Review

- Building up SML one construct at a time via precise definitions
  - Constructs have syntax, type-checking rules, evaluation rules
  - And reasons they're in the language
  - Evaluation converts an expression to a value
- So far:
  - Variable bindings
  - Several expression forms: addition, conditionals, ...
  - Several types: int bool unit
- Today:
  - Brief discussion on aspects of learning a PL
  - Functions, pairs, and lists [almost enough for all of HW1]

Five different things

1. Syntax: How do you write language constructs?
3. Idioms: What are typical patterns for using language features to express your computation?
4. Libraries: What facilities does the language (or a well-known project) provide “standard”? (E.g., file access, data structures)
5. Tools: What do language implementations provide to make your job easier? (E.g., REPL, debugger, code formatter, …)

These are 5 separate issues
- In practice, all are essential for good programmers
- Many people confuse them, but shouldn’t

Our Focus

This course focuses on semantics and idioms

- Syntax is usually uninteresting
  - A fact to learn, like “The American Civil War ended in 1865”
  - People obsess over subjective preferences [yawn]
- Libraries and tools crucial, but often learn new ones on the job
  - We’re learning language semantics and how to use that knowledge to do great things

Function definitions

Functions: the most important building block in the whole course
- Like Java methods, have arguments and result
- But no classes, this, return, etc.

Example function binding:

```ml
(* Note: correct only if y>=0 *)
f

fun pow (x : int, y : int) =
  if y=0
  then 1
  else x * pow(x,y-1)
```

Note: The body includes a (recursive) function call: `pow(x,y-1)`

Function bindings: 3 questions

- Syntax: `fun x0 (x1 : t1, ... , xn : tn) = e`
  - (Will generalize in later lecture)
- Evaluation: A function is a value! (No evaluation yet)
  - Adds x0 to environment so later expressions can call it
  - (Function-call semantics will also allow recursion)
- Type-checking:
  - Adds binding `x0 : (t1 * ... * tn) -> t` if:
  - Can type-check body `e` to have type `t` in the static environment containing:
    - “Enclosing” static environment (earlier bindings)
    - `x1 : t1, ... , xn : tn` (arguments with their types)
    - `x0 : (t1 * ... * tn) -> t` (for recursion)
More on type-checking

- New kind of type: \((t_1 \times \ldots \times t_n) \rightarrow t\)
  - Result type on right
  - The overall type-checking result is to give \(x_0\) this type in rest of program (unlike Java, not for earlier bindings)
  - Arguments can be used only in \(e\) (unsurprising)

- Because evaluation of a call to \(x_0\) will return result of evaluating \(e\), the return type of \(x_0\) is the type of \(e\)

- The type-checker “magically” figures out \(t\) if such a \(t\) exists
  - Later lecture: Requires some cleverness due to recursion
  - More magic after hw1: Later can omit argument types too

---

Function Calls

A new kind of expression: 3 questions

Syntax: \(e_0\ (e_1, \ldots, e_n)\)
  - (Will generalize later)
  - Parentheses optional if there is exactly one argument

Type-checking:
  - If:
    - \(e_0\) has some type \((t_1 \times \ldots \times t_n) \rightarrow t\)
    - \(e_1\) has type \(t_1\), \(\ldots\)
    - \(e_n\) has type \(t_n\)
  - Then:
    - \(e_0(e_1, \ldots, e_n)\) has type \(t\)

Example: \(\text{pow}(x, y-1)\) in previous example has type \(\text{int}\)

---

Function-calls continued

Evaluation:

1. (Under current dynamic environment,) evaluate \(e_0\) to a function \(\text{fun} \ x_0 \ (x_1 : t_1, \ldots, x_n : t_n) = e\)
   - Since call type-checked, result will be a function

2. (Under current dynamic environment,) evaluate arguments to values \(v_1, \ldots, v_n\)

3. Result is evaluation of \(e\) in an environment extended to map \(x_1 \rightarrow v_1, \ldots, x_n \rightarrow v_n\)
   - ("An environment" is actually the environment where the function was defined, and includes \(x_0\) for recursion)

---

Example, extended

```
fun pow (x : int, y : int) = 
  if y=0 
  then 1 
  else x * pow(x,y-1) 

fun cube (x : int) = 
  pow (x,3) 

val sixtyfour = cube 4 
val fortytwo = pow(2,4) + pow(4,2) + cube(2) + 2 
```

---

Some gotchas

Three common “gotchas”

- Bad error messages if you mess up function-argument syntax
- The use of \* in type syntax is not multiplication
  - Example: \(\text{int} \times \text{int} \rightarrow \text{int}\)
  - In expressions, \* is multiplication: \(x \times \text{pow}(x,y-1)\)
- Cannot refer to later function bindings
  - That’s what the rules say
  - Helper functions must come before their uses
  - Need special construct for mutual recursion (later)

---

Recursion

- If you’re not yet comfortable with recursion, you will be soon 🙏
  - Will use for most functions taking or returning lists
- “Makes sense” because calls to same function solve “simpler” problems
- Recursion more powerful than loops
  - We won’t use a single loop in ML
  - Loops often (not always) obscure simple, elegant solutions
**Tuples and lists**

So far: numbers, booleans, conditionals, variables, functions
- Now ways to build up data with multiple parts
- This is essential
- Java examples: classes with fields, arrays

Rest of lecture:
- Tuples: fixed "number of pieces" that may have different types
- Lists: any "number of pieces" that all have the same type

Later: Other more general ways to create compound data

---

**Pairs (2-tuples)**

We need a way to build pairs and a way to access the pieces

**Build:**
- Syntax: \((e_1, e_2)\)
- Evaluation: Evaluate \(e_1\) to \(v_1\) and \(e_2\) to \(v_2\); result is \((v_1, v_2)\)
  - A pair of values is a value
- Type-checking: if \(e_1\) has type \(t_1\) and \(e_2\) has type \(t_2\), then the pair expression has type \(t_1 \times t_2\)
  - A new kind of type, the pair type

---

**Examples**

Functions can take and return pairs

```
fun swap (pr : int*bool) = 
  (#2 pr, #1 pr)

fun sum_two_pairs (pr1 : int*int, pr2 : int*int) = 
  (#1 pr1) + (#2 pr1) + (#1 pr2) + (#2 pr2)

fun div_mod (x : int, y : int) = 
  (x div y, x mod y)
```

---

**Tuples**

Actually, you can have tuples with more than two parts
- A new feature: a generalization of pairs

```
(e_1, e_2, ..., e_n)
```

```
t_1 \times t_2 \times \cdots \times t_n
```

```
#1 e, #2 e, #3 e, ...
```

Homework 1 uses triples of type \(\text{int} \times \text{int} \times \text{int}\) a lot

---

**Nesting**

Pairs and tuples can be nested however you want
- Not a new feature: implied by the syntax and semantics

```
val x1 = (7, (true, 9)) (* int \times (\text{bool} \times \text{int}) *)
val x2 = #1 (#2 x1)) (* bool *)
val x3 = (#2 x1) (* bool \times \text{int} *)
val x4 = ((3, 5), ((4, 8), (0, 0))) (* (\text{int} \times \text{int}) \times (\text{int} \times \text{int}) \times (\text{int} \times \text{int}) *)
```
Lists

- Despite nested tuples, the type of a variable still “commits” to a particular “amount” of data
- In contrast, a list can have any number of elements
- But unlike tuples, all elements have the same type

Need ways to build lists and access the pieces...

Building Lists

- The empty list is a value: 
  
- In general, a list of values is a value; elements separated by commas: 
  
- If \( e_1 \) evaluates to \( v \) and \( e_2 \) evaluates to a list \( [v_1, \ldots, v_n] \), then \( e_1 :: e_2 \) evaluates to \( [v, \ldots, v_n] \)

Accessing Lists

Until we learn pattern-matching, we will use three standard-library functions

- \( \text{null } e \) evaluates to true if and only if \( e \) evaluates to \( [] \)
- If \( e \) evaluates to \( [v_1, v_2, \ldots, v_n] \) then \( \text{hd } e \) evaluates to \( v_1 \)
- (raise exception if \( e \) evaluates to \( [] \))

- If \( e \) evaluates to \( [v_1, v_2, \ldots, v_n] \) then \( \text{tl } e \) evaluates to \( [v_2, \ldots, v_n] \)
- (raise exception if \( e \) evaluates to \( [] \))
- Notice result is a list

Type-checking list operations

Lots of new types: For any type \( t \), the type \( t \text{ list} \) describes lists where all elements have type \( t \)

- Examples: \( \text{int list} \) \( \text{bool list} \) \( \text{int list list} \) \( \text{(int * int) list} \) \( \text{(int list * int) list} \)

- So \( [] \) can have type \( t \text{ list} \) list for any type
- SML uses type \( \text{a list} \) to indicate this (“quote a” or “alpha”)
- For \( e_1 :: e_2 \) to type-check, we need a \( t \) such that \( e_1 \) has type \( t \)
  and \( e_2 \) has type \( t \text{ list} \). Then the result type is \( t \text{ list} \)

- \( \text{null } : \text{ a list } \to \text{ bool} \)
- \( \text{hd } : \text{ a list } \to \text{ a} \)
- \( \text{tl } : \text{ a list } \to \text{ a list} \)

Example list functions

```ml
fun sum_list (lst : int list) = 
  if null lst 
  then 0 
  else hd(lst) + sum_list(tl(lst))

fun countdown (x : int) = 
  if x=0 
  then [] 
  else x :: countdown (x-1)

fun append (lst1 : int list, lst2 : int list) = 
  if null lst1 
  then lst2 
  else hd (lst1) :: append (tl(lst1), lst2)
```

Recursion again

Functions over lists are usually recursive

- Only way to “get to all the elements”
- What should the answer be for the empty list?
- What should the answer be for a non-empty list?
  - Typically in terms of the answer for the tail of the list!

Similarly, functions that produce lists of potentially any size will be recursive

- You create a list is out of smaller lists
Lists of pairs

Processing lists of pairs requires no new features. Examples:

```haskell
fun sum_pair_list (lst : (int*int) list) = 
  if null lst 
  then 0 
  else #1(hd lst) + #2(hd lst) + sum_pair_list(tl lst)

fun firsts (lst : (int*int) list) = 
  if null lst 
  then [] 
  else #1(hd lst) :: firsts(tl lst)

fun seconds (lst : (int*int) list) = 
  if null lst 
  then [] 
  else #2(hd lst) :: seconds(tl lst)

fun sum_pair_list2 (lst : (int*int) list) = 
  (sum_list (firsts lst)) + (sum_list (seconds lst))
```