Implementing Languages

- At a very high level there are 2 ways to implement a language $A$
  - Write an *interpreter* in language $B$ that reads, analyzes, and immediately evaluates programs written in language $A$
  - Write a *compiler* in language $B$ that translates a program written in language $A$ into some other language $C$ (and have an implementation of $C$ available)
Homework 4: Implement MUPL

- MUPL – “Made-Up Programming Language”
  - Basically a small subset of Scheme
  - Most interesting feature: higher-order functions
- HW4 is to write an interpreter for this language
Encoding A Language

- Suppose we want to process “-(2+2)”
- Compilers and interpreters both read (parse) linear program text and produce an *abstract syntax tree* representation
  - For this example using a made-up AST syntax: (make-negate (make-add (make-const 2) (make-const 2)))
- A *parser* turns the linear input into an AST
  - For hw4 we’ll write ASTs directly – no parser needed
An Interpreter

- An interpreter: a “direct” implementation created by writing out the evaluation rules for the language in another language

- For HW4:
  - MUPL programs encoded in Scheme data structures (use define-struct definitions in starter code)
  - Interpreter written in Scheme
Variables & Environments

- Languages with variables or parameters need interpreters with environments
- “Environment”: a name -> value map
  - For MUPL, names are “strings”
  - For MUPL, environment is an association list – a list of (name value) pairs
    - Lookup function is in the starter code
Evaluation

- The core of the interpreter is (eval-prog p)
  - Recursively evaluate program p in an initially empty environment (function applications will create bindings for sub-expressions)
  - Example: To evaluate addition, evaluate subexpressions in the same environment, then add the resulting values
Implementing Higher-Order Functions

- The magic: How is the right environment available to make lexical scope working?
- Lack of magic: implementation keeps it around
Higher-Order Functions

Details

- The interpreter has a “current environment”
- To evaluate a function expression (lambda, called “fun” in MUPL)
  - Create a closure, which is a pair of the function and the “current environment”
- To apply a function (really to apply a closure)
  - Evaluate the function body but use the environment from the closure instead of the “current environment”
Functions with Multiple Arguments

- A MUPL simplification: functions can only have a single (optional) parameter
- Sounds like a restriction, but it isn’t really
- Idea: rewrite multiple-argument functions as higher-order functions that take an argument and return a function to process the rest
  - This is currying – we’ve seen it already
Currying Example

- Suppose we have: \( \text{lambda } (x \ y) \ (+ \ x \ y) \)
  - Application: \(((\text{lambda } (x \ y) \ (+ \ x \ y)) \ 3 \ 4)\)

- Rewrite as:
  - \( \text{lambda } (x) \ (\text{lambda } (y) \ (+ \ x \ y)) \)
  - Application: \(((\text{lambda } (x) \ (\text{lambda } (y) \ (+ \ x \ y))) \ 3 \ 4)\)

- So multiple arguments only buy convenience, but no additional power