```
fun append (xs,ys) =
    if xs=[]
    then ys
    else (hd xs)::append(tl xs,ys)

fun map (f,xs) =
    case xs of
        [] => []
      | x::xs' => (f x)::(map(f,xs'))

val a = map (increment, [4,8,12,16])
val b = map (hd, [[8,6],[7,5],[3,0,9]])
```

Programming Languages

Dan Grossman
2013

An Equivalent Structure

Equivalent implementations

A key purpose of abstraction is to allow *different implementations* to be *equivalent*

- *No* client can tell which you are using
- So can improve/replace/choose implementations later
- Easier to do if you *start* with more abstract signatures (reveal only what you must)

Now:

Another structure that can also have signature **`RATIONAL_A`**, **`RATIONAL_B`**, or **`RATIONAL_C`**

- But only *equivalent* under **`RATIONAL_B`** or **`RATIONAL_C`** (ignoring overflow)

Next:

A third equivalent structure implemented very differently

Equivalent implementations

Example (see code file):

- `structure Rational2` does not keep rationals in reduced form, instead reducing them "at last moment" in `toString`
  - Also make `gcd` and `reduce` local functions

- Not equivalent under `RATIONAL_A`
  - `Rational1.toString(Rational1.Frac(9,6)) = "9/6"`
  - `Rational2.toString(Rational2.Frac(9,6)) = "3/2"`

- Equivalent under `RATIONAL_B` or `RATIONAL_C`
  - Different invariants, but same properties
  - Essential that type `rational` is abstract