CSE 341:
Programming Languages

Section AC with Nate Yazdani

agenda

- mutual recursion
- module system

mutual recursion

- what if we need a function **f** to call **g**, and a function **g** to call **f**

- this happens more often than you might think!

- a silly example, that sadly doesn't work :-(

```
fun even x =
    x = 0 orelse odd (x - 1)
fun odd x =
    x <> 0 andalso (x = 1 orelse even (x - 1))
```

mutual recursion

- SML has a special keyword to help us out

```
fun even x =
    x = 0 orelse odd (x - 1)
and odd x =
    x <> 0 andalso (x = 1 orelse even (x - 1))
```

- also works with mutually recursive **datatype** bindings

```
datatype even = Zero | ESucc of odd
and      odd = OSucc of even
```

mutual recursion

- SML has a special keyword to help us out

```
fun even x =
    x = 0 orelse odd (x - 1)
and odd x =
    x <> 0 andalso (x = 1 orelse even (x - 1))
```

- also works with mutually recursive **datatype** bindings

I fully admit that this is a contrived example :-)

```
datatype even = Zero | ESucc of odd
and      odd = OSucc of even
```

module system

- good for organizing your code and managing *namespaces*

- good for maintaining *invariants*

```
structure name = struct bindings end
```

practice with modules!

```
signature QUEUE = sig
  type 'a queue
  exception Underflow
  val empty : 'a queue
  val isEmpty : 'a queue -> bool
  val enqueue : 'a * 'a queue -> 'a queue
  val dequeue : 'a queue -> 'a * 'a queue
  val map : ('a -> 'b) -> 'a queue -> 'b queue
end
```

work together to design an SML module that implements the **QUEUE** abstract data type

practice with modules!

```
structure Queue :> QUEUE = struct
  exception Underflow
  type 'a queue = 'a list * 'a list
  val empty = ([], [])
  fun isEmpty ([], []) = true
    | isEmpty (_, _) = false
  fun enqueue (v, (en, de)) = (v :: en, de)
  fun dequeue ([], []) = raise Underflow
    | dequeue (en, v :: de) = (v, (en, de))
    | dequeue (en, []) =
      dequeue([], List.rev en)
  fun map f (en, de) =
    (List.map f en, List.map f de)
end
```