CS403: Programming Languages

Recursive and Iterative Processes

Key ideas

- recursive functions
- recursive processes
- iterative processes
- tail recursion
- recognizing tail recursion
- optimizing tail recursion

Recursive processes - performing exponentiation

Let's write a Scheme procedure for computing the raising of a number by an exponent. Scheme (nominally) has no looping constructs, so we use recursion instead. It's almost always a good thing to write a set of recurrence relations first. Here's a set for exponentiation.

$base^0 = 1$
$base^{exp} = base * base^{exp-1}$

Remember, we must have a terminating case, otherwise the recurrence goes on forever. Recurrence relations are easily translated into Scheme:

```
(define (raise base exp)
    (if (= exp 0)
        1
        (* base (raise base (- exp 1)))
        )
    )
```

Let's investigate the evaluation of the expression:

```
(raise 2 3)
```

First the interpreter evaluates:

```
(raise 2 3)
```

To do so, it needs to compute the expression:

```
(* 2 (raise 2 2))
```

which in turns requires evaluation of:

```
(* 2 (* 2 (raise 2 1)))
```

and:

```
(* 2 (* 2 (* 2 (raise 2 0))))
```

At this point, the interpreter can directly calculate (raise 2 0) and in the normal scheme of things, places that return value on the stack in the right place, yielding:

```
(* 2 (* 2 (* 2 1)))
```

The expression shortens as the interpreter bounds up the stack

```
(* 2 (* 2 2))
```

followed by:

```
(* 2 4)
```

and finally:

```
8
```

which is the value the interpreter returns. Note that the number of recursive calls is proportional to the exponent, so the procedure executes in $O(n)$ time, where n is the size of the exponent. It also uses $O(n)$ space, since each of the recursive calls takes up successive space on the stack. Compare this to a loop-based implementation in a procedural language, which takes $O(n)$ time (here the time is dominated by the number of multiplications) but $O(1)$ space. Next time we'll look at a Scheme implementation that uses $O(1)$ space as well.

Iterative processes - performing exponentiation

Last time, we developed a version of the exponentiation function that was not only a recursive function, but implemented a recursive process as well:

```
(define (raise base exp)
    (if (= exp 0)
        1
        (* base (raise base (- exp 1)))
        )
    )
```

Here is an alternate version which implements an iterative process:

```
(define (raise base exp)
    (define (inner_raise total base count)
        (if (= count 0)
            total
            (inner_raise (* total base) base (- count 1))
            )
        )
    (inner_raise 1 base exp)
    )
```

Let's investigate again the evaluation of the expression:

```
(raise 2 3)
```

First the interpreter evaluates

```
(raise 2 3)
```

Now, it needs to compute the expression:

```
(inner_raise 1 2 3)
```

which in turns requires evaluation of:

```
(inner_raise 2 2 2)
```

and then:

```
(inner_raise 4 2 1)
```

and finally:

```
(inner_raise 8 2 0)
```

At this point, the interpreter returns 8. Because no data is needed from the previous call to *inner_raise*, the current call to *inner_raise* can completely overlay the previous call on the call stack. Thus the call stack does not grow and the function executes in constant space. Of course, the compiler or interpreter must implement the overlay process. Scheme does so while C, C++, and Java compilers do not. This is known as tail recursion and is one of the things that makes Scheme so efficient. You can hand-optimize procedural code, however. Here is *inner_raise*, written as an recursive process in a C.

```
long
inner_raise(long total, int base, int count)
    {
    if (count == 0)
        return total;

    return inner_raise(total*base, base, count-1);
    }
```

Since C does not optimize tail recursion, this process runs in $O(n)$ space. We can hand optimize *inner_raise*, however. We simply place a label at the top, update the formal parameters, and then jump to the top. Here is the optimized code:

```
long
inner_raise(long total, int base, int count)
    {
TOP:
    if (count == 0)
        return total;

    total = total*base
    count = count-1;
    goto TOP;
    }
```

This simulates overwriting the formal parameters on the stack. We have converted *inner_raise* from implementing a recursive process to implementing an iterative process. Of course, normally we would write this as a loop. The point to be gained here is that if you prefer to write code in a recursive style, Scheme allows you to do so without loss of efficiency. Another point to be gained is that there is actually no difference between loops and recursion; they are one and the same.
Here is another example of a recursive calculation that can be implemented as a recursive or an iterative process. The task is to calculate the n^{th} Fibonacci number.

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

and

```
(define (fibi n)
    (define (inner_fib last current count)
        (if (= count 0)
            last
            (inner_fib current (+ last current) (- count 1))
            )
        )
    (inner_fib 0 1 n)
    )
```

At this point, you ought to be able to tell the nature of the process each function implements.

Recognizing tail recursion

Tail recursion is easy to recognize. If the last thing a function does before exiting is a recursive call, then that *particular* call is a tail recursive call. In *fibr*, the last thing that is done before leaving *fibr* is an addition, so *fibr* is not tail recursive. However, the last thing *inner_fib* does before exiting is call *inner_fib*. Therefore, *inner_fib* is tail recursive. Since the last thing *fibi* does is call *inner_fib*, since *inner_fib* is tail recursive, *fibi* is tail recursive as well.
Note, it is possible for a function to be sometimes tail recursive and sometimes not, as in this example:

```
(define (f x)
    (cond
        ((= x 0)
            0
            )
```

```
            ((= x 1)
                1
                )
            ;; tail recursive call
            ((= (remainder x 2) 0)
                (f (- x 1))
                )
            ;; non-tail recursive call
            (else
                (+ (f (- x 1)) (f (- x 2)))
                )
            )
        )
```

Here is another interesting example:

```
    (define (app a b)
        (cond
            ((null? a) b)
            ((null? b) a)
            (else (app (cdr a) (app (cdr a) (cdr b))))
            )
        )
```

Although the outer recursive call to *app* is tail recursive, the inner recursive call is not. Thus *app* generates a recursive process.