A Self-Updating Stream

In the next part of the course, we will build up to a programming paradigm called **logic programming**. Logic programming is considered more *declarative* than the other programming paradigms that you are used to, meaning that we tend to describe *what* the desired outputs look like, rather than *how* the computation should perform the task.

Logic programming is related to **artificial intelligence**. In particular, we will solve problems (e.g. Sudoku) by enumerating through possible solutions, then filtering those possibilites for actual solutions. With problems like these, we will often work with a large (or even infinite) *search space*. So, we start this section by reviewing our definition of **streams**.

Review: Stream Operations (from Prof. David Liu's notes)

Streams are lazy lists. These are the stream operations that we developed in the past few weeks. We developed these operations to somewhat match the list operations that we are used to. Recall that `s-cons` and `make-stream` are macros. This is so that their arguments are not evaluated eagerly.

```
; Empty stream value, and check for empty stream.
(define s-null 's-null)
(define (s-null? stream) (equal? stream s-null))

#|
(s-cons <first> <rest>)
  <first>: A Racket expression.
  <rest>: A stream (e.g., s-null or another s-cons expression).

  Returns a new stream whose first value is <first>, and whose other
  items are the ones in <rest>. Unlike a regular list, both <first>
  and <rest> are wrapped in a thunk, so aren't evaluated until called.
|#
(define-syntax s-cons
  (syntax-rules ()
    [(s-cons <first> <rest>)
     (cons (thunk <first>) (thunk <rest>))]))

; These two define the stream-equivalents of "first" and "rest".
; We need to use `car` and `cdr` (similar to `first` and `rest`)
; for a technical reason that isn't important for this course.
; Note that s-rest both accesses the "rest thunk" and calls it,
; so that it does indeed return a stream.
(define (s-first stream) ((car stream)))
(define (s-rest stream) ((cdr stream)))

; Make a stream. We use macros to so that the elements of the stream
; are not evaluated until they are used.
(define-syntax make-stream
  (syntax-rules ()
    [(make-stream) s-null]
    [(make-stream <first> <rest> ...)
     (s-cons <first> (make-stream <rest> ...))]))
```

Python Generators

Many kinds of data are naturally represented using streams: for example, you could represent all tweets as a stream, or stock prices over time, or the temperature readings from a weather station.

As such, the *idea* of a stream (decoupled from our Racket implementation) is actually quite general and pervasive. Different programming languages have different ways of representing streams, and use the idea of lazy evaluation in different ways. To motivate our next topic, let's take a look at how Python implements generators.

A **generator function** in Python is a way of creating an iterator with possibly infinite number of values. Such a function uses the `yield` keyword, rather than the `return` keyword. Here is a simple example:

```
def my_gen():
    n = 1
    print("yielding 1...")
    yield n
    n += 1
    print("yielding 2...")
    yield 2
    n += 1
    print("yielding 3...")
    yield 3
    print("finished")
```

Calling the function `my_gen` yields an iterator, which can be looped over using a `for` loop (e.g. `for x in my_gen(): ...`). We can also use the Python function `next` to obtain the results one by one.

```
>>> g = my_gen()
>>> next(g)
yielding 1...
1
>>> next(g)
yielding 2...
2
>>> next(g)
yielding 3...
3
>>> next(g)
finished
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Every time the function `next` is called, Python continues execution of the generator function until the next `yield` is reached. Once that happens, Python returns the yielded value, and suspends execution of the function. The values of local variables like `n` are preserved, and are used the next time we call `next`.

Like our Racket streams, Python generators can be used to generate an infinite sequence of values.

```
define repeat(n):
    while True:
        yield n
```

This generator repeats the value `n` an infinite number of times:

```
>>> g = repeat(3)
>>> next(g)
3
>>> next(g)
3
>>> next(g)
3
>>> next(g)
3
```

A self-updating stream (with mutation)

The Python generator has an interesting and simple syntax that we would like to replicate using streams. In particular, we would like a Racket syntax `next!` that behaves like this:

```
> (define s (make-stream 1 2 3))
> (next! s)
1
> (next! s)
2
> (next! s)
3
> (next! s)
'DONE
> s
's-null
```

The exclamation mark `!` at the end of `next!` is pronounced "bang", and is used to signal a function that uses *mutation*. As you might have guessed, `next!` uses mutation to update the value of `s`. To be more specific, we would like `next!` to do the following:

When the stream `s` is non-empty, we would like to

- Update the stream to `s-rest` of the stream
- Return the `s-first` of the stream

In other words, when `s` is nonempty, `(next! s)` is equivalent to the following code:

```
(let* ([tmp s])          ; save a handle to `s`
  (begin
    (set! s (s-rest s))  ; re-bind `s` to the `(s-rest s)`
    (s-first tmp)))      ; return the first value, saved from earlier
```

When the stream `s` is empty, then we will simply return the symbol `'DONE`.

How can we implement `next!`? Since we are trying to mutate a variable, `next!` *cannot* possibly be a function. (Why?) Instead, we will define `next!` as a macro.

```
(define-syntax next!
  (syntax-rules ()
    [(next! <g>)
     (if (s-null? <g>)
         'DONE
         (let* ([tmp <g>])
           (begin
             (set! <g> (s-rest <g>))
             (s-first tmp))))]))
```

Let's check that this macro can be used with other streams, including infinite streams. We'll write a `repeat` function in Racket to create an infininte stream, and take items from that stream:

```
> (define (repeat n) (s-cons n (repeat n)))
> (define g (repeat 3))
> (next! g)
3
> (next! g)
3
> (next! g)
3
> (next! g)
3
```

One nice feature of streams is that their values are evaluated lazily. Errors will not appear until an invalid stream element is reached, and are isolated to the single invalid element.

```
> (define g (make-stream 1 2 (/ 3 0) 4))
> (next! g)
1
> (next! g)
2
> (next! g)
ERROR /: division by zero
> (next! g)
4
> (next! g)
'DONE
```

Acknowledgements

Portions of these notes are derived from the CSC324 Notes by Prof. David Liu. The portions that are directly duplicated are indicated. Other portions are still borrowed heavily from Prof. Liu's work. https://www.cs.toronto.edu/~david/