CS 360
Programming Languages
Day 15 –
Delayed Evaluation & Streams

Swift

The truth comes out!

- Everything that looks like a function call in Racket is not necessarily a function call.
- Everything that looks like a function call is either
  - A function call (as we thought).
  - Or a “special form.”
- Special forms: define, let, lambda, if, cond, and, or, …
- Why can’t these be functions?
- Recall the evaluation model for a function call:
  - **`(f e1 e2 e3…)`**: evaluate **`e1 e2`** … to obtain values **`v1 v2`**…, then evaluate **`f`** to get a closure, then evaluate the body of the closure with its arguments bound to **`v1 v2`**…
  - Why would this not work for defining **`if`**?

Evaluation strategies

- Every programming language uses an ***evaluation strategy*** to figure out two things:
  - ***when*** to evaluate the arguments of a function call (or other operation), and
  - ***what kind of value*** to pass to the function.

- You have explored the "what kind of value" issue in CS142:
  - ***pass by value*** versus ***pass by reference***.
  - There are others: e.g., pass by name.

- When to evaluate arguments?
  - Most PLs use ***eager evaluation*** (args are evaluated completely before being passed to the function).
  - Today we will explore ***delayed*** or ***lazy evaluation***.

Delayed evaluation

- In Racket, function arguments are ***eager***.
  Special form arguments are ***lazy***.
  - Delay evaluation of the argument until we really need its value.

- Why wouldn't these functions work?

```
(define (my-if-bad x y z)
  (if x y z))

(define (fact-wrong n)
    (my-if-bad (= n 0)
               1
               (* n (fact-wrong (- n 1)))))
```

Thunks

- We know how to delay evaluation: put expression in a function definition!
  - Because defining a function doesn't run the code until later.

- A zero-argument function used to delay evaluation is called a *thunk.*
  - As a verb: *thunk the expression.*

- This works (though silly to re-define **`if`** like this):

```
(define (my-if x y z)
  (if x (y) (z)))

(define (fact n)
    (my-if (= n 0)
           (lambda () 1)
           (lambda () (* n (fact (- n 1))))))
```

Try this one

- Write a function called **`while`** that takes two arguments:
  - a thunk called **`condition`**
  - a thunk called **`body`**

- This function should emulate a while loop: test the **`condition`**, and if it's true, run the **`body`**. Then test the **`condition`** again, and if it's still true, run the **`body`** again. Continue until the **`condition`** is false.
  - You will likely need to use **`(begin)`**.
  - The while function itself may return whatever you want.

- Using your while function, write a while loop that prints the numbers 1 to 10.

- Define a function called **`my-length`** that takes one list argument. **`my-length`** should return the length of the list argument. Use your while loop.

Thunks

- Think of a thunk as a "promise" to "evaluate this expression as soon as we really need the value."
- ```
  (define result
    (compute-answer-to-life-univ-and-everything))
  ```
  – Would take a really long time to calculate result.
- ```
  (define result
    (lambda ()
      (compute-answer-to-life-univ-and-everything)))
  ```
  – Note that just by defining a variable to hold the result doesn't mean we "really" need it yet.
- ```
  (if (= (result) 42)
    (do something) (do something else))
  ```
  – Now we need the value, so we compute it with **`(result)`**.
  ```

Avoiding expensive computations

Thunks let you skip expensive computations if they aren't needed.

```
(define result
  (lambda ()
     (compute-answer-to-life-univ-and-everything)))
(if (want-to-know-answer?)
  (display (result)) (display “save time”))
```

Don't compute the answer to life, the universe, and everything unless you really want to know.

- Pro: More flexible than putting the computation itself inside of the if statement.
- Con: Every time we call **`(result)`**, we compute the answer again! (Time waste, assuming the answer doesn't change)

```scheme
; simulate a long computation time
(define (compute-answer-to-life)
  (begin (sleep 3) 42))

; create a thunk for the answer
(define answer
   (lambda () (compute-answer-to-life)))

(answer) ; 3 second pause, then 42
(answer) ; 3 second pause again, then 42
```

Best of both worlds

- Assuming our expensive computation has no side effects, ideally we would:
  - Not compute it until needed.
  - Remember the answer so future uses don't re-compute (memoization).

- This is known as ***lazy evaluation***.

- Languages where most constructs, including function calls, work this way are called *lazy languages* (e.g., Haskell).

- Racket by default is an eager language, but we can add support for laziness.

Best of both worlds

- Here is our strategy for introducing optional laziness into an eager language:

- Create a data structure called a ***promise*** to represent a computation that may or may not take place at some point in the future.
  - Promises must store a thunk (the code for the computation),
  - something representing whether or not the thunk has been evaluated yet,
  - and the result of the thunk if it has been evaluated.

- Promises are not specific to Racket (though they appear a lot in similar functional languages). Other languages call them ***futures*** (e.g., Python, Java, C++).

Implementing promises

We will use a mutable pair to implement the promise data structure.

The car will always be a boolean, the cdr will be one of two things:

- **#f** in **car** means **cdr** is an unevaluated thunk.
- **#t** in **car** means **cdr** is the result of evaluating the thunk.

```
(define (make-promise thunk)
  (mcons #f thunk))

(define (eval-promise p)
  (if (mcar p)
      (mcdr p)
      (begin (set-mcar! p #t)
             (set-mcdr! p ((mcdr p)))
             (mcdr p))))
```

make-promise: create a promise data type for the thunk argument.

eval-promise: return result of thunk (either run it and save the return value for later, or return previously-saved value).

Using promises

```
; simulate a long computation time
(define (compute-answer-to-life)
  (begin (sleep 3) 42))

; create a promise to hold a thunk for the answer
(define answer2
   (make-promise
      (lambda () (compute-answer-to-life))))

(eval-promise answer2) ; 3 second pause, then 42
(eval-promise answer2) ; instant 42
```

Racket promises

- Making our own promise data structure is still clunky because we have to explicitly wrap the thunk in a lambda.

- Racket has built-in promises (yay!)
  - **`(delay e)`**: special form that is equivalent to our make-promise.
    - (No extra lambda needed, b/c **`delay`** is a special form).

  - **`(force p)`**: equivalent to our function eval-promise.
    - Evaluates a promise (something returned by **`delay`**) to compute whatever the value of **`e`** is. Also caches the value so future forces will be very fast, even if the evaluation of the original expression is slow.

```scheme
(define (compute-answer-to-life)
   (begin (sleep 3) 42))

(define answer3 (delay (compute-answer-to-life)))
(force answer3) ; 3 second pause, then 42
(force answer3) ; instant 42
```

Lazy lists, or streams

- One common use of delayed evaluation is to create a “lazy list,” or a “stream.”

- By convention, a stream is just like a Racket list in that it consists of two parts: the car and the cdr.
  - Only difference is that the cdr is lazy (car is not usually lazy).
  - In other words, the cdr is a promise to return the rest of the stream when its really needed.

- We do this by creating a function that creates a cons cell where the car is normal but the cdr is lazy.

Streams

- **`stream-cons`**: a special form that creates a new pair where the car is eager but the cdr is lazy.
  - alternatively, think of this as creating a new stream from a new first element and an existing stream.
  - just like regular cons creates a new list from a new first element and an existing list:
    - **`(cons 1 '(2 3))`** ➔ **`'(1 2 3)`**

- ```
  (define (stream-cons first rest)
     (cons first (delay rest))
  ```

  the above definition is correct in spirit, though wrong in syntax because we need to make **`stream-cons`** a special form so that **`rest`** won't be evaluated when stream-cons is called.
  ```

Streams

```
(define-syntax-rule (stream-cons first rest)
    (cons first (delay rest)))

(define (stream-car stream)
  (car stream))

(define (stream-cdr stream)
  (force (cdr stream)))

(define the-empty-stream '())

(define (stream-null? stream)
  (null? stream))
```

This is how you create a special form.

Let's try it out