PAUL G. ALLEN SCHOOL
OF COMPUTER SCIENCE & ENGINEERING

CSE 341
Section 4

Autumn 2018

With thanks to 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)

Deja vu?

We have similar things in Java!

It’s called interface!

Let’s implement a bank!

A bank should be able…

1. To open a new account
2. To deposit money
3. To withdraw money

```
1  public interface BankInterface {
2      // an account is being stored in some format that we don't know
3      public Account newAccount(String name, double initialDeposite);
4      public Account deposit(Account account, double amount);
5      public Account withdraw(Account account, double amount);
6  }
```

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

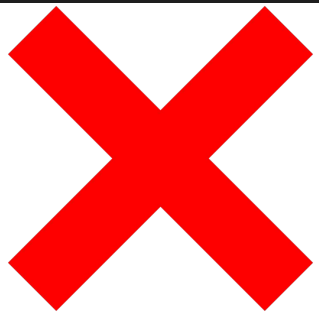
```
signature sigA =
sig
  type b
  val c : string -> string
end
```

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
```



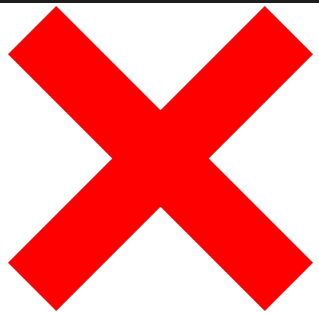
Will it match?

```
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
  - List.filter
  - List.foldl

Practice: flatten

- **Type:**
  - `'a list list -> 'a list`
- **Behavior:**
  - Does this look familiar?
  - Returns concatenation of list of lists.

Code: flatten

```
fun concat(acc, xs) = xs @ acc
fun flatten xs = List.foldl  concat [] xs
```

Alternative 1: op@

```
fun flatten2 xs = List.foldl (op@) [] xs
```

- Does this work? Why/why not?
- **This returns the reversed concatenation!**

Alternative 2: better style

```
val flatten3 = List.foldl concat []
```

- Does this work? Why/why not?
- **Nope, value restriction :(**

Practice: flat_map

- **Type:**
  - `'a list list -> 'a list`
- **Behavior:**
  - Does this look familiar?
  - Returns the concatenation of a list of list as one list.

Code: flat_map

```
fun flat_map f xs =
  case xs of
      [] => []
   |  x::xs' => (f x) @ flat_map f xs'
```

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**

Code: only_valid

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