CSE 341
Section 4

Spring 2019

With thanks to Daniel Snitkovskiy, Nick Mooney & Spencer Pearson

Today’s Agenda

- Mutual Recursion
- Module System Example
- Practice with Currying and High Order Functions

Mutual Recursion

- What if we need function f to call g, and function g to call f?
- This is a common idiom

```
fun earlier x =
    ...
    later x
    ...
fun later x =
    ...
    earlier x
    ...
```

Unfortunately this does not work ☹️

Mutual Recursion Workaround

- We can use higher order functions to get this working
- It works, but there has got to be a better way!

```
fun earlier f x =
    ...
    f x
    ...
fun later x =
    ...
    earlier later x
```

Mutual Recursion with `and`

- SML has a keyword for that
- Works with mutually recursive `datatype` bindings too

```
fun earlier x =
    ...
    later x
    ...
and later x =
    ...
    earlier x
    ...
```

Module System

- Good for organizing code, and managing namespaces (useful, relevant)
- Good for maintaining invariants (interesting)
- Hide implementation details

Deja vu?

We have similar things in Java!

It’s called interface!

Let’s implement an encoder!

An encoder should…

1. Be able to encrypt a message
2. Be able to decrypt a message
3. Never allow user to create an encrypted message directly

Matching signature and struct

Rules:

- Everything in signature must in struct
- Type in signature and type in struct must match
- Must specify type if type in signature is unspecified

Matching signature and struct

```
signature sigA =
sig
  type b
  val c : string -> string
end
```

Will it match?

```
structure structA1 :> sigA =
struct
    type b = int * int
    val c = fn s => 341
end
```

Matching signature and struct

```
signature sigA =
sig
  type b
  val c : string -> string
end
```

Will it match?

```
structure structA2 :> sigA =
struct
    exception a
    val c = fn s => s
end
```

Matching signature and struct

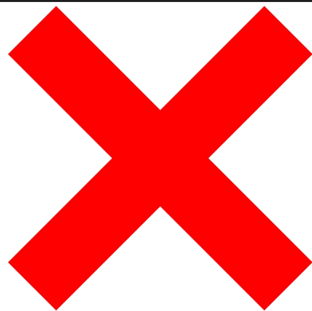
```
signature sigA =
sig
  type b
  val c : string -> string
end
```

Will it match?

```
structure structA3 :> sigA =
struct
    exception a
    type b = real * real
    val c = fn s => s
end
```

Matching signature and struct

```
signature sigB =
sig
  exception a of int
  type b = string * string
  type c
end
```



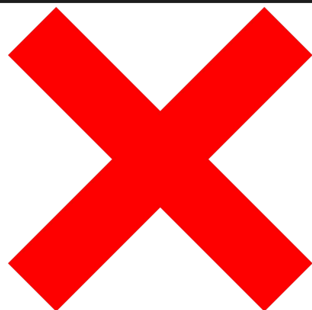
Will it match?

```
structure structB1 :> sigB =
struct
    exception a
    type b = string * string
    type c = int * real
end
```

Matching signature and struct

```
signature sigB =
sig
  exception a of int
  type b = string * string
  type c
end
```

Will it match?



```
structure structB2 :> sigB =
struct
    type b = string * string
    type c = int * real
end
```

Matching signature and struct

```
signature sigB =
sig
  exception a of int
  type b = string * string
  type c
end
```

Will it match?

```
structure structB3 :> sigB =
struct
    exception a of int
    type b = string * string
    datatype c = cse of int
end
```

Matching signature and struct

```
signature sigB =
sig
  exception a of int
  type b = string * string
  type c
end
```

Will it match?

```
structure structB4 :> sigB =
struct
    exception a of int
    type b = string * string
    type c = int * real
end
```

Interesting Examples of Invariants

- Ordering of operations
  - e.g. insert, then query
- Data kept in good state
  - e.g. fractions in lowest terms
- Policies followed
  - e.g. don't allow shipping request without purchase order

Currying and High Order Functions

- Some examples:
  - List.map:
    - `('a -> 'b) -> 'a list -> 'b list`
  - List.filter:
    - `('a -> bool) -> 'a list -> 'a list`
  - List.foldl:
    - `('a * 'b -> 'b) -> 'b -> 'a list -> 'b`

Practice: only_valid

- **Type:**
  - `(int * int) list -> (int * int) list`
- **Behavior:**
  - Does this look familiar?
  - Returns a list of int tuples with the elements of the input list of int tuples that match a certain criteria.
  - **Let’s just say the criteria is that both ints add up to 17**
    - `e.g. only_valid [(1,16),(2,5)] === [(1,16)]`

Code: only_valid

```
fun is_valid(x,y) = x + y = 17
val only_valid = List.filter is_valid
```

Practice: product_valid

- **Type:**
  - `(int * int) list -> bool`
- **Behavior:**
  - Returns a bool indicating whether all the products of elements in each tuple with both elements are positive are divisible by five.
  - `e.g. product_valid [(1,15),(~2,15)] === true  (since 15 mod 5 = 0)`
  - `e.g. product_valid [(1,13),(~2, ~2)] === false (since 13 mod 5 <> 0)`

Code: product_valid

```
fun is_valid(x,y) = x > 0 andalso y > 0
val only_valid = List.filter is_valid
val prods = List.map (fn (a, b) => a * b)
fun checker (prod, tst) =
   tst andalso (prod mod 5 = 0)
fun product_valid lst =
   List.foldl checker true (prods (only_valid lst))
```