

# CptS 355

## Sept. 3, 2004

6th September 2004

### 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

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

Convention in commenting Postscript functions

```
tosbefore functionname tosafter
```

That is, describe what is on the stack before using the function, then the function name, then what is on the stack after the function is done.

```
% n sumints (sum(n))
/sumints { % straightforward recursive def
% note how the stack eventually has n+1 items on it
  dup
  0 eq
  {}
  {dup 1 sub sumints add}
  ifelse
} def
```

Potential problem: the stack grows until the base case is reached, possibly becoming very large (for large values of  $n$ ). To try to avoid this let's consider using a dictionary to hold the parameter value. Notice that, like last time, we'll need to add a dictionary to the dictionary stack for each recursive call so there is a separate copy of  $i$  for each invocation.

```

/sumints { % named parameter
% now the *dict* stack eventually has n+1 dictionaries on it
  1 dict begin
    /i exch def
    i 0 eq
    {0}
    {i 1 sub sumints i add}
  ifelse
end
} def

```

This has actually made the problem worse: instead of a large stack of values we now have a large dictionary stack. The solution is a technique which we'll call *accumulating parameters* which will allow 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` examples are not tail recursive. In the first example, after the call to `sumints` there is still an `add` operation that has to be done. In the second example, there is still a push of `i` followed by an `add` operation and an `end` operation.

```

/sumints { % tail recursion using an accumulating parameter
% uses a nested function
% uses constant stack space
  1 dict begin
    /sumints2 { % sumsofar ntogo sumints2 (sumsofar + (sum (ntogo)))
      2 dict begin
        /ntogo exch def          %local definition
        /sumsofar exch def      %local definition
        ntogo 0 eq
        {sumsofar end}          %final answer left on stack
        {
          ntogo sumsofar add
          ntogo 1 sub
          end
          sumints2              %recursive call
        }
      ifelse
    } def
  } def

```

```

    0 exch sumints2          %initial call
  end
} def

```

In this third version of sumints an accumulating parameter, sumsofar, 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, sumints2, is used. Notice how it is defined locally to (inside of) sumints.

Note also how there are 3 end operations but only 2 begins. Why? Because begin and end are executable operations in PostScript. In order to make sumints2 tail recursive end must be used before the recursive call to sumints2 in the false branch. So there must also be an end in the true branch.

### Horner's rule

Another example: suppose you want to evaluate a polynomial

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

(Note that this is useful in the CA problem to convert a sequence of 3 adjacent b and w values to a number between 0 and 7 so you can do an array lookup for the next-generation value of a cell.)

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
}

```