CSE 341
Section 7

Emily Leland (Not Nick)

Spring 2017

Adapted from slides by Nicholas Shahan, Sunjay Cauligi, and Dan Grossman

Today’s Agenda

- Building a MUPL Interpreter
  - Assume Correct Syntax
  - Check for Correct Semantics
  - Evaluating the AST
- MUPL “Macros”
- Debugging MUPL
- Eval, Quote, and Quasiquote
- (Maybe) Variable Number of Arguments
- (Maybe) Apply

Building a MUPL Interpreter

- Skipping the parsing phase ← Do Not Implement
- Interpreter written in Racket
    - Racket is the “Metalanguage”
- MUPL code represented as an AST
    - AST nodes represented as Racket structs
- Can assume AST has valid syntax
- Can ***NOT*** assume AST has valid semantics

Correct Syntax Examples

Given this syntax:

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

We can need to evaluate these MUPL programs:

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

Incorrect Syntax Examples

Given this syntax:

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

We can assume we won't see MUPL programs like:

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

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 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

Evaluating the AST

- **`eval-exp`** should return a MUPL value
- MUPL 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)
```

Macros Review

- Extend language syntax (allow new constructs)
- Written in terms of existing syntax
- Expanded before language is actually interpreted or compiled

MUPL “Macros”

- Interpreting MUPL using Racket as the metalanguage
- MUPL is represented as Racket structs
- In Racket, these are just data types
- Why not write a Racket function that returns MUPL ASTs?

MUPL “Macros”

If our MUPL Macro is a Racket function

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

Then the MUPL code

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

Expands to

```
(add (int 1) (int 7))
```

`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

`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

```
(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))
```

- You may also see the backtick ` character used as syntactic sugar for **`quasiquote`**
- The comma character **,** is used as syntactic sugar for **`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"
```

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
```