CS450 - Structure of Higher Level Languages

Lexical Scoping, Recursion versus Iteration

September 14, 2020

Cond – Reminder

Remember how **cond** works:

```
(cond (<condition> <exp> <exp> <exp> ...)
      (<condition> <exp> <exp> <exp> ...)
      (<condition> <exp> <exp> <exp> ...)
              ...
      (else <exp> <exp> <exp> ...)
      ;; this clause is optional
      )
```

Let's Use It

```
(define (count1 x)
  (cond ((= x 0) (display x))
        (else (display x)
              (count1 (- x 1))  )))


(define (count2 x)
  (cond ((= x 0) (display x))
        (else (count2 (- x 1))
              (display x)      )))
```

Let's Evaluate It

Now evaluate

```
==> (count1 4)
```

and

```
==> (count2 4)
```

Let's Evaluate It

Let's print them both out:

```
(count1 4)
(display 4)   ;; prints 4
(count1 3)
(display 3)   ;; prints 3
(count1 2)
(display 2)   ;; prints 2
(count1 1)
(display 1)   ;; prints 1
(count1 0)
(display 0)   ;; prints 0
```

so it prints out

43210

Let's Evaluate It

```
(count2 4)
(count2 3)(display 4)
(count2 2)(display 3)(display 4)
(count2 1)(display 2)(display 3)(display 4)
(count2 0)(display 1)(display 2)(display 3)(display 4)
(display 0)(display 1)(display 2)(display 3)(display 4)
```

so it prints out

```
01234
```

Analysis

- Notice the difference in behavior at run-time: `count2` cannot display anything until the end – it has to store up all the display actions until the final `(count2 0)` is evaluated.
- What actually happens is that these actions are in effect pushed onto a run-time stack.
- `count1`, on the other hand, does not need to defer any operations, and so doesn't need to push anything on a stack as it executes.

Tail Recursion

- The way you can tell which of these behaviors will happen is to look at the nature of the recursive call in each case:
- In `count1`, the recursive call to `count1` is the last thing that is executed in the body of `count1`.
- Thus by the time the recursive call is made, all the rest of the body has been executed (or "evaluated"), and there is nothing that needs to be deferred.
- This is called *tail recursion*. It leads to a run-time behavior that does not defer any operations.
- In other languages such behavior is normally written as a loop.
- In Scheme, we tend to write this recursively, but since this is tail recursion, SICP calls this kind of code *iterative*.
- We, however, along with everyone else in the world, call it *tail recursion*.

Tail Recursion

- In `count2`, the recursive call to `count2` in the body of `count2` is followed by another expression to evaluate.
- Thus, this code is not tail recursive, and the evaluation of that following expression needs to be deferred. This kind of code is called *recursive* by SICP.
- So you have to be a bit careful when reading the book: when the authors use the term *recursive*, they do not mean *tail recursive*.

- Thus, both `count1` and `count2` are syntactically recursive. But `count1` is tail recursive while `count2` is not.
- As we will see again and again, tail recursion is the natural way to represent iterative computations (that is, computations written as loops in other programming languages) in Scheme.
- In most computer languages on the other hand, iterative computations must be represented by special iteration constructs like **for** loops in C or **do** loops in Fortran.
- But in Scheme, they can be represented by *tail-recursive* procedures, which (by the Scheme standard) must be implemented as iterations.
- We'll talk a lot more about this as the course goes on.

Newton's Method

- This is the "divide-and-average" method for finding square roots, which is extremely efficient (the number of correct digits approximately doubles with each iteration)
- For example, to find $\sqrt{2}$, we first pick any initial guess, say 1, and proceed as follows:

Guess	**Quotient**	**Average**
1	$\frac{2}{1} = 2$	$\frac{2+1}{2} = 1.5$
1.5	$\frac{2}{1.5} = 1.33333$	$\frac{1.33333+1.5}{2} = 1.4167$
1.4167	$\frac{2}{1.4167} = 1.4118$	$\frac{1.4118+1.4167}{2} = 1.4142$
1.4142	...	...

In Scheme

```
(define (sqrt-iter guess x)
;; This implements the iteration.
  (if (good-enough? guess x)
      guess
      (sqrt-iter (improve guess x)  x)  ))

(define (improve guess x)
  (average guess (/ x guess))  )

(define (average x y)
  (/ (+ x y) 2)  )
```

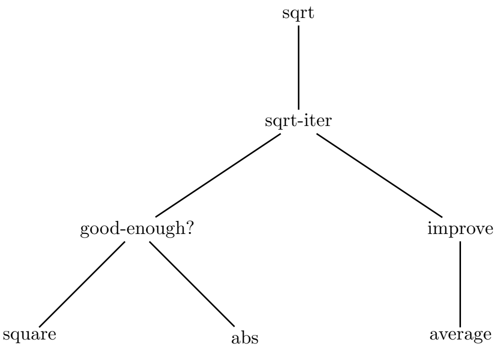
In Scheme

```
(define (good-enough? guess x)
;; guess, x could be a, b (for instance) here
  (< (abs (- (square guess) x)) .001)  )
  ;; .001 is a "magic number"

(define (sqrt x)
;; Here's where we start.  We package up
;; the data and start the iteration.
  (sqrt-iter 1 x)  )
```

Call Graph

Here is the *call graph* for the square root algorithm we just wrote down:



Call Graph

The book rewrites the Scheme code above like this:

```
(define (sqrt x)
  (define (good-enough? guess x)
    (< (abs (- (square guess) x)) .001) )
  (define (improve guess x)
    (average guess (/ x guess)) )
  (define (average x y)
    (/ (+ x y) 2) )
  (define (sqrt-iter guess x)
    (if (good-enough? guess x)
        guess
        (sqrt-iter (improve guess x) x) ))
  (sqrt-iter 1 x))
```

Call Graph

- All the user cares about is the function sqrt.
- The internal details can be hidden, as they are here.
- Actually, it would be even better to write the code in a way that accurately reflects the call graph, like this:

```
(define (sqrt x)
  (define (sqrt-iter guess x)
    (define (good-enough? guess x)
      (< (abs (- (square guess) x)) .001) )
    (define (improve guess x)
      (define (average x y)
        (/ (+ x y) 2))
      (average guess (/ x guess)))
    (if (good-enough? guess x)
        guess
        (sqrt-iter (improve guess x)))) ; end sqrt-iter
  (sqrt-iter 1 guess))
```

Lexical Scoping

- Variables get looked up in the innermost scope in which they are found.
- This is called *lexical scoping*.
- Now we can remove the variables available from an outer scope.
- These are all the inner `x`'s *except* the `x` in `average`, as well as all the `guess`'s inside `sqrt-iter`.

Lexical Scoping

```
(define (sqrt x)
  (define (sqrt-iter guess)
    (define (good-enough?)
      (< (abs (- (square guess) x)) .001) )
    (define (improve)
      (define (average x y)
        (/ (+ x y) 2))
      (average guess (/ x guess)))
    (if (good-enough?)
        guess
        (sqrt-iter (improve)))) ; end sqrt-iter
  (sqrt-iter 1))
```

Note that `x` was originally *bound* in `improve`, but now it is *free* (its value is not passed in, but is obtained from an outer scope).

A second application: the Euclidean algorithm

Euclid's algorithm computes the GCD (greatest common divisor) of two numbers *a* and *b*:

a	*b*
206	40
40	6
6	4
4	2
2	0

so the GCD of 206 and 40 is 2.

A second application: the Euclidean algorithm

algorithm in Scheme, using the primitive procedure `remainder`:

```
(define (gcd a b)
  (if (= b 0)
      a
      (gcd b (remainder a b)) ))
```

Theorem (Lamé, 1845)

If Euclid's algorithm requires k steps, then the smaller of the two input numbers is $\geq$ the k^{th} Fibonacci number.

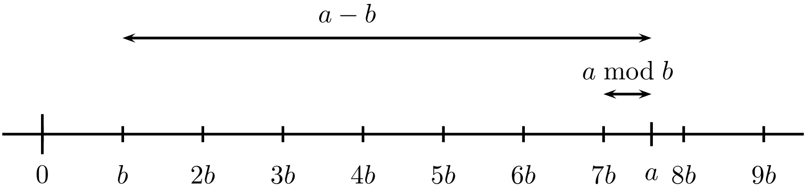
Proof of Lamé's theorem

- Lamé's theorem can be proved by induction.
- We start out the algorithm with two numbers a and b, where $a > b$.
- Let us set out the computation as follows:

$$n-1 \text{ steps} \left\{ \begin{array}{cccc} n & F_n & a & b \\ n-1 & F_{n-1} & b & a \bmod b \\ n-2 & F_{n-2} & a \bmod b & . \\ \vdots & \vdots & \vdots & \vdots \\ 3 & 2 & x & y \\ 2 & 1 & y & z \\ 1 & 1 & z & 0 \end{array} \right.$$

Proof of Lamé's theorem

- Here the number of steps in the process is $k = n - 1$. (That is, there are $n - 1$ steps to get from the top row to the bottom.)
- If $0 < b < a$ (as is true here), then $a - b \geq a \bmod b$.



Proof of Lamé's theorem

- This is simply because $a \bmod b$ is what is left after you subtract as many b's from a as you can.
- Since $b < a$, you can subtract at least 1 b, so $a - b \geq a \bmod b$.
- Thus $a \bmod b + b \leq a$.
- That is, in the second column from the right, the top element is $\geq$ the sum of the next two elements below it.
- By the same reasoning, this property holds all the way down that column. Further, we know that y and z must be ≥ 1.

Proof of Lamé's theorem

Working back up, we see that we can put in $\geq$ signs:

$$
n-1 \text{ steps} \left\{
\begin{array}{ccccc}
n & F_n & \leq & a & b \\
n-1 & F_{n-1} & \leq & b & a \bmod b \\
n-2 & F_{n-2} & \leq & a \bmod b & . \\
\vdots & \vdots & \vdots & \vdots & \vdots \\
3 & 2 & \leq & x & y \\
2 & 1 & \leq & y & z \\
1 & 1 & \leq & z & 0
\end{array}
\right.
$$

and so we see that $a \geq F_n$, and $b \geq F_{n-1}$. Since the number of steps in the algorithm is just $k = n-1$, we have $b \geq F_{n-1} = F_k$, which is what the theorem asserts, and we are done.

Recursion versus iteration: factorials

Let us consider the factorial function $n! = 1 \cdot 2 \ldots n$. We can compute this in Scheme as follows:

```
(define (factorial n)
  (if (= n 1)
      1
      (* n (factorial (- n 1))) ))
```

How Does It Work

```
(factorial 6)
(* 6 (factorial 5))
(* 6 (* 5 (factorial 4)))
(* 6 (* 5 (* 4 (factorial 3))))
(* 6 (* 5 (* 4 (* 3 (factorial 2)))))
(* 6 (* 5 (* 4 (* 3 (* 2 (factorial 1))))))
(* 6 (* 5 (* 4 (* 3 (* 2 1)))))
(* 6 (* 5 (* 4 (* 3 2))))
(* 6 (* 5 (* 4 6)))
(* 6 (* 5 24))
(* 6 120)
720
```

How Does It Work

- This is an example of what the authors of our text call a *recursive* procedure.
- As we explained before, they call it recursive because the operations are deferred.
- We keep saving the numbers 6, 5, and so on until the very end, when they get all multiplied together.
- But suppose we didn't really save them – suppose we kept multiplying them as we went on, and passed the partial products on as a parameter to the function?
- Then there would be nothing to collect at the end.

How Does It Look Like

```
(define (factorial n)
  (fact-iter 1 n) )

(define (fact-iter product count-down)
  (if (= count-down 1)
      product
      (fact-iter (* product count-down)
                 (- count-down 1) )))
```

How Does It Look Like

```
(factorial 6)
(fact-iter 1 6)
(fact-iter 6 5)
(fact-iter 30 4)
(fact-iter 120 3)
(fact-iter 360 2)
(fact-iter 720 1)
720
```

The Book Version

- The book gives a similar version, except that it counts up instead ofdown.
- Note that in this version we need a third argument to fact-iter, because we are comparing `count` to `max-count`, rather than to 1

```
(define (factorial n)
  (fact-iter 1 1 n) )

(define (fact-iter product count max-count)
  (if (> count max-count)
      product
      (fact-iter (* count product)
                 (+ count 1)
                 max-count) ))
```

The Book Version

```
(factorial 6)
(fact-iter 1 1 6)
(fact-iter 1 2 6)
(fact-iter 2 3 6)
(fact-iter 6 4 6)
(fact-iter 24 5 6)
(fact-iter 120 6 6)
(fact-iter 720 7 6)
720
```

Recursion vs. Iteration

- So as we saw before with `count1`, although these versions of the `factorial` procedure are syntactically recursive, none of the operations are deferred – we don't accumulate a big list of "things to do".
- In an actual implementation, these "things" would be accumulated on the stack.
- For this reason, these new computations are called *iterative*.
- And in fact it is easy to see that the code for both these versions of `factorial` is tail-recursive.

Exercise

1. Rewrite this last version of `factorial` so that `fact-iter` is an internal definition.
2. Show that the third argument to `fact-iter` can then be eliminated.

Recursion versus iteration: the Fibonacci Numbers

For another example of recursion, let us compute the Fibonacci numbers:

0	1	2	3	4	5	6	7	8	n
0	1	1	2	3	5	8	13	21	F_n

The recursive (mathematical) definition of these numbers is as follows:

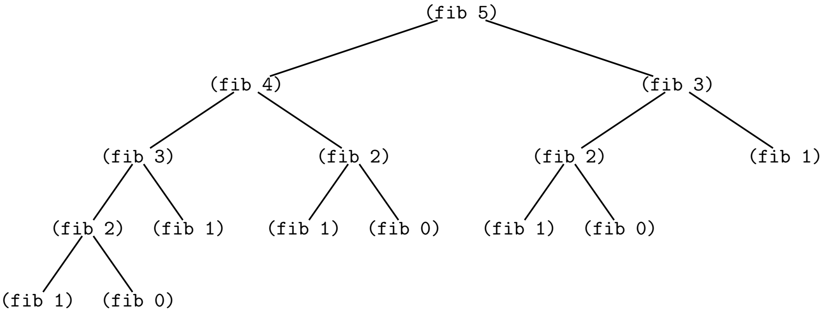
$$\text{fib}(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ \text{fib}(n-1) + \text{fib}(n-2) & \text{otherwise} \end{cases}$$

The Fibonacci Numbers in Scheme

```
(define (fib n)
  (cond ((= n 0) 0)
        ((= n 1) 1)
        (else (+ (fib (- n 1)) (fib (- n 2)))) ))
```

What Could Possibly Go Wrong?

Well, everything...



What Could Possibly Go Wrong?

- This kind of process is called *tree recursion* and is extremely inefficient. In fact, the number of leaves in the tree is F_{n+1}.
- You may know that there is a clever formula for F_n:

$$F_n = \frac{1}{\sqrt{5}}\left(\left(\frac{\sqrt{5}+1}{2}\right)^n - \left(\frac{\sqrt{5}-1}{2}\right)^n\right)$$

Now $(\sqrt{5}+1)/2 = 1.618\ldots$, and $abs(\sqrt{5}-1)/2 < 1$, so for large n,

$$F_n \cong \frac{1}{\sqrt{5}} 1.618^n$$

- which is an exponential.
- So this recursive method is a perfectly terrible way of computing the Fibonacci numbers. We say that this computation is an $O(1.6^n)$ computation.

Iterative Fibonacci

- On the other hand, there is an iterative way we can perform this computation.
- We do this, as before, by saving at each step the results we need to compute the next step:

```
(define (fib n)
  (fib-iter 1 0 n) )

(define (fib-iter fn-1 fn-2 count)
  (if (= count 0)
      fn-2
      (fib-iter (+ fn-1 fn-2) fn-1 (- count 1)) ))
```

Iterative Fibonacci

Here's how it works out:

```
(fib 5)
(fib-iter 1 0 5)
(fib-iter 1 1 4)
(fib-iter 2 1 3)
(fib-iter 3 2 2)
(fib-iter 5 3 1)
(fib-iter 8 5 0)
5
```

and we see in this case that the computation is $O(n)$, which is a vast improvement.

Recursion Vs. Iteration: Exponentiation

- Suppose we want to compute b^n. (b stands for *base*.)
- We will assume that both b and n are non-negative integers.
- A naive way to compute this is recursively: we know that $b^n = b * b^{n-1}$, so we can write...

```
(define (expt b n)
  (if (= n 0)
      1
      (* b (expt b (- n 1))) ))
```

- This is recursive, because the call to expt is deferred in the tail of the computation.
- The computation is $O(n)$ in time and $O(n)$ in space.

Recursion Vs. Iteration: Exponentiation

On the other hand, we can compute this iteratively (i.e., using tail-recursion):

```
(define (expt-iter b counter product)
  (if (= counter 0)
      product
      (expt-iter b
                 (- counter 1)
                 (* b product)) ))
```

This iterative (i.e., tail-recursive) procedure is $O(n)$ in time, but only $O(1)$ in space.

Recursion Vs. Iteration: Exponentiation

An even better way to perform this computation is to use a method of successive squaring. We use the fact that

$$b^n = \begin{cases} \left(b^{\frac{n}{2}}\right)^2 & \text{if } n \text{ is even} \\ b \cdot b^{n-1} & \text{if } n \text{ is odd} \end{cases}$$

```
(define (fast-expt b n)
  (cond ((= n 0) 1)
        ((even? n) (square (fast-expt b (/ n 2))))
        (else (* b (fast-expt b (- n 1))))) )
```

- This procedure, even though it is not tail-recursive, is $O(\log_2 n)$ in both space and time.
- So it's a little worse in space, but a lot better in time.
- **Exercise:** Can you show that this procedure is $O(\log_2 n)$ in time?