Delayed evaluation

Application: Working with streams

Lazy evaluation with nullary functions
Main idea

- To delay the execution of a sequence of definitions and expressions, wrap it in `(lambda () ...)`

$\Rightarrow$ a nullary function whose body is executed only when we call it.

Example

`sum12` is a nullary function with body `(+ 1 2)`. To trigger the computation of the body, we must call `(sum12)`:

```
> (define sum12 (lambda () (+ 1 2)))
> sum12
#<procedure:sum12>
> (sum12)
3
```

Lexical closures

Function values are lexical closures: They remember the values of the variables from the context when they were created.

Example (A function that returns a nullary function as result)

```
> (define (sum x y) (lambda () (+ x y)))
> (define sum34 (sum 3 4))
> sum34
#<procedure>
```

- `sum34` is the value of the nullary function `(lambda () (+ x y))`.
- When `sum34` was created, `x` was `3`, and `y` was `4` $\Rightarrow$ `sum34` will "remember" these values for `x` and `y`:

```
> (sum34) ; compute 3+4
7
```

Lazy evaluation with `delay/force`

Example (delayed evaluation of sum)

The conventional definition of `sum` is:

```
> (define sum (lambda (x y) (+ x y)))
```

To delay it's evaluation, call

```
> (define s (delay (sum 1 2)))
> s
#promise
```

To perform a delayed computation *c*, call `(force` *c*`)`:

```
> (force s)
3
```

1. Whenever we wish to delay some computation *body*, we can wrap it in the body of a nullary function:

   `> (define` ***delayed-work*** `(lambda ()` *body*`))`

2. When we wish to perform the computation of *body*, we call the nullary function

   `> (`*delayed-work*`)`

With this technique, we have full control of the evaluation process:

- ▷ We can delay computations and execute them only when really needed.

This way of computing is called lazy evaluation.

Stream: a finite representation of an infinite list, where we know how to generate new elements from previous elements.

Examples of steeams:

All ones: `(1 1 1 ...)`
: Next element is always 1.

Natural numbers: `(0 1 2 3 ...)`
: Next element is successor or previous one.

Fibonacci numbers: `(1 1 2 3 5 8 13 ...)`
: Every element, except first two, is sum of previous two elements.

Prime numbers: `(2 3 5 7 11 13 ...)`
: Every next element is the first natural number different from 1, which is not a multiple of previous elements. (This is the idea of the Sieve of Erathostenes)

Application: Streams
Main idea

How to represent in a finite way a stream?

$(a_1 \;\; a_2 \;\; a_3 \;\; \ldots)$

How to represent in a finite way a stream?

$(a_1 \;\; a_2 \;\; a_3 \;\; \ldots)$

$(a_1 \;\; \ldots \;\; a_k \;\; . \;\; gen)$

where *gen* is a nullary function that can generate more elements on demand:

- (gen) computes $(a_{k+1} \;\; \ldots \;\; a_\ell . \;\; gen')$
- *gen* is called the stream generator.
- A generator is just a function, and function *gen* is recognised with `(procedure?` *gen*`)`

Streams

Examples

```
(define gen-ones
   (lambda () (cons 1 gen-ones)))
; stream of all ones
(define all-ones (cons 1 gen-ones))

(define (gen-nats n)
      (cons n (lambda () (gen-nats (+ n 1)))))

; stream of all naturals
(define nats (gen-nats 0))

>  all-ones
'(1 . #<procedure:gen-ones>)

> naturals
'(0 . #<procedure>)
```

Working with streams

Utility functions: s-take and s-filter

```
; list of first n elements from stream s
(define (s-take n s)
  (cond [(= n 0) '()]
        [(procedure? s) ; s is the stream generatoe
            (s-take n (s))]
        [#t (cons (car s) (s-take (- n 1) (cdr s)))]))

; stream of all elements of s which satisfy predicate p
(define (s-filter p s)
  (cond [(procedure? s) ; s is the stream generator
        (s-filter p (s))]
        [(p (car s))
            (cons (car s)
                  (lambda () (s-filter p (cdr s))))]
        [#t (s-filter p (cdr s))]))

> (s-take 5 (s-filter even? nats))
'(0 2 4 6 8)
```

Working with streams

Utility functions: s-map

```
(s-map f s)
```

- takes as inputs a stream s and a function that computes a value for any element of s
- returns the stream obtained by applying function f to all elements of s

```
(define (s-map f s)
   (if (procedure? s) (s-map f (s))
       (cons (f (car s))
             (lambda () (s-map f (cdr s))))))
```

```
>  (define cubes
     (s-map (lambda (x) (* x x x)) nats))
> (s-take 7 cubes)
'(0 1 8 27 64 125 216)
```

Utility functions on streams of numbers

s-add

(s-add s1 s2)

- takes as inputs two streams of numbers

  s1 = (a_1 a_2 ...)

  s2 = (b_1 b_2 ...)

- returns the stream $a_1 + b_1$ $a_2 + b_2$ ...)

```
(define (s-add s1 s2)
  (cond [(procedure? s1) (s-add (s1) s2)]
        [(procedure? s2) (s-add s1 (s2))]
        [#t (cons (+ (car s1) (car s2))
                  (lambda ()
                    (s-add (cdr s1) (cdr s2))))]))
```

```
> ; stream of numbers n^2 + n for all n
  (define ns (s-add (s-map (lambda (x) (* x x)) nats)
                    nats))
> (s-take 6 ns)
'(0 2 6 12 20 30)
```

Stream of Fibonacci numbers

Useful observation: the stream `fib` of Fibonacci numbers has the following useful property:

- Adding streams `fib` and `(cdr fib)` yields `(cddr fib)`

$$
\begin{array}{rcllll}
\texttt{fib} & = & f_0 & f_1 & f_2 & \dots \quad + \\
(\texttt{cdr fib}) & = & f_1 & f_2 & f_3 & \dots \\
\hline
f_0 \;\; f_1 & & f_2 & f_3 & f_4 & \dots
\end{array}
$$

- Once we know the first two elements f_0 and f_1, we can start generating the rest of the stream:

```
(define fib
    (cons 1
          (cons 1
                (lambda () (s-add fib (cdr fib))))))

> (s-take 10 fib)
'(1 1 2 3 5 8 13 21 34 55)
```