CSE341 – Section 7

ASTs, Interpreters, MUPL

Sunjay Cauligi

February 21st, 2013

Legal vs. Nonlegal ASTs

Consider the Following

```
(add 3 4)
(add (const 3) (const 4))
(add (const 3) (bool #t))
```

Legal vs. Nonlegal ASTs

Consider the Following

```
(add 3 4)
(add (const 3) (const 4))
(add (const 3) (bool #t))
```

- Syntax vs. semantics

Legal vs. Nonlegal ASTs

Consider the Following

```
(add 3 4)
(add (const 3) (const 4))
(add (const 3) (bool #t))
```

- Syntax vs. semantics
- No need to check for syntax

Legal vs. Nonlegal ASTs

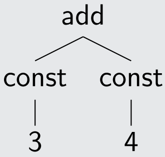
Consider the Following

```
(add 3 4)
(add (const 3) (const 4))
(add (const 3) (bool #t))
```

- Syntax vs. semantics
- No need to check for syntax
- Must check semantics

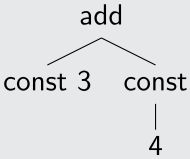
Checking Semantics

Nice Case



Checking Semantics

Nice Case



add

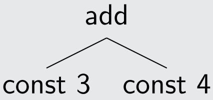
const 3

const

4

Checking Semantics

Nice Case



Checking Semantics

Nice Case

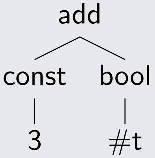
const 7

Checking Semantics

Nice Case

const 7

Not Nice Case



add

const bool

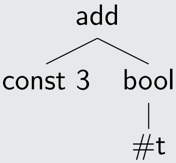
3 #t

Checking Semantics

Nice Case

const 7

Not Nice Case

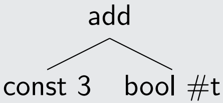


Checking Semantics

Nice Case

const 7

Not Nice Case



Checking Semantics

Nice Case

const 7

Not Nice Case

Error: add applied to non-number!

Valid Assumptions

Allowed to Assume

- Input AST is “valid”
- Each node in AST has right “types”
  - Remember that nodes such as `add` and `multiply` take *ASTs*, not numbers!
- Illegal input ASTs may crash the interpreter – this is OK

Valid Assumptions

Allowed to Assume

- Input AST is "valid"
- Each node in AST has right "types"
  - Remember that nodes such as `add` and `multiply` take *ASTs*, not numbers!
- Illegal input ASTs may crash the interpreter – this is OK

Need to Check

- Return types from subexpressions
- E.g. `(add (const 3) (bool #t))` is a legal AST, but has a wrong value being passed to `add`

Reviewing Macros

What is a Macro?

- Extends language syntax (allows new constructs)
- Written in terms of *existing syntax*

Reviewing Macros

What is a Macro?

- Extends language syntax (allows new constructs)
- Written in terms of *existing syntax*
- Expanded before language is actually interpreted/compiled

MUPL “Macros”

A Clever Trick

- Interpreting MUPL using Racket
- MUPL is represented as Racket structs

MUPL “Macros”

A Clever Trick

- Interpreting MUPL using Racket
- MUPL is represented as Racket structs
    - In Racket, these are just more data types

MUPL "Macros"

A Clever Trick

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

MUPL “Macros”

A Clever Trick

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

Note on Hygiene

Implementing “macros” in this manner doesn’t give very good macro hygiene

Racket's quote function

Quoting a Set of Tokens

- Syntactically, Racket statements can be thought of as lists of tokens
- `(+ 3 4)` is a plus sign, a '3', and a '4'

Racket's quote function

Quoting a Set of Tokens

- 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*

Racket's quote function

Quoting a Set of Tokens

- 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*

Examples

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

Self Interpretation

Notes on "eval"

- Many languages provide an `eval` function or something similar
- Performs interpretation/compilation *at runtime*

Self Interpretation

Notes on "eval"

- Many languages provide an `eval` function or something similar
- Performs interpretation/compilation *at runtime*
  - Needs full language implementation during runtime

Self Interpretation

Notes on "eval"

- Many languages provide an `eval` function or something similar
- Performs interpretation/compilation *at runtime*
  - Needs full language implementation during runtime

Use of eval

- It's useful, but there's usually a better way
- Makes analysis, debugging difficult

Eval in Racket

Racket's "eval" function

- Racket's `eval` operates on lists of tokens
  - Like those generated from `quote`

Eval in Racket

Racket's "eval" function

- Racket's `eval` operates on lists of tokens
  - Like those generated from `quote`

Examples

```
(define quoted (quote (+ 3 4)))
(eval quoted) => 7
(define bad-quoted (quote (+ 3 #t)))
(eval bad-quoted) => Error
```

Quasiquoting

Quasiquoting

- Inserts evaluated tokens into a “quote”
- Convenient for generating dynamic token lists

Quasiquoting

Quasiquoting

- Inserts evaluated tokens into a “quote”
- Convenient for generating dynamic token lists

Examples

```
(quasiquote (+ 3 (unquote (+ 2 2)))) => '(+ 3 4)
(quasiquote (+ 3 (unquote (quote (I love CSE 338))))) => '(+ 3 (I love CSE 338))
```

Quasiquoting

Quasiquoting

- Inserts evaluated tokens into a “quote”
- Convenient for generating dynamic token lists

Examples

```
(quasiquote (+ 3 (unquote (+ 2 2)))) => '(+ 3 4)
(quasiquote (+ 3 (unquote (quote (I love CSE 338))))) => '(+ 3 (I love CSE 338))

(quasiquote (+ (unquote (eval (quote (- 5 2))))
               (unquote (eval (quasiquote (+ (unquote (/ 4 2)) 2)))))) => '(+ 3 4)
```

Cute Little Typographical Shortcuts

```
'(a b c) <=> (quote (a b c))
`(a b ,(+ 2 2) d) <=>
  (quasiquote (a b (unquote (+ 2 2)) d))
(λ (x) (+ x 1)) <=> (lambda (x) (+ x 1))
```