Side Effects

3 December 2020
Warning: Dangerous Side Effects

Do *not* use anything in this lecture on Assignment 5 or on Exam 3!
What’s the signature for big-bang?
What’s the signature for `big-bang`?

```
;; big-bang : X -> X
```
What’s the signature for \texttt{big-bang}?

\texttt{;; \texttt{big-bang} : X \rightarrow X}

Does this tell the whole story?
**big-bang** is a *function* – it takes one or more inputs and returns an output – but it has a notable *side effect*: It displays a simulation.

To control how this simulation gets draw, we also provide **big-bang** with several special clauses specifying the functions to use, e.g., `[to-draw ...]`
We’ve been working in a paradigm called *functional programming*, which avoids side effects.

This makes many things simpler, but it’s not the only way to program.
If we switch to the *Advanced Student* language, we gain access to other functions with side effects.

One of these is **printf**:

```scheme
> (printf "hi")
hi
> (define X "hello")
> (printf X)
hello
```
printf is so named because it can print formatting strings.

E.g., to print a string where you include a number in it, you can do this:

```scheme
> (define N 42)
> (printf "The answer is~%...~%~A!" N)
The answer is
...
42!
```

~A is replaced with an argument that comes after the string. ~% is replaced with a newline.
If all we could do was print, side effects wouldn’t be very important.

But – wait – there’s more!
Destructive programming
Destructive programming with \texttt{set!}

The special form \texttt{set!} changes the value of a local or global symbol \textit{destructively}. That is, its previous value is permanently lost.

\begin{verbatim}
(set! \langle name \rangle \langle newvalue \rangle)
\end{verbatim}

The name is not evaluated (otherwise it would be replaced by its current value), but its binding is changed to \texttt{\langle newvalue \rangle}, e.g.,

\begin{verbatim}
> (define x 4)
> x
4
> (set! x 5)
(void)
> x
5
\end{verbatim}
set! vs define

We can use define at the top-level or in local to define a new value.

But it won’t work correctly in the body of a function; you’ll get an error.
Using `set!`, write a function that updates a global variable, `max-val`, to be the largest number in a list.
(define max-val #false)

;; find-max! : [List-of Number] -> [List-of Number]
;; Update max-val to be the largest number in the
;; input list.
(define (find-max! lon)
  (map (λ (x)
         (if (or (false? max-val)
                 (> x max-val))
             (set! max-val x)
             ;; Do nothing...
             (set! max-val max-val)
         lon))
lon))
(define max-val #false)

;; find-max! : [List-of Number] -> [List-of Number]
;; Update max-val to be the largest number in the
;; input list.
(define (find-max! lon)
  (map (λ (x)
         (when (or (false? max-val)
                   (> x max-val))
           (set! max-val x)))
       lon))

when is like if without an “else”. 
When we want to run a function on each element of a list, but we *only care about the side effect* of running it, we can use *for-each* instead of *map*.
(define max-val #false)

;; find-max! : [List-of Number] -> None
;; Update max-val to be the largest number in the
;; input list.
(define (find-max! lon)
  (for-each (λ (x)
                (when (or (false? max-val)
                          (> x max-val))
                     (set! max-val x)))
            lon))
Where are the test cases?

Because we’re interested in the side effect the function has on things outside of itself rather than what it returns, it can be very hard to test programs like this.

To test `find-max!`, we’d change the state of the program. We’d need to reset `max-val` before and after every test.

This is one of the reasons the functional programming paradigm we’ve been using is a good thing!
Well, it seems like it would have been better to write

\[ \text{find-max : [List-of Number] \rightarrow Number} \]

When might you \textit{really} want to use destructive programming?
Well, you might want to use a global variable to store your grocery list:

```
(define groceries
  (list "carrots" "avocados" "hummus"))
```
You could add items to your grocery list by making a new list every time, e.g.,

\[
\text{(define \texttt{groceries-v2}} \text{ (cons "bread" groceries))}
\]

But that’s not really what we want; we want to actually \textit{change} the list.
The best approach is to write a function using `set!`:

```
(define (add-grocery! item)
  (set! groceries
    (cons item groceries)))
```

When this is called, the global variables `groceries` is defined to be a new list made by `cons`-ing the item onto the existing list:

```
> (add-grocery! "chocolate")
(void)
> groceries
(list "chocolate" "carrots" "avocados" "hummus")
```
To change values that are parts of *data structures*, we need more specific mutator functions than `set!`. 
Destructive programming and vectors
A **vector** is Racket’s version of the **array** data type found in many programming languages, which is an ordered collection of elements that can be accessed by an integer index.

Usually all elements of an array have the same type, though in Racket’s vectors they don’t need to.

In most languages, the actual values are stored in the array. In Racket, pointers are stored, which means that the contents of a vector can be of any size without needing to make a bigger vector.
Making vectors

Unlike lists, vectors are not constructed incrementally; they are created with a fixed length in one shot.

#(item1 item2 ...) creates an immutable vector. Like quote notation for lists, the arguments to # are shielded from evaluation.

> '(2 3 4 (+ 2 3))
(list 2 3 4 (list '+ 2 3))
> #(2 3 4 (+ 2 3))
(vector-immutable 2 3 4 (list '+ 2 3))
Making vectors

Unlike lists, vectors are not constructed incrementally; they are created with a fixed length in one shot.

The `vector` function is like the `list` function, letting you specify the items, which are evaluated:

```
> (list 2 3 4 (+ 2 3))
(list 2 3 4 5)
> (vector 2 3 4 (+ 2 3))
(vector 2 3 4 5)
```
We can access vector elements by their index using `vector-ref`. This is like the `list-ref` function we’ve seen, but `list-ref` needs to walk through the list to find an element:

```scheme
> (define v1 (vector "a" "b" "c"))
> (vector-ref v1 1)
"b"
```
We can destructively modify vector elements by their index using `vector-set!`.

```scheme
> (define v1 (vector "a" "b" "c"))
> (vector-set! v1 1 "beta")
> v1 (vector "a" "beta" "c")
```
Lists vs vectors

Since lists and vectors have essentially the same functionality, why do we have both?

Efficiency!

Under the hood, lists are linked sequences of cons cells, with the rest of each cons cell holding the address of the next cell in the sequence. These cons cells might be scattered all over the computer’s memory.

In contrast, the entries in a vector are allocated all at once and are stored in one contiguous block of memory. This makes for fast access times to arbitrary vector elements.
Lists are good for doing list-based recursion, especially if we’re not creating lots of new cons cells in the process.

Constant-time (i.e., fast) operations for lists are `first`, `rest`, `cons`, and `empty?`.

Slow operations are `length`, `list-ref`, and `append`.

Vectors, on the other hand, are good for other purposes in which we jump from place to place in the vector, erasing and re-writing the contents of entries in any order.

Constant-time (i.e., fast) operations for vectors are `vector-ref`, `vector-set!`, and `vector-length`.

There are no equivalents to `first`, `rest`, or `cons` for vectors. Instead of doing recursion until the vector is empty, you need to do recursion over a position number, often ending when the position number is equal to the length of the vector.
;; vector-swap! : Vector, Natural, Natural -> None
;; Swap the values stored at the indices in the vector
(define (vector-swap! vec i j)
  (local [[(define tmp (vector-ref vec i))]
            (begin
              (vector-set! vec i (vector-ref vec j))
              (vector-set! vec j tmp))))

> (define v (vector "a" "b" "c"))
> (vector-swap! v 0 2)
> v
(vector "c" "b" "a")
Destructive programming and structures
We’ve seen the use of `set-vector!` for vectors, but what about the structures we define ourselves?
When we define a structure with `define-struct`, it does even more than we learned previously.

In addition to defining a constructor, type-checker predicate, and selectors for each field, it also defines a mutator function for each field.
For example, if we define a structure `person` to represent a person’s names:

```
(define-struct person [first-name last-name])
```

Then Racket defines `set-person-first-name!` and `set-person-last-name!` for us:

```
> (define indy (make-person "Henry" "Jones"))
> (set-person-first-name! indy "Indiana")
> (person-first-name indy)  
"Indiana"
```
(define-struct student
    [first-name last-name grad-class])

> (define edna (make-student "Edna" "Millay" 1917))
> (student? edna)
#true
> (student-first-name edna)
"Edna"
> (define grace (make-student "Grace" "Hopper" 1928))
> (define liz (make-student "Elizabeth" "Bishop" 1934))
> (define students (list edna grace liz))
> (map student-grad-class students)
(list 1917 1928 1934)
What if we wanted to [travel back in time and] hold everyone back? I.e., everyone’s graduating a year later.
> (map student-grad-class students)
(list 1917 1928 1934)
> (for-each (lambda (x)
       (set-student-grad-class! x
       (add1 (student-grad-class x)))))
students)
> (map student-grad-class students)
(list 1918 1929 1935)
Remember: Side effects can let us do cool things – but they also lose major benefits of functional programming.

In functional programming, all that matters is what a function takes as input and what it produces as output – nothing else in the world is relevant.

This has made our functions (relatively) easy to test, which is important for making reliable programs!
Acknowledgments

This lecture incorporates material from:

Simon Ellis
Luke Hunsberger
Gordon S. Novak Jr
Jennifer Walter