CSE 341: *Programming Languages*

Section AC with Nate Yazdani

agenda

- guidance for homework 5 (MUPL)
  - syntax
  - semantics
  - evaluation
  - syntactic sugar

- more Racket
  - `eval`, `quote`, and `quasiquote`
  - RackUnit
  - variadic procedures
  - `apply`

Change how we do this

- Previous version of **`eval_exp`** has type **`exp -> int`**
- From now on will write such functions with type **`exp -> exp`**
- Why? Because will be interpreting languages with multiple kinds of results (ints, pairs, functions, …)
  - Even though much more complicated for example so far
- How? See the ML code file:
  - Base case returns entire expression, e.g., **`(Const 17)`**
  - Recursive cases:
    - Check variant (e.g., make sure a **`Const`**)
    - Extract data (e.g., the number under the **`Const`**)
    - Also return an **`exp`** (e.g., create a new **`Const`**)

New way in Racket

See the Racket code file for coding up the same new kind of "**`exp -> exp`**" *interpreter*

- Using lists where car of list encodes "what kind of exp"

Key points:

- Define our own constructor, test-variant, extract-data functions
  - Just better style than hard-to-read uses of **`car`**, **`cdr`**
- Same recursive structure without pattern-matching
- With no type system, no notion of "what is an exp" except in documentation
  - But if we use the helper functions correctly, then okay
  - Could add more explicit error-checking if desired

syntax of MUPL

- no parsing this time
  - already seen enough of that :-)

- MUPL programs are abstract syntax trees (ASTs)
  - composed of Racket `struct`s as nodes

- interpreter can assume that the given AST is valid, *i.e.*, conforms to the specification of MUPL syntax

- *however*, even a syntactically correct program could have invalid semantics!

valid syntax

for this abstract syntax:

n is a Racket integer

```
(struct int (n) #:transparent)
(struct add (e1 e2) #:transparent)
(struct mif (e1 e2 e3) #:transparent)
(struct mtrue () #:transparent)
(struct mfalse () #:transparent)
```

each `ei` is a subexpression

your interpreter should support *valid* ASTs, like these:

```
(int 341)
(add (int 99) (int 1))
(if (mtrue) (int 1) (add (int 10) (int 1)))
```

invalid syntax

for this abstract syntax:

```
(struct int (n) #:transparent)
(struct add (e1 e2) #:transparent)
(struct mif (e1 e2 e3) #:transparent)
(struct mtrue () #:transparent)
(struct mfal
```

can literally crash — that's totally fine

your interpreter can ignore *invalid* ASTs, like these:

```
(int “dan then dog”)
(mif #t (int 1) (int 0))
(int (add (int 1) (int 0)))
```

semantics of MUPL

- a MUPL program (AST) might be *syntactically* valid, but it still may not be *semantically* valid
  - for instance, **`(add (mtrue) (int 0))`**
- your interpreter should detect these cases and report an error in terms of the language, *not* the implementation
  - for instance, “error: arguments to **`add`** must be **`int`** values”
- your interpreter should ensure that every result from a recursive call is the sort of MUPL value expected
  - if any MUPL value works, then no need to check

evaluation of MUPL programs

- **`eval-exp`** should return a MUPL value
  - a MUPL value just evaluates to itself
  - a MUPL expression (that isn't a value) evaluates based on how its MUPL subexpressions evaluate

probably going to need some recursion!

```
(eval-exp (int 341)) ⇓ (int 341)
```

```
(eval-exp (add  ...                         (int 100)
```

"left thing computes to right thing"

```
(eval-exp (mif (mtrue)
               (add (int 1) (int 2))      ⇓ (int 3)
               (mfalse)))
```

macros review

- extend language syntax
- expressed in terms of existing syntax
- expanded before the program is evaluated (*i.e.*, interpreted or compiled)

“macros” for MUPL

- we’re interpreting MUPL (the object language) inside of Racket (the metalanguage)
- the syntax of MUPL programs is represented with Racket **`struct`**s
- to Racket, a MUPL program is just data
- Why not write Racket functions that return MUPL ASTs?

“macros” for MUPL

- let’s call this Racket function a MUPL macro:

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

- now, this MUPL code

```
(++ (int 101))
```

- evaluates (in Racket) to this MUPL AST:

```
(add (int 1) (int 101))
```

quotation

- syntactically, Racket code can be thought of as a (possibly nested) list of tokens
- for instance, **`(+ 1 2)`** is **`+`**, then **`1`**, and then **`2`**
- **`quote`**-ing a parenthesized expression or prefixing it with **‘** gives you that list:

```
(+ 1 2) ; evaluates to 7
(quote (+ 1 2)) ; evaluates to ‘(+ 1 2)
(quote (+ 1 #t)) ; evaluates to ‘(+ 1 #t)
(+ 1 #t) ; error!
```

quasiquotation

- **`quasiquote`** or **`` ` ``** (the backtick) lets you evaluate part of the syntax with **`unquote`** or **`,`**
- more precisely, **`unquote`** escapes **`quasiquote`** back to evaluated Racket
- without **`unquote`**, **`quasiquote`** is equivalent to plain **`quote`**

```
(quasiquote (unquote (+ 1 2 3))) ; 6
(quasiquote (cse (unquote (+ 3 338)))) ; ‘(cse 341)
```

self-interpretation

- many languages provide an `eval` function or something like it
- evaluates syntax at runtime, possibly with interpretation or possibly with compilation
- can be useful, but there’s often a better way
- self-interpretation makes reasoning about your code difficult, both for computers (*e.g.*, analyses) and for people (*e.g.*, debugging)

self-interpretation

- Racket's **`eval`** works on nested lists of tokens
- **`quote`** and **`quasiquote`** generate such lists
- **`eval`** treats the given list as the syntax of a Racket program and (tries to) evaluate it

```
(define quoted
  (quote (+ 1 2 (+ 3 4)))) ; '(+ 1 2 (+ 3 4))
(eval quoted) ; 10
```

RackUnit

- unit testing built into Racket standard library
  - http://docs.racket-lang.org/rackunit/
- provides functions to make testing your code easier: **`check-eq?`**, **`check-true`**, **`check-exn`**, and many more

variadic functions

- "variadic" functions (like **+**) accept a variable number of arguments
- you can define your own, if you'd like:

```
(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))))
```

function application

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
```