Programming Abstractions

Week 12-2: Promises

Stephen Checkoway

Finishing up macros

Consider `switch`

```
(switch exp [case-1 exp-1] ... [case-n exp-n])
```

The behavior we want is

- exp is evaluated;
- the result is compared against each of `case-1` through `case-n` in order;
- if the result is equal to `case-i` then the value of the expression is `exp-i`

It should behave the same as

```
(let ([result exp])
  (cond [(equal? result case-1) exp-1]
        ...
        [(equal? result case-n) exp-n]))
```

Let's define a switch syntax!

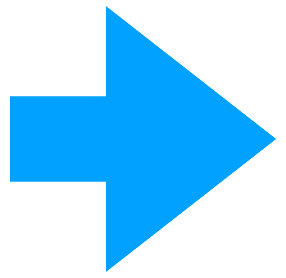
```
(define-syntax switch
  (syntax-rules ()
    [(_ exp [case case-exp] ...)
     (let ([result exp])
       (cond [(equal? result case) case-exp] ...))]))

(switch (- 2 1)
        [0 "zero"]
        [1 "one"]
        [2 "two"])
```

Let's define a switch syntax!

```
(define-syntax switch
  (syntax-rules ()
    [(_ exp [case case-exp] ...)
     (let ([result exp])
       (cond [(equal? result case) case-exp] ...))]))
```

```
(switch (- 2 1)
        [0 "zero"]
        [1 "one"]
        [2 "two"])
```



```
(let ([result (- 2 1)])
  (cond [(equal? result 0) "zero"]
        [(equal? result 1) "one"]
        [(equal? result 2) "two"]))
```

What is the value of this?

```
(define-syntax switch
  (syntax-rules ()
    [(_ exp [case case-exp] ...)
     (let ([result exp])
       (cond [(equal? result case) case-exp] ...))]))

(switch 3
        [0 "zero"]
        [1 "one"]
        [2 "two"])
```

A. 3

B. `"three"`

C. `void`

D. It's an error

Let's add an [else exp] to switch

We want to support an else

```
(switch 3
        [0 "zero"]
        [1 "one"]
        [2 "two"]
        [else "something else"])
```

As we've currently implemented switch, this won't work

- Why not?

Let's add an [else exp] to switch

We want to support an else

```
(switch 3
        [0 "zero"]
        [1 "one"]
        [2 "two"]
        [else "something else"])
```

As we've currently implemented switch, this won't work

- Why not?

```
(let ([result 3])
  (cond [(equal? result 0) "zero"]
        [(equal? result 1) "one"]
        [(equal? result 2) "two"]
        [(equal? result else) "something else"]))
```

First attempt

```
(define-syntax switch
  (syntax-rules ()
    [(_ exp [case case-exp] ... [else else-exp])
     (let ([result exp])
       (cond [(equal? result case) case-exp] ...
             [else else-exp]))]
    [(_ exp [case case-exp] ...)
     (switch exp [case case-exp] ... [else (void)])]))
```

Recursive macros are fine!

Two rules, each with a pattern and a matching transformation

Idea: a `(switch …)` without an `[else …]` matches the second rule;
a `(switch …)` with an `[else …]` matches the first rule

Trying it out

```
(switch 3
        [0 "zero"]
        [1 "one"]
        [2 "two"]
        [else "something else"])
```

returns `"something else"`

Success?

Not quite

```
(switch 3
        [0 "zero"]
        [1 "one"]
        [2 "two"])
```

returns `"two"!`

The problem is this `switch` matches the first pattern

```
(_ exp [case case-exp] ... [else else-exp])
```

We need to inform Racket that `else` is not a pattern variable and is meant to be matched literally

Not quite

```
(switch 3
        [0 "zero"]
        [1 "one"]
        [2 "two"])
```

```
(let ([result 3])
  (cond [(equal? result 0) "zero"]
        [(equal? result 1) "one"]
        [2 "two"]))
```

returns `"two"!`

The problem is this `switch` matches the first pattern

```
(_ exp [case case-exp] ... [else else-exp])
```

We need to inform Racket that `else` is not a pattern variable and is meant to be matched literally

Literal matches

```
(syntax-rules (literal ...) [pattern transform] ...)
```

The first argument to `syntax-rules` is a list of words to match literally

`else` is not a pattern variable; it's matched literally

```
(define-syntax switch
  (syntax-rules (else)
    [(_ exp [case case-exp] ... [else else-exp])
     (let ([result exp])
       (cond [(equal? result case) case-exp] ...
             [else else-exp]))]
    [(_ exp [case case-exp] ...)
     (switch exp [case case-exp] ... [else (void)])]))
```

Second attempt

```
(switch 3
        [0 "zero"]
        [1 "one"]
        [2 "two"])
```

Result is `void`

```
(let ([result 3])
  (cond [(equal? result 0) "zero"]
        [(equal? result 1) "one"]
        [(equal? result 2) "two"]
        [else (void)]))
```

```
(switch 3
        [0 "zero"]
        [1 "one"]
        [2 "two"]
        [else "blah"])
```

Result is `"blah"`

```
(let ([result 3])
  (cond [(equal? result 0) "zero"]
        [(equal? result 1) "one"]
        [(equal? result 2) "two"]
        [else "blah"]))
```

Macros match arguments, not evaluate

When a macro is being evaluated, the arguments are matched against the pattern but they aren't evaluated

```
(switch 1
        [0 (displayln "zero")]
        [1 (displayln "one")]
        [2 (displayln "two")]
        [else (displayln "something else")])
```

This prints `one`

If the arguments were evaluated (well, it'd be an error because 0 isn't a procedure) but it'd also print out `zero`, `one`, `two`, `something else`

Hygienic macros?

Macros in other languages can introduce variables that shadow variables used in the arguments (unhygienic)

```
(define-syntax value-of-var
  (syntax-rules ()
    [(_ var) (let ([x 0]) var)]))
(let ([x 10])
  (value-of-var x))
```

If Scheme used textual replacement, the `let` would become

```
(let ([x 10])
  (let ([x 0]) x))
```

which would have value 0

Scheme macros are hygienic so the actual value is 10

Promises

Promises

Some new Scheme special forms

`(delay exp)` returns an object called a *promise*, without evaluating `exp`

`(force promise)` evaluates the promised expression and returns its value

- A promised expression is evaluated only once, no matter how many times it is evaluated!

Example

```
(define foo
  (delay
    (begin
      (displayln "Promise is evaluated")
      2)))

(force foo) ; prints "Promise is evaluated"; returns 2
(force foo) ; returns 2
(force foo) ; returns 2
```

Example

`begin` not needed in Racket
`delay` allows arbitrary number of expressions

```
(define foo
  (delay
    (begin
      (displayln "Promise is evaluated")
      2)))

(force foo) ; prints "Promise is evaluated"; returns 2
(force foo) ; returns 2
(force foo) ; returns 2
```

Implementing delay and force

Before we talk about *why* we might want this, let's talk about how we can implement it

First attempt: define delay as a procedure that returns a procedure

```
(define (delay exp)
  (λ ()
    exp))

(define (force promise)
  (promise))
```

What goes wrong with this definition?

```
(define (delay exp)
  (λ ()
    exp))

(define (force promise)
  (promise))
```

A. When you know what goes wrong, select this choice

Evaluation isn't delayed

```
(delay
  (displayln "Lazy evaluation would be nice"))
```

Since `delay` was implemented as a procedure, its argument is evaluated when `delay` is called

`force` will correctly return the value, but it was already computed; we need to delay the computation until `force` is called

We need a macro!

Let's think about what we want

We want

```
(delay exp)
```

to become something like

```
(λ () exp)
```

Second attempt: define delay as a macro which produces a λ

```
(define-syntax delay
  (syntax-rules ()
    [(_ exp) (λ () exp)]))

(define (force promise)
  (promise))
```

Example

```
(define foo
  (delay
    (begin
      (displayln "This time, it's lazy!")
      10)))
```

This successfully defines foo as

```
(λ ()
  (begin
    (displayln "This time, it's lazy!")
    10))
```

and it doesn't evaluate until `(force foo)`

What goes wrong with this definition?

```
(define-syntax delay
  (syntax-rules ()
    [(_ exp) (λ () exp)]))

(define (force promise)
  (promise))
```

A. When you know what goes wrong, select this choice

Each time we force the promise, it's evaluated

```
(force foo) ; prints "This time it's lazy"; returns 10
(force foo) ; prints "This time it's lazy"; returns 10
(force foo) ; prints "This time it's lazy"; returns 10
```

We're going to need some mutation

We need to remember two things

- Has the promise been forced yet?
- If so, what was the value?

What we really want

We want

```
(delay exp)
```

to become something like

```
(let ([evaluated #f]
      [value 0])
  (λ ()
    (if evaluated
        value
        (begin
          (set! value exp)
          (set! evaluated #t)
          value))))
```

When the result is forced (i.e., called) the first time

- `exp` will be evaluated
- `value` will be set to the result
- `evaluated` will be set to `#t`
- `value` is returned

On subsequent calls

- value is returned

When would we use promises?

We can build an infinite data structure like an infinite list

- An infinite list of primes
- The Fibonacci sequence

If our language supports concurrent execution (i.e., multiple computations happening at the same time), we can model a long-running computation as a promise

- Creating the promise doesn't actually delay evaluation, it starts a *thread* that performs the computation
- Forcing the promise causes the current thread to wait until the computing thread has finished before returning the answer

Promises in other languages

JavaScript has `async` which starts some potentially long-running calculation or (more typically) starts loading a resource from the Internet and returns a promise

This is paired with `await` which waits for the promise to finish computing/resource to download and returns the answer

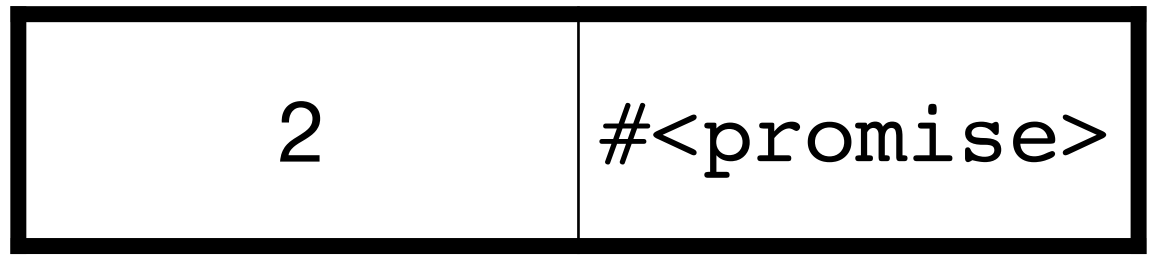
Rust has something similar

Let's build an infinite list of primes

First, we need to think about how we want to represent this

Let's use a `cons` cell where

- the `car` is a prime; and
- the `cdr` is a promise which will return the next `cons` cell

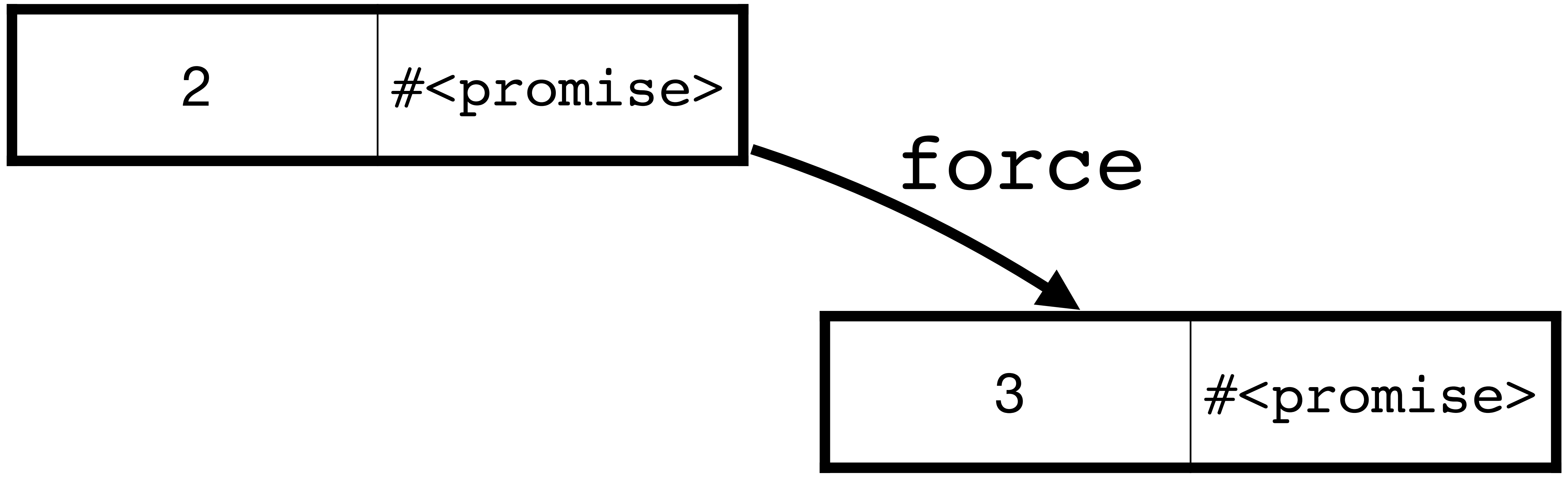


Let's build an infinite list of primes

First, we need to think about how we want to represent this

Let's use a `cons` cell where

- the `car` is a prime; and
- the `cdr` is a promise which will return the next `cons` cell

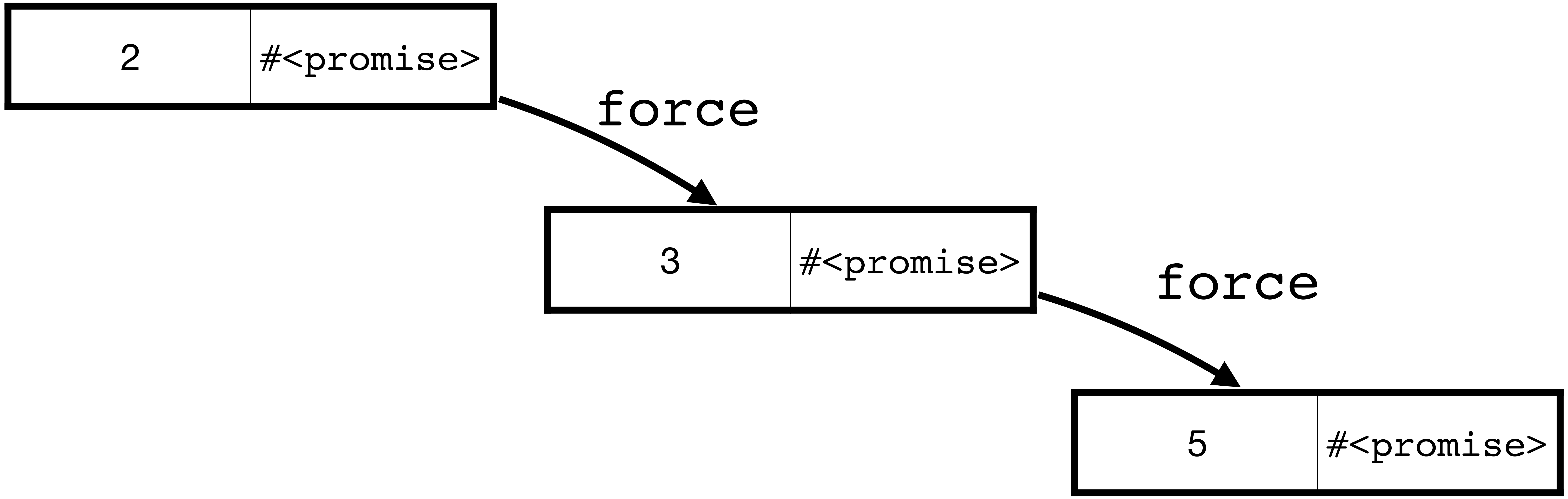


Let's build an infinite list of primes

First, we need to think about how we want to represent this

Let's use a `cons` cell where

- the `car` is a prime; and
- the `cdr` is a promise which will return the next `cons` cell



The uninteresting piece: checking primality

```
(define (prime? n)
  (cond [(= n 2) #t]
        [(even? n) #f]
        [else (not
               (ormap
                (λ (m) (zero? (remainder n m)))
                (range 3
                       (add1 (exact-floor (sqrt n)))
                       2)))]))
```

Does the simple thing and checks if dividing n by any odd m up to $\sqrt{n}$ gives remainder 0

The interesting piece: building the list

`next-prime` checks if n is prime and if so, returns a `cons` cell containing n and a promise to construct the next one; otherwise it recurses on n+2

```
(define (next-prime n)
  (cond [(prime? n) (cons n
                          (delay (next-prime (+ n 2))))]
        [else (next-prime (+ n 2))]))
```

`primes` returns a cons cell containing 2 and a promise to construct the next one

```
(define (primes)
  (cons 2
        (delay (next-prime 3))))
```

Infinite list in action!

```
> (define prime-lst (primes))
> prime-lst
'(2 . #<promise>)
> (force (cdr prime-lst))
'(3 . #<promise>)
> (force (cdr (force (cdr prime-lst))))
'(5 . #<promise>)
> prime-lst
'(2 . #<promise!(3 . #<promise!(5 . #<promise>)>)>)
```

Using our list

```
(define (print-until n prime-lst)
  (let ([prime (car prime-lst)])
    (if (<= prime n)
        (begin
          (displayln prime)
          (print-until n (force (cdr prime-lst))))
        prime-lst))) ; Return the remainder of the list
```

Using our list

```
> (print-until 15 prime-lst)
2
3
5
7
11
13
'(17 . #<promise>)
```