ML: a quasi-functional language with strong typing

- Conventional syntax:
  - `val x = 5;` (*user input*)
  - `val x = 5: int` (*system response*)
  - `fun len lis = if (null lis) then 0 else 1 + len (tl lis);`
    - `val len = fn : 'a list -> int`
- Type inference for local entities
  - `x * x * x;`
    - `val it = 125: int` (* it denotes the last computation*)
Operations on lists

Similar to LISP, with \((\text{hd, tl, ::})\) instead of \((\text{car, cdr, cons})\)

\[
\text{fun append} \ (x, y) = \text{if null} \ (x) \ \text{then} \ y \\
\quad \text{else hd} \ (x) :: \text{append} \ (\text{tl} \ (x), y);
\]

> \quad \text{val append} = \text{fn} : \text{‘a list} * \text{‘a list} -\rightarrow \text{‘a list}

(* a function that takes a pair of lists and yields a list *)

- \quad \text{fun upto} \ (m, n) = 
  \quad \quad \text{if} \ m > n \ \text{then} \ [\ ] \ \text{else} \ m :: \text{upto} \ (m+1, n);

\[
\text{val upto} = \text{fn} : \text{int} * \text{int} -\rightarrow \text{int list}
\]

(*a function that takes two numbers and yields a list *)
Patterns

A simple mechanism to describe and name parts of a structure:

```plaintext
fun prod [ ] = 1  (* if list is empty *)
| prod (n::ns) = n * prod (ns);  (* n is first element *)
```

- A list can be described by its head and its tail
  ```plaintext
  fun maxl [m] : int = m
  | maxl (m::n::ns) if m > n then maxl (m ::ns)
     else maxl (n :: ns)
  ```

- > ** warning: patterns not exhaustive
Recursion and iteration

- A function that returns the first n elements of a list:
  
  ```suffix
  fun take ([ ], n) = [ ]
  | take (x::xs, n) = if n > 0 then x::take (xs, n-1) else [ ];
  ```

  **Non-recursive version:** introduce an accumulator argument:

  ```suffix
  fun itake ([ ], _, taken) = taken (* _ is pattern for anything *)
  | itake (x::xs, n, taken) =
      if n > 0 then itake (xs, n-1, x::taken)
      else taken;
  > var itake = fn: 'a list * int * 'a list -> 'a list
  ```
Block structure and nesting

• **Let** provides local scope:

```plaintext
fun findroot (a, x, acc) =
  (* standard Newton-Raphson *)
  let val nextx = (a / x + x) / 2.0 (*next approximation *)
  in
  if abs (x - nextx) < acc*x then nextx
    else findroot (a, nextx, acc)
  end;
  (*tail recursion rather than iteration *)
```
A classic in functional form: quicksort

```ml
fun quick [ ] = [ ]
  | quick [x:real] = [x]
  | quick (a::bs) =
    let fun partition (left, right [ ]) =
      (quick left) @ (quick right)
    in
      partition (left, right, x::xs) =
        if x <= a then partition (x::left, right, xs)
        else partition (left, x:: right, xs)
    in
      partition ([ ], [ ], bs) end;
>  val quick : fn : real list -> real list;
```
Functionals

fun exists pred [] = false
   | exists pred (x::xs) = (pred x) or else exists pred xs;

(* pred is a predicate : a function that delivers a boolean *)

val exists : fn : ('a -> bool) -> 'a list -> bool

fun all pred [] = true
   | all pred (x::xs) = (pred x) and also all pred xs;

fun filter pred [] = []
   | filter (x::xs) =
      if pred x then x :: filter pred xs
      else filter pred xs;

val filter = fn : ('a -> bool) -> 'a list -> 'a list
Currying: partial bindings

a b c means (a b) c: (a b) yields a function that is applied to c
- fun app2 x y = if null x then y
  = else (hd x) :: app2 (tl x) y;
val app2 = fn : ‘a list -> ‘a list -> a’list

(*a function that given a list yields a function that takes a list and yields a list *)

- val ap123 = app2 [1, 2, 3];
val ap123 = fn : int list -> int list
- ap123 [10, 20, 30];
val it = [1, 2, 3, 10, 20, 30]
A single formal is sufficient

If function takes more than one argument, say that it takes a single composite one:

```plaintext
fun exp (x, n) = if n = 0 then 1
  else x * exp (x, n-1);
```

```plaintext
val exp = fn : int * int -> int
```

Single argument is a **tuple** (int, int)
Can also define records with named components
The type system

Primitive types: int, char, real, string
Constructors: list, array, product (record), function
Every legal expression has a corresponding type expression
the interpreter builds the type expression for each input
type checking requires that type expression of functions and their arguments match, and that type expression of the context of a call match the result type of the function being called
Type inference

- fun incr x = x + 1;
  val incr = fn : int -> int
  - because of its appearance in (x+1), x must be integer

- fun add2 x = incr (incr x);
  val add2 = fn : int -> int
  - incr returns an integer, so add2 does as well
  - x is argument of incr, so must be integer

  val wrong = 10.5 + incr 7;

  Error: operator and operand don’t agree
Polymorphism

- fun len x = if null x then 0
  = else 1 + len (tl x);

• works for any kind of list. What is its type expression?

  val len : fn = ‘a list -> int

• ‘a denotes a type variable: Implicit universal quantification:

  for any a, len applies to a list of a’s.

  - fun copy lis = if null lis then nil
  = else hd (lis) :: copy (tl lis);
Type inference and unification

- Type expressions are built by solving a set of equations
  - fun reduce f lis init = if null lis then init
    = else f ((hd lis), reduce f (tl lis) init)
  
  null : ‘a list -> bool
  hd : ‘b list -> ‘b
  tl : ‘c list -> ‘c list
  apply : (‘d -> ‘e) * ‘d -> ‘e
  
  let b be the type of init. Let a be the element type of lis. Then f
  takes an a and a b and returns a b.
  - val reduce = fn : (‘a * ‘b -> ‘b) -> (‘a list) -> ‘b -> ‘b
Unification algorithm

A type variable can be unified with another variable
   ‘a unifies with ‘b  =>  ‘a and ‘b are the same
A type variable can be unified with a constant
   ‘a unifies with int  =>  all occurrences of a mean int
A type variable can be unified with an expression
   ‘a unifies with ‘b list
   ‘a does not unify with ‘a list
A constant can be unified with itself  int is int
An expression can be unified with another expression if the constructors are identical and if the arguments can be unified (recursive):
   (int -> int) list unifies with ‘a list, ‘a is a function on integers
Type system does not handle overloading well

- **fun** plus \( x \, y = x + y; \)

- operator is overloaded, cannot be resolved from context (error in some versions, defaults to int in others)

- Can use explicit typing to select interpretation:

  - fun mix (x, y ,z) = x * y + z:real;
  mix : (real * real * real) -> real
Parametric polymorphism vs. generics

A function whose type expression has type variables applies to an **infinite set of types**.

Equality of type expressions means structural equivalence.

All applications of a polymorphic function use the same body: no need to instantiate.

- let val ints = [1, 2, 3];
  val strs = ["this", "that"];
  in
  len ints + len strs (* int list -> int, string list -> int *)
  end;
val it = 5: int
User-defined types and inference

A user-defined type introduces constructors:

```
datatype tree = leaf of int | node of tree * tree
```

- `leaf` and `node` are type constructors

  can define functions by pattern:

  ```
  - fun sum (leaf (t)) = t
  = | sum (node (t1, t2)) = sum t1 + sum t2;
  val sum = fn : tree -> int
  ```
Parameterized datatypes

- \textbf{fun} flatten (leaf (t)) = [t]

= \text{ | flatten (node (t1, t2)) = flatten (t1) \@ flatten (t2);}

\textbf{flatten: tree \rightarrow int list}

- \textbf{datatype} ‘a gentree = leaf of ‘a

= \text{ | node of ‘a gentree * ‘a gentree;}

- \textbf{val} names = node (leaf ("this"), leaf ("that"));

\textbf{val} names = … string gentree