CSE 341
Section 7
Fall 2018

Adapted from slides by Nicholas Shahan, Dan Grossman, and Tam Dang
Outline

• Interpreting LBI (Language Being Implemented)
  • Assume Correct Syntax
  • Check for Correct Semantics
  • Evaluating the AST

• LBI “Macros”

• Eval, Quote, and Quasiquote

• Variable Number of Arguments

• Apply
Building an LBI Interpreter

• We are skipping the parsing phase ← Do Not Implement
  - Can be skipped because AST (“Abstract Syntax Tree”) nodes represented as Racket structs.

• LBI vs. Metalanguage:
  - MUPL is the LBI.
  - Racket is the “metalanguage”.

```
  Call
   /\     \\
  Function Constant
 /   \      |
|     |     4
|     |
Var X  Var X
```

```
  +
 /\  \\
Var X  Var X
```

```
x
```
A larger language example...

(struct const (int) #:transparent)
(struct negate (e1)#:transparent)
(struct add (e1 e2) #:transparent)
(struct bool (b)#:transparent)
(struct multiply (e1 e2)#:transparent)
(struct eq-num (e1 e2)#:transparent)
(struct if-then-else (e1 e2 e3)#:transparent)

LBI → (add (const 1) (const 1))
Metalanguage → Racket structs/operations on structs/the above code.
Correct Syntax Examples

Using these Racket structs...

\[(\text{struct const (int) #:transparent})\]
\[(\text{struct add (e1 e2) #:transparent})\]
\[(\text{struct if-then-else (e1 e2 e3) #:transparent})\]

...we can interpret these LBI programs:

\[(\text{const 34})\]
\[(\text{add (const 34) (const 30)})\]
\[(\text{if-then-else (bool #t) (const 10) (const 20)})\]
Incorrect Syntax Examples

While using these Racket structs...

(struct const (int) #:transparent)
(struct add (e1 e2) #:transparent)
(struct if-then-else (e1 e2 e3) #:transparent)

...we can assume we won’t see LBI programs like:

(const "dan then dog")
(add 5 4)
(if-then-else (bool '(1 2)) (const 5) (bool #f))

Illegal input ASTs may crash the interpreter - this is OK
Racket vs. LBI

Structs in Racket, when defined to take an argument, can take any Racket value:

(struct const (int) #:transparent)
(struct add (e1 e2) #:transparent)
(struct if-then-else (e1 e2 e3) #:transparent)

But in LBI, we restrict const to take only an integer value, add to take two LBI expressions, and so on…

(const “dan then dog”)
(add 5 4)
(if-then-else (bool `(1 2)) (const 5) (bool #f))

Illegal input ASTs may crash the interpreter - this is OK
Check for Correct Semantics

What if the program is a legal AST, but evaluation of it tries to use the \textit{wrong} kind of value?

\begin{verbatim}
(struct const (int) #:transparent)
(struct add (e1 e2) #:transparent)
(struct if-then-else (e1 e2 e3) #:transparent)
\end{verbatim}

This is invalid LBI syntax that we need to check for...

\begin{verbatim}
(add (const 1) (bool #t))
(if-then-else (const 5) (const 5) (bool #f))
\end{verbatim}

• You should detect this and give an error message that is not in terms of the interpreter implementation
Evaluating the AST

• `eval-exp` should return a LBI value
• LBI values all evaluate to themselves
• Otherwise, we haven’t interpreted far enough

(const 7) \textcolor{green}{; evaluates to (const 7)}
(add (const 3) (const 4)) \textcolor{green}{; evaluates to (const 7)}
Evaluating the AST

- What’s wrong with this implementation of eval? (other than it being called “eval-exp-wrong”... )
Evaluating the AST

- It doesn’t recursively check for semantic correctness!
  - Let’s see a better version of this...
Macros Review

• Extend language syntax (allow new constructs)
• Written in terms of existing syntax
• Expanded before language is actually interpreted or compiled
LBI “Macros”

- Interpreting LBI using Racket as the metalanguage
- LBI is made up of Racket structs
- In Racket, these are just data types
- Why not write a Racket function that returns LBI ASTs?
LBI “Macros”

If our LBI Macro is a Racket function

```
(define (++ exp) (add (const 1) exp))
```

Then the LBI code

```
(++) (++) (const 7))
```

Expands to

```
(add (const 1) (add (const 1) (const 7)))
```
**LBI “Macros”**

If our LBI Macro is a Racket function

```
(define (andalso e1 e2) (if-then-else e1 e2 (bool #f)))
```

Then the LBI code

```
(andalso (bool #t) (bool #t))
```

Expands to

```
(if-then-else (bool #t) (bool #t) (bool #f))
```
• Syntactically, Racket statements can be thought of as lists of tokens

• \((+ 3 4)\) is a “plus sign”, a “3”, and a “4”

• \texttt{quote-ing} a parenthesized expression produces a list of tokens
quote Examples

(+ 3 4) ; 7

; '(+ 3 4)
(quote (+ 3 4))
'(+) 3 4)

; '(+ 3 #t)
(quote (+ 3 #t))
'(+) 3 #t)
quasiquote

• Inserts evaluated tokens into a quote
• Convenient for generating dynamic token lists
• Use unquote to escape a quasiquote back to evaluated Racket code
• A quasiquote and quote are equivalent unless we use an unquote operation
Self Interpretation

• Many languages provide an `eval` function or something similar
• Performs interpretation or compilation at runtime
  • Needs full language implementation during runtime
• It's useful, but there's usually a better way
• Makes analysis, debugging difficult
eval

• Racket's eval operates on lists of tokens
• Like those generated from quote and quasiquote
• Treat the input data as a program and evaluate it
Variable Number of Arguments

• Some functions (like +) can take a variable number of arguments

There is syntax that lets you define your own:

(define fn-any
  (lambda xs ; any number of args
    (print xs)))
(define fn-1-or-more
  (lambda (a . xs) ; at least 1 arg
    (begin (print a) (print xs))))
(define fn-2-or-more
  (lambda (a b . xs) ; at least 2 args
    (print a) (print b) (print xs)))
apply

- Applies a list of values as the arguments to a function in order by position

```scheme
(define fn-any
  (lambda xs ; any number of args
    (print xs)))

(apply fn-any (list 1 2 3 4))

(apply + (list 1 2 3 4)) ; 10
(apply max (list 1 2 3 4)) ; 4
```