**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*:

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

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)`
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(2,2+2) + 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: int * int -> int
  – In expressions, * is multiplication: x * pow(x,y-1)

• Cannot refer to later function bindings
  – That’s simply ML’s rule
  – 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
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

fun x0 (x1 : t1, ..., xn : tn) = e

• New kind of type: (t1 * ... * tn) -> t
  – Result type on right
  – The overall type-checking result is to give x0 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 x0 will return result of evaluating e, the return type of x0 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 $(t_1 \ * \ \ldots \ * \ t_n) \rightarrow t$
- $e_1$ has type $t_1$, ..., $e_n$ 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 int
**Function-calls continued**

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

**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
   \( x_1 \) to \( v_1 \), \( \ldots \), \( x_n \) to \( v_n \)
   
   (“An environment” is actually the environment where the function was defined, and includes \( x_0 \) for recursion)
**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

Now:
- *Tuples*: fixed “number of pieces” that may have different types

Then:
- *Lists*: any “number of pieces” that all have the same type

Later:
- Other more general ways to create compound data
Pairs (2-tuples)

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 \(t_a\) and \(e_2\) has type \(t_b\), then the pair expression has type \(t_a * t_b\)
  – A new kind of type
Pairs (2-tuples)

Need a way to build pairs and a way to access the pieces

Access:

• Syntax: \( \#1 \ e \) and \( \#2 \ e \)

• Evaluation: Evaluate \( e \) to a pair of values and return first or second piece
  – Example: If \( e \) is a variable \( x \), then look up \( x \) in environment

• Type-checking: If \( e \) has type \( ta \star tb \), then \( \#1 \ e \) has type \( ta \) and \( \#2 \ e \) has type \( tb \)
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)

fun sort_pair (pr : int*int) =
    if (#1 pr) < (#2 pr)
      then pr
      else (#2 pr, #1 pr)
Tuples

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

  * (e₁,e₂,…,eₙ)
  * t₁ * t₂ * … * tₙ
  * #₁ e, #₂ e, #₃ e, …

Homework 1 uses triples of type *int*|*int*|*int* a lot
Nesting

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

\[
\begin{align*}
\text{val } & \text{x1} = (7, (\text{true}, 9)) \quad (* \text{ int } * (\text{bool}*\text{int}) \ *) \\
\text{val } & \text{x2} = \#1 (\#2 \ \text{x1}) \quad (* \text{ bool } \ *) \\
\text{val } & \text{x3} = (\#2 \ \text{x1}) \quad (* \text{ bool}*\text{int} \ *) \\
\text{val } & \text{x4} = ((3, 5), ((4, 8), (0, 0))) \\
& \quad (* (\text{int}*\text{int})*((\text{int}*\text{int})*\text{int})*\text{int}) \ *)
\end{align*}
\]
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 all list 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:

  \[
  [v_1, v_2, \ldots, v_n]
  \]

• 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 \([v, \ldots, v_n]\)

  \(e_1::e_2\) (* pronounced "cons" *)
Accessing Lists

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

- `null e` evaluates to `true` if and only if `e` evaluates to `[]`
- If `e` evaluates to `[v1,v2,...,vn]` then `hd e` evaluates to `v1`
  - (raise exception if `e` evaluates to `[]`)
- If `e` evaluates to `[v1,v2,...,vn]` then `tl e` evaluates to `[v2,...,vn]`
  - (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: int list  bool list  int list list  (int * int) list  (int list * int) list

- So [] can have type \( t \text{ list list} \) for any type
  - SML uses type 'a list to indicate this (“quote a” or “alpha”)

- For e1: e2 to type-check, we need a t such that e1 has type t and e2 has type t list. Then the result type is t list

- null : 'a list -> bool
- hd : 'a list -> 'a
- tl : 'a list -> 'a list
Example list functions

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

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

fun append (xs : int list, ys : int list) =
  if null xs
  then ys
  else hd (xs) :: append (tl(xs), ys)
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 out of smaller lists
Lists of pairs

Processing lists of pairs requires no new features. Examples:

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

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

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

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