**18. Nondeterminism**

**1. Introduction**

Nondeterminism is a algorithm to solve a problem by just defining it. The nondeterminism program selects choices automatically that match the conditions. This technique suits logical programing.

For instance, following code returns a pear of two numbers in that sum of them is a prime number. One number is selected from '(4 6 7) and the other from '(5 8 11),

(let ((i (amb 4 6 7))

(j (amb 5 8 11)))

(if (prime? (+ i j))

(list i j)

(amb)))

;Value 23: (6 5)

(amb 4 6 7) returns a proper value from 4, 6, and 7 and (amb 5 8 11) returns a proper value from 5, 8, and 11. (amb) represents the false as there is no value to be selected.

Actually, amb performs a depth first search. In practice, (amb c1 c2 c3 ...) makes a searching path that checks c1, c2, c3, ... in order and (amb) backtracks. Thus, nondeterminism is a abstraction that hide searching from programmers. Once we get amb, we can write logical programs easily without thinking what computers do.

**2. Implementation of nondeterminism**

The backtrack used in the nondeterminism is implemented as a chain of closures which is connected by current continuation. The chain is represented by a global parameter **fail** which is a function that rewrite itself.

**2.1. Implementation as a function**

As a first step, I implement the nondeterminism using a function (named **choose**) as shown in [code 1]. First I define a global parameter **fail**, whose initial value is a function that return 'no-choice to the top level (lines 22 – 26). Then the chain of closure is implemented by re-defining **fail** in the functionchoose. Backtrack is represented by calling previous **fail**.

The function choose behaves like as follows:

1. if no choice, call (fail).
2. if any,
   1. stores the fail as fail0 and call the current continuation.
   2. re-defines the fail in the continuation. The new fail get back itself to the fail0 and apply choose to the rest of choices.
   3. returns the first choice to the out side of the continuation.

[code 1]

01: ;;; abbreviation for call-with-current-continuation

02: (define call/cc call-with-current-continuation)

03:

04: ;;; This function is re-assigned in `choose' and `fail' itself.

05: (define fail #f)

06:

07: ;;; function for nondeterminism

08: (define (choose . ls)

09: (if (null? ls)

10: (fail)

11: (let ((fail0 fail))

12: (call/cc

13: (lambda (cc)

14: (set! fail

15: (lambda ()

16: (set! fail fail0)

17: (cc (apply choose (cdr ls)))))

18: (cc (car ls)))))))

19:

20: ;;; write following at the end of file

21: ;;; initial value for fail

22: (call/cc

23: (lambda (cc)

24: (set! fail

25: (lambda ()

26: (cc 'no-choice)))))

Let's see if the choose can find a Pythagorean triple. Function pythag is to find the triple. It returns the list if it find, otherwise calls choose with no argument to backtrack.

[sample 1]

(define (sq x)

(\* x x))

;Value: sq

;;; Pythagorean triples

(define (pythag a b c)

(if (= (+ (sq a) (sq b)) (sq c))

(list a b c)

(choose)))

;Value: pythag

(pythag (choose 1 2 3) (choose 3 4 5) (choose 4 5 6))

;Value 16: (3 4 5)

**2.2. Implementation as a macro**

To give S-expressions to the nondeterminism operator, the operator should be defined as a macro. For instance, while a function an-integer-starting-from shown in [sample 2] should be return a proper integer greater than or equal to **n**, it does not work because the argument is evaluated immediately if choose (defined as function) is used to define the function.

[sample 2]

(define (an-integer-starting-from n)

(choose n (an-integer-starting-from (1+ n))))

;Value: an-integer-starting-from

(an-integer-starting-from 1)

;Aborting!: maximum recursion depth exceeded

To achieve this, let's define a nondeterminism macro **amb** with consulting the definition of choose shown in [code 1]. The macro amb has the same structure as that of choose and calls itself recursively.

Lines 1 – 5 and 20 – 26 in the [code 1] are reused in the following codes.

When the [code 2] is compiled using the MIT-Scheme, the compiler warns like:

;Warning: Possible inapplicable operator ()

But it works properly. The code works also on the [Petite Chez Scheme](http://www.scheme.com/petitechezscheme.html). Even I have not tried other Scheme implementation, this definition of the amb could work on them if they follow R5RS. You can download a MIT-Scheme specific definition of the amb from [here](http://www.shido.info/lisp/scheme_amb.zip). The compiler of the MIT-Scheme does not warn on the specific definition.

[code 2]

01: ;;; nondeterminism macro operator

02: (define-syntax amb

03: (syntax-rules ()

04: ((\_) (fail))

05: ((\_ a) a)

06: ((\_ a b ...)

07: (let ((fail0 fail))

08: (call/cc

09: (lambda (cc)

10: (set! fail

11: (lambda ()

12: (set! fail fail0)

13: (cc (amb b ...))))

14: (cc a)))))))

The macro definition, amb, works properly for arguments of S-expressions as well as those of values.

[sample 3]

(define (an-integer-starting-from n)

(amb n (an-integer-starting-from (1+ n))))

;Value: an-integer-starting-from

(an-integer-starting-from 1)

;Value: 1

(amb)

;Value: 2

(amb)

;Value: 3

Implementations of amb in [Teach Yourself Scheme in Fixnum Days](http://www.ccs.neu.edu/home/dorai/t-y-scheme/t-y-scheme-Z-H-16.html#node_sec_14.2) or [Dave Herman Code](http://www.ccs.neu.edu/home/dherman/code/amb.ss) expand the parameter using ',@(map .... )'. Even they are straight forward definition, they are somehow complicated because they use call/cc twice. The recursive definition shown in [code 2] is simpler, even the expanded S-expression becomes some how complicated.

**2.3. Utilities for logical programing**

[code 3] shows utilities for logical programing, which makes the programs concise.

[code 3]

01: ;;; returning all possibilities

02: (define-syntax set-of

03: (syntax-rules ()

04: ((\_ s)

05: (let ((acc '()))

06: (amb (let ((v s))

07: (set! acc (cons v acc))

08: (fail))

09: (reverse! acc))))))

10:

11: ;;; if not pred backtrack

12: (define (assert pred)

13: (or pred (amb)))

14:

15: ;;; returns arbitrary number larger or equal to n

16: (define (an-integer-starting-from n)

17: (amb n (an-integer-starting-from (1+ n))))

18:

19: ;;; returns arbitrary number between a and b

20: (define (number-between a b)

21: (let loop ((i a))

22: (if (> i b)

23: (amb)

24: (amb i (loop (1+ i))))))

**(set-of s)**

It returns all the possibilities that satisfy **s**. The macro behaves like as follows:

1. (Line 5) A list (acc) is defined, which holds the result that satisfies **s**.
2. (Line 6) The result of **s** is assigned to **v** and is pushed to **acc**. If the result is pushed directly without **v** (like (set! acc (cons s acc))), Only the last value is stored in the acc because **s** uses a continuation. The **s** changes the value of fail.
3. (Lines 7,8) After that, it backtracks by calling fail. The function fail behaves as if it is called at line 6 because it uses call/cc.
4. (Line 9) When all the possible choices are found, it calls (reverse! acc) and returns the all the possibilities.

The definition assume that the amb searches from the leftmost argument.

**(assert pred)**

It backtracks if **pred** is not satisfied.

**(an-integer-starting-from n)**

It returns integers starting from **n** nondeterminately.

**(number-between a b)**

It returns integers between **a** and **b** nondeterminately.

[example 4] shows how to to use the set-of. All the prime numbers below 20 are obtained.

[example 4]

(define (prime? n)

(let ((m (sqrt n)))

(let loop ((i 2))

(or (< m i)

(and (not (zero? (modulo n i)))

(loop (+ i (if (= i 2) 1 2))))))))

(define (gen-prime n)

(let ((i (number-between 2 n)))

(assert (prime? i))

i))

(set-of (gen-prime 20))

;Value 12: (2 3 5 7 11 13 17 19)

**3. An example of logical programmings**

Let's solve [the problem 4.42. in the SIPS](http://mitpress.mit.edu/sicp/full-text/book/book-Z-H-28.html#%_sec_4.3.2) as an example. The question is like as follows:

Five school girls took an exam. As they think that their parents are too much interested in their score, they promise that they write one correct and one wrong informations to their parents. Followings are parts of their letters concerning their result:

|  |  |
| --- | --- |
| Betty: | Kitty was the second and I third. |
| Ethel: | I won the top and Joan the second. |
| Joan: | I was the third and poor Ethel the last. |
| Kitty: | I was the second and Mary the fourth. |
| Mary: | I was the fourth. Betty won the top. |

Guess the real order of the five school girls.

[code 4] shows the program to solve the problem.

[code 4]

01: (define (xor a b)

02: (if a (not b) b))

03:

04: (define (all-different? . ls)

05: (let loop ((obj (car ls)) (ls (cdr ls)))

06: (or (null? ls)

07: (and (not (memv obj ls))

08: (loop (car ls) (cdr ls))))))

09:

10: ;;; SICP Exercise 4.42

11: (define (girls-exam)

12: (let ((kitty (number-between 1 5))

13: (betty (number-between 1 5)))

14: (assert (xor (= kitty 2) (= betty 3)))

15: (let ((mary (number-between 1 5)))

16: (assert (xor (= kitty 2) (= mary 4)))

17: (assert (xor (= mary 4) (= betty 1)))

18: (let ((ethel (number-between 1 5))

19: (joan (number-between 1 5)))

20: (assert (xor (= ethel 1) (= joan 2)))

21: (assert (xor (= joan 3) (= ethel 5)))

22: (assert (all-different? kitty betty ethel joan mary))

23: (map list '(kitty betty ethel joan mary) (list kitty betty ethel joan mary))))))

24:

25: ;;; Bad answer for ex 4.42

26: (define (girls-exam-x)

27: (let ((kitty (number-between 1 5))

28: (betty (number-between 1 5))

29: (mary (number-between 1 5))

30: (ethel (number-between 1 5))

31: (joan (number-between 1 5)))

32: (assert (xor (= kitty 2) (= betty 3)))

33: (assert (xor (= kitty 2) (= mary 4)))

34: (assert (xor (= mary 4) (= betty 1)))

35: (assert (xor (= ethel 1) (= joan 2)))

36: (assert (xor (= joan 3) (= ethel 5)))

37: (assert (all-different? kitty betty ethel joan mary))

38: (map list '(kitty betty ethel joan mary) (list kitty betty ethel joan mary))))

* **(xor a b)** returns #t when
  + **a** is #t and **b** #f or
  + **a** #f and **b** #t.

It returns #f otherwise.

* **(all-different? . ls)** returns #t when all the items in **ls** are different.
* **(girls-exam)** is the main function to solve the puzzle. It returns a list of names and positions. Calling assert step by step after assigning parameters prunes dead-end branches efficiently, which makes run time shorter. **(girls-exam-x)** is a bad example. It calls assert after assigning all the parameters. In such a case, numerous dead-end branches are searched. The run time of the (girls-exam-x) is more than ten times larger than that of (girls-exam) as shown in [sample 5].

[sample 5]

(define-syntax cpu-time/sec

(syntax-rules ()

((\_ s)

(with-timings

(lambda () s)

(lambda (run-time gc-time real-time)

(write (internal-time/ticks->seconds run-time))

(write-char #\space)

(write (internal-time/ticks->seconds gc-time))

(write-char #\space)

(write (internal-time/ticks->seconds real-time))

(newline))))))

;Value: cpu-time/sec

(cpu-time/sec (girls-exam))

.03 0. .03

;Value 14: ((kitty 1) (betty 3) (ethel 5) (joan 2) (mary 4))

(cpu-time/sec (girls-exam-x))

.341 .29 .631

;Value 15: ((kitty 1) (betty 3) (ethel 5) (joan 2) (mary 4))

**4. Summary**

You can write programs that seems to be second-sighted when you use nondeterminism and the technique is applied to logical programming and parsing. Notice that you cannot use the code shown in this chapter if the searching path has loops. See [SICP 4.3.](http://mitpress.mit.edu/sicp/full-text/book/book-Z-H-28.html#%_sec_4.3) for detailed information on this matter.

I consulted [Teach Yourself Scheme in Fixnum Days](http://www.ccs.neu.edu/home/dorai/t-y-scheme/t-y-scheme-Z-H-16.html) to write this chapter.

You can download the code in this chapter from [here](http://www.shido.info/lisp/scheme_amb.zip).