Today: More ML

- Let
- Defining New Types
- Polymorphism
- Higher-Order Functions
- Signatures
- References
- Case Statements
- Exceptions
- Records

Let

```
- fun xten(x:real) =
= let val x5 = x*x*x*x*x
=     fun square x = x * x
= in
=     square x5
= end;

- fun xten(x:real) =
= let val x5 = x*x*x*x*x
=     val x2 = x * x
= in
=     square x5
= end;
```


Last Time: Basic ML

- Simple Operators: integers, floating point values, strings
- Recursive and mutually recursive functions
- Tuples: pairs, triples, quadruples, ...
- Lists: [1,2,3], ::, @, ...
- Simple Pattern matching

Let

```
- fun xten(x:real) = x*x*x*x*x*x*x*x*x*x;
 val xten = fn :  real -> real

- xten(2.0);
 val it = 1024.0 :  real

- fun xten(x:real) =
= let val x5 = x*x*x*x*x in
=   let val x10 = x5 * x5 in
=     x10
=   end;
= end;

val xten = fn :  real -> real
```

Let

```
fun f(x:real) =
let val x4 = x*x*x*x
    val x6 =
        let val x3 = x*x*x in
          x3 * x3
        end
in
    x6 * x4
end;
```


Datatypes

A datatype declaration gives you:

1. A new type.
2. A new collection of data constructors.

```
- datatype money =
    Nickel
  | Dime
  | Quarter
  | Bill of int;

datatype money = Bill of int | Dime | Nickel | Quarter

- Nickel;;
val it = Nickel : money
- Bill 10;;
val it = Bill 10 : money
```


Datatypes

A common programming paradigm in ML:

1. Think about the structure of data.
2. Declare a new datatype that describes that structure.
3. Write (recursive) functions that are defined on the structure of the data.

```
- fun value(Nickel) = 5
=    | value(Dime) = 10
=    | value(Quarter) = 25
=    | value(Bill(x)) = x*100;

val value = fn :  money -> int

- value(Bill(10));
 val it = 1000 :  int
```


Recursive Datatypes

```
datatype intTree =
  Leaf of int
| Node of intTree * intTree;

datatype intTree = Leaf of int | Node of intTree * intTree

fun treelist (Leaf x) = [x]
  | treelist (Node (t1,t2)) = (treelist t1) @ (treelist t2)
```

A Fancier Tree to List Conversion

The previous tree to list function took O(n^2) time to execute. This one takes O(n).

```
fun treelist2 t =
  let fun loop (Leaf x) cont = x :: (cont ())
        | loop (Node (t1,t2)) cont =
               loop t1 (fn () => loop t2 cont)
  in
    loop t
  end;
```

The inner `loop` is called a *higher-order* function because it takes a function (`cont`) as an argument. This example also shows how to create *anonymous* functions: `fn x => ...`

Higher-Order Functions

- We also consider functions that return functions to be higher-order:

```
- fun add(x:int) = let
=   fun g(y) = x + y
= in
=   g
= end;
val add = fn : int -> int -> int
- val z = add(4);
val z = fn : int -> int
- z(5);
val it = 9 : int
```


Polymorphic Functions

A function is *polymorphic* if the types of its parameters are partially or completely flexible.

- Completely Flexible:

```
- fun identity(x) = x;
val identity = fn : 'a -> 'a
- identity(2);
val it = 2 : int
- identity([1,2]);
val it = [1,2] : int list
```

- Partially Flexible:

```
- fun reverse(nil) = nil
= | reverse (x::xs) = reverse(xs) @ [x];
val reverse = fn : 'a list -> 'a list
```


Polymorphic, Higher-Order Functions

- By using polymorphism together with higher-order functions, you can often write very general code.

```
(* map : ('a -> 'b) * 'a list -> 'b list *)

fun map f nil = nil
  | map f (x::xs) = (f x)::(map f xs);

(* val square = fn : int -> int *)

fun square(x:int) = x * x;

val squares =  map square [1,2,3,4];
```

- Arguments of higher-order functions are often one-use functions, and so one normally makes them anonymous functions (if they aren't too big).

```
val sq2 = map (fn x => x * x) [1,2,3];
```

Function Composition

- C(x) = F(G(x))

```
(* comp : ('a -> 'b) * ('c -> 'a) -> 'c -> 'b *)
fun comp(F, G) = let
  fun C(x) = F(G(x))
in
  C
end;

- fun F(x) = x + 3;
val F = fn : int -> int
- fun G(x) = x * 10;
val G = fn : int -> int
- val H = comp(F, G);
val H = fn : int -> int
- H(10);
val it = 103 : int
```


Polymorphic Type Definitions

- Use 'a or 'b, etc. to represent any type in polymorphic datatype declarations:

```
datatype 'a btree =
  Leaf of 'a
| Node of 'a btree * 'a btree;

- val x = Node(Leaf(2), Leaf(3));
val x = Node (Leaf 2,Leaf 3) : int btree

- val y = Node(Leaf("a"), Node(Leaf("b"), Leaf("c")))
val y = Node (Leaf "P.U.",Node (Leaf "cs",Leaf "320"))
  : string btree

- val z = Node (Leaf y, Leaf y);
val z = ... : string btree btree
```


Modularity

ML has one of the most sophisticated module systems in the world.

- *Structures* encapsulate the implementation of a program component.
- Structures encapsulate type definitions as well as value definitions
- *Signatures* serve as "types" for structures.
- Signatures state the available types and give types to terms.
- *Opaque* signatures implement "information hiding" by making certain definitions inaccessible to user.

Structures and Signatures

```
structure someStruct = struct
=   val a = 5;
=   fun listCount(nil) = 0
=   |   listCount(x::xs) = 1 + listCount(xs);
=   datatype color = Red | Green | Blue;
=   val green = Green;
=   val blue = Blue;
=   val red = Red;
=   fun isGreen Green = true
=     | isGreen _ = false;
= end;
structure someStruct : sig
    datatype color = Blue | Green | Red
    val a : int
    val listCount : 'a list -> int
    val green : color
    val blue : color
    val red : color
    val isGreen : color -> bool
  end
```

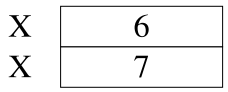

Signatures

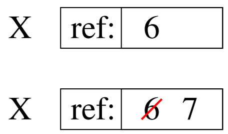
Signatures often serve to make some type abstract and to hide components of a structure:

```
signature S = sig
   type color;
   isGreen : color -> bool
   green : color
   blue : color
   red : color
end;

- structure T <: S = someStruct;
```


References

```
- val x = 6;
val x = 6 : int
- val x = 7;
val x = 7 : int

- val x = ref 6;
val x = ref 6 : int ref
- x;
val it = ref 6 : int ref
- !x;
val it = 6 : int
- x := 7;
val it = () : unit
- !x;
val it = 7 : int
```





Case Statement

```
- datatype money = Nickel | Dime | Quarter | Bill of int;
datatype money = Bill of int | Dime | Nickel | Quarter
- fun value(Nickel) = 5
= |   value(Dime) = 10
= |   value(Quarter) = 25
= |   value(Bill(x)) = x * 100;
val value = fn : money -> int

- fun value(x) =
=   case x of
=       Nickel => 5
=     | Dime => 10
=     | Quarter => 25
=     | Bill(x) => x * 100;
val value = fn : money -> int
```

Records

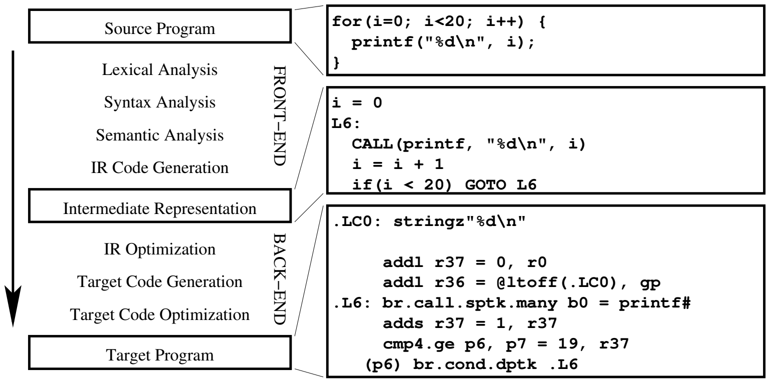
- *Records* are a more general form of a tuple.

```
- val x = {name="John", age=21, ID=1111};
val x = {ID=1111,age=21,name="John"} :
    {ID:int, age:int, name:string}
- #name(x);
val it = "John" : string
- #age(x);
val it = 21 : int
```

- Tuples are equivalent to numbered records:

```
- val y = ("a","b","c");
val y = ("a","b","c") : string * string * string
- val y = {1="a",2="b",3="c"};
val y = ("a","b","c") : string * string * string
- #3(y);
val it = "c" : string
```


The Compiler



- Lexical Analysis: Break into tokens (think words, punctuation)
- Syntax Analysis: Parse phrase structure (think document, paragraphs, sentences)
- Semantic Analysis: Calculate meaning

Exceptions

- *Exceptions* allow errors to be caught easily regardless of function return type.
- For example: n! not defined for $n < 0$:

```
- exception negativeInt;
exception negativeInt

- fun fact(n) = if n = 0 then 1
=               else if n > 0 then n * fact(n-1)
=                    else raise negativeInt;
val fact = fn :  int -> int

- fact(~1);
uncaught exception negativeInt raised
```


Records

- Can use pattern-matching with records.

```
- fun getAge(person,
=            nil:{name:string,age:int,ID:int} list) =
=       raise notFound
= |   getAge(person, {name=n, age=a, ID=i}::xs) =
=       if person = n then a
=       else getAge(person, xs);
```

- Rather than creating new variable to represent value of a field, may use field name itself. Helps increase clarity.

```
- fun getAge(person,
=            nil:{name:string,age:int,ID:int} list) =
=       raise notFound
= |   getAge(person, {name, age, ID}::xs) =
=       if person = name then age
=       else getAge(person, xs);
```