Lecture 17: Scheme

July 20th, 2022

Cooper Bedin

Announcements

- Midterm grades have been released, and regrade requests are due Friday at 11:59PM PST
- A couple things to keep in mind re: regrade requests
  - Please don't ask us to add partial credit items—you should only submit a regrade request if you believe your work was graded incorrectly *according to the rubric that is already there*
  - Once you have received a response to your regrade request, that is a final decision—please don't contest it
  - If you're unsure whether you should submit a regrade request or not, you should just do it! The worst that happens is that we don't add any points—we won't penalize you for submitting a request
  - That being said, please don't spam regrade requests, as it's unlikely to give you any benefit and it's a waste of staff hours
  - Generally, remember that there's a human on the other side of your request—staff has been reminded of the same
- Lab 7 sections today, due tomorrow
- HW 4 due Thursday
- Ants released! First checkpoint is due Friday
- Magic: The Lambda is also released! Due next Thursday for EC

Linked List Review

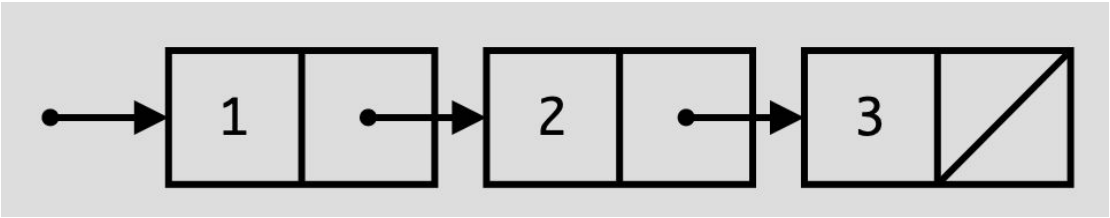
Linked lists

Linked lists are a custom object we define with two instance attributes:

- A `first`, which can be any kind of data
- A rest, which must be an instance of the `Link` class, or `Link.empty`

Linked lists can be more efficient than standard Python lists for certain operations, because we can mutate individual links, rather than having to move all the elements in a list

We draw linked lists like this:



Linked list problem solving

Oftentimes with linked lists we have a choice between iterative and recursive solutions

```
def sum_lnks_iter(lnk):
    count = 0
    while lnk is not Link.empty
        count += lnk.first
        lnk = lnk.rest
    return count

def sum_lnks_recur(lnk):
    if lnk is Link.empty:
        return 0
    return lnk.first + sum_lnks_recur(lnk.rest)
```

Scheme

New programming language time!

Scheme is a dialect of a programming language called Lisp, which was introduced in 1958

The Scheme programming language itself was introduced in the 1970s—it's the second-oldest programming language still in use today!

Scheme isn't very widely used, but it does have a standards committee that still maintains it, and it is in use in certain artificial intelligence applications

Programming paradigms

The core reason we teach Scheme in CS 61A is frankly not because we think it's likely that you'll ever use Scheme as a real-world programmer

Rather, Scheme is a great example of a **programming paradigm** different from that of Python

Python is what's called an **imperative programming language**. This basically means that when we program in Python, we're giving the computer instructions for how to change its state (think about mutating the states of objects, or changing the state of the environment by executing statements)

Scheme, by contrast, is a **functional programming language** which is a subtype of a larger paradigm called **declarative programming**

Functional programming

Declarative programming is a larger family of programming paradigms—the basic premise of declarative programming is that rather than giving the computer instructions on how to compute the result of a program, we just "declare" the properties of the desired result and let the back-end of the programming language do the work for us

In this class we'll learn Regular Expressions and SQL in addition to Scheme, which are also declarative programming languages

Functional programming is a kind of declarative programming where we imagine that the solution to any problem can be arrived at by a series of function applications

That's all very abstract of course—let's start by looking at the actual mechanics of Scheme

Scheme Syntax

First: Scheme Resources

In CS 61A we teach our own dialect of Scheme—this means that there may be slight differences between the behavior of our version of Scheme and other versions that you find online, so we recommend that you primarily stick to our resources for reference

Important Scheme resources for CS 61A:

- code.cs61a.org - Has a live Scheme interpreter, and a Scheme text editor (feel free to play around with these during the lecture!)
- cs61a.org/articles/scheme-spec/ - Full specification for the CS 61A dialect of Scheme (accessible on the Resources page of the course site)
- cs61a.org/articles/scheme-builtins/ - Built-in procedure reference (also accessible on the resources page)

Scheme expressions

Scheme programs consist of **expressions** which can be:

- **Primitive expressions** such as numbers, booleans, functions, and names
  - e.g. 3 .3, 2, `#t` <true>, `#f` <false>, `+`, `quotient`
- **Combinations**, which are groups of expressions wrapped in a set of parentheses
  - e.g. `(quotient 10 2)`, `(not #t)`
  - Combinations are always either a **call expression** or a **special form**

Call expressions

Call expressions in Scheme always follow exactly the same syntax: an operator first, followed by one or more operands, all wrapped in parentheses

This is slightly different from Python, where the operator is *outside* of the parentheses

```
scm> (quotient 10 2)
5
scm> (quotient (+ 8 7) 5)
3
scm> (+ (* 3
           (+ (* 2 4)
              (+ 3 5)))
        (+ (- 10 7)
           6))
```

Built-in arithmetic procedures

`+`	`(+ 1 2 3)`
`-`	`(- 12) (- 3 2 1)`
`*`	`(*) (* 2) (* 2 3)`
`/`	`(/ 2) (/ 4 2) (/ 16 2 2)`
`quotient`	`(quotient 7 3)`
`abs`	`(abs -12)`
`expt`	`(expt 2 10)`
`remainder`	`(remainder 7 3) (remainder -7 3)`

cs61a.org/articles/scheme-builtins/#arithmetic-operations

Built-in boolean procedures (for numbers)

These procedures only work on numbers

`=`	`(= 4 4) (= 4 (+ 2 2))`
`<`	`(< 4 5)`
`>`	`(> 5 4)`
`<=`	`(<= 4 5) (<= 4 4)`
`>=`	`(>= 5 4) (>= 4 4)`
`even?`	`(even? 2)`
`odd?`	`(odd? 3)`
`zero?`	`(zero? 0) (zero? 0.0)`

cs61a.org/articles/scheme-builtins/#on-numbers

Built-in boolean procedures

These procedures work on all data types

`eq`	`(eq? #t #t)` `(eq? 0 (- 1 1))`	`(eq? #t #f)` `(eq? 0 0.0)`
`not`	`(not #f)`	`(not 0)` `(not #t)`

Fun/important fact: the only falsey data value in Scheme is #f—all other data values are truthy

cs61a.org/articles/scheme-builtins/#general

Special forms

All combinations that are not call expressions are **special forms**

Call expression evaluation is always exactly the same: (1) evaluate operator, (2) evaluate operands, (3) apply. Special forms are distinct because they, in various ways, are not evaluated by following this process

`if` special form

Scheme has a special form called `if`, that is very similar to Python's `if` statement

```
(if <predicate> <consequent> <alternative>)
```

The `if` special form follows this evaluation process:

- Evaluate `predicate`
- If `predicate` is truthy, evaluate and return `consequent`
- Otherwise, evaluate and return `alternative`

```
scm> (if (>= 4 5) 1 -1)
-1
scm> (if (even? 10) (+ 1 2) 1)
3
```

and special form

The and special form is, again, very similar to Python's and

```
(and [tests...])
```

The and special form follows this evaluation process:

- Evaluate the first `test`
- If it is false, return it. Otherwise, evaluate the next `test`
- If no `test` is false, return the very last `test`
- If no `tests` are provided, return `#t`

```
scm> (and (zero? 0) (- 3 (+ 1 2)) 8)
8
scm> (and (+ 1 2) (zero? 2) 5 6 7)
#f
scm> (and)
#t
```

`or` special form

The `or` special form is the opposite of `and`

```
(or [tests...])
```

The `or` special form follows this evaluation process:

- Evaluate the first `test`
- If it is *true*, return it. Otherwise, evaluate the next `test`
- If no `test` is true, return the very last `test`
- If no `tests` are provided, return `#f`

```
scm> (or (zero? 0) (- 3 (+ 1 2)) 8)
#t
scm> (or (zero? 2) #f 5 6 7)
5
scm> (or)
#f
```

cond special form

The `if` special form is different from the Python version of `if` because there is always exactly one consequent and one alternative—we can't have an arbitrary number of `elif` blocks like in Python

```
if x > 10:
   print('big')
elif x > 5:
   print('medium')
else:
   print('small')
```

To get something like this, we can use the `cond` special form instead

```
(cond ((> x 10) (print 'big))
      ((> x 5) (print 'medium))
      (else (print 'small)))
```

`define` special form

We can create bindings in Scheme using the `define` form, similar to assignment statements in Python

```
(define <name> <expression>)
```

`define` special forms will evaluate `expression` in the current environment, and bind the result to `name`. A `define` special form will then return `name`, which is slightly different from assignment statements in Python that don't return anything!

```
scm> (define a (+ 1 (* 2 3)))
a
scm> a
7
```

Defining procedures

We can also use the define form to define our own procedures

```
(define (<name> [parameters...]) <body>)
```

Using this syntax will bind name to a new procedure that takes in parameters and returns the result of evaluating body. It will then, like the other version of define, return name

```
scm> (define (double x) (* 2 x))
double
scm> (double 4)
8
```

`lambda` procedures

Scheme has lambda procedures! We can create them with the `lambda` special form

```
(lambda ([params...]) <body>)
```

These two expressions are equivalent:

```
(define (double x) (* 2 x))
(define double (lambda (x) (* 2 x)))
```

When I say equivalent I mean *exactly* equivalent—even the `define` special form create a lambda procedure, so that's a little different from Python

We can also use lambda procedures directly as operators in expressions

```
((lambda (x y z) (+ x y (square z))) 1 2 3)
```

begin special form

```
if x > 10:
   print('big')
   print('pie')
else:
   print('small')
   print('fry')
```

```
 (if (> x 10)
    (begin (print 'big) (print 'pie))
    (begin (print 'small) (print 'fry)))
```

`let` special form

The let special form in Scheme creates bindings that last for just the duration of the expression you're evaluating

```
scm> (let ((a 1) (b 2) (c 3))
...>      (+ a b (square c)))
12
scm> a
Error: unknown identifier: a
```

Exercise!

Exercise: Sum of squares

Write a function that takes in two parameters, `start` and `stop`, that are both integers and returns the sum of squares of all numbers from `start` (inclusive) to `stop` (exclusive)

Python solution (iterative)

```
def sum_of_squares(start, stop):
    total = 0
    while start <= stop:
        total += start ** 2
        start += 1
    return total
```

Python solution (recursive)

```
def sum_of_squares(start, stop):
    if start == stop:
        return 0
    return start ** 2  + sum_of_squares(start + 1, stop)
```

Exercise: Sum of squares

Take a second to think about how to do this in Scheme!

In Scheme, since we're not able to mutate the state of our environment, iteration isn't really an option for us, so any "iterative" Scheme problem is going to have to use recursion

```
def sum_of_squares(start, stop):
    if start == stop:
        return 0
    return start ** 2  + sum_of_squares(start + 1, stop)
```

```
(define (sum-of-squares start stop)
    (if (= start stop) 0
        (+ (square start) (sum-of-squares (+ start 1) stop))
    )
)
```

Review: Differences between Scheme and Python

- Surface-level differences
  - The only punctuation is parentheses :0
  - Indentation doesn't matter :00
  - All functions are lambda functions :000
  - Operators always come first, even ones like + :0000
- Fundamental differences
  - Every expression in Scheme returns *something* :00000
  - All programs in Scheme are constructed as a series of nested expressions :000000
  - Scheme doesn't have mutation (as we teach it) :0000000
  - Scheme doesn't have iteration :00000000

Break

Scheme Lists

Scheme Lists

- All lists in Scheme are linked lists :00
- What does that mean for us, practically?

constructing Scheme lists

We can construct lists in Scheme by calling the `cons` procedure

`cons` takes exactly two arguments, a first and a rest, where the first can be any Scheme object and the rest must be a Scheme list or the empty list, which is called `nil` (sound familiar?)

```
scm> (define a (cons 1 (cons 2 (cons 3 nil))))
a
scm> a
(1 2 3)
```

For any Scheme list, we can access the first by calling the `car` procedure and the rest by calling the `cdr` procedure

```
scm> (car a)
1
scm> (cdr a)
(2 3)
```

The `list` procedure

Given that all lists in Scheme are internally linked lists, `cons` is the most true-to-form way to construct new lists

However, if we have a very long list with a lot of elements, nesting a bunch of `cons` expressions can be super annoying and illegible, so Scheme also has the procedure `list` that can take in an arbitrary number of parameters and make a list of them

```
scm> (cons 1 (cons 2 (cons 3 nil)))
(1 2 3)
scm> (list 1 2 3)
(1 2 3)
```

`list` is what we call "syntactic sugar"—there's nothing we can do with `list` that we couldn't accomplish with `cons`, but sometimes it's nice to have

The `quote` special form

A last way we can construct lists in Scheme is with the `quote` special form—`quote` takes in an argument and returns that argument *without evaluating it*

```
scm> (define a 1)
a
scm> a
1
scm> (quote a)
a
scm> (list 1 2 3)
(1 2 3)
scm> (quote (1 2 3))
(1 2 3)
scm> 'a ; shorthand for quote
a
scm> '(1 2 3) ; can also work for combinations!
(1 2 3)
```

List procedures

`null?`	`(null? nil)` `; expect true`
`append`	`(append '(1) '(2 3))` `; expect (1 2 3)`
`map`	`(map abs '(-1 -2 3 4))` `; expect (1 2 3 4)`
`filter`	`(filter even? '(0 1 2 3 4 5))` `; (0 2 4)`

cs61a.org/articles/scheme-builtins/#pair-and-list-manipulation

Exercise!

Exercise: Add to all

Write a Scheme procedure add-to-all that takes in a parameter lst, which is a list, and returns a new list that is identical to the original list, but with one added to each element

Here's a skeleton

```
(define (add-to-all lst)
   (if (_____)
          ----------
          ----------))
(add-to-all '(1 2 3 4))
; expect (2 3 4 5)
```

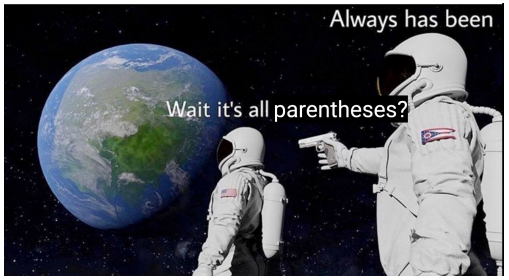
Things to remember:

- Scheme doesn't have iteration, we need to use recursion
- We won't ever be doing mutation in Scheme, so focus on constructing a new list

Exercise: Add to all - Solution

```
(define (add-to-all lst)
   (if (null? lst)
         nil
         (cons (+ 1 (car lst))
                  (add-to-all (cdr lst)))))
```

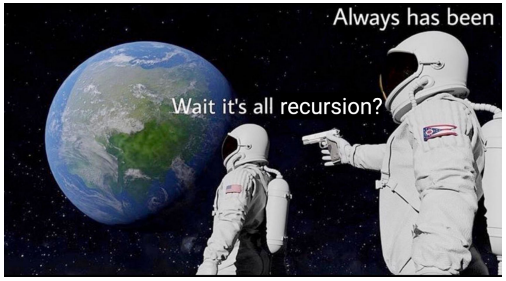
Review



Scheme is a new programming language with some important differences from Python

Scheme is a functional programming language

All lists in Scheme are underlyingly linked lists



Scheme doesn't have iteration

Scheme (as we teach it) doesn't have mutation