# CSE 341: Programming Languages

Dan Grossman

Spring 2008

Lecture 2— 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 first homework problems 1–6, 8, 10
  - (rest after Friday)
  - And we will learn better constructs soon
  - `andalso`, `orelse` also quite useful, especially in problem 1

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

Recall a program is a sequence of bindings...

# Function Definitions

... A second kind of binding is for functions (kinda like Java methods without fields, classes, statements, ...)

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

Typing rules:

1. Context for `e` is (the function's context extended with) `x1:t1`, ..., `xn:tn` *and*:
2. `x0 : (t1 * ... * tn) -> 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: *A function is a value.*

# Function Applications (a.k.a. Calls)

Syntax: `e0 (e1,...,en)` (parens optional for one argument)

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

1. `e0` must have some type `(t1 * ... * tn) -> t`
2. `ei` must have type `ti` (for i=1, ..., i=n)
3. `e0 (e1,...,en)` has type `t`

Evaluation rules:

1. `e0` evaluates to a function $f$ in the application's environment
2. `ei` evaluates to value `vi` in the application's environment
3. result is $f$'s body evaluated in an environment extended to bind `xi` to `vi` (for i=1, ..., i=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 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
- 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.
- 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: `(e1,e2)`
- If `e1` has type `t1` and `e2` has type `t2` (in current context), then `(e1,e2)` has type `t1*t2`.
  - (I wish it were `(t1,t2)`, but it isn't.)
- If `e1` evaluates to `v1` and `e2` evaluates to `v2` (in current environment), then `(e1,e2)` evaluates to `(v1,v2)`.
  - (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: ______________

# Tuples

Actually, you can have *tuples* with any number of parts:

- `(e1,e2,...,en)`
- `t1 * t2 * ... * tn`
- `#n e` for any number `n`

Homework 1 uses `int * int * int`.

# 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 (a value)
- More generally, `[v1,v2,...,vn]` is a length $n$ list
- If `e1` evaluates to `v` and `e2` evaluates to a list `[v1,v2,...,vn]`, then `e1::e2` evaluates to `[v,v1,v2,...,vn]` (a value).
- `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` and `tl e` evaluates to `[v2,...,vn]`.
  - 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: `int list`, `(int*int) list`, `(int list) list`.

What are the type rules for `::`, `null`, `hd`, and `tl`?

- Possible exceptions do not affect the type.

Hmmm, that does not explain the type of `[]` ?

- It can have any list 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: `null`, `hd`, and `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 the answer be for a non-empty list? (*Typically in terms of the 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?

# Sharing, no mutation, etc.

Does `tl` copy the list or share its result with the tail of its argument?

What about our elegant `append`?

It doesn't matter!!!

- *All* that worrying you did in Java about aliasing, object identity, copying versus updating, equal vs. same-object is only relevant when you have assignment statements!
  - A great reason not to use them.

In ML, if `append ([1,2],[3,4,5])` produces `[1,2,3,4,5]`, you cannot tell how much sharing there is, so you don't have to think about it.

- Implementation tends to get the efficiency of sharing (`tl` is fast and doesn't make a "new" list), but without mutation there are no complications.