
'(define (md2)
  (let ((baker (amb 1 2 3 4))
        (cooper (amb 2 3 4 5))
        (fletcher (amb 2 3 4))
        (miller (amb 1 2 3 4 5))
        (smith (amb 1 2 3 4 5)))
    (require (> miller cooper))
    (require (not (= (abs (- smith fletcher)) 1)))
    (require (not (= (abs (- fletcher cooper)) 1)))
    (require
     (distinct? (list baker cooper fletcher miller smith)))
    (list (list 'baker baker)
          (list 'cooper cooper)
          (list 'fletcher fletcher)
          (list 'miller miller)
          (list 'smith smith)))))
```

This runs 3 times faster than even the fast version from exercise 4.39

An even further improvement can be made, however. We can suspend the generation of possible floors to people only until restrictions on this people appear:

```
'(define (md3)
  (let ((cooper (amb 2 3 4 5))
        (miller (amb 1 2 3 4 5)))
    (require (> miller cooper))
    (let ((fletcher (amb 2 3 4)))
      (require (not (= (abs (- fletcher cooper)) 1)))
      (let ((smith (amb 1 2 3 4 5)))
        (require (not (= (abs (- smith fletcher)) 1)))
        (let ((baker (amb 1 2 3 4)))
          (require
            (distinct? (list baker cooper fletcher miller smith)))
          (list (list 'baker baker)
          (list 'cooper cooper)
          (list 'fletcher fletcher)
          (list 'miller miller)
          (list 'smith smith)))))))
```

This runs much faster than the previous version. All improvements combined,`md3` runs aboiyt 30 times as fast as the original `multiple-dwelling`.

Exercise 4.41

Here's a naive implementation:

```
(defun baker (f) (first f))
(defun cooper (f) (second f))
(defun fletcher (f) (third f))
(defun miller (f) (fourth f))
(defun smith (f) (fifth f))

(deflex *constraints*
  (list
    #'(lambda (f)
        (/= (baker f) 5))
    #'(lambda (f)
        (/= (fletcher f) 5))
    #'(lambda (f)
        (/= (fletcher f) 1))
    #'(lambda (f)
        (/= (cooper f) 1))
    #'(lambda (f)
        (> (miller f) (cooper f)))
    #'(lambda (f)
        (/= (abs (- (smith f) (fletcher f))) 1))
    #'(lambda (f)
        (/= (abs (- (cooper f) (fletcher f))) 1))))

(let ((options (permutations '(1 2 3 4 5))))
  (dolist (constraint *constraints*)
    (setf options
      (remove-if-not constraint options)))

  (format t "~a~%" options))
=>
((3 2 4 5 1))
1
```

It first builds the list of all options, by using the`permutations` function, and then applies the constraints one by one, shortening the list of possibilities each time. `permutations` is defined as follows:

```
(defun permutations (bag)
  "Returns a list of all permutations of
  the input list"
  (if (null bag)
    '(())
    (mapcan
      #'(lambda (e)
          (mapcar
            #'(lambda (p) (cons e p))
            (permutations
              (remove e bag :count 1 :test #'eq))))
      bag)))
```

Note that the `distinct?` constraint is not required now, since `permutations` returns a list of distinct permutations.

Exercise 4.42

From the book it's not entirely clear *how* we're expected to solve this puzzle, but I assume the authors meant using `amb`. So, first let's add some utilities:

```
(interpret
  '(define (distinct? items)
    (cond ((null? items) true)
          ((null? (cdr items)) true)
          ((member (car items) (cdr items)) false)
          (else (distinct? (cdr items))))))
```

Additionally, the following primitive is added to the evaluator:

```
    (list 'xor (lambda (a b) (and (or a b) (not (and a b)))))
```

Now the problem formulation:

```
(interpret
  '(define (liars)
    (let ((betty (amb 1 2 3 4 5))
          (ethel (amb 1 2 3 4 5))
          (joan (amb 1 2 3 4 5))
          (kitty (amb 1 2 3 4 5))
          (mary (amb 1 2 3 4 5)))
    (require (distinct? (list betty ethel joan kitty mary)))
    (require (xor (= kitty 2) (= betty 3)))
    (require (xor (= ethel 1) (= joan 2)))
    (require (xor (= joan 3) (= ethel 5)))
    (require (xor (= kitty 2) (= mary 4)))
    (require (xor (= mary 4) (= betty 1)))
    (list (list 'betty betty)
   (list 'ethel ethel)
   (list 'joan joan)
   (list 'kitty kitty)
   (list 'mary mary)))))
```

Result:

```
(interpret '(liars)) =>
((betty 3) (ethel 5) (joan 2) (kitty 1) (mary 4))
```

Note: This answer is due to Brian Maissy. It appears correct but throws a stack overflow with my CLISP (which, as I've mentioned before) isn't too great with deep recursions. If you manage to reproduce the result or make it run in CLISP or another system, drop me a line.

Exercise 4.43

As always, I will begin with the most naive solution, putting efficiency aside for a moment. Once I have a working correct version, I can optimize it.

Perhaps there are several ways to represent the data set of this puzzle, but I chose the variables to be the girls' last

names:

```
(interpret
  '(define (yachts)
    (let ((gabrielle (amb 'moore 'downing 'hall 'barnacle 'parker))
          (lorna (amb 'moore 'downing 'hall 'barnacle 'parker))
          (rosalind (amb 'moore 'downing 'hall 'barnacle 'parker))
          (melissa (amb 'moore 'downing 'hall 'barnacle 'parker))
          (maryann (amb 'moore 'downing 'hall 'barnacle 'parker)))
      (require (not (eq? gabrielle 'barnacle)))
      (require (not (eq? lorna 'moore)))
      (require (not (eq? rosalind 'hall)))
      (require (eq? melissa 'barnacle))
      (require (eq? maryann 'moore))
      (require
        (cond ((eq? gabrielle 'moore) (eq? lorna 'parker))
              ((eq? gabrielle 'downing) (eq? melissa 'parker))
              ((eq? gabrielle 'hall) (eq? rosalind 'parker))
              (else false)))
      (require
        (distinct? (list gabrielle lorna rosalind melissa maryann)))
      (list (list 'gabrielle gabrielle)
            (list 'lorna lorna)
            (list 'rosalind rosalind)
            (list 'melissa melissa)
            (list 'maryann maryann)))))
```

With this representation, all the constrains given in the puzzle are simple, except the last one, which gave me quite some trouble.

> Gabrielle's father owns the yacht that is named after Dr. Parker's daughter.

This constraint is complex, because it intermixes a lot of the variables. To translate it into a constraint on the variables I've chosen, I had to first apply some logic in my head. To see why it translates to:

```
(require
  (cond ((eq? gabrielle 'moore) (eq? lorna 'parker))
        ((eq? gabrielle 'downing) (eq? melissa 'parker))
        ((eq? gabrielle 'hall) (eq? rosalind 'parker))
        (else false)))
```

Consider that the puzzle gives us the names of the yachts of all the friends[1]. Since this constraint talks about Gabrielle's father's yacht, we can infer the name of Parker's daughter by keeping in mind who owns which yacht.

Anyway, running this I get the result:

```
 ((GABRIELLE HALL)
  (LORNA DOWNING)
  (ROSALIND PARKER)
  (MELISSA BARNACLE)
  (MARYANN MOORE))
```

So the answer is: "Lorna's father is Colonel Downing". If we remove the constraint on Mary Ann's last names, we get two solutions: the one we got with the constraint, and this one:

```
 ((GABRIELLE MOORE)
  (LORNA PARKER)
  (ROSALIND DOWNING)
  (MELISSA BARNACLE)
  (MARYANN HALL))
```

Now, on to the optimization. Measuring the speed of the original implementation first:

```
* (time (dotimes (i 100 t) (interpret '(yachts))))

Evaluation took:
  2.875 seconds of real time
  2.78125 seconds of user run time
  0.03125 seconds of system run time
  [Run times include 0.109 seconds GC run time.]
  0 calls to %EVAL
  0 page faults and
  322,418,992 bytes consed.
```

I'll remove all the static constraints, and instead will list only partial possibility lists in the first place. For instance, it is very inefficient to first set `maryann` to `(amb 'moore 'downing 'hall 'barnacle 'parker)` and then require `(eq? maryann 'moore)`. It is much better to just set `maryann` to `moore` in the first place. So the optimized version is:

```
(interpret
  '(define (yachts2)
    (let ((gabrielle (amb 'moore 'downing 'hall 'parker))
          (lorna (amb 'downing 'hall 'barnacle 'parker))
          (rosalind (amb 'moore 'downing 'barnacle 'parker))
          (melissa 'barnacle)
          (maryann 'moore))
      (require
        (cond ((eq? gabrielle 'moore) (eq? lorna 'parker))
              ((eq? gabrielle 'downing) (eq? melissa 'parker))
              ((eq? gabrielle 'hall) (eq? rosalind 'parker))
              (else false)))
      (require
        (distinct? (list gabrielle lorna rosalind melissa maryann)))
      (list (list 'gabrielle gabrielle)
            (list 'lorna lorna)
            (list 'rosalind rosalind)
            (list 'melissa melissa)
            (list 'maryann maryann)))))
```

And it runs *much* faster:

```
* (time (dotimes (i 100 t) (interpret '(yachts2))))

Evaluation took:
  0.078 seconds of real time
  0.0625 seconds of user run time
  0.015625 seconds of system run time
  [Run times include 0.016 seconds GC run time.]
  0 calls to %EVAL
  0 page faults and
  6,586,888 bytes consed.
```

A 46-fold improvement. Not bad!

Exercise 4.44

First let's build some scaffolding. This function will enumerate the sequence from 1 to `n`:

```
(interpret
  '(define (enumerate-seq n)
    (define (enum-seq-iter seq)
      (if (> seq n)
        '()
        (cons seq
              (enum-seq-iter (+ seq 1)))))
    (enum-seq-iter 1)))
```

And this one is a list version of `amb`. Instead of picking one of its arguments, it picks one of the elements from the list it's given:

```
(interpret
  '(define (list-amb lst)
    (if (null? lst)
      (amb)
      (amb (car lst) (list-amb (cdr lst))))))
```

Now to chess-specific functions:

```
(interpret
  '(define (nth i lst)
    (cond ((null? lst) '())
          ((= i 0) (car lst))
          (else (nth (- i 1) (cdr lst))))))

(interpret
  '(define (attacks? row1 col1 row2 col2)
    (cond ((= row1 row2) true)
          ((= col1 col2) true)
          ((= (abs (- col1 col2))
              (abs (- row1 row2))) true)
          (else false))))

(interpret
  '(define (safe-kth? k pos)
    (let ((kth-col (nth k pos))
          (pos-len (length pos)))
      (define (safe-iter i)
        (cond ((= i pos-len) true)
              ((= i k) (safe-iter (+ i 1)))
              (else
                (let ((ith-col (nth i pos)))
                  (if (attacks? i ith-col k kth-col)
                    false
                    (safe-iter (+ i 1)))))))
      (safe-iter 0))))
```

The most important function is `safe-kth?`. It is given a row number and a position and finds out whether the queen in the given row is safe in relation to all the other queens. The position is a list of columns. For example: `(1 3 5 4 7 8 6 2)` means there queen in the first row is in column 1, the queen in the second row is in column 3, etc. The `k` given to `safe-kth?` is 0-based.

Finally:

```
(interpret
  '(define (queens n)
    (define (queens-iter pos i)
      (cond ((> i (- n 1)) pos)
            (else
              (let ((new-col (list-amb (enumerate-seq n))))
                (let ((new-pos (append pos (list new-col))))
                  (require (safe-kth? i new-pos))
                  (queens-iter new-pos (+ i 1)))))))
    (queens-iter '() 0)))
```

Will provide the solutions for the n-queens problem for any `n`. The algorithm is simple: generate a new queen in the next row, one at a time, and check if it's safe in relation to all the other queens. Compare this to the solution of exercise 2.42 – `amb` replaces all the complex filtering done there.

Concluding – usage of `amb` for puzzles

As we've seen, `amb` is very handy for solving puzzles like the ones presented in this section so far. `amb`'s main usefullness comes from its including a search on all possibilities, implicitly. To see how cumbersome it is to perform this search explicitly, compare with the solution of exercise 4.41

Like many other constructs presented in the book, `amb` is a language abstraction that makes a complex idea transparent

and easy to use. In what follows, we'll use it for quite a different purpose – parsing natural language.

Exercise 4.45

```
;;; Amb-Eval input:
(parse '(the professor lectures to the student in the class with the cat))

;;; Starting a new problem
;;; Amb-Eval value:
(SENTENCE (SIMPLE-NOUN-PHRASE (ARTICLE THE) (NOUN PROFESSOR))
 (VERB-PHRASE
  (VERB-PHRASE
   (VERB-PHRASE (VERB LECTURES)
    (PREP-PHASE (PREP TO) (SIMPLE-NOUN-PHRASE (ARTICLE THE) (NOUN STUDENT))))
   (PREP-PHASE (PREP IN) (SIMPLE-NOUN-PHRASE (ARTICLE THE) (NOUN CLASS))))
  (PREP-PHASE (PREP WITH) (SIMPLE-NOUN-PHRASE (ARTICLE THE) (NOUN CAT)))))

;;; Amb-Eval input:
try-again

;;; Amb-Eval value:
(SENTENCE (SIMPLE-NOUN-PHRASE (ARTICLE THE) (NOUN PROFESSOR))
 (VERB-PHRASE
  (VERB-PHRASE (VERB LECTURES)
   (PREP-PHASE (PREP TO) (SIMPLE-NOUN-PHRASE (ARTICLE THE) (NOUN STUDENT))))
  (PREP-PHASE (PREP IN)
   (NOUN-PHRASE (SIMPLE-NOUN-PHRASE (ARTICLE THE) (NOUN CLASS))
    (PREP-PHASE (PREP WITH) (SIMPLE-NOUN-PHRASE (ARTICLE THE) (NOUN CAT)))))))

;;; Amb-Eval input:
try-again

;;; Amb-Eval value:
(SENTENCE (SIMPLE-NOUN-PHRASE (ARTICLE THE) (NOUN PROFESSOR))
 (VERB-PHRASE
  (VERB-PHRASE (VERB LECTURES)
   (PREP-PHASE (PREP TO)
    (NOUN-PHRASE (SIMPLE-NOUN-PHRASE (ARTICLE THE) (NOUN STUDENT))
     (PREP-PHASE (PREP IN) (SIMPLE-NOUN-PHRASE (ARTICLE THE) (NOUN CLASS))))))
  (PREP-PHASE (PREP WITH) (SIMPLE-NOUN-PHRASE (ARTICLE THE) (NOUN CAT)))))

;;; Amb-Eval input:
try-again

;;; Amb-Eval value:
(SENTENCE (SIMPLE-NOUN-PHRASE (ARTICLE THE) (NOUN PROFESSOR))
 (VERB-PHRASE (VERB LECTURES)
  (PREP-PHASE (PREP TO)
   (NOUN-PHRASE
    (NOUN-PHRASE (SIMPLE-NOUN-PHRASE (ARTICLE THE) (NOUN STUDENT))
     (PREP-PHASE (PREP IN) (SIMPLE-NOUN-PHRASE (ARTICLE THE) (NOUN CLASS))))
    (PREP-PHASE (PREP WITH) (SIMPLE-NOUN-PHRASE (ARTICLE THE) (NOUN CAT)))))))

;;; Amb-Eval input:
try-again

;;; Amb-Eval value:
(SENTENCE (SIMPLE-NOUN-PHRASE (ARTICLE THE) (NOUN PROFESSOR))
 (VERB-PHRASE (VERB LECTURES)
  (PREP-PHASE (PREP TO)
   (NOUN-PHRASE (SIMPLE-NOUN-PHRASE (ARTICLE THE) (NOUN STUDENT))
    (PREP-PHASE (PREP IN)
     (NOUN-PHRASE (SIMPLE-NOUN-PHRASE (ARTICLE THE) (NOUN CLASS))
      (PREP-PHASE (PREP WITH)
       (SIMPLE-NOUN-PHRASE (ARTICLE THE) (NOUN CAT)))))))))
```

```
;;; Amb-Eval input:
try-again

;;; there are no more values of
(PARSE '(THE PROFESSOR LECTURES TO THE STUDENT IN THE CLASS WITH THE CAT))
```

Exercise 4.46

`parse-word` advances the `*unparsed*` list from left to right, because it knows that the parser works this way. Without being sure of the order, we couldn't implement `parse-word`.

I'm not 100% sure of my answer to this exercise. If you have a better one, let me know.

Exercise 4.47

Yes, the change Louis suggests works. However, when we interchange the order of expressions in the `amb`:

```
(interpret
  '(define (parse-verb-phrase)
    (amb
      (list 'verb-phrase
            (parse-verb-phrase)
            (parse-prepositional-phrase)))
      (parse-word verbs)))
```

The program enters an infinite loop. This can be explained by considering how `amb` works. Whenever it's called, it first tries to pick its first option. However, note that in this case picking the first option means calling `amb` again, with the same first option. This is a recursion without a stop condition.

Exercise 4.48

I'll add the possibility to prepend adjectives to nouns:

```
(interpret
  '(define adjectives '(adjective gray tired nice)))

(interpret
  '(define (parse-simple-noun-phrase)
    (amb
      (list 'simple-noun-phrase
            (parse-word articles)
            (parse-word nouns))
      (list 'adjective-noun-phrase
            (parse-word articles)
            (parse-word adjectives)
            (parse-word nouns)))))
```

Now calling:

```
(interpret
  '(parse '(The tired professor lectures to a gray cat)))
=>
(SENTENCE (ADJECTIVE-NOUN-PHRASE (ARTICLE THE) (ADJECTIVE TIRED) (NOUN PROFESSOR))
 (VERB-PHRASE (VERB LECTURES)
  (PREP-PHASE (PREP TO) (ADJECTIVE-NOUN-PHRASE (ARTICLE A) (ADJECTIVE GRAY) (NOUN CAT)))))
```

Exercise 4.49

I'll change `parse-word` to the following:

```
(interpret
  '(define (parse-word word-list)
    (require (not (null? *unparsed*)))
    (let ((found-word (random-list-element (cdr word-list))))
      (set! *unparsed* (cdr *unparsed*))
      (list (car word-list) found-word))))
```

With these support functions:

```
(interpret
  '(define (nth i lst)
    (cond ((null? lst) '())
          ((= i 0) (car lst))
          (else (nth (- i 1) (cdr lst))))))

(interpret
  '(define (random-list-element lst)
    (nth (random (length lst)) lst)))
```

Note that I "loaned" `random` from Common Lisp's library[2]. Now I can provide a "template" and random sentences will be generated:

```
(parse '(The professor lectures))
=>
(SENTENCE (SIMPLE-NOUN-PHRASE (ARTICLE A) (NOUN CAT)) (VERB STUDIES))
...
(SENTENCE (SIMPLE-NOUN-PHRASE (ARTICLE THE) (NOUN STUDENT)) (VERB SLEEPS))
...
(SENTENCE (SIMPLE-NOUN-PHRASE (ARTICLE THE) (NOUN PROFESSOR)) (VERB LECTURES))
...
```

[1] Except Parker's yacht, but that can be inferred by elimination to be Mary Ann.

[2] I always do this by attaching the function I want to the list of `primitive-procedures` in the core `evaluator`.

Back to top