Functions

Some function definitions:
- fun square(x:int):int = x * x;
  val square = fn : int -> int
- fun swap(a:int, b:string):string*int = (b,a);
  val swap = fn : int * string -> string * int

Some function types:
  int -> int
  int*string -> string*int
  • in general: T_arg(s) -> T_result(s)
  • * binds tighter than ->

Some function calls:
- square(3);
  val it = 9 : int
- swap(3 * 4, "billy" ^ "bob);
  val it = ("billybob",12) : string * int

Expression-orientation

Function body is a single expression
fun square(x:int):int = x * x
  • not a statement list
  • no return keyword needed

Like equality in math
  • a call to a function is equivalent to its body,
    after substituting its formals for the actuals in the call
  square(3) ⇔ (x*x)[x→3] ⇔ 3*3

There are no statements in ML, only expressions
  • simplicity, regularity, and orthogonality in action

What would be statements in other languages
are recast as expressions in ML

Expression Orientation: A Big Idea

If expression

General form:
if test then e1 else e2
  • return value of either e1 or e2,
    based on whether test is true or false
  • cannot omit else part

- fun max(x:int, y:int):int =
  = if x >= y then x else y;
  val max = fn : int * int -> int

Like ?: operator in C
  • don’t need the separate if statement

Static typechecking of if expression

What are the rules for typechecking an if expression?
What’s the type of the result of if?

Some basic principles of typechecking:
  • values are members of types
  • the type of an expression must include all the values that
    might possibly result from evaluating that expression at
    run-time

Requirements on each if expression:
  • the type of the test expression must be bool
  • the type of the result of the if must include whatever values
    might be returned from the if
  • the if might return the result of either e1 or e2

A solution: e1 and e2 must have the same type,
  and that type is the type of the result of the if expression
Let expression

An expression that introduces a new nested scope with local variable declarations
- unlike {...} statements in C, which don't compute results

General form:
```markdown
let val id_1 : type_1  = e_1
...
val id_n : type_n  = e_n
in
  e_body
end
```
- `type_i` are optional; they'll be inferred

Evaluates each \( e_i \) and binds it to \( id_i \), in turn
- each \( e_i \) can refer to the previous \( id_1..id_{i-1} \) bindings

Evaluates \( e_{body} \) and returns it as the result of the `let` expression
- can refer to all the \( id_1..id_n \) bindings

The \( id_i \) bindings disappear after \( e_{body} \) is evaluated
- they're in a nested, local scope

Example scopes

- `val x = 3;
val x = 3 : int`
- `fun f(y:int):int =`
- `let
  val z = x + y
  val x = 4
  in
  (let
   val y = z + x
   in
   x + y + z
   end)
  = end;
val f = fn : int -> int`
- `val x = 5;
val x = 5 : int`
- `f(x);`