

CptS 355

Feb. 4, 2005

4th February 2005

Recursive problems and recursive solutions

Let's consider the problem of computing the sum of integers 1 through n

$$\sum_{i=1}^n i$$

First, a mathematical convention

$$\sum_{i=1}^0 = 0$$

and note the equivalences

$$\sum_{i=1}^n i = 1 + \sum_{i=2}^n i = \sum_{i=1}^{n-1} i + n$$

```
; (sumints n)
; sum of integers from 1 to n
(define (sumints n)
  (if (eq? 0 n)
      0
      (+ n (sumints (- n 1)))
  )
)
```

Notice how there is a large stack of function calls building up until the base case is reached. The solution is a technique which called *accumulating parameters* which allows us to implement recursive functions so they are *tail recursive*. **Definition:** a function is *tail recursive* if at the point where it makes the recursive call there is no more work to do for the current invocation. The previous two sumints example is not tail recursive. After the call to sumints there is still an add operation that has to be done.

```

; (sumints n) - returns sum of integers from 1 to n
; uses a nested function
(define (sumints n)
  ; sumtr is t_ail r_recursive
  (define (sumtr n acc)
    (if (eq? n 0)
        acc
        (sumtr (- n 1) (+ acc n))))
  )
  (sumtr n 0)
)

```

In this version of `sumints` an accumulating parameter, `acc`, is used. Because we want `sumints` to have the same calling conventions as before, in order to use the accumulating parameter technique an auxiliary function, `sumtr`, is used. Notice how it is defined locally to (inside of) `sumints`.

Horner's rule

Another example: suppose you want to evaluate a polynomial

$$\sum_{i=0}^n a_i x^i$$

Computing all the powers of x separately is computationally inefficient, to say nothing of ugly. There is a better way. Observe

$$\sum_{i=0}^n a_i x^i = a_0 + x \sum_{i=1}^n a_i x^{i-1} = a_0 + x(a_1 + x(a_2 + \dots x(a_n) \dots))$$

Suggesting an evaluation method:

```

sum := a_n
for (i=n-1; i>=0; i--) {
  sum := x*sum + a_i
}

```

Fold

Functions taking other functions as arguments are a standard programming technique in functional programming. We have talked elsewhere about the `map` function which returns a list consisting of elements obtained by applying a function to each element of another list. That is, the value of `(map f (e1 e2 ... en))` is a list `((f e1) (f e2) ... (f en))`.

```
(define (map f l)
  (if (null? l)
      ()
      (cons (f (car l)) (map f (cdr l)))
  )
)
```

Another example is `(fold f b l)` which computes

$$f(f(\dots f(f(b, l_1), l_2), \dots, l_{n-1}), l_n)$$

The effect of fold is easier to see in infix notation. Suppose `f` is `+`. Then the result of `(fold + 0 l)` is

$$(((\dots((b + l_1) + l_2) + \dots + l_{n-1}) + l_n)$$