Lecture 20: Scheme II

Brian Hou
July 26, 2016

Announcements

- Project 3 is due today (7/26)
- Homework 8 is due tomorrow (7/27)
- Quiz 7 on Thursday (7/28) at the beginning of lecture
  - May cover mutable linked lists, mutable trees, or Scheme I
- Opportunities to earn back points
  - Hog composition revisions due tomorrow (7/27)
  - Maps composition revisions due Saturday (7/30)
  - Homework 7 AutoStyle portion due tomorrow (7/27)

Roadmap

Introduction

Functions

Data

Mutability

Objects

Interpretation

Paradigms

Applications

- This week (Interpretation), the goals are:
  - To learn a new language, Scheme, in two days!
  - To understand how interpreters work, using Scheme as an example

The **let** Special Form (demo)

- The **let** special form defines local variables and evaluates expressions in this new environment

```
scm> (define x 1)
x
scm> (let ((x 10) (y 20))
       (+ x y))
30
scm> x
1
```

Tail Recursion

Factorial (Again) (demo)

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

```
(define (fact n)
  (define (helper n prod)
    (if (= n 0)
      prod
      (helper (- n 1) (* n prod))))
  (helper n 1))
```

```
scm> (fact 10)
scm> (fact 1000)
```

Tail Recursion

The Revised[7] Report on the Algorithmic Language Scheme:

> "Implementations of Scheme are required to be properly tail-recursive. This allows the execution of an iterative computation in constant space, even if the iterative computation is described by a syntactically recursive procedure."

How? Eliminate the middleman!

```
(define (fact n)
  (define (helper n prod)
    (if (= n 0) prod (helper (- n 1) (* n prod))))
  (helper n 1))
```

Tail Calls

- A procedure call that has not yet returned is *active*
- Some procedure calls are *tail calls*
- Scheme implementations should support an *unbounded number* of active tail calls using only a *constant* amount of space
- A tail call is a call expression in a tail context:
  - The last body sub-expression in a **lambda**
  - The consequent and alternative in a tail context **if**
  - All non-predicate sub-expressions in a tail context **cond**
  - The last sub-expression in a tail context **and**, **or**, **begin**, or **let**

Tail Contexts

- A tail call is a call expression in a tail context:
  - The last body sub-expression in a **lambda**
  - The consequent and alternative in a tail context **if**
  - All non-predicate sub-expressions in a tail context **cond**
  - The last sub-expression in a tail context **and**, **or**, **begin**, or **let**

```
(define (fact n)
  (define (helper n prod)
    (if (= n 0) prod (helper (- n 1) (* n prod))))
  (helper n 1))
```

Example: Length

```
(define (length s)
  (if (null? s) 0
    (+ 1 (length (cdr s)))))
```

Not a tail context

- A call expression is not a tail call if more computation is still required in the calling procedure
- Linear recursive procedures can often be rewritten to use tail calls

```
(define (length-tail s)
  (define (length-iter s n)
    (if (null? s) n
      (length-iter (cdr s) (+ 1 n))))
  (length-iter s 0))
```

Lazy Computation

Lazy Computation (demo)

- Lazy computation means that computation of a value is delayed until that value is needed
    - In other words, values are computed *on demand*

```
>>> r = range(11111, 1111111111)
>>> r[20149616]
20160726
```

Streams

- Streams are lazy Scheme lists: the rest of a list is computed only when needed

`(car (cons 1 2)) -> 1`	`(car        (cons-stream 1 2)) -> 1`
`(cdr (cons 1 2)) -> 2`	`(cdr-stream (cons-stream 1 2)) -> 2`
`(cons 1 (cons 2 nil))`	`(cons-stream 1 (cons-stream 2 nil))`

Streams (demo)

- Streams are lazy Scheme lists: the rest of a list is computed only when needed
- Errors only occur when expressions are evaluated

```
(cons-stream 1 (/ 1 0))   -> (1 . #[promise (not forced)])

(car        (cons-stream 1 (/ 1 0))) -> 1

(cdr-stream (cons-stream 1 (/ 1 0))) -> ERROR
```

Infinite Streams (demo)

- An integer stream is a stream of consecutive integers
- The rest of the stream is not computed when the stream is created

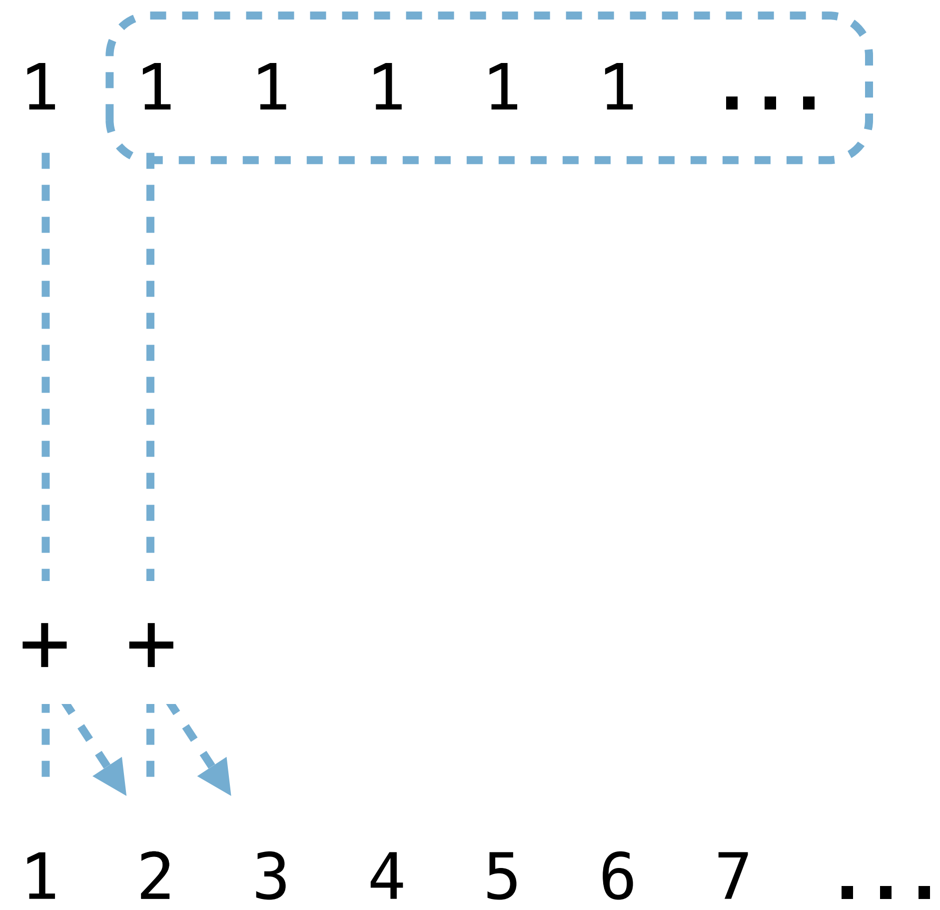
```
(define (int-stream start)
  (cons-stream
    start
    (int-stream (+ start 1))))
```

Recursively Defined Streams (demo)

```
(define ones (cons-stream 1 ones))

(define (add-streams s1 s2)
  (cons-stream
    (+ (car s1) (car s2))
    (add-streams
      (cdr-stream s1)
      (cdr-stream s2))))

(define ints
  (cons-stream 1
    (add-streams ones ints)))
```



A Stream of Primes (demo)

- For a prime k, any larger prime cannot be divisible by k
- Idea: Filter out all numbers that are divisible by k
- This idea is called the Sieve of Eratosthenes

2, 3, ~~4~~, 5, ~~6~~, 7, ~~8~~, ~~9~~, ~~10~~, 11, ~~12~~, 13


Break!

Symbolic Programming

Symbolic Programming

procedure

```
(define (square x) (* x x))
```

```
'(define (square x) (* x x))
```

list

- Lists can be manipulated with **car** and **cdr**
- Lists can created and combined with **cons, list, append**
- We can rewrite Scheme procedures using these tools!

List Comprehensions in Scheme (demo)

```
exp                      ((* x x) for x in '(1 2 3 4) if (> x 2))
(car exp)                                                  (* x x)
(car (cddr exp))                                                 x
(car (cddr (cddr exp)))                                  '(1 2 3 4)
(car (cddr (cddr (cddr exp))))                             (> x 2)
```

```
(list 'lambda (list 'x) '(* x x))
                                  (lambda (x) (* x x))
(list 'lambda (list 'x) '(> x 2))
                                  (lambda (x) (> x 2))
```

```
   (map (lambda (x) (* x x))
        (filter (lambda (x) (> x 2)) '(1 2 3 4)))
```

More Symbolic Programming

Rational numbers!

Summary

- Tail call optimization allows some recursive procedures to take up a constant amount of space — just like iterative functions in Python!
- Streams can be used to define implicit sequences
- We can manipulate Scheme programs (as lists) to create new Scheme programs
  - This is one huge language feature that has contributed to Lisp's staying power over the years
  - Look up "macros" to learn more!