MAU11602
Lecture 14
2026-02-19

Recursion

We'd like to able to define functions recursively like

```
fun f n = if   n = 0 orelse n = 1
          then n
          else f (n - 2) + f (n - 1)
```

This closely mirrors the way we normally describe the Fibonacci sequence: the 0'th and 1'st Fibonacci numbers are 0 and 1 and each later Fibonacci number is the sum of the previous two.

I mentioned earlier that `fun` is shorthand for creating an anonymous function and then naming it with a let expression so the code above should be equivalent to

```
val f = fn n => if   n = 0 orelse n = 1
                then n
                else f (n - 2) + f (n - 1)
```

There's a problem with variable scope though. The variable `f` appears in the subexpression

```
fn n => if n = 0 orelse n = 1 then n else f (n - 2) + f (n - 1)
```

but there's no `f` in scope, so how should this be evaluated?

Recursion, continued

The scope problem is a symptom of a different problem. The way we've defined let expressions and anonymous functions has the useful feature that all functions evaluate to a value no matter what arguments they are given.

If we define `f` as above, with the obviously intended meaning that the `f`'s in the else branch refer to the same `f` that's being defined, this function won't evaluate to anything when given a negative argument. It will just loop forever.

We'd like functions to terminate and we'd like to be able to define the Fibonacci sequence in the usual way, but we can't have both. We have to choose.

Pretty much all programming language choose to allow recursion, at the expense of possible non-termination.

This gives us a new way for expressions to fail to evaluate, distinct from things like division by zero.

It also requires a new definition of observational equivalence: two expressions are observationally equivalent if they have the same behaviour in any context, where same behaviour could mean evaluation to the same value or could mean both loop forever.

Pattern matching definition

The definition

```
fun f n = if   n = 0 orelse n = 1
          then n
          else f (n - 2) + f (n - 1)
```

is valid syntax in SML, which allows recursion, but it's not idiomatic. The idiomatic version uses pattern matching:

```
fun f1 0 = 0
  | f1 1 = 1
  | f1 n = f1 (n - 2) + f1 (n - 1)
```

This corresponds even more closely to the usual description of the Fibonacci sequence.

I use SML-like syntax mainly because it's close to standard mathematical usage.

I'll use pattern matching from now on to avoid conditionals and case expressions where they're not needed.

The function `f1` is observationally equivalent to the original `f` because both evaluate to the n'th Fibonacci number for non-negative n, and both fail to evaluate at all for negative n.

Partial functions

`f` and `f1` are what are called *partial functions*, meaning they fail to produce a value for all values of their arguments. Division is also a partial function, but for different reasons. The opposite of a partial function is a *total function*.

Sequences are functions. They just happen to be functions with a domain which is a subset of the integers.

Considered as a function on the integers, the Fibonacci sequence should really be total, not partial, because we can define f in such a way that $f(n) = f(n-2) + f(n-1)$ for *all* values of n:

```
fun f2 0 = 0
  | f2 1 = 1
  | f2 n = if   n > 0
           then f2 (n - 2) + f2 (n - 1)
           else f2 (n + 2) - f2 (n + 1)
```

This function terminates for all integer arguments, as we can prove by induction on the absolute value of the argument, so `f2` is a total function, unlike `f` and `f1`.

Partial functions, continued

We fixed the problem with the original definitions of the Fibonacci function, but there are some reasons for concern.
- We proved `f2` is total, but we can't expect a compiler to check that it's total. To a compiler `f2` looks very similar to `f1`. Before recursion the lexer, parser and type checker only allowed us to try to evaluate expressions which are guaranteed to terminate. So introducing totality has shifted the burden from the compiler to the programmer.
- Not all recursively defined functions are naturally total. The factorial function, for example, is naturally defined recursively by

```
fun g 0 = 1
  | g n = g ( n - 1 ) * n
```

This will also not terminate when given a negative argument, but here there's no way to define $n!$ for all integers n in a way which preserves its characteristic property: $(n+1)! = n! \cdot (n+1)$.

Efficiency

Our definition of the Fibonacci function has the advantage of clarity–it is transparently equivalent to the usual mathematical definition–but is horribly inefficient:

$$
\begin{aligned}
f(4) &= f(2) + f(3) \\
&= f(0) + f(1) + f(3) \\
&= 0 + 1 + f(3) \\
&= 1 + f(3) \\
&= 1 + f(1) + f(2) \\
&= 1 + 1 + f(2) \\
&= 2 + f(2) \\
&= 2 + f(0) + f(1) \\
&= 2 + 0 + 1 \\
&= 2 + 1 \\
&= 3
\end{aligned}
$$

More efficiency

Memoisation could solve our inefficiency problem, but we don't have the tools to implement it yet and there are simpler options.
We can compute less by computing more. Let's define a few auxialiary functions. function h which computes the n'th and $n+1$'st Fibonacci number together, as an ordered pair. It uses three auxiliary functions i and j, which compute the next or previous pair corresponding to a given pair, and k, which accesses the left element of an ordered pair

```
fun i (x, y) = (y, x + y)
fun j (x, y) = (y - x, x)
fun k (x, y) = x
fun h 0 = (0, 1)
  | h n = (if n > 0 then i else j) (h (n - 1))
fun f3 n = k (h n)
```

More efficiency, continued

Now

$$
\begin{aligned}
f(4) &= k(h(4)) \\
&= k(i(h(3))) \\
&= k(i(i(h(2)))) \\
&= k(i(i(i(h(1))))) \\
&= k(i(i(i(i(h(0)))))) \\
&= k(i(i(i(i(0,1))))) \\
&= k(i(i(i(1,1)))) \\
&= k(i(i(1,2))) \\
&= k(i(2,3)) \\
&= k(3,5) \\
&= 3
\end{aligned}
$$

Still more efficiency

In this small example `f3` doesn't look much better than `f2` did, but for large n it is much better. `f2` takes a number of steps which grows exponentially with n while `f3` takes a number of steps which grows linearly in n.

We can improve the efficiency further though. The evaluation trace on the previous slide kept growing to the right, as more and more pending function evaluations accumulated. We can fix this, and also reduce the number of auxiliary functions.

```
fun h (0, (x, y)) = x
  | h (n, (x, y)) = if   n > 0
                    then h (n - 1, (y, x + y))
                    else h (n + 1, (y - x, x))
fun f4 n = h (n, (0, 1))
```

Still more efficiency, continued

Now our evaluation trace is shorter and doesn't grow to the right.

$$\begin{aligned} f(4) &= h(4,(0,1)) \\ &= h(3,(1,1)) \\ &= h(2,(1,2)) \\ &= h(1,(2,3)) \\ &= h(0,(3,5)) \\ &= 3 \end{aligned}$$

Height corresponds (roughly) to execution time and width corresponds (roughly) to memory usage.

`f4` has linear time complexity, like `f3`, but `f4` has constant space usage, while `f3` had linear space usage.

As we gain in efficiency we start to lose a bit of clarity though. Is it still obvious that what we're computing is the Fibonacci sequence?

Further efficiency gains are possible, but I won't pursue them here.