What is an ML program?

A sequence of bindings from names to expressions.

Build powerful progs by composing simple constructs.

Build rich exprs from simple exprs

Build rich types from simple types

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:

(* 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: \texttt{int \* int \rightarrow int}
  – In expressions, \* is multiplication: \texttt{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 \textit{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

How to talk about functions precisely?

3 Step ML Language Construct Recipe
3 Step ML Language Construct Recipe

1. Syntax
   - How do we write programs with this construct?

2. Typechecking Rules (Static Semantics)
   - When is use of this construct well typed?

3. Evaluation (Dynamic Semantics)
   - What happens when I run this construct?

More on type-checking

- New kind of type: \((t_1 \times \ldots \times t_n) \to t\)
  - 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 bindings: 3 step recipe

1. Syntax: \(\text{fun } x_0 (x_1 : t_1, \ldots, x_n : t_n) = e\)
   - (Will generalize in later lecture)

2. Evaluation: A function is a value! (No evaluation yet)
   - Adds \(x_0\) to environment so \(\text{later expressions can call it}\)
   - (Function-call semantics will also allow recursion)

   - Add binding \(x_0 : (t_1 \times \ldots \times t_n) \to t\) if:
     - “Enclosing” static environment (earlier bindings)
     - \(x_1 : t_1, \ldots, x_n : t_n\) (arguments with their types)
     - \(x_0 : (t_1 \times \ldots \times t_n) \to t\) (for recursion)

Function Calls

A new kind of expression: 3 questions

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

2. Type-checking:
   - \(e_0\) has some type \((t_1 \times \ldots \times t_n) \to t\)
   - \(e_1\) 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

3. Evaluation:

A. (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

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

C. 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:

1. Syntax: $(e_1, e_2)$

2. Type-checking: If $e_1$ has type $t_a$ and $e_2$ has type $t_b$, then the pair expression has type $t_a \times t_b$
   - A new kind of type

3. 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

Access:

1. Syntax: $\#_1 e$ and $\#_2 e$

2. Type-checking: If $e$ has type $t_a \times t_b$, then $\#_1 e$ has type $t_a$ and $\#_2 e$ has type $t_b$

3. 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
**Pairs (2-tuples)**

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

**Access:**

1. Syntax: \( \#1 \, e \) and \( \#2 \, e \)

2. Type-checking: If \( e \) has type \( ta \ast tb \) and \( \#2 \, e \) has type \( ta \)

3. 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

**Examples**

Functions can take and return pairs

\[
\begin{align*}
\text{fun } \text{swap} \, (pr : \text{int}*\text{bool}) &= \left(\#2 \, pr, \#1 \, pr\right) \\
\text{fun } \text{sum_two_pairs} \, (pr1 : \text{int}*\text{int}, pr2 : \text{int}*\text{int}) &= \left(\#1 \, pr1 + \#2 \, pr1 + \#1 \, pr2 + \#2 \, pr2\right) \\
\text{fun } \text{div_mod} \, (x : \text{int}, y : \text{int}) &= \left(\frac{x}{y}, \text{x mod y}\right) \\
\text{fun } \text{sort_pair} \, (pr : \text{int}*\text{int}) &= \begin{cases} 
pr & \text{if } \#1 \, pr < \#2 \, pr \\
\left(\#2 \, pr, \#1 \, pr\right) & \text{else}
\end{cases}
\end{align*}
\]

**Tuples**

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

- \((e_1, e_2, \ldots, e_n)\)
- \(ta \ast tb \ast \ldots \ast tn\)
- \(\#1 \, e, \#2 \, e, \#3 \, e, \ldots\)

Homework 1 uses triples of type \text{int}*\text{int}*\text{int} a lot

**Nesting**

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

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

Should this be true?

\[(1, (2, 3)) = ((1, 2), 3)\]

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 \([v, \ldots, v_n]\)

\[
e_1::e_2 (* \text{ pronounced "cons" } *)
\]

Accessing Lists

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

- \( \text{null } e \) evaluates to \text{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: \( \text{int list} \), \( \text{bool list} \), \( \text{int list list} \), \( (\text{int \times \text{int}) list} \), \( (\text{int list \times \text{int list)}) \)

- So \([\,] \) can have type \( t \text{ list list} \) for any type
- SML uses type \( 'a \text{ list} \) to indicate this (“tick 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 \( t \text{ list} \).

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

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:

```
fun sum_pair_list (xs : (int\times int) \text{list}) = 
  if null xs 
  then 0 
  else #1(hd xs) + #2(hd xs) + sum_pair_list(tl xs)

fun firsts (xs : (int\times int) \text{list}) = 
  if null xs 
  then [] 
  else #1(hd xs) :: firsts(tl xs)

fun seconds (xs : (int\times int) \text{list}) = 
  if null xs 
  then [] 
  else #2(hd xs) :: seconds(tl xs)

fun sum_pair_list2 (xs : (int\times int) \text{list}) = 
  (sum_list (firsts xs)) + (sum_list (seconds xs))
```