CSE 341: Programming Languages

Winter 2005
Lecture 17—varargs and apply, implementing higher-order functions
Today:

- Some “easy” Scheme odds and ends
- Implementing higher-order functions and exceptions
Scheme varargs

In Scheme, functions can:

- Take exactly \( n \) arguments, for any \( n \geq 0 \)
  - Examples: \texttt{cons} \((n = 2)\), \texttt{null?} \((n = 1)\)

- Take \textit{n or more} arguments, for any \( n \geq 0 \)
  - Examples: \texttt{+} \((n = 0)\), \texttt{string=?} \((n = 2)\)

For user-defined functions taking 0 or more arguments:

\[
\text{(define f (lambda x e)) ; no parens on x, x is a list}
\]
\[
(f 3 4 "hi" (list 2 4))
\]

For user-defined functions taking \( n > 0 \) or more arguments:

\[
\text{(define g (lambda (x y . z) e)) ; note ., z is a list}
\]
\[
(g 3 4) (g 3 4 5) (g 3 4 5 6)
\]

Really just sugar: implicitly put arguments in a list.
Implementing Languages

Mostly 341 is about language meaning, not “how can an implementation do that”, but it’s important to “dispel the magic”.

At super high-level, there are two ways to implement a language $A$:

- Write an *interpreter* in language $B$ that evaluates a program in $A$
- Write a *compiler* in language $B$ that translates a program in $A$ to a program in language $C$ (and have an implementation of $C$)

In theory, this is just an implementation decision.

Issue: Higher-order functions and exceptions seem less straightforward.
Implementing Higher-Order Functions

The magic: How is the “right environment” around for lexical scope (the environment from when the function was defined)?

Lack of magic: Implementation keeps it around!

Interpreter:

- An interpreter has a “current environment”
- To evaluate a function (expression), create a closure (value), a pair of the function and the environment.
- Application will now apply a closure to an argument: Interpret function body, but instead of using “current environment”, use closure’s environment extended with the argument.

Note: This is a direct “coding” of the semantics we defined several weeks ago.
Compiling Higher-Order Functions

The key to the interpreter approach: The interpreter has an explicit environment and can “change” it to implement lexical scope.

We can also compile to a language without free variables: Instead of an implicit environment, we pass an explicit environment to every function.

- As with interpreter, we build a closure to evaluate functions.
- But all functions now take one extra argument.
- Application passes a closure’s code its own environment for the extra argument.
- Evaluating variables uses this extra argument.

Plus: Lots of data-structure optimizations so variable-lookup is fast (often a read from a known-size record).
Implementing Exceptions

Implementing exceptions (e.g., `(make-handle e1 e2)`) is:

- easier: dynamically scoped
- harder: have to “immediately transfer control elsewhere”

In addition to the current environment, we have a “current handler”, i.e., where to transfer control to when raising an exception.

Calling a function does not change the handler (dynamic scope).

Installing a nested handler changes the handler for evaluating a subexpression (e.g., e1).

In our example, what to do if e1 raises an exception it doesn’t handle?

- Evaluate e2, under environment and handler we had when we started evaluating e1.
- Return this result for the evaluation of `(make-handle e1 e2)`.
Implementing exceptions, continued

The hard part: “Stop what you’re doing” and evaluate e2. Interpreter approaches:

- “Bubble-up”: For every subexpression, interpreter returns a one-of type “normal value” or “exception”. (Slow, cumbersome, straightforward.)

- “Control transfer”: Use the interpreter-language (e.g., Scheme) to do what you need (e.g., \texttt{let/cc}). (Elegant, unobtrusive, requires powerful interpreter-language.)

Compiler approaches the same in theory, but if target language is assembly, bubbling up can be less cumbersome: Special code can treat the call-stack as a data object and explicitly pop until reaching handler.