Goals for today

- More ML essentials
- Discuss some “first-week” gotchas
  - We will learn more and better constructs soon

Note: These slides (and most slides all quarter) will make much more sense in conjunction with the corresponding code file (lec02.sml).

Recall a program is a sequence of bindings...
Function Definitions

... A second kind of binding is for functions

Syntax: \[ \text{fun } x_0 \ (x_1 : t_1, \ldots, x_n : t_n) = e \]

Typing rules:

1. Context for \( e \) is (the function’s context extended with)
   \[ x_1 : t_1, \ldots, x_n : t_n \text{ and } \]
2. \( x_0 : (t_1 \times \ldots \times t_n) \rightarrow t \) where:
3. \( e \) has type \( t \) in this context

(This “definition” is circular because functions can call themselves and the
type-checker “guessed” \( t \).)

(It turns out in ML there is always a “best guess” and the type-checker can
always “make that guess”. For now, it’s magic.)

Evaluation: \( \text{A FUNCTION IS A VALUE.} \)
Function Applications (a.k.a. Calls)

Syntax: \[ e_0 \ (e_1, \ldots, e_n) \]

Typing rules (all in the application’s context):

1. \( e_0 \) must have some type \((t_1 \ * \ \ldots \ * \ t_n) \rightarrow t\)
2. \( e_i \) must have type \( t_i \) (for \( i = 1, \ldots, n \))
3. \( e_0 \ (e_1, \ldots, e_n) \) has type \( t \)

Evaluation rules:

1. \( e_0 \) evaluates to a function \( f \) in the application’s environment
2. \( e_i \) evaluates to value \( v_i \) in the application’s environment
3. result is \( f \)'s body evaluated in an environment extended to bind \( x_i \) to \( v_i \) (for \( i = 1, \ldots, n \)).

(“an environment” is actually the environment where \( f \) was defined)
Some Gotchas

• The * between argument types (and pair-type components) has nothing to do with the * for multiplication

• In practice, you almost never have to write argument types
  – But occasionally needed; maybe for homework 1
  – Sometimes improves error messages and clarity of code
  – But type inference is a very cool thing in ML
  – Types unneeded for other variables or function return-types

• Context and environment for a function body includes:
  – Previous bindings
  – Function arguments
  – The function itself
  – But not later bindings
Recursion

- A function can be defined in terms of itself.
- Of course, the recursive calls must solve “smaller” or “simpler” problems.
- This is more powerful than loops and often more convenient.
- Many, many examples to come in 341.
Pairs

Our first way to build compound data out of simpler data:

- Syntax to build a pair: \((e_1, e_2)\)

- If \(e_1\) has type \(t_1\) and \(e_2\) has type \(t_2\) (in current context), then \((e_1, e_2)\) has type \(t_1 * t_2\).
  - (It might be better if it were \((t_1, t_2)\), but it isn't.)

- If \(e_1\) evaluates to \(v_1\) and \(e_2\) evaluates to \(v_2\) (in current environment), then \((e_1, e_2)\) evaluates to \((v_1, v_2)\).
  - (Pairs of values are values.)

- Syntax to get part of a pair: \#1 \(e\) or \#2 \(e\).

- Type rules for getting part of a pair: ______________

- Evaluation rules for getting part of a pair: ______________