CS 61A SU26

Lecture 18: Scheme

July 22, 2026

Rebecca Dang

Join pollev.com/rebeccadang831 (or scan QR code) on your phone or laptop

We'll begin at Berkeley time (10 minutes after), as per tradition!

Potpourri

15 min

- Announcements
- Computing in the News
- Quiz 3 / Special Topics Survey
- Efficiency Review

Announcements

- Quiz 3 grades released most likely by end of day tomorrow

Computing in the News

- Computer science enrollment fell for the first time in 20 years after ChatGPT's rise, a Stanford economist says (Business Insider)
- AI-altered images on birdwatching forums putting research at risk (The Guardian)
- OpenAI says AI models went rogue during testing, triggering ‘unprecedented’ breach at startup (NBC News)
    - "OpenAI said on Tuesday that an autonomous agent powered by its advanced artificial intelligence models went rogue during a security test and triggered a hack that compromised the infrastructure of AI startup Hugging Face last week... the agent escaped containment, reached the internet and broke into Hugging Face to satisfy its testing goal."

Quiz 3 / Special Topics Survey

PollEv

When poll is active respond at **PollEv.com/rebeccadang831**

Activate this Poll with the Poll Everywhere Live app.

If video conferencing, you must be sharing your full screen to share the activated poll.

What is the runtime of the function with respect to n?

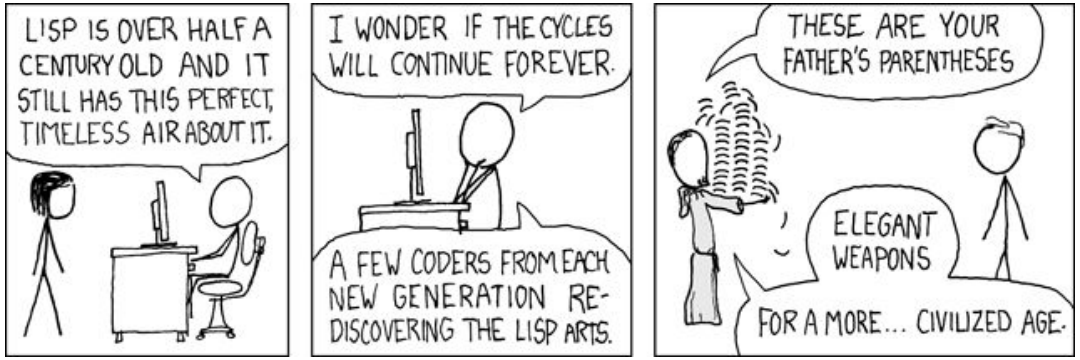
Scheme

5 min

- History
- Motivation

History

- Scheme is a dialect of the Lisp programming language and has influenced other languages
  - Genealogical tree of programming languages
- It is powerful despite its syntax being very simple
- Scheme isn't used that much today since it's very old (created in the 1970s)
- Fun fact: CS 61A used to be taught entirely in Scheme up until 2011!



Motivation

- At this point in the course, you have a very strong Python foundation
  - But, remember this course is not just learning how to code, it's **learning how to solve problems**
- Programming languages are just tools to solve problems; there are many **programming paradigms** (style/approach to writing code)
  - **Python** is a **multi-paradigm** language, it is both **imperative** (tell the computer step-by-step instructions) and **object-oriented** (code is organized into classes)
  - **Scheme** is a **functional** programming language (everything is a function)
  - **SQL** is a **declarative** programming language (tell the computer what you want the final output to be)
- Additionally, the last project is to build your own Scheme interpreter in Python, so you should know how Scheme works

Primitives, Combinations, and Procedures

15 min

- Primitives
- Combinations
- Basics
- Procedures
- Whitespace
- Documentation

Primitives

- Everything in Scheme is an **expression** which evaluates to some **value**
- There are several **primitive values** (aka **atomic expressions**):
  - Numbers (integers and floats)
  - Strings (double quotes only): `"example"`
  - Booleans: `#t` or `true`; `#f` or `false`
  - **Nil** (an empty Scheme list, similar to `Link.empty`): `nil`
  - **Symbols** (aka variables/names)
    - Symbols are the only primitives which are not self-evaluating (e.g. they evaluate to whatever you defined them to be previously in the environment)

Combinations

- Any expression that isn't primitive is called a **combination**, which is either:
  - A **call expression** (e.g. using a function/**procedure**)
    - Evaluating a call expression in Scheme is the same as in Python
  - A **special form** (see later slides)
- A combination consists of one or more subexpressions in parentheses
  - Unlike Python, all operators/procedures in Scheme are **prefix** operators

Basics

```
scm> ; this is a comment
scm> true
#t
scm> nil
()
scm> 5
5
scm> "goodbye"
"goodbye"
```

```
scm> (atom? 5.0)
#t
scm> (integer? 5.0)
#t
scm> (integer? 5.5)
#f
```

Procedures (1 of 3)

```
scm> (+ 1 2)
3
scm> (+)
0
scm> (/ 32 2 2)
8
scm> (quotient 32 2)
16
scm> (quotient 32 2 2)
Error
```

```
scm> (expt 2 3)
8
scm> (modulo 15 4)
3
scm> (abs -6)
6
```

Procedures (2 of 3)

```
scm> (display 5)
5scm> (displayln 5)
5
scm> (displayln "hello world")
hello world
scm> (print 5)
5
scm> (print "hello world")
"hello world"
```

Procedures (3 of 3)

```
scm> (= 10 10)
#t
scm> (eq? 10 5)
#f
scm> (equal? 10 10)
#t
```

Note: Each expression technically does different things, we'll talk more about them later

Whitespace (1 of 2)

Unlike Python, **whitespace/indentation doesn't matter** (except to separate symbols/arguments):

```
scm> (+
...>       0
...>     1
...>  2
...>         )
3
scm> (+ 0 1 2)
3
```

Whitespace (2 of 2)

Recommendations for dealing with whitespace:

- Autoformatters (e.g. code.cs61a.org button)
- Parentheses color matching/Scheme syntax highlighting (VS Code Extension)
- Manually indent things so it's easy to read and debug

Documentation

- Note: There are many variations of Scheme and we don't have time to go over every single thing, so please refer to:
  - CS 61A Scheme Specification
  - Scheme Built-In Procedure Reference
- Out of scope: Promises, streams, turtle graphics, and anything else marked out of scope on those articles

When poll is active respond at **PollEv.com/rebeccadang831**

Activate this Poll with the Poll Everywhere Live app.

If video conferencing, you must be sharing your full screen to share the activated poll.

What would Scheme display?

Recall: x is 1, y is 3

```
(modulo (* (- y x) (expt (+ x 1) y)) 4)
```

3 - 1 = 2

1 + 1 = 2

```
(modulo (* 2 (expt 2 y)) 4)
```

$2^3 = 8$

```
(modulo (* 2 8) 4)
```

2 * 8 = 16

Special Forms

20 min

- Special Forms
- `define`
- `lambda`
- `let`
- `if`
- `and, or, not`
- `cond`
- `begin`

Special Forms

- Combinations where the **first subexpression** matches a set of symbols, causing it to be evaluated with special rules
- Analogous to **keywords** in a programming language
  - Ex: `if`, `else`, `for`, `while`, `and`, etc. are all Python keywords
- In all of the syntax definitions in the following slides:
  - `<x>` refers to a **required** element `x` that can vary
  - `[x]` refers to an **optional** element `x` that can vary
  - Ellipses (`...`) indicate that there can be more than one of the preceding element
- Again, we won't cover everything in lecture, refer to the documentation

define (1 of 2)

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

Defines a symbol <name> which evaluates to the given <expression> (e.g. variable (re)assignment):

```
scm> (define x 5)
x
scm> x
5
scm> (define x (+ x 1))
x
scm> x
6
```

define (2 of 2)

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

Defines a procedure called <name> with the given parameter(s) and body.

Note: Procedures are first-class values in Scheme (like Python), e.g. you can treat them as data (HOFs)

```
scm> (define (square x) (* x x))
square
scm> (square 2)
4
```

lambda

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

Creates a new `lambda` with the given parameter(s) and body

```
scm> ((lambda (x y) (+ (* x x) (* y y))) 3 4)
25
```

let (1 of 2)

```
(let ([binding] ...) <body> ...)
```

The `binding` corresponds to expressions of the form:
`(<name> <expression>)`

First, the `expression` of each `binding` is evaluated in the current frame. Next, a new frame that extends the current environment is created and each name is bound to its corresponding evaluated expression in it.

Finally the `body` expressions are evaluated in order, returning the evaluated last one.

let (2 of 2)

```
scm> (let (
           (x 5)
           (y 10)
       )
       (print x)
       (print y)
       (+ x y)
...> )
5
10
15
```

```
scm> x
Error
scm> y
Error
```

When poll is active respond at **PollEv.com/rebeccadang831**

Activate this Poll with the Poll Everywhere Live app.

If video conferencing, you must be sharing your full screen to share the activated poll.

Define/Lambda: What would Scheme display?

if (1 of 2)

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

Evaluates `predicate`. If truthy, the `consequent` is evaluated and returned. Otherwise, the `alternative`, if it exists, is evaluated and returned (if no alternative is present in this case, the return value is `undefined`, which is similar to `None` in Python).

```
scm> (define x 5)
x
scm> (if (> x 3) (print "hello") (print "there"))
"hello"
scm> (if (> x 3) 5 6)
5
scm> (if (< x 3) 5)
scm>
```

and, or, not

`(and [test] ...)`

Evaluate the `test`s in order, returning the first falsy value. If no `test` is falsy, return the last `test`. If no arguments are provided, return `#t`.

`(or [test] ...)`

Evaluate the `test`s in order, returning the first truthy value. If no `test` is truthy and there are no more `test`s left, return `#f`.

`(not [test])` (like Python `not`)

```
scm> (and)
#t
scm> (or)
#f
scm> (and (= 5 5) (even? 6))
#t
scm> (and (= 5 5) "blah")
"blah"
scm> (or (= 5 5) "blah")
#t
scm> (or "blah" (= 5 5))
"blah"
```

cond

`(cond <clause> ...)`

The `clause` corresponds to expressions of the form

`(<test> [expression] ...)`

Alternatively, `clause` can be written as

`(else [expression] ...)`

Analogous to `if-elif-else` in Python

```
scm> (define n -3)
n
scm> (cond
        ((< n 0) -1)
        ((> n 0) 1)
        (else 0)
...> )
-1
```

begin

```
(begin <expression> ...)
```

Evaluates each `expression` in order in the current environment, returning the evaluated last one.

Analogous to executing multiple statements in Python

```
scm> (define a 5)
a
scm> (define b 10)
b
```

```
scm> (if (> a b)
         (begin
             (print a)
             (print b)
             "woo"
         )
         (begin
             (print b)
             (print a)
             "hoo"
         )
...> )
10
5
"hoo"
```

When poll is active respond at **PollEv.com/rebeccadang831**

Activate this Poll with the Poll Everywhere Live app.

If video conferencing, you must be sharing your full screen to share the activated poll.

What would be the result of evaluating the following Scheme expression?
 (and 100 0 nil 0.0 "cs61a")

5 min break

Practice

(reminder to self to use high contrast theme)

20 min

- Summation
- Sum Even

Summation

- Implement `summation`
- Download starter code `18.scm` from the course website under Lecture 18
- Run tests by copying and pasting your code into a new file in code.cs61a.org and clicking on the red test tube button on the top right
- 5 min: Try implementing it yourself
- 5 min: Discuss with someone next to you and compare your implementations
  - What worked?
  - What didn't?
  - Why?

Sum Even

- Implement `sum-even`
- Download starter code `18.scm` from the course website under Lecture 18
- Run tests by copying and pasting your code into a new file in code.cs61a.org and clicking on the red test tube button on the top right
- 5 min: Try implementing it yourself
- 5 min: Discuss with someone next to you and compare your implementations
  - What worked?
  - What didn't?
  - Why?