Chapter 8 - A bigger Example - nprime, atoi

In this section we consider a particular example, that of finding the nth prime number. At the end, the source (without documentation) for a string to integer function is given.

nprime

(The style used for code in this chapter is changed for purposes of readability)

This example finds the nth prime number, using the following method:
- keep a list of all the primes we have found;
- try each new number against the list we have found so far to see if prime;
  (only need to check up to the square root of the number we are trying)
- repeat until have enough primes.

```
(* we have a header here to get things started *)
nprime[n:int] -> primeaux[,0,n,2];

(* primeaux[primes found so far, sqrt of latest try, number to go, latest try *)
(* none to go, so finished *)
primeaux[l,s,0,k] -> k-1;
(* update the square root *)
primeaux[l,s,n,k]::k>s*s -> primeaux[l,s+1,n,k];
(* see if we've found one *)
primeaux[l,s,n,k]::primetest[l,s,k] -> primeaux[.l,k,s,n-1,k+1];
(* otherwise try the next *)
primeaux[l,s,n,k] -> primeaux[l,s,n,k+1];

(* primetest[list to try dividing against, sqrt of latest try, number we are testing *)
(* didn't divide any, therefore prime *)
primetest[,s,k] -> true;
(* didn't divide any less than the square root, so prime *)
primetest[a,.b,s,k]::a>s -> true;
(* a division, so not prime *)
primetest[a,.b,s,k]::k%a=0 -> false;
(* didn't divide that number, try the next *)
primetest[a,.b,s,k] -> primetest[b,s,k];

nprime[2000] -> 17389
```

Time taken on LC III: 8009 ticks.

Here is the same code with types added (which speeds up all the arithmetic) and the comments removed.

```
nprime[n:int] -> primeaux[,0,n,2];

primeaux[l:lis,s:int,0,k:int] -> k-1;
```

```
primeaux[l:lis,s:int,n:int,k:int]::k>s*s -> primeaux[l,s+1,n,k];
primeaux[l:lis,s:int,n:int,k:int]::primetest[l,s,k] -> primeaux[.l,k,s,n-1,k+1];
primeaux[l:lis,s:int,n:int,k:int] -> primeaux[l,s,n,k+1];

primetest[,s:int,k:int] -> true;
primetest[a:int,.b,s:int,k:int]::a>s -> true;
primetest[a:int,.b,s:int,k:int]::k%a=0 -> false;
primetest[a:int,.b,s:int,k:int] -> primetest[b,s,k];

nprime[2000] -> 17389
```
Time taken on LC III: 7079 ticks

Although the above (ReWrite) example has everything clearly typed, and tail recursion keeps the stack use very low, there is a problem that the Pascal version (see below) doesn't share - in the third line of `primeaux`,

```
primeaux[l:lis,s:int,n:int,k:int]:primetest[l,s,k] -> primeaux[.l,k,s,n-1,k+1];
```

the parameter `l` is used in the condition (as well as on the right hand side), which because of unique references means that the whole list of prime found so far has to be copied each time through the loop.

The following version avoids this problem by passing the whole list to `primetest`, which doesn't just dispose of the primes that it has tested against so far, but keeps them and returns the complete list of results as well as the result for whether the new number is prime or not. `primeaux2` is 'glue' code that takes the result from `primetest` and adds the new number to the list or not accordingly.

Note: that this code makes use of some of the more unusual feature of ReWrite:

- `primetest` returns 2 results, as an arbitrary number of results can be returned;
- the list of primes is not passed as an actual list to `primetest` but it turned into a lot of extra arguments, and because the argument list ends with a `.b`, this picks up all the primes and puts them into `b`. This improves the code speed slightly.

```
nprime[n:int] -> primeaux[,0,n,2];

(* primeaux[primes found so far, sqrt, number found, this try] *)
primeaux[l:lis,s:int,0,k:int] -> k-1;
primeaux[l:lis,s:int,n:int,k:int]::k>s*s -> primeaux[l,s+1,n,k];
primeaux[l:lis,s:int,n:int,k:int] -> primeaux2[primetest[s,k,,.l],s,n,k];

(* primeaux2[primes found so far, is prime?, sqrt, number found, this try] *)
primeaux2[l:lis,false,s:int,n:int,k:int] -> primeaux[l,s,n,k+1];
primeaux2[l:lis,true,s:int,n:int,k:int] -> primeaux[.l,k,s,n-1,k+1];

(* primeaux2[sqrt, this try, list of primes checked, primes unchecked] *)
primetest[s:int,k:int,l:lis] -> l,true;
primetest[s:int,k:int,l:lis,a:int,.b]::a>s -> .l,a,.b,true;
primetest[s:int,k:int,l:lis,a:int,.b]::k%a=0 -> .l,a,.b,false;
primetest[s:int,k:int,l:lis,a:int,.b] -> primetest[s,k,.l,a,.b];
```

```
nprime[2000] -> 17389
```
Time taken on LC III: 134 ticks - better than a 50-fold improvement!

It is hoped that some future version or ReWrite will be able to do this sort of code transformation automatically as an optimisation, however that is some way off.

Just for an example of how the typing helps, the above example with all the type information removed takes much longer (so putting in all those `int`s is worth it):

```
nprime[2000] -> 17389
```
Time taken on LC III: 556 ticks

Here is another implementation of the same algorithm, using features new to ReWrite 0.2.5 (the unrestricted use of `splice` on the left)

This version keeps track of four values:

* the list of primes less than or equal to the square root of the current number (this is the list to check for divisors in),
* the list of other primes found so far,
* the number of primes found so far,
* the current number being checked.

```
nprime[n:int] -> primeaux[,,n,2];

(* primeaux[(primes found so far <= sqrt[k]), (primes found so far > sqrt[k]), number found, this try] *)
(* found as many primes as we want, so finished *)
primeaux[l1:lis,l2:lis,0,k:int] -> k-1;
(* update the list of primes <= sqrt[k] if necessary *)
primeaux[l1:lis,s,.l2,n:int,k:int]::k>=s*s -> primeaux[.l1,s,l2,n,k];
(* not prime if and only if one of the primes <= sqrt[k] divides k *)
primeaux[.a,b:int,.c,l2:lis,n:int,k:int]::k%b=0 -> primeaux[.a,b,.c,l2,n,k+1];
(* otherwise prime, so add to list *)
primeaux[l1:lis,l2:lis,n:int,k:int] -> primeaux[l1,.l2,k,n-1,k+1];
```

```
nprime[2000] -> 17389
```
Time taken on LC III: 108 ticks

Just for a comparison, here is a Pascal program using the same algorithm:

```
function nprime (n: longint): longint;

    var
        s, k, i, t: longint;
        f: boolean;
        a: array[0..2000] of longint;

begin
```

```
        s := 0;
        k := 2;
        i := 0;
        while i < n do
            begin
                while s * s < k do
                    s := s + 1;
                t := 0;
                f := true;
                while (t < i) and f and (a[t] <= s) do
                    begin
                        f := (k mod a[t]) <> 0;
                        t := t + 1;
                    end;
                if f then
                    begin
                        a[i] := k;
                        i := i + 1;
                    end;
                k := k + 1;
            end;
        nprime := k - 1;
    end;
```

`nprime(2000)` -> `17389`
Time taken on LC III: 32 ticks

This is much faster, but longer and (I think) has less readable code.

atoi2

This converts strings to integers
Note that `atoi2` is not the version or `atoi` used internally, this version has been updated to use the new character type.

```
atoi2[.x] -> atoixx[xatoi[.x]];

atoixx[x:int] -> x;
atoixx[.x] -> "error - not a number";

xatoi["-",.n] -> xnegx[xatoi[.n]];
xatoi["$",.a] -> xatoih[0,.a];
xatoi["%",.a] -> xatoib[0,.a];
xatoi[x:char,.a]::xisdigit[x] -> xatoix[0,x,.a];
xatoi[.a] -> ,.a;

xnegx[,.rest] -> ,.rest;
xnegx[a:int,.rest] -> -a,.rest;
```

```
xatoix[x:int,d:char,.n]::xisdigit[d] -> xatoix[10*x+d:int-"0":int,.n];
xatoix[x:int,.n] -> x,.n;

xatoih[x:int,d:char,.n]::xisdigit[d] -> xatoih[16*x+d:int-"0":int,.n];
xatoih[x:int,d:char,.n]::d>="A" & d<="F" -> xatoih[16*x+10+d:int-"A":int,.n];
xatoih[x:int,d:char,.n]::d>="a" & d<="f" -> xatoih[16*x+10+d:int-"a":int,.n];
xatoih[x:int,.n] -> x,.n;

xatoib[x:int,"0",.n] -> xatoib[2*x,.n];
xatoib[x:int,"1",.n] -> xatoib[2*x+1,.n];
xatoib[x:int,.n] -> x,.n;

xisdigit[d:char] -> d>="0" & d<="9";
```