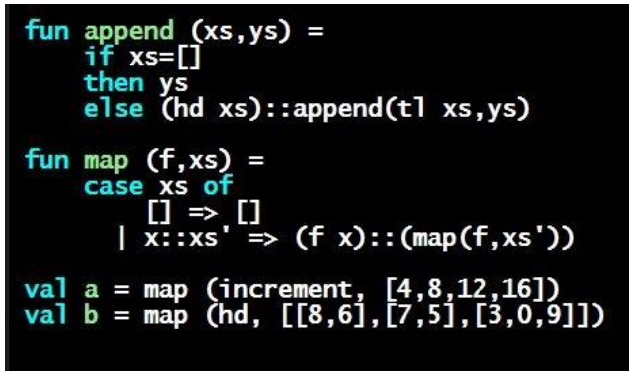

Programming Languages

Dan Grossman

2013

Signatures and Hiding Things

Signatures

- A *signature* is a type for a module
  - What bindings does it have and what are their types
- Can define a signature and ascribe it to modules – example:

```
signature MATHLIB =
sig
val fact : int -> int
val half_pi : real
val doubler : int -> int
end

structure MyMathLib :> MATHLIB =
struct
fun fact x = …
val half_pi = Math.pi / 2.0
fun doubler x = x * 2
end
```

In general

- Signatures

```
signature SIGNAME =
sig types-for-bindings end
```

  - Can include variables, types, datatypes, and exceptions defined in module

- Ascribing a signature to a module

```
structure MyModule :> SIGNAME =
struct bindings end
```

  - Module will not type-check unless it matches the signature, meaning it has all the bindings at the right types
  - Note: SML has other forms of ascription; we will stick with these [opaque signatures]

Hiding things

Real value of signatures is to to *hide* bindings and type definitions

- So far, just documenting and checking the types

Hiding implementation details is the most important strategy for writing correct, robust, reusable software

So first remind ourselves that functions already do well for some forms of hiding…

Hiding with functions

These three functions are totally equivalent: no client can tell which we are using (so we can change our choice later):

```
fun double x = x*2
fun double x = x+x
val y = 2
fun double x = x*y
```

Defining helper functions locally is also powerful

- Can change/remove functions later and know it affects no other code

Would be convenient to have "private" top-level functions too

- So two functions could easily share a helper function
- ML does this via signatures that omit bindings…

Example

Outside the module, **`MyMathLib.doubler`** is simply unbound

- So cannot be used [directly]
- Fairly powerful, very simple idea

```
signature MATHLIB =
sig
val fact : int -> int
val half_pi : real
end

structure MyMathLib :> MATHLIB =
struct
fun fact x = …
val half_pi = Math.pi / 2.0
fun doubler x = x * 2
end
```