CSE 341:
Programming Languages

Dan Grossman
Spring 2004
Lecture 2—ML functions, pairs, and lists
What is a programming language?

Here are separable concepts for defining and evaluating a language:

- **syntax**: how do you write the various parts of the language?
- **semantics**: what do programs mean? (One way to answer: what are the evaluation rules?)
- **idioms**: how do you typically use the language to express computations?
- **libraries**: does the language provide “standard” facilities such as file-access, hashtables, etc.? How?
- **tools**: what is available for manipulating programs in the language?
Our focus

This course: focus on semantics and idioms to make you a better programmer

Reality: Good programmers know semantics, idioms, libraries, and tools

Libraries are crucial, but you can learn them on your own.
Goals for today

- Add some more absolutely essential ML constructs
- Discuss lots of “first-week” gotchas
- Enough to do your homework
  - Though section and Friday will help
  - And we will learn better constructs soon

Note: These slides make much more sense in conjunction with lec2.sml.
Functions

- Recall a program is a sequence of bindings
- A second kind of binding is for functions, e.g.:
  \[
  \text{fun } x_0 \ (x_1 : t_1, \ldots, x_n : t_n) = e
  \]
- Function name and arguments are available in function body \(e\)
- Function type includes types of arguments and results
- Function \textit{application} can be written \(x_0 \ (e_1, \ldots, e_n)\)
- Type of (legal) application is type of function-result
- Application evaluation: \(x_0 \ (e_1, \ldots, e_n)\) evaluates to \(v\) if \(e_1, \ldots, e_n\) evaluate to \(v_1, \ldots, v_n\) and \(e\) evaluates to \(v\) under an environment extended to bind \(x_1\) to \(v_1\) ... \(x_n\) to \(v_n\)
- (We’ll come back to which environment we extend.)
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 you do for the way we will use pairs in homework 1
  - And it can improve error messages and your understanding
  - But type inference is a very cool thing in ML
  - Types unneeded for other variables or function return-types
- 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.
- This “makes sense” if the calls to itself (recursive calls) solve “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 environment), then \((e_1, e_2)\) has type \(t_1 \times t_2\).
  - (I wish 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 are a new type of value.)

- 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: ____________
Lists

We can have pairs of pairs of pairs... but we still “commit” to the amount of data when we write down a type.

Lists can have *any* number of elements:

- [ ] is the empty list
- More generally, \([v_1, v_2, \ldots, v_n]\) is a length \(n\) list
- If \(e_1\) evaluates to \(v\) and \(e_2\) evaluates to a list \([v_1, v_2, \ldots, v_n]\), then \(e_1 : : e_2\) evaluates to \([v, v_1, v_2, \ldots, v_n]\).
- \texttt{null} \(e\) evaluates to true if and only if \(e\) evaluates to [ ]
- If \(e\) evaluates to \([v_1, v_2, \ldots, v_n]\), then \(\texttt{hd} \ e\) evaluates to \(v_1\) and \(\texttt{tl} \ e\) evaluates to \([v_2, \ldots, v_n]\).
  - If \(e\) evaluates to [ ], a *run-time exception* is raised (this is different than a type error; more on this later)
List types

A given list's elements must all have the same type.

If the elements have type \( t \), then the list has type \( t \ list \). Examples: \( \text{int list, int*int list, int list list} \).

What are the type rules for \( :: \), \texttt{null}, \texttt{hd}, and \texttt{tl}?  

- Possible exceptions do not affect the type.

Hmmm, that does not explain the type of \( [] \) ?  

- It can have any type, which is indicated via \( 'a \ list \).
- That is, we can build a list of any type from \( [] \).

*Polymorphic* types are 3 weeks ahead of us.

- Teaser: \texttt{null}, \texttt{hd}, and \texttt{tl} are not keywords!
Recursion again

Functions over lists that depend on all list elements will be recursive:

- What should the answer be for the empty list?
- What should they do for a non-empty list? (In terms of answer for the tail of the list.)

Functions that produce lists of (potentially) any size will be recursive:

- When do we create a small (e.g., empty) list?
- How should we build a bigger list out of a smaller one?