Standard ML (SML)

Marley Alford

CMSC 305

Intro to SML

- Functional programming language
- Compile-time type-checking
- polymorphic type inference
- Automatic storage management for data structures and functions
- Pattern matching
- Has a precise definition.

```
fun factorial 0 = 1
  | factorial n = n * factorial (n - 1)
```

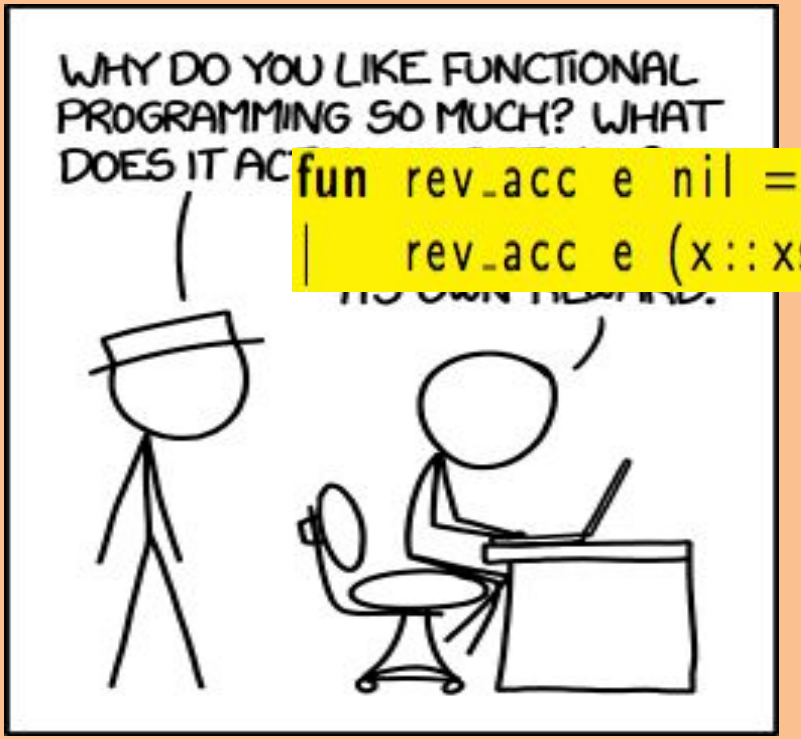
Intro to SML

- Functional programming language
- Compile-time type-checking
- polymorphic type inference
- Automatic storage management for data structures and functions
- Pattern matching
- Has a precise definition

```
fun factorial 0 = 1
  | factorial n = n * factorial (n - 1)
```

Functional

- Computation by evaluation of expressions, rather than execution of commands.



Syntax

SML

```
val foo = fn : int * int -> int
```

Haskell

```
foo :: int -> int -> int
```

Syntax

SML

```
(* comment *)
```

Haskell

```
--comment
```

Syntax

SML

```
fun factorial n =
        if n = 0 then 1 else n * factorial (n-1)
```

Haskell

```
factorial n = if n = 0 then 1 else n * factorial (n-1)
```

Intro to SML

- Functional programming language
- Compile-time type-checking
- polymorphic type inference
- Automatic storage management for data structures and functions
- Pattern matching
- Has a precise definition

```
fun factorial 0 = 1
  | factorial n = n * factorial (n - 1)
```

Intro to SML

- Functional programming language
- Compile-time type-checking
- polymorphic type inference
- Automatic storage management for data structures and functions
- Pattern matching
- Has a precise definition

```
fun factorial 0 = 1
  | factorial n = n * factorial (n - 1)
```

Type-Checking

- No implicit conversions between types.

Intro to SML

- Functional programming language
- Compile-time type-checking
- polymorphic type inference
- Automatic storage management for data structures and functions
- Pattern matching
- Has a precise definition

```
fun factorial 0 = 1
  | factorial n = n * factorial (n - 1)
```

Intro to SML

- Functional programming language
- Compile-time type-checking
- polymorphic type inference
- Automatic storage management for data structures and functions
- Pattern matching
- Has a precise definition

```
fun factorial 0 = 1
  | factorial n = n * factorial (n - 1)
```

Intro to SML

- Destructive update

```
signature ARRAY =
   sig
     type 'a array

     val array : int * 'a -> 'a array

     val fromList : 'a list -> 'a array
          exception Subscript
     val sub : 'a array * int -> 'a
     val update : 'a array * int * 'a -> unit

     val length : 'a array -> int

     ...
   end
```

Intro to SML

- Destructive update

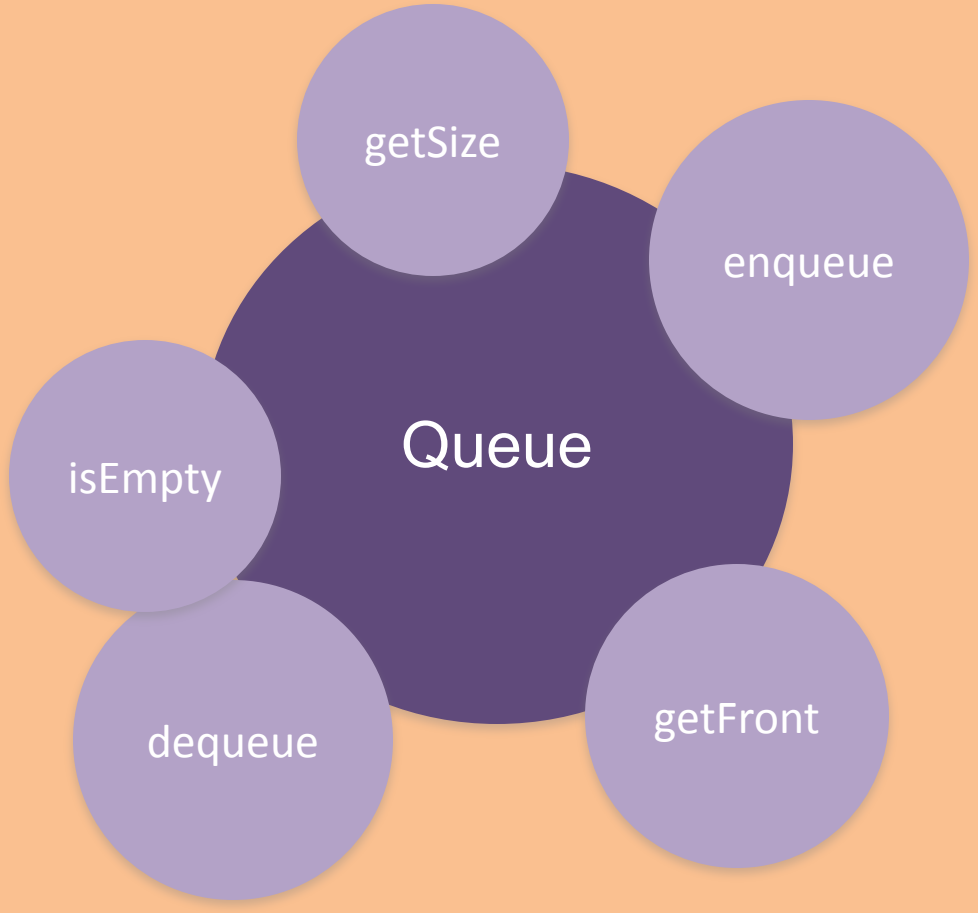
```
signature ARRAY =
   sig
     type 'a array  (An 'a array is a mutable fixed-length sequence of elements of type 'a. *)

     val array : int * 'a -> 'a array  (* array(n,x) is a new array of length n whose elements are all equal to x. *)
     val fromList : 'a list -> 'a array (* fromList(lst) is a new array containing the values in lst *)
          exception Subscript  (* indicates an out-of-bounds array index *)
     val sub : 'a array * int -> 'a (* sub(a,i) is the ith element in a. If i is out of bounds, raise Subscript *)
     val update : 'a array * int * 'a -> unit (* update(a,i,x); Set the ith element of a to x. Raise Subscript if i is not a legal index into a *)
     val length : 'a array -> int (* length(a) is the length of a *)

     ...
   end
```

SML vs. Haskell

- Pattern matching
- Hindley-Milner Type Inference 
- Parametric Polymorphism 
- Ad-Hoc Polymorphism 
- Monads 
- Syntactic Sugar
- Use of “_” as a wildcard variable

Modules

- Modules - Structures



Intro to SML

- Modules - Functors

```
functor PQUEUE(type Item
               val > : Item * Item -> bool
              ):QueueSig =
struct
   type Item = Item
   exception Deq
   fun insert e [] = [e]:Item list
     | insert e (h :: t) =
       if e > h then e :: h :: t
                else h :: insert e t
   abstype Queue = Q of Item list
   with
      val empty         = Q []
      fun isEmpty (Q []) = true
        | isEmpty _      = false
      fun enq(Q q, e)    = Q(insert e q)
      fun deq(Q(h :: t)) = (Q t, h)
        | deq _          = raise Deq
   end
end;
```

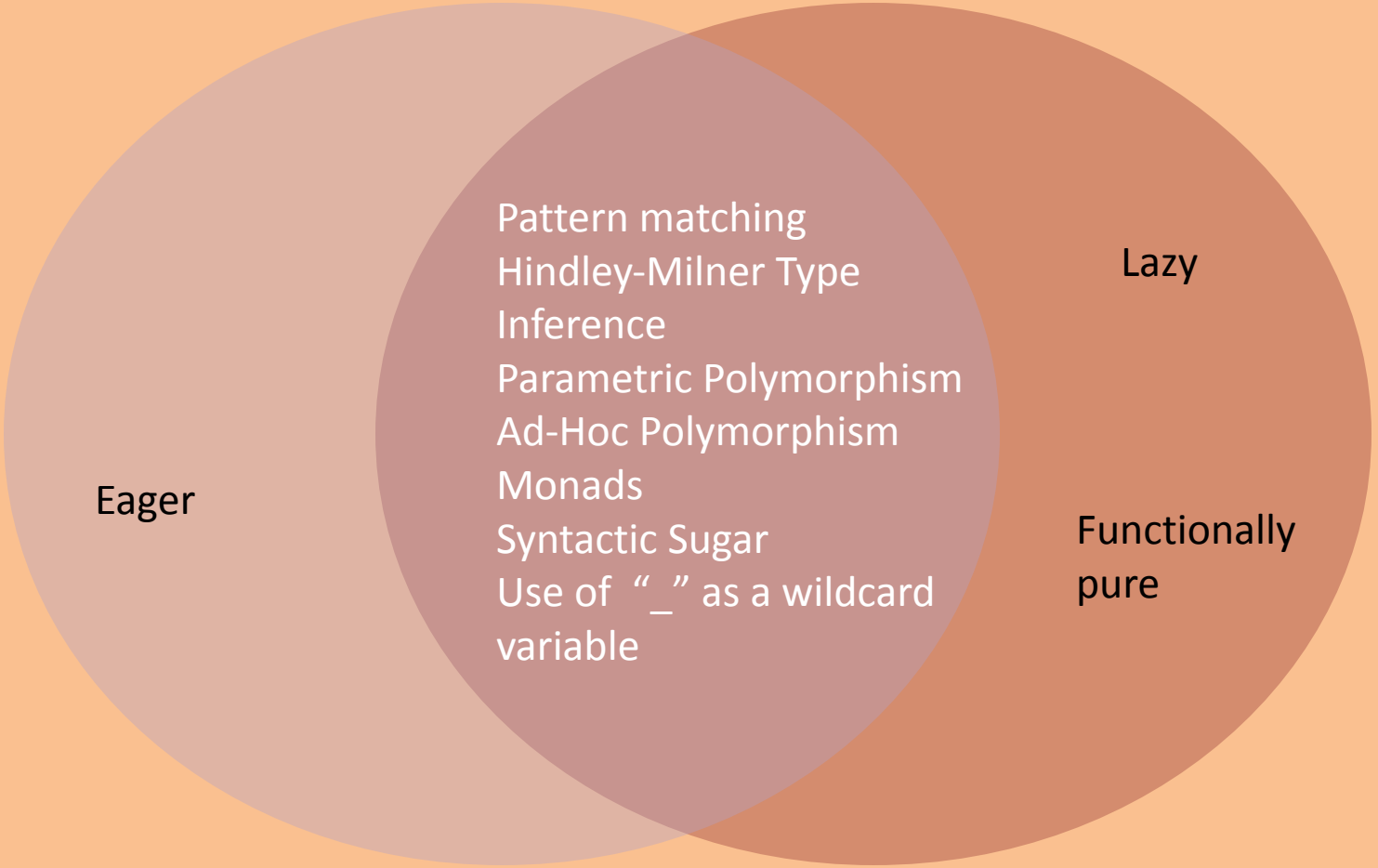
```
structure IntPQ = PQUEUE(type Item = int
                         val op >  = op > : int * int -> bool)
```

```
signature OrdSig =
sig
   type Item
   val > : Item * Item -> bool
end;
```

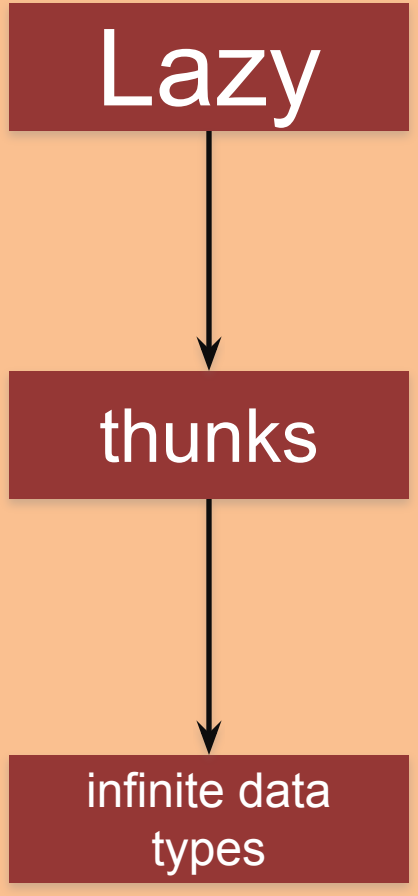
```
infix 4 ==
structure IntItem =
struct
   type Item = int
   val op == = (op = : int * int -> bool)
   val op >  = (op > : int * int -> bool)
end;
```

```
structure IntPQ = PQUEUE(IntItem)
```

SML vs. Haskell



SML vs. Haskell



SML vs. Haskell

- SML
- Not functionally pure
- Eager/Strict
- Less ‘user friendly’
- Interactive Interpreter

- Haskell
- Functionally pure
- Lazy
- More ‘user friendly’
- Interactive Shell

SML vs. Haskell

- Haskell - more mindful of modern software development practices, and less 'theoretically pure' than SML.
- SML – More powerful module system, and more concise and reusable code.

- *Both use but Haskell takes this further, providing sugaring for Monads and Arrows, and advanced pattern matching.*

Thank You!

Infinite Lists

Unique to Haskell:

```
inf = 1 : map (+1) inf
```

[1,2,3,4,5,...]

```
ones = 1 : ones
```

[1,1,1,1,1,...]

Tail Recursion

```
fun rev_acc e nil = e
|   rev_acc e (x::xs) = rev_acc (x::e) xs
```

This function is tail recursive:

- no computation happens after the recursive call
- value of recursive call is the return value
- thus, no variables are referenced after recursive call

This kind of recursion is actually iteration in disguise!