CS 61A Discussion 8: **Scheme**

March 23, 2017

Announcements

- Ants is due today. Finish it! Also turn it in!
- HW 6 is due tomorrow. Finish it! Also turn it in!
- HW party today from 6:30-8:30p in 247 Cory.
- Midterm scores are out. Good job everyone!
  (Submit regrades by Sunday 4/2 or forever hold your peace.)

116 Attendance: `cinnamon_twist`
140 Attendance: `vanilla_bark`

Scheme

As long as 61A lives, it lives

(also I don't know who these people are, so don't ask)

First of all, why are we even learning this <*insert choice words here*> language?

- To get experience with different languages and paradigms (**functional programming**!)
- So you guys can write an interpreter for it (project 4 hype)

All right, I'm sold. So how do we learn this <*insert choice words here*> language?

- Test code in Jack's interpreter (scheme.cs61a.org)
- Read Kevin's guide (tiny.cc/kevin_scheme)
- Check out the Scheme reference (tiny.cc/scheme_ref)
- Gaze at infographics (tiny.cc/scheme_illu)
- *Write your own code*! The more you write, the better you will become

Important point for the future

- Everything in Scheme is either a **primitive** or a **combination**.
- Primitives (atomic elements) look like numbers and symbols: `3, '4, …`
- Combinations look like Scheme lists: **`(define x 6)`**
- Note: in a meta sense, **`(define x 6)`** is really just a well-formed list containing the elements **`'define`**, **`'x`**, and **`6`**!

A couple of random notes

…that it won't hurt to remember

- The only false values in Scheme are **`#f`**, **`False`**, and **`false`**. Everything else is true!

- Symbols are immutable strings; think of them like variable names or the code itself. They are case-insensitive.

- To get a symbol, use the quote operator:

```
scm> (define x 6)
x
scm> ‘x ; the symbol x
x
scm> x ; VALUE of the symbol x
6
```

Definitions

- `define` binds a value to a name (just like = in Python). Note that `define` always returns the symbol that has just been assigned a value!
- `(define x 6) ; variable`
- `(define (identity x) x) ; procedure`

Call Expressions

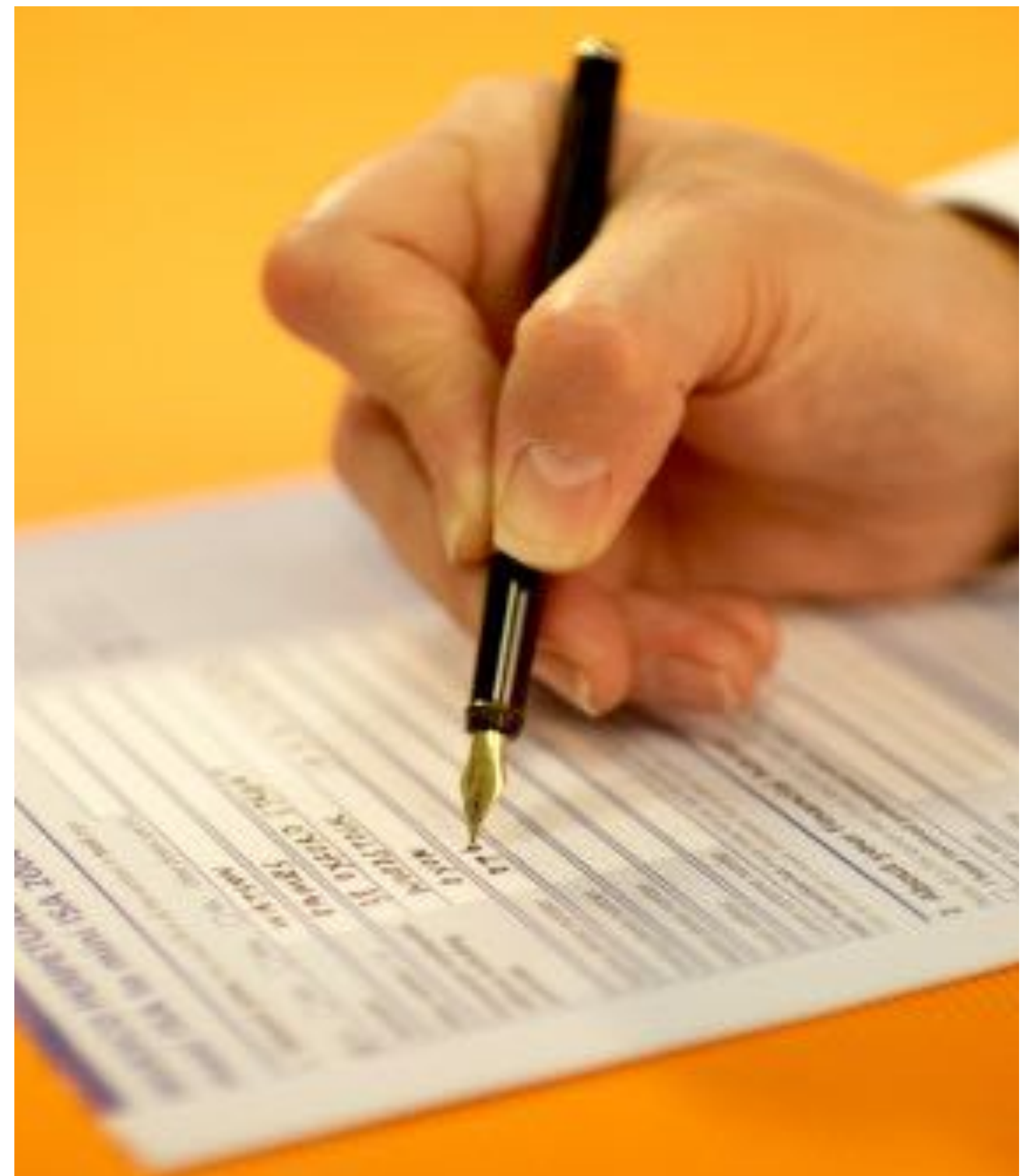
- `(<procedure> <arguments>)`
- **Evaluation** (same as Python):
  - Evaluate the operator, then evaluate each of the operands.
  - Apply the operator to the evaluated operands.

Testing for equality

- `=` is used for numbers (and numbers only!)
- **`equal?`** is used for everything else (…although it actually works for numbers as well)

Special Forms

- `define, if, and, or, not, lambda, let`
- Special forms look like function calls (**because they're surrounded by parentheses**), but don't follow the same execution process



Lambdas

- `(lambda (<params>) <expression>)`

- Much like Python, lambdas are **first-class function values** and you create new frames when you call them.

- Much like Python, `<expression>` isn't evaluated until the lambda is called.

WWSP?

```
((lambda (x) (x x)) (lambda (y) 4))
```

WWSP?

```
((lambda (x) (x x)) (lambda (y) 4))
```

```
Answer: 4
```

Let

```
(let ((<symbol-1> <value-1>)
      ...
      (<symbol-n> <value-n>))
      <body>)
```

is equivalent to

```
((lambda (<symbol-1> … <symbol-n>)
         <body>)
 <value-1> … <value-n>)
```

It’s basically saying “assign *these* variables to *these*
values, and then execute this code with those assignments
in effect.”

Pairs

- **`(cons <elt1> <elt2>)`** creates a pair containing the elements **`<elt1>`** and **`<elt2>`**.
- **`(car pair)`** selects the first element of a pair.
- **`(cdr pair)`** selects the second element of a pair.
- **`nil, (), ‘()`** are equivalent and represent the empty list.

Well-formed lists

- A **well-formed list** is a sequence of pairs where the second element of each pair is ALWAYS either another pair or `nil`.

- **Malformed list**: a sequence of pairs where the second element of ANY of those pairs is something other than another pair or `nil`.

Connecting the dots

- The dot delimits the **first** and **second** element of a pair.
- Note that well-formed lists do not have dots in their final interpreter output.
- Rules for displaying a pair in interpreter format:
  - Use a dot to separate the first and second elements of a pair.
  - If the second element is also a pair (i.e. the dot is immediately followed by an open parenthesis), then remove the dot and the next set of parentheses.

- In this way, `(cons 1 (cons 1 2))` becomes `(1 . (1 . 2))` and finally `(1 1 . 2)` when we break it down into the interpreter’s final output.

The `list` operator

- `(list <args>)` takes zero or more arguments and returns a **well-formed list** of its arguments (i.e. each argument is in the `car` field of its respective pair).

- `(list <arg1> <arg2>) → (<arg1> <arg2>)`

- **Quoting** does the same thing... but expressions that are not self-evaluating (i.e. variables or procedure calls) will not be evaluated.

The difference between `list` and ‘

```
scm> (define a 1)
a
scm> (define b 2)
b
scm> (list a b)
(1 2)
scm> ‘(a b)
(a b)
```

List examples

```
scm> (equal? ‘(1 2) (list 1 2))
true
scm> ‘(1 . (2 3))
(1 2 3)
scm> ‘(define (square x) (* x x))
(define (square x) (* x x))
```

append

A useful procedure for **concatenating lists** that never seems to officially get covered. Takes in zero or more **lists** (not list **elements**!), and returns a single well-formed list containing all the elements of the input lists, in order.

```
scm> (append <lst1> <lst2> …)
(<lst1 elements> <lst2 elements> …)
```

If you pass in no arguments, it returns `nil`. It also has robust behavior for random `nil`s as arguments:

```
scm> (append nil '(1 (2)) nil '(3) nil nil '(5))
(1 (2) 3 5)
```

thanks for coming everyone!

:)