**CSE341: Programming Languages**

**Lecture 17**
Implementing Languages Including Closures

Dan Grossman  
Autumn 2017

---

**Typical workflow**

```
(fn x => x + x) 4
```

**Parsing**

```
abstract syntax (tree)
```

**Possible errors / warnings**

Type checking?

```
Rest of implementation
```

---

**Interpretor or compiler**

So "rest of implementation" takes the abstract syntax tree (AST) and "runs the program" to produce a result.

Fundamentally, two approaches to implement a PL $B$:

- Write an interpreter in another language $A$:
  - Better names: evaluator, executor
  - Take a program in $B$ and produce an answer (in $B$)
- Write a compiler in another language $A$ to a third language $C$:
  - Better name: translator
  - Translation must preserve meaning (equivalence)

We call $A$ the metalanguage:
- Crucial to keep $A$ and $B$ straight

---

**Reality more complicated**

Evaluation (interpreter) and translation (compiler) are your options:
- But in modern practice have both and multiple layers

A plausible example:
- Java compiler to bytecode intermediate language
- Have an interpreter for bytecode (itself in binary), but compile frequent functions to binary at run-time
- The chip is itself an interpreter for binary
  - Well, except these days the x86 has a translator in hardware to more primitive micro-operations it then executes

DrRacket uses a similar mix

---

**Sermon**

Interpretor versus compiler versus combinations is about a particular language implementation, not the language definition

So there is no such thing as a "compiled language" or an "interpreted language":
- Programs cannot "see" how the implementation works

Unfortunately, you often hear such phrases:
- "C is faster because it's compiled and LISP is interpreted"
- This is nonsense; politely correct people
- (Admittedly, languages with "eval" must "ship with some implementation of the language" in each program)
Skipping parsing

- If implementing PL B in PL A, we can skip parsing
  - Have B programmers write ASTs directly in PL A
  - Not so bad with ML constructors or Racket structs
  - Embeds B programs as trees in A

```
(define (eval-exp e)
  (cond [(const? e) e]
        [(negate? e) (...)]
        [(add? e) (...)]
        [(multiply? e) (...)]...)
```

Already did an example!

- Let the metalanguage A = Racket
- Let the language-implemented B = "Arithmetic Language"
- Arithmetic programs written with calls to Racket constructors
- The interpreter is eval-exp

```
(define (eval-exp e)
  (cond [(const? e) e]
        [(negate? e) (...)]
        [(add? e) (...)]
        [(multiply? e) (...)]...)
```

What we know

- Define (abstract) syntax of language B with Racket structs
  - B called MUPL in homework
- Write B programs directly in Racket via constructors
- Implement interpreter for B as a (recursive) Racket function

Now, a subtle-but-important distinction:
- Interpreter can assume input is a "legal AST for B"
  - Okay to give wrong answer or inscrutable error otherwise
- Interpreter must check that recursive results are the right kind of value
  - Give a good error message otherwise

Legal ASTs

- "Trees the interpreter must handle" are a subset of all the trees Racket allows as a dynamically typed language
- Can assume "right types" for struct fields
  - const holds a number
  - negate holds a legal AST
  - add and multiply hold 2 legal ASTs
- Illegal ASTs can "crash the interpreter" – this is fine

```
(multiply (add (const 3) "uh-oh") (const 4))
(negate -7)
```

Interpreter results

- Our interpreters return expressions, but not any expressions
  - Result should always be a value, a kind of expression that evaluates to itself
  - If not, the interpreter has a bug
- So far, only values are from const, e.g., (const 17)
- But a larger language has more values than just numbers
  - Booleans, strings, etc.
  - Pairs of values (definition of value recursive)
  - Closures
  - ...

Example

See code for language that adds booleans, number-comparison, and conditionals:

```
(struct const (int) #:transparent)
(struct negate (e) #:transparent)
(struct add (el e2) #:transparent)
(struct multiply (el e2) #:transparent)
```

```
(struct bool (b) #:transparent)
(struct eq-num (e1 e2) #:transparent)
(struct if-then-else (e1 e2 e3) #:transparent)
```

What if the program is a legal AST, but evaluation of it tries to use the wrong kind of value?
- For example, "add a boolean"
  - You should detect this and give an error message not in terms of the interpreter implementation
  - Means checking a recursive result whenever a particular kind of value is needed
- No need to check if any kind of value is okay
Dealing with variables

- Interpreters so far have been for languages without variables
  - No let-expressions, functions-with-arguments, etc.
  - Language in homework has all these things
- This segment describes in English what to do
  - Up to you to translate this to code
- Fortunately, what you have to implement is what we have been stressing since the very, very beginning of the course

The Set-up

So now a recursive helper function has all the interesting stuff:

```scheme
(define (eval-under-env e env)
  (cond … ; case for each kind of expression
  )) ;

- Recursive calls must "pass down" correct environment
```

Then `eval-exp` just calls `eval-under-env` with same expression and the empty environment

On homework, environments themselves are just Racket lists containing Racket pairs of a string (the MUPL variable name, e.g., "x") and a MUPL value (e.g., `(int 17)`)

The best part

- The most interesting and mind-bending part of the homework is that the language being implemented has first-class closures
  - With lexical scope of course
- Fortunately, what you have to implement is what we have been stressing since we first learned about closures...

Higher-order functions

The "magic": How do we use the "right environment" for lexical scope when functions may return other functions, store them in data structures, etc.?

Lack of magic: The interpreter uses a closure data structure (with two parts) to keep the environment it will need to use later

```
(struct closure (env fun) #:transparent)
```

Evaluate a function expression:

- A function is not a value; a closure is a value
  - Evaluating a function returns a closure
  - Create a closure out of (a) the function and (b) the current environment when the function was evaluated

Evaluate a function call:

```scheme
...```
Function calls

\[(\text{call } e_1 \ e_2)\]

- Use current environment to evaluate \(e_1\) to a closure
  - Error if result is a value that is not a closure
- Use current environment to evaluate \(e_2\) to a value
- Evaluate closure’s function’s body in the closure’s environment, extended to:
  - Map the function’s argument-name to the argument-value
  - And for recursion, map the function’s name to the whole closure

This is the same semantics we learned a few weeks ago “coded up”

Given a closure, the code part is only ever evaluated using the environment part (extended), not the environment at the call-site

Is that expensive?

- **Time** to build a closure is tiny: a struct with two fields
- Space to store closures might be large if environment is large
  - But environments are immutable, so natural and correct to have lots of sharing, e.g., of list tails (cf. lecture 3)
  - Still, end up keeping around bindings that are not needed
- Alternative used in practice: When creating a closure, store a possibly-smaller environment holding only the variables that are free variables in the function body
  - Free variables: Variables that occur, not counting shadowed uses of the same variable name
  - A function body would never need anything else from the environment

Free variables examples

\[
\begin{align*}
\text{(lambda } () \ (+ \ x \ y \ z)) & \ ; \ \{x, \ y, \ z\} \\
\text{(lambda } (x) \ (+ \ x \ y \ z)) & \ ; \ \{y, \ z\} \\
\text{(lambda } (x) \ (if \ x \ y \ z)) & \ ; \ \{y, \ z\} \\
\text{(lambda } (x) \ (let \ ([y \ 0]) \ (+ \ x \ y \ z))) & \ ; \ \{z\} \\
\text{(lambda } (x \ y \ z) \ (+ \ x \ y \ z)) & \ ; \ \{\} \\
\text{(lambda } (x) \ (+ \ (let \ ([y \ z]) \ (+ \ y \ y)))) & \ ; \ \{y, \ z\}
\end{align*}
\]

Computing free variables

- So does the interpreter have to analyze the code body every time it creates a closure?
- No: Before evaluation begins, compute free variables of every function in program and store this information with the function
- Compared to naïve store-entire-environment approach, building a closure now takes more time but less space
  - And time proportional to number of free variables
  - And various optimizations are possible
- [Also use a much better data structure for looking up variables than a list]

Optional: compiling higher-order functions

- If we are compiling to a language without closures (like assembly), cannot rely on there being a “current environment”
- So compile functions by having the translation produce “regular” functions that all take an extra explicit argument called “environment”
- And compiler replaces all uses of free variables with code that looks up the variable using the environment argument
  - Can make these fast operations with some tricks
- Running program still creates closures and every function call passes the closure’s environment to the closure’s code

Recall…

Our approach to language implementation:

- Implementing language \(B\) in language \(A\)
- Skipping parsing by writing language \(B\) programs directly in terms of language \(A\) constructors
- An interpreter written in \(A\) recursively evaluates

What we know about macros:

- Extend the syntax of a language
- Use of a macro expands into language syntax before the program is run, i.e., before calling the main interpreter function
Put it together

With our set-up, we can use language A (i.e., Racket) functions that produce language B abstract syntax as language B "macros"

- Language B programs can use the "macros" as though they are part of language B
- No change to the interpreter or struct definitions
- Just a programming idiom enabled by our set-up
  - Helps teach what macros are
  - See code for example "macro" definitions and "macro" uses
    - "macro expansion" happens before calling `eval-exp`

Hygiene issues

- Earlier we had material on hygiene issues with macros
  - (Among other things), problems with shadowing variables when using local variables to avoid evaluating expressions more than once
- The "macro" approach described here does not deal well with this