# ML Overview

Main references are:

- are Harper's TR (online) and
- Kathleen Fischer's lecture notes (online at www-cs.stanford)
- Alex Aiken's 264 lecture notes (online)

There is a class newsgroup: `ucb.class.cs263`

# Chacteristics of ML

- “Functional language with imperative features”
  - eager, as opposed to lazy evaluation:
  - side effects (evaluation order is important)
  - Here ”functional” means ”has lots of functions”.
  - Has concurrent extentions.
- Expression-based.
- Strongly typed, with a highly flexible type system.
  - Every expression has a type.
  - No casts/loopholes.
  - Polymorphism.
  - Abstract data types.
- Statically scoped.
  - Exceptions. (Milner’s Turing Award citation mentions: ”type-safe exception mechanism”)
  - Module system.

# Advantages of strong typing

- Documentation.
- Earlier detection of (some) errors.
- Faster code may be generated.

Useful for programming in the large and readability. A criticism of strong typing is programmer overhead.

Note distinction between static vs. dynamic types and language that are have static types but with loopholes in the type system.

# Summary of Scalar Types

Atomic Types:

- `Unit`

  `( ) :  unit`

  Used as the result of functions executed merely for side-effects (ie, procedures) and as arguments to functions of no arguments.

- `Bool`

  `true, false :  bool, if...  then...  else...`

  The "else" branch is required, because it is a conditional expression, not a conditional command. Both branches must have the same types.

- Integers: `0,1,2,...,-1,-2,...`, with operation `+, mod, *,...`

- Strings

  `"Rainy day"`

- Real

  `1.0, 2.0, ...`

  Real constants differ from integer constants, because the type inference system has no other way to distinguish between them.

# ML Interactive session

Input:

```
- 9 + 3;
val it = 12 : int

- (1, true, 8);
val it = (1,true,8) : int * bool * int

- 1 :: 2 :: 3 :: nil;
val it = [1,2,3] : int list

- {foo="Foo",bar=3};
val it = {bar=3,foo="Foo"} : {bar:int, foo:string}
```

The variable `it` always refers to the result of the last expression evaluation. In addition to scalar types there are tuples (ordered heterogeneous), lists (homogeneous), records (named heterogeneous).

# Naming

All variables must be declared before use, done using the keyword `val`. Variables may be declared local to other variable declarations:

```
- local
     val x = 10
  in
     val u = x + x
  end;
```

or local to an expression evaluation.

```
- let
     val x = 10
  in
     x + x
  end;
```

In both cases definitions nest, so innermost declarations mask outer level ones in case of conflict.

# Function Declarations

General form is:

```
fun f( < pattern > ) = < exp >
```

```
- fun f(x,y) = x + y;
```

The argument to f is a tuple, which is broken down by pattern matching into its first and second arguments.

```
- fun length(nil) = 0
    | length( x :: xs ) = 1 + length(xs);

val length = fn : 'a list -> int
```

The function length is written via a *clausal definition.* When applied to an argument, the clauses are pattern-matched in the order they are written. Because length is guaranteed to get a list as its argument by the type system, these two clauses cover all values that could possibly be passed to this function.

# Exceptions

```
- val (left, right) = (3, true);
val left = 3 : int
val right = true : bool

- val l = [1, 2, 3];
- val hd :: tl = l;

exception nthex;
- fun nnth (x :: _ , 0) = x
=  | nnth (x :: xs, n) = if n > 0 then nnth (xs, n-1)
=                                 else raise nthex
=  | nnth _ = raise nthex;
val nnth = fn : 'a list * int -> 'a

- nnth([1, 2, 3], 2);
val it = 3 : int
```

## Modules

*Module* is a loose term roughly meaning a related set of functions and values and an interface. Modules address one of the fundamental naming problems: large name spaces.

The purpose of the ML module system is to allow one to organize coherent interfaces and ensure safety.

An ML "structure" is a collection of ML functions, values, types, exceptions, (sub)structures, etc. An example:

```
structure Foo =
    struct
            type value = int;
            fun leq(i:val,j:val) = i < j;
            fun add1(i:int) = i + 1;
            fun id(x) = x;
            exception Bad;
    end;
```

# Modules

Two rules for ML Modules:

1. components are uniquely named
2. components are valid ML declarations

A structure may contain free or bound variables and may be nested. Structure provide a hierarchical name space. E.g., we refer the `add1` function in `Foo` as `Foo.add1`.

# Types for Modules

The "type" of a structure is a signature.

```
signature FooSig =
        sig
                type value;
                fun add1: int -> int;
                fun id: string -> string;
                exception Bad;
        end;
```

A signature declaration looks suspiciously like a structure declaration, but the two entities are talking about completely different things. May help to think of a signature as a set of structures.

# Matching Structures

What does it mean for a signature and a structure to "match"? There are two requirements:

1. If there is a component with name X in the signature, then there is a component with name X in the structure.
2. The types of components with same name match, in the sense that the type in the signature is an instance (i.e., less general than) the type in the structure.

For example, the signature FooSig and structure Foo match, because all components named in FooSig appear in Foo and have compatible types. However, note that FooSig does not expose the function Foo.leq; this function is hidden by the interface. In addition, FooSig restricts the type of the function "id" to work only on strings.

# History

- LCF: Logic For Computable Function
  * Scott 1969 paper.
  * Milner 1970-1971 project at Stanford.
  * continued 1972-?? at University of Edinburgh.
- ML: Meta-Language of LCF System
- LCF is a logic for describing/proving properties about computable functions. ML is the language for writing programs that construct proofs of LCF formulas.
- Main notion of such programs: Proof tactics.

# Proof Tactics

A proof tactic is a function that, given a formula, attempts to find a proof of that formula. In the system, there is a type for formulas and a type for proofs. Type of tactics: `Formula -> Proof`.

Language ideas to support this usage:

1. The type system guarantees proof correctness.
   If p:proof, then p is a correct proof.
2. Since tactics may fail, the language has an exception mechanism.

```
                     succeed and return proof
tactic(formula) = /
                  \
                     fail, raising an exception.
```

3. `f :  A -> B` means For all `x` in `A`, either `f(x)` terminates normally with result in `B` or f(x) diverges or raises an exception.
4. Exception handlers must preserve type correctness.
5. Emphasis on exception mechanisms and higher-order functions.