CSE 341: Section 7

Yuma & Taylor
(Slides credit: Eric Mullen)

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*
- Interpreter written in Racket
  - Racket is the “metalanguage”
- LBI code represented as an AST
  - AST nodes represented as Racket structs
  - Allows us to skip the parsing phase
- Can assume AST has valid syntax
- Can NOT assume AST has valid semantics

Correct Syntax Examples

Using these Racket structs…

```
(struct int (num) #:transparent)
(struct add (e1 e2) #:transparent)
(struct ifnz (e1 e2 e3) #:transparent)
```

We can define these LBI programs:

```
(int 34)
(add (int 34) (int 30))
(ifnz (add (int 5) (int 7)) (int 12) (int 1))
```

Incorrect Syntax Examples

Using these Racket structs…

```
(struct int (num) #:transparent)
(struct add (e1 e2) #:transparent)
(struct ifnz (e1 e2 e3) #:transparent)
```

We *can* define these LBI programs *(but they are incorrect!)*:

```
(int "dan then dog")
(int (ifnz (int 0) (int 5) (int 7)))
(add (int 8) #t)
(add 5 4)
```

Valid LBI programs

Using these Racket structs…

```
(struct int (num) #:transparent)
(struct add (e1 e2) #:transparent)
(struct ifnz (e1 e2 e3) #:transparent)
```

Racket structs can take any value; we restrict the domain of valid forms in LBI.

```
(int "dan then dog")
(int (ifnz (int 0) (int 5) (int 7)))
(add (int 8) #t)
(add 5 4)
```

Racket vs. LBI syntax

Using these Racket structs…

```
(struct int (num) #:transparent)
(struct add (e1 e2) #:transparent)
(struct ifnz (e1 e2 e3) #:transparent)
```

This *is* valid Racket syntax, but it *is not* valid LBI syntax.

```
(int "dan then dog")
(int (ifnz (int 0) (int 5) (int 7)))
(add (int 8) #t)
(add 5 4)
```

Evaluating LBI

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

```
(int 7) ; evaluates to (int 7)
(add (int 3) (int 4)) ; evaluates to (int 7)
```

Demo!

Checking for Correct Semantics

- What if the program is a legal AST, but evaluation of it tries to use the wrong kind of value?
- For example, “add an integer and a function”
- You should detect this and give an error message that is not in terms of the interpreter implementation
- We need to check that the type of a recursive result is what we expect
  - No need to check if any type is acceptable

Macros

- 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 Macros is a Racket function

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

- Then the LBI code

```
(++ (int 17))
```

- Expands to

```
(add (int 1) (int 17))
```

Quote

- Syntactically, Racket statements can be thought of as lists of tokens
- `(+ 3 4)` is a “plus sign”, a “3”, and a “4”
- **`quote`**-ing a parenthesized expression produces a list of tokens

```
(+ 3 4) ; 7
(quote (+ 3 4)) ; '(+ 3 4)
(quote (+ 3 #t)) ; '(+ 3 #t)
(+ 3 #t) ; Error
```

Quote

- Syntactically, Racket statements can be thought of as lists of tokens
- `(+ 3 4)` is a “plus sign”, a “3”, and a “4”
- **`quote`**-ing a parenthesized expression produces a list of tokens

```
(+ 3 4) ; 7
`(+ 3 4) ; '(+ 3 4)
`(+ 3 #t) ; '(+ 3 #t)
(+ 3 #t) ; Error
```

Quasiquote

```
(quasiquote (+ 3 (unquote(+ 2 2)))) ; '(+ 3 4)
(quasiquote
  (string-append
   "I love CSE"
   (number->string
   (unquote (+ 3 338)))))
; '(string-append "I love CSE" (number->string 341))
```

Quasiquote

```
`(+ 3 ,(+ 2 2)) ; '(+ 3 4)

`(string-append
   "I love CSE"
   (number->string
   ,(+ 3 338))))
; '(string-append "I love CSE" (number->string 341))
```

`eval` & `apply`

Demo!