SICP section 1.2.1

June 26, 2007 at 05:14 Tags SICP

Section 1.2.1

The section begins with presenting two approaches to computing factorial. The first is a recursive process (translated to CL, as usual):

```
(defun factorial-rec (n)
  (if (= n 1)
      1
      (* n (factorial-rec (1- n)))))
```

The second uses an iterative process, although it is coded recursively as well:

```
(defun factorial-iter-aux (product counter max-count)
  (if (> counter max-count)
      product
      (factorial-iter-aux (* counter product)
                          (+ counter 1)
                          max-count)))

(defun factorial-iter (n)
  (factorial-iter-aux 1 1 n))
```

The discussion that follows in the book about the difference between iterative and recursive processes is priceless, a must read for anyone not intimately familiar with the subject. I won't repeat it here, but instead will focus on the implementation details.

This topic is a great place to discuss another subtle difference between Scheme and CL. To make the process generated by the `factorial-iter-aux` function really iterative, the compiler must support what's called *tail call optimization*, meaning that it must know to translate the recursive call in `factorial-iter-aux` into a jump instead of a function call.

Well, the Scheme standard requires compilers to implement tail call optimization. The CL standard does not, although many implementations do. I use CLISP 2.41 on Windows, and lets see how it handles this, using the `disassemble` command. For the recursive process:

```
[10]> (disassemble  factorial-rec)

Disassembly of function FACTORIAL-REC
(CONST 0) = 1
1 required argument
0 optional arguments
No rest parameter
No keyword parameters
12 byte-code instructions:
0     L0
0     (LOAD&PUSH 1)
1     (CONST&PUSH 0)                      ; 1
2     (CALLSR&JMPIF 1 45 L16)             ; =
6     (LOAD&PUSH 1)
7     (LOAD&DEC&PUSH 2)
9     (JSR&PUSH L0)
11    (CALLSR 2 55)                       ; *
14    (SKIP&RET 2)
16    L16
16    (CONST 0)                           ; 1
17    (SKIP&RET 2)
NIL
```

Although these “assembly” codes are CLISP-specific, most should look familiar to anyone who has learned any assembly. Note the `CALLSR` calls – these call system functions, and the comment to the right helpfully shows which. But what is really interesting is the `JSR` on line 9 – this is the *jump subroutine* command which recursively calls `factororial-rec`. It is coupled with a `PUSH` – which grows the stack. So indeed, a recursive process is generated here.

On the other hand, if we dissassemble `factorial-iter-aux`:

```
[19]> (disassemble 'FACTORIAL-ITER)

Disassembly of function FACTORIAL-ITER
3 required arguments
0 optional arguments
No rest parameter
No keyword parameters
13 byte-code instructions:
0     L0
0     (LOAD&PUSH 2)
1     (LOAD&PUSH 2)
2     (CALLSR&JMPIF 1 48 L18)             ; >
6     (LOAD&PUSH 2)
7     (LOAD&PUSH 4)
8     (CALLSR&PUSH 2 55)                  ; *
11    (LOAD&INC&PUSH 3)
13    (LOAD&PUSH 3)
14    (JMPTAIL 3 7 L0)
18    L18
18    (LOAD 3)
19    (SKIP&RET 4)
NIL
```

We see `JMPTAIL` on line 14 – which is the tail recursive jump. Indeed,CLISP implements the tail call optimization. To see one difference between them, we can run the two factorial functions on some large input (say 4000). While `factorial-rec` will fail with a stack overflow, `factorial-iter` won't[1] – this is because the stack does not grow.

Looping in Common Lisp

I don't know about you, but to me `factorial-rec` seems more natural than `factorial-iter`. This is probably because the tail recursive approach is not the best for representing the solution in iterative form. Generally, Common Lisp programmers prefer to use the built-in looping constructs[2] of the language instead of explicit tail recursion for iteration. Apart from being simpler to understand, it also removes the need for defining an auxiliary function (such as `factorial-iter-aux`) whenever a loop is needed. Here are a few definitions of the factorial function with CL loops:

```
; This is a bit contrived because DOTIMES always counts from 0 to n - 1,
; so we have to ignore 0.
;
(defun factorial-dotimes (n)
  (let ((product 1))
    (dotimes (i (+ n 1))
      (when (> i 0)
        (setq product (* product i))))
    product))

; DO is a flexible looping construct. By using DO* that sequentually
; performs the variable settings we can avoid an explicit temporary
; 'product' with a LET. But DO can also be used as a simple for loop
; of 'C'
;
(defun factorial-do (n)
  (if (= n 0)
    1
    (do* ((i 1 (1+ i))
          (product 1 (* product i)))
        ((= i n) product))))

; LOOP is the most powerful iteration construct of them all. A mini-language
; in itself, it is a bit of a controversy in the CL world.
; There are probably several ways to achieve this with a loop, and this is
; not the best one...
;
(defun factorial-loop (n)
  (loop
    for i from 1 to n
    and product = 1 then (* product i)
    finally (return product)))
```

I think that for someone who has written some CL code and is generally familiar with the looping macros, this code should be clear and self-descriptive.

Exercise 1.9

The first + is recursive, the second iterative. It is quite easy to figure out without explicitly going through the substitution model, actually. Just try to see if calling itself is absolutely the last thing a function does.

```
(defun add-rec (a b)
  (if (= a 0)
    b
    (1+ (add-rec (1- a) b))))

; (add-rec 4 5)
; (1+ (add-rec 3 5))
; (1+ (1+ (add-rec 2 5)))
; (1+ (1+ (1+ (add-rec 1 5))))
; (1+ (1+ (1+ (1+ (add-rec 0 5)))))
; (1+ (1+ (1+ (1+ 5))))
; (1+ (1+ (1+ 6)))
; (1+ (1+ 7))
; (1+ 8)
; 9

(defun add-iter (a b)
  (if (= a 0)
      b
      (add-iter (1- a) (1+ b))))

; (add-iter 4 5)
; (add-iter 3 6)
; (add-iter 2 7)
; (add-iter 1 8)
; (add-iter 0 9)
; 9
```

Exercise 1.10

The Ackermann's function is a very interesting mathematical curiosity. It is recursive in nature (note that there is a double recursion, so it's not tail recursive) and produces astoundingly huge numbers very quickly. Here are a couple of places to read more about it[3]. Anyway, here are the expansions:

```
(defun A (x y)
  (cond ((= y 0) 0)
        ((= x 0) (* 2 y))
        ((= y 1) 2)
        (t (A (- x 1)
              (A x (- y 1))))))

; (A 1 10)
; (A 0 (A 1 9))
; (* 2 (A 1 9))
; (* 2 (A 0 (A 1 8)))
; (* 2 (* 2 (A 1 8)))
; (* 2 (* 2 ... 10 times
; So, (A 1 10) = 2^10 = 1024

; (A 2 4)
; (A 1 (A 2 3))
; we saw that (A 1 x) is 2^x
; let's see what's (A 2 3)
; (A 1 (A 2 2)
; (A 1 (A 1 (A 2 1)))
; (A 1 (A 1 2)) => 2^(2^2) => 16
; So, (A 2 4) = 2^16 = 65536

; (A 3 3)
; (A 2 (A 3 2))
; (A 2 (A 2 (A 3 1)))
; (A 2 (A 2 2))
; (A 2 4)
; from above, this is 65536
; So, (A 3 3) = 2^16 = 65536
```

And here is the solution for the second part:

```
(defun f (n) (A 0 n))
; 2n

(defun g (n) (A 1 n))
; 2^n

(defun h (n) (A 2 n))
; 2^2^... (n times)
```

We can see from this how fast the Ackermann function grows. Even`(A 2 n)` produces huge numbers quickly. It's fascinating to ponder what `(A 3 n)` is like. Actually, I think that from the progression of `f, g, h` we can guess, but it's not something our mathematical notation is capable of expressing explicitly. Something like the 4th dimension in space, which we can play with mathematically but can't really gasp intuitively.

Footnotes

[1] If you're using CLISP, you must compile `factorial-iter` in order to see tail call optimization in

action. The interpreter doesn't implement it for uncompiled functions.

[2] "Built-in" is a question of definition here. The looping constructs are actually CL macros, and you could define them yourself if they weren't there.

[3] Although the definition SICP gives for Ackermann's function seems non-standard, it's the same general idea anyway.

For comments, please send me an email.

Back to top