Outline

• LBI (Language Being Implemented)
• LBI “Macros”
• Eval, Quote, and Quasiquote
• Variable Number of Arguments
• Apply
LBI (Language Being Implemented)

• Yesterday in lecture, we saw we can define a “Programming Language” inside racket by structs

• We will talk about how to do evaluation on these LBIs tomorrow

• Show struct definition examples
Macros Review

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

• 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 (int 1) exp))
```

Then the LBI code

```
(++) (int 7)
```

Expands to

```
(add (int 1) (int 7))
```
LBI “Macros”

• We are just generating expressions of LBI, so expressions in LBI are still composed of the original structs

• If we have an eval function, we don’t need extra code to evaluate these “macros”
• 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
(quote (+ 3 4)) ; '(+ 3 4)
(quote (+ 3 #t)) ; '(+ 3 #t)
(+ 3 #t) ; Error

• You may also see the single quote ` character used as syntactic sugar
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
quasiquote Examples

\[
\begin{align*}
&(\text{quasiquote } (+ \ 3 \ (\text{unquote}(+ \ 2 \ 2)))) \ ; \ '(+ \ 3 \ 4) \\
&(\text{quasiquote} \\
&\quad\text{(string-append} \\
&\quad\quad"\text{I love CSE}"
\quad\text{(number->string} \\
&\quad\quad\quad(\text{unquote} (+ \ 3 \ 338)))))) \\
&\ ; \ '(\text{string-append} \ "\text{I love CSE}" \ (\text{number->string} \ 341))
\end{align*}
\]

- You may also see the backtick ` character used as syntactic sugar for \texttt{quasiquote}
- The comma character \texttt{,} is used as syntactic sugar for \texttt{unquote}
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
**eval examples**

```
(define quoted (quote (+ 3 4)))
(eval quoted) ; 7

(define bad-quoted (quote (+ 3 #t)))
(eval bad-quoted) ; Error

(define qquoted (quasiquote (+ 3 (unquote(+ 2 2)))))
(eval qquoted) ; 7

(define big-qquoted
  (quasiquote
    (string-append
      "I love CSE"
      (number->string
        (unquote (+ 3 338)))))
  (eval big-qquoted) ; “I love CSE341”
```
RackUnit

• Unit testing is built into the standard library
  • [http://docs.racket-lang.org/rackunit/](http://docs.racket-lang.org/rackunit/)
• Built in test functions to make testing your code easier
  • Test for equality, `check-eq?`
  • Test for True, `check-true`
  • Test for raised exception, `check-exn`
  • and many more
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
    (begin (print a) (print a) (print xs))))
```
apply

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

```
(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
```