2013W1-lecture7

October 2, 2013

Contents

1 Question of the Day

Here's Racket's documentation on the `case` form:

```
(case <val-expr> <case-clause> ...)

  <case-clause> = [(<datum> ...) <then-body> ...+]
                | [else <then-body> ...+]
```

Evaluating `<val-expr>` yields the value we check against the cases. The entire expression's value is the result of evaluating either (1) the `<then-body>` of the first `<case-clause>` with a `<datum>` which would, if quoted, be equal to the value of the `<val-expr>` or (2) the `<then-body>` of the `else` clause if no `<datum>` matches. Example:

```
(case (+ 7 5)
   [(1 2 3) false]
   [(10 11 12) true])
```

evaluates to `true` because `12` appears in the second clause.

Here's the question. People often use case clauses with quoted symbols in place of the parenthesized `<datum>` syntax. Does it work? What do these evaluate to?

```
(define (try-it x)
  (case x
    ['+ false]
    [else true]))

(try-it '+)
(try-it 'quote)
```

```
  SOLUTION
 Let's just run it!

 Spoiler: they both evaluate to =false=.

 Why?!  Can you figure it out?
```

2 Logistics

2.1 Programming Assignment #2

Due one week from today.

This one should be easy for your team. We may release the next project on Monday.

BE SURE to read the marking comments posted on Piazza!

2.1.1 Who does not have a team?

2.2 Quiz/Conceptual Assignment #2: Wednesday

See the assignments area of the course website for a running example we'll use in the quiz. Think about it a bit beforehand!

2.3 Programming Assignment #3

Coming soon. This will involve desugaring the *entire* ParselTongue language to a core language implementation we provide. Your test cases from PA2 will be absolutely *invaluable* in this process, particularly if they're simple, well-targeted at specific semantic issues, and well-documented.

3 Functions

file:2013W1-lecture7-functions.rkt

- TODO 1: finish subst
- TODO 3: implement div/id/app interp cases
- TODO 2: function test cases. For our test cases, we'd like to explore what happens..
  - in "normal" cases where we call a natural-seeming function and it does its thing
  - when a function calls another function
  - when a function calls itself (CAN'T REALLY DO RECURSION W/OUT CONDITIONALS)
  - when a function does not use its parameter
  - when a function uses its parameter twice (how can we tell?)
    - Just look for multiple references to the identifier. It will always have one *binding site* where it's defined, but it may be referenced multiple times.
  - when a function's "actual parameter" is an identifier?
  - when we try to use an identifier that is not bound anywhere?
  - when we try to use an identifier that's bound in the function that calls the current function

        * If this is allowed, it means our language has *dynamic scope* for identifiers. In other words, an identifier binding made in a particular part of the program may be used in a totally unrelated part of the program just because this part of the program calls a function in that part (or calls a function that calls a function that ... that calls a function in that part). If that sounds good, consider: How would you like it if calling something like `Math.min` in Java could use (and maybe modify the values of) identifiers in your program?
        * Fortunately, we'll use *static scope* in which an identifier binding is only available in a region of the program defined by its static structure (its "shape on the page").
    - Also, how could we tell the difference between "lazy" and "eager" evaluation semantics? Can we test for the appropriate semantics? What do we need to be able to do such a test?
        * `(test/exn (... '(useless x)) "")`

- TODO 4: revisit lazy eval
- TODO 5: add a `debug` expression

3.1 Useful templates for interp, desugar, and the like

3.1.1 Working with a user-defined type with more than one variant

```
; Note: some fields of some variants may themselves be of type
; UserDefinedType.  For those fields, we expect to make a recursive
; call.  We use field2 of variant2 as an example below.
(define (foo [x : UserDefinedType]) : ...
  (type-case UserDefinedType x
    [variant1 (field1 field2 ...) (... field1 field2 ...)]
    [variant2 (field1 field2 field3 ...) (... field1 (foo field2) field3 ...)]
    ...))
```

3.1.2 Working with a user-defined type with ONE variant

```
(define-type UserDefinedType1Variant
  [variant (field1 : ..) (field2 : ..) ...])

; No need for type-case because only one variant!
```

```
; Each of the variant's fields is available to us.
(define (fun-for-user-defined-type-1-variant x)
  (... (variant-field1 x) (variant-field2 x) ...))
```

4 What have we learned today?

- From QotD: not much :). A little more about desugaring and "leaking" a desugaring to the user. We can see now that the `quote` form gets desugared *before* the `case` form even gets to check its own syntax. If the desugaring went after syntax checking, then the `case` form would complain that it did not receive a parenthesized list of `<datum>` forms.

- Functions

  - Explain in context four of the key features of a function:
    * delaying evaluation of the function body,
    * allowing evaluation of the function body from (possibly distant) locations in the program,
    * blocking out bindings from the function's dynamic context, and
    * "allowing in" a dynamic value via the parameter(s).
  - Formal parameters vs. actual parameters
    * Distinguish between formal and actual parameters. (Formal parameters are the names given to argument values within the function; actual parameters are the argument expressions whose values (under eager evaluation) are bound to the formal parameters.)
    * Identify formal and actual parameters in context.
  - Function calls vs. function definitions
    * Distinguish between a function call (also known as function application) and a function definition. (A function call is application of a previously defined function to actual parameters, which causes the function's body to be evaluated. A function definition creates a function—prepares it to be used—with formal parameters and a body but does not evaluate the body yet!)
    * Identify function calls and definitions in context.
  - Bound vs. unbound identifiers

    * Define bound and unbound identifiers, including illustrating their meanings with examples.
    * Identify bound and unbound identifiers in context.
    * Define *scope* with respect to identifiers. (The scope of an identifier's binding is the portion of the program in which that binding is visible. In Racket, bindings use lexcical (AKA static) scope. However, for the curious willing to suffer some complexity, Racket does have a facility for something like dynamic scope; see particularly the "with a fox" example in http://docs.racket-lang.org/guide/parameterize.html.)

- Eager evaluation vs. lazy evaluation
    * Explain the basic difference between lazy and eager evaluation. (In lazy evaluation, actual parameters are not evaluated before proceeding with a function call. So, formal parameters are bound to the actual parameter *expressions*, not their *values*. In eager evaluation, actual parameters are evaluated prior to proceeding with a function call, and the function's formal parameters are bound to the resulting values.)
    * Be able to distinguish between lazy and eager evaluation regimes by evaluating (and possibly providing a test case) that behaves differently under the two regimes.
    * Alter existing substitution and interpreter code to switch back and forth between lazy and eager evaluation regimes.