```
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

A Module Example

A larger example [mostly see the code]

Now consider a module that defines an Abstract Data Type (ADT)

- A type of data and operations on it

Our example: rational numbers supporting **`add`** and **`toString`**

```
structure Rational1 =
struct
datatype rational = Whole of int | Frac of int*int
exception BadFrac

(*internal functions gcd and reduce not on slide*)

fun make_frac (x,y) = …
fun add (r1,r2) = …
fun toString r = …
end
```

Library spec and invariants

Properties [externally visible guarantees, up to library writer]

- Disallow denominators of 0
- Return strings in reduced form (“4” not “4/1”, “3/2” not “9/6”)
- No infinite loops or exceptions

Invariants [part of the implementation, not the module’s spec]

- All denominators are greater than 0
- All `rational` values returned from functions are reduced

More on invariants

Our code maintains the invariants and relies on them

Maintain:

- **`make_frac`** disallows 0 denominator, removes negative denominator, and reduces result
- **`add`** assumes invariants on inputs, calls **`reduce`** if needed

Rely:

- **`gcd`** does not work with negative arguments, but no denominator can be negative
- **`add`** uses math properties to avoid calling **`reduce`**
- **`toString`** assumes its argument is already reduced