Legal vs. Nonlegal ASTs

Consider the Following

\[
\begin{align*}
&\text{(add 3 4)} \\
&\text{(add (const 3) (const 4))} \\
&\text{(add (const 3) (bool #t))}
\end{align*}
\]
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

(\texttt{add 3 4})
(\texttt{add (const 3) (const 4)})
(\texttt{add (const 3) (bool \#t)})

- Syntax vs. semantics
- No need to check for syntax
Legal vs. Nonlegal ASTs

Consider the Following

- $(\text{add} \ 3 \ 4)$
- $(\text{add} \ (\text{const} \ 3) \ (\text{const} \ 4))$
- $(\text{add} \ (\text{const} \ 3) \ (\text{bool} \ #t))$

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

**Nice Case**

```
  add
  └── const
      └── 3
  └── const
      └── 4
```
Nice Case

```
add
  const 3  const
    |
  4
```

Sunjay Cauligi  CSE341 – Section 7
Checking Semantics

Nice Case

```
add
  const 3
  const 4
```
Nice Case

const 7
Checking Semantics

**Nice Case**

```
const 7
```

**Not Nice Case**

```
add
  const
  3
  bool
  #t
```
Checking Semantics

**Nice Case**

\[ \text{const 7} \]

**Not Nice Case**

\[
\text{add} \\
\text{const 3} \quad \text{bool} \\
\text{#t}
\]
Checking Semantics

Nice Case

const 7

Not Nice Case

add

const 3  bool  #t
## Checking Semantics

<table>
<thead>
<tr>
<th>Nice Case</th>
<th>Not Nice Case</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>const 7</code></td>
<td><strong>Error: add applied to non-number!</strong></td>
</tr>
</tbody>
</table>
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
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
A Clever Trick

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

\[(+ 3 4) \Rightarrow 7\]
\[(quote (+ 3 4)) \Rightarrow '(+ 3 4)\]
\[(quote (+ 3 #t)) \Rightarrow '(+ 3 #t)\]
\[(+ 3 #t) \Rightarrow Error\]
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
Racket’s “eval” function

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

Examples:

```racket
(define quoted (quote (+ 3 4)))
(eval quoted) => 7

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

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

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

**Examples**

```racket
(quasiquote (+ 3 (unquote (+ 2 2)))) => '(+ 3 4)
(quasiquote (+ 3 (unquote (quote (I love CSE 338))))) => '(+ 3 (I love CSE 338))
```
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))
(\(x\) (+ x 1)) <=> (lambda (x) (+ x 1))