Building Interpreters: Recap

CS 440: Programming Languages
Michael Lee <lee@iit.edu>
Agenda

- Overview
- Some implementation details
  - Reader & Parser
  - Identifier bindings
  - Evaluation strategy
  - Scope selection
  - Desugaring
Overview
“Traditional” Interpreter Workflow
Our Implementation

Source code (Plain text) → Reader → Parser → Syntax tree

Parser → Syntax tree

Desugarer → Syntax tree (core language) → Evaluator

Our Implementation
Compilation Workflow

Source code (Plain text) → Scanner → Tokens → Parser → Syntax tree

Analysis / Optimization → Syntax tree → Code Generator → Bytecode / Machine code

IR
Some implementation details
Reader (Racket built-in)

```
(let ([x 44] [y 10]) (* x y))
```

Abstract Syntax Tree (AST)
Parser

(let ((x 44) (y 10)) (* x y))

(let-exp (list (var-exp 'x) (var-exp 'y))
          (list (int-exp 44) (int-exp 10))
          (arith-exp "TIMES" (var-exp 'x) (var-exp 'y)))
(define (parse-sexp)
  (match sexp
    [(? integer?)
      (int-exp sexp)]
    [(? symbol?)
      (var-exp sexp)]
    [(list '+ lhs rhs)
      (arith-exp "PLUS" (parse lhs) (parse rhs))]
    [(list '* lhs rhs)
      (arith-exp "TIMES" (parse lhs) (parse rhs))]
    [(list 'let (list (list id val) ...) body)
      (let-exp (map parse id)
                (map parse val)
                (parse body))])))

Parser

- Maps internal syntax tree nodes to recursive calls for additional parsing
- Calls match shape of the syntax tree
- **Recursive descent parser**
- Decorates syntax tree with metadata useful for evaluation
Identifier bindings

- `let` and `lambda` forms bind identifiers within specific scopes.

- An expression’s *environment* comprises all bindings in effect when it is evaluated.

```scheme
(let ([x 44])
  (let ([y 10])
    (* x y)))

(let ([f (lambda (x)
          (* x 10))])
  (f 44))
```
Identifier bindings

- We use an association list to represent an environment
- E.g., '((x . 44) (y . 10))
- Immutable structure: bindings are prepended when recursing
Identifier bindings

\[
\text{eval}\quad \text{env} = '() \\
\quad \text{let}\ ([x\ 44]) \\
\quad \text{let}\ ([y\ 10]) \\
\quad (*\ x\ y) \\
\quad \text{eval}\quad \text{env} = '((x\ .\ 44)) \\
\quad \text{eval}\quad \text{env} = '((y\ .\ 10)\ (x\ .\ 44))
\]
Identifier bindings

\[
\begin{aligned}
\text{eval} & \quad \text{env}=\emptyset \\
(\text{let } ([x \ 44])) & \quad \text{eval} \quad \text{env}=((x \ . \ 44)) \\
(\text{let } ([y \ 10])) & \quad \text{eval} \quad \text{env}=((y \ . \ 10) \ (x \ . \ 44)) \\
(\text{let } ([x \ 54])) & \quad \text{eval} \quad \text{env}=((x \ . \ 54) \ (y \ . \ 10)) \\
(* \ x \ y) & \quad \text{(shadowed)}
\end{aligned}
\]
Identifier bindings

\[
\text{eval}
\quad ((\lambda (x) (* x 10)) 44)
\quad \text{env}='()$
\]

\[
\text{eval}
\quad \text{env}='((x . 44))
\]

\[
(* x 10)
\]
let/lambda equivalence

- Note that all let forms can be written as lambda applications!

\[
\begin{align*}
\text{(let ([x 44]) (* x 10))} & \Leftrightarrow \text{((lambda (x) (* x 10)) 44)} \\
\text{(let ([x 44] [y (+ 3 7)]) (* x y))} & \Leftrightarrow \text{((lambda (x y) (* x y)) 44 (+ 3 7))}
\end{align*}
\]
Evaluation strategies

- Question: **when** do we evaluate expressions in binding forms?

- E.g., `(let ([x (+ 1 2)]) ...)`
  ```
  ((lambda (x) ...) (+ 1 2))
  ```

- Two general strategies: **Eager** (aka **Strict**) and **Lazy**
Eager evaluation

- Evaluate *before* binding the identifier

- aka **call-by-value**: evaluated “value” is passed as arg to function

```
(match expr
 [(var-exp id)
  (cdr (assoc id env))]
[(let-exp (list (var-exp id)) (list exp) body)
  (eval body
   (cons (cons id (eval exp env)) env))]
[(app-exp f arg)
  (match-let ([[lambda-exp id body)]
    (eval-env f env))
   (eval body
    (cons (cons id (eval arg env)) env)))])
```
Lazy evaluation

- Evaluate the expression only when needed
- aka call-by-name: un-evaluated expression “name” is passed
- An efficient version may cache (memoize) evaluated results instead of re-evaluating

```
(match expr
 [(var-exp id)
  (eval (cdr (assoc id env)) env))]
 [(let-exp (list (var-exp id)) (list exp) body)
  (eval body
   (cons (cons id exp) env))]
 [(app-exp f arg)
  (match-let [[(lambda-exp id body)]
               (eval f env)]
    (eval body
     (cons (cons id arg) env)))]
```
Eager vs. Lazy

- Eager evaluation is much more common in modern languages
  - More predictable behavior; easier to analyze program requirements
  - Often more efficient than a non-memoizing lazy evaluator
- Lazy evaluation may avoid doing unnecessary work (e.g., unreferenced identifiers in a function)
  - Control flow can be implemented via regular functions
  - Infinite / partially defined data structures are easy to define
Scope selection

- Question: **which bindings** (for free variables) are used when evaluating a function (lambda)?

  - E.g., `(let ([f (let ([x 44])
                   (lambda (y)
                   (* x y)))]))

    `(let ([x 33])
        (f 10))`

- Two strategies: **Dynamic** and **Lexical**
Dynamic binding

- Use the scopes in effect where the function is **called**
- I.e., free variables are looked up in the *dynamic environment*

```
(let ([f (let ([x 5])
            (lambda (y)
                (* x y)))]
     (+ (let ([x 4])
         (f 10))
     (let ([x 3])
         (f 10))))

> 70
```
Lexical binding

- Use the scopes in effect where the function is **defined**
- I.e., a function captures or “closes over” bindings in its **lexical environment**
- Lexically bound functions = **Closures**

\[
\begin{align*}
\text{let } ([f \text{ (let } ([x \ 5]) \\
\quad \text{ (lambda (y)} \\
\quad \quad \quad (* \ x \ y))))]) \\
\quad (+ \text{ (let } ([x \ 4]) \\
\quad \quad \quad (f \ 10)) \\
\quad \quad \quad \text{ (let } ([x \ 3]) \\
\quad \quad \quad \quad \quad (f \ 10)))))
\end{align*}
\]

\[> \ 100\]
Closure implementation

- A closure couples a function with its lexical environment
- An efficient version would only keep required bindings
- Critical for languages with first-class functions
- Functions may outlive their defining environment, but need to hang onto bindings!

(match expr
  [(lambda-exp id body)
   (closure id body env)]
  [(app-exp f arg)
   (match-let ([(closure id body clenv)
                 (eval f env)]
                [arg-val
                 (eval arg env)])
     (eval body
      (cons (cons id arg-val) clenv)))]
Desugaring

- Question: how to add syntactic elements (and associated semantics)?
- Option 1: update parser & evaluator — all syntax is first class
- Option 2: translate new syntactic elements into core language
  - Performed during “desugaring” passes (syntactic sugar → core syntax)
  - Keeps core language small and easy to reason about / test!
Desugaring

- E.g., (lambda (x y z ...) body)

(define (desugar exp)
  (match exp
    [(arith-exp op lhs rhs)    
      (arith-exp op (desugar lhs) (desugar rhs))]
    [(let-exp ids vals body)  
      (let-exp ids (map desugar vals) (desugar body))]
    [(lambda-exp ids body)     
      (foldr (lambda (id body)    
                 (lambda-exp id body))    
             (desugar body) ids)]
    [else exp]))
What did we leave out?

- Parsing!
- Language independent intermediate representations (e.g., LLVM)
- Optimizations (e.g., lean/fast environments, efficient execution)
- Memory management
- Code generation (transpiling, bytecode/machine code generation)
- Take **CS 443: Compiler Construction** (Prof Muller)!