CSE 341: Programming Languages

Winter 2006
Lecture 3—Let bindings, options, and benefits of no mutation
Let bindings

Motivation: Functions without local variables can be poor style and/or really inefficient.

Syntax: \texttt{let \, b1 \, b2 \ldots \, bn \, in \, e \, end} where each \texttt{bi} is a \textit{binding}.

Typing rules: Type-check each \texttt{bi} and \texttt{e} in context including previous bindings. Type of whole expression is type of \texttt{e}.

Evaluation rules: Evaluate each \texttt{bi} and \texttt{e} in environment including previous bindings. Value of whole expression is result of evaluating \texttt{e}.

Elegant design worth repeating:

- Let-expressions can appear anywhere an expression can.
- Let-expressions can have any kind of binding.
  - Local functions can refer to any bindings \textit{in scope}. 


CSE341 Winter 2006, Lecture 3
More than style

Exercise: hand-evaluate bad_max and good_max for lists [1, 2], [1, 2, 3], and [3, 2, 1].
Summary and general pattern

Major progress: recursive functions, pairs, lists, let-expressions

Each has a syntax, typing rules, evaluation rules.

Functions, pairs, and lists are very different, but we can describe them in the same way:

- How do you create values? (function definition, pair expressions, empty-list and ::)
- How do you use values? (function application, #1 and #2, null, hd, and tl)

This (and conditionals) is enough for your homework though:

- andalso and orelse might help
- You might want options (next slide)
- Soon: much better ways to use pairs and lists (pattern-matching)
Options

“Options are like lists that can have at most one element.”

- Create a \( t \) option with \( \text{NONE} \) or \( \text{SOME} \ e \) where \( e \) has type \( t \).
- Use a \( t \) option with \( \text{isSome} \) and \( \text{valOf} \)

Why not just use (more general) lists? An interesting style trade-off:

- Options better express purpose, enforce invariants on callers, maybe faster.
- But cannot use functions for lists already written.
You want to change something?

There is no way to mutate (assign to) a binding, pair component, or list element.

How could the lack of a feature make programming easier? In this case:

- Amount of sharing is indistinguishable
  - Aliasing irrelevant to correctness!

- Bindings are invariant across function application
  - Mutation breaks compositional reasoning, a (the?) intellectual tool of engineering