# For-loops for free

In other words, students' calculations above already deploy a CbC (correct by construction) "for-loop" implementation of multiplication:

```
a .* n = for (a+) 0 n
```

something to be encoded (much later!) imperatively, eg. in C:

```
int mul(int a, int n)
{
int s=0; int i;
for (i=1;i<n+1;i++) {s += a;}
return s;
};
```

# Not so immediate for-loops

Now consider the challenge of encoding the square function, $sq\ n = n^2$. Following the same approach, let students first recall known facts about squares, including Newton's binomial formula:

$$\begin{aligned} 0^2 &= 0 \\ 1^2 &= 1 \\ (a+b)^2 &= a^2 + 2ab + b^2 \end{aligned}$$

Playing the same game, the following will be obtained:

$$\begin{aligned} sq\ 0 &= 0 \\ sq\ (n+1) &= sq\ n + \underbrace{2n+1}_{odd\ n} \end{aligned}$$

**By the way:** students aware that $n^2$ is the sum of the first $n$ odd numbers.

# Not so immediate for-loops

- However, $sq$ is not a for-loop because each additive contribution $odd\ n = 2n + 1$ is dependent on $n$.
- What about $odd$ itself? Ask the students to try and exploit "its maths",

$$\begin{aligned} odd\ 0 &= 1 \\ odd(n+1) &= 2 + odd\ n \end{aligned}$$

  which lead immediately to for-loop $for\ (2+)\ 1$.
- Still, students don't know what to do with $sq$. What can we do about this?

# Two-variable for-loops

By putting *sq* and *odd* side by side,

```
sq 0 = 0                            odd 0 = 1
sq (n + 1) = sq n + odd n           odd (n+1) = 2 + odd n
```

observe that both functions share the same input pattern and can thus run "together", co-operating with each other. Thus proceed to **tupling**,

$$\langle sq, odd\rangle x \quad = \quad (sq\ x, odd\ x)$$

only to exploit "the maths" of this pair of functions:

$$\langle sq, odd\rangle 0 \ = \ (0, 1)$$
$$\langle sq, odd\rangle(i+1) = let\ (s, o) \ = \ \langle sq, odd\rangle\ i\ in\ (s+o, 2+o)$$

Clearly, this is for-loop $for((s, o) \mapsto (s+o, 2+o))(0, 1)$ which computes $i^2$ on variable $s$ and $odd\ i$ on variable $o$. Thus the code which follows:

# Calculation

$$\begin{cases} sq \cdot in = [\underline{0}\ , +] \cdot \mathsf{F}\langle sq, odd \rangle \\ odd \cdot in = [\underline{1}\ , (2+) \cdot \pi_2] \cdot \mathsf{F}\langle sq, odd \rangle \end{cases}$$

$\Leftrightarrow$ { mutual recursion law }

$$\langle sq, odd \rangle = (\!|\langle [\underline{0}\ , +], [\underline{1}\ , (2+) \cdot \pi_2] \rangle|\!)$$

$\Leftrightarrow$ { exchange law }

$$\langle sq, odd \rangle = (\!|[\langle \underline{0}, \underline{1} \rangle\ , \langle +, (2+) \cdot \pi_2 \rangle]|\!)$$

$\Leftrightarrow$ { $(\!|[\underline{i}\ , b]|\!) = \mathit{for}\ b\ i$ (going pointwise into for-loop) }

$$\langle sq, odd \rangle = \mathit{for}\ \langle +, (2+) \cdot \pi_2 \rangle\ (0, 1)$$

$\Leftrightarrow$ { unfolding loop body }

$$\langle sq, odd \rangle = \mathit{for}\ \lambda(s, o).(s + o, 2 + o)\ (0, 1)$$

# Two-variable for-loops

C code for *sq* (and *odd*, implicitly):

```
int sq(int n)
{
int s=0; int i; int o=1;
for (i=1;i<n+1;i++) {s+=o; o+=2;}
return s;
};
```

## Learning outcome

The number of **variables** required by a for-loop implementation of a given function over the natural numbers is the number of **mutually recursive functions** which such given function "unfolds" into once "their maths" are inspected.