CS 61A
DISCUSSION 7

October 27, 2016

TOPICS FOR TODAY

- Scheme
- Yeah that’s about it

PRELIMINARY NOTES

Announcements

- P/NP deadline is tomorrow. Don’t P/NP if you want to major in CS
- HW9 deadline extended to Monday

Midterm 2 Recap

It had a lot of tricky parts (and 7b was just tricky as a whole). Any questions or comments?

Learning Scheme: "*What happens if…?*"

- TEST IT IN THE INTERPRETER
- It's easy now! (scheme.cs61a.org)

To draw box-and-pointer diagrams, you can use `(demo 'autopair)` in the above interpreter

Learning Scheme: "*What happens if...?*" (cont.)

Scheme reference:
http://inst.eecs.berkeley.edu/~cs61a/su16/proj/scheme/scheme-spec.html

Built-in procedure reference:

http://inst.eecs.berkeley.edu/~cs61a/su16/proj/scheme/scheme-primitives.html

“Cute” Scheme Infographics

- List constructors
- Functions as data
- Common list errors

SCHEME

“Why are we learning Scheme?”

- To get experience with **different programming languages** (functional programming!)
- Similar to Python: **dynamically typed**, **strict evaluation order**, **first-class functions**
- Mainly (IMO) because **we can write an interpreter for it**
- Scheme: “the world’s most unportable programming language” haha (…until CS 61A Project 4 becomes a standard?)

(Disclaimer for people trying to sue me)

A large portion of the following slide content has been borrowed from the unprinted portion of this week's discussion packet.

Everything in Scheme

Everything in Scheme is either a **primitive** or a **combination**!

Combinations are formatted as Scheme lists…!

Take the combination `(define a 4)`…

For the purposes of program interpretation, this is just a well-formed list containing the elements `‘define`, `‘a`, and 4!

Primitives

- `2`, `2.1`, `#t`, ...
- <u>the only false value in Scheme</u> is `#f` (or, equivalently, `False` / `false`). **Everything else** is true!

Primitives are **self-evaluating**…! They're automatically evaluated, and they evaluate to themselves. Also, `'<primitive>` is equivalent to `<primitive>`.

Symbols

- **Symbols** are immutable strings (where there can only be one copy of any given symbol).
- Think of them like variable names; *think of them like the code itself*. That's how we'll use them in our study of interpreters.

It's less complicated than it might seem at first!

Symbols, cont.

- “*There can only be one copy of any given symbol*”
- This applies very easily to variable names if you’re perceiving symbols that way. (*Think of what happens if you assign to a variable name twice! There can’t be more than one variable with the same name in memory.*)

Symbols are case-insensitive and composed of the following characters:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789!$%&*/:<=>?@^_~+-.
```

Symbols, cont.

To get a symbol object, use the **quote** operator:

```
scm> (define x 4)
x
scm> ‘x; the symbol x
x
scm> x ; VALUE of the symbol x
4
```

Symbol usage

Symbols are mainly used as **keys in data structures**; you could keep a dictionary from symbols to values if you were using symbols as variable names :)

But this is interpreter stuff.

You don't really have to worry about this yet...

So as a tl;dr: generally, you don't really have to think about manipulating symbols for normal 61A implementations (i.e. non-interpreter stuff).

Defining variables and procedures

(**Procedure**: the word for a function in Scheme)

`define` binds a **value** to a **name** (just like `=` in Python). Note that `define` always returns a symbol of the name that has just had a value assigned to it!

```
(define a 4) ; variable
(define (identity x) x) ; procedure
```

Definition Syntax

```
(define <var name> <value>)
(define (<fn name> <params>) <body>)
```

More examples:

```
scm> (define a 3)
a
scm> (define (foo x) (+ x 2))
foo
scm> (foo a)
5
```

p1, Q1

Call Expressions

```
(<procedure> <arguments>)
```

Evaluation (same as in Python):

- Evaluate the operator, then evaluate each of the operands
- Apply the operator to the evaluated operands

Important built-in functions

`+, -, *, /`

`equal?, =, >, >=, <, <=`

Testing for equality

`=` is used for numbers (and numbers only!)

`equal?` is used for everything else (...although it actually works for numbers as well)

p2, Q2

Special Forms

Expressions that look like function calls (because they're surrounded by **parentheses**!), but have special functionality in that they don't follow normal order of evaluation

`define, if, and, or, not, lambda, let`

Basic special form usage

`if` syntax: `(if <condition> <true-result> <false-result>)`

```
scm> (if (> 4 5) (/ 1 0) 42)
42

scm> (or #t (/ 1 0))
true
scm> (and False (/ 1 0))
false
scm> (and 1 2 3)
3
```

p2, Q3

Lambdas and defining functions

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

Notes:

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

- `<expression>` isn’t evaluated until the lambda is called.

`(define (<fn> <params>) <body>)` is automatically translated to `(define <fn> (lambda (<params>) <body>))`.

Lambda examples

```
scm> (define square (lambda (x) (* x x)))
square
scm> (square 4)
16
scm> ((lambda (x y) (+ x y)) 6 7)
13
```

Quick Quiz (WWSP)

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

Quick Quiz (WWSP)

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

Quick Quiz #2 (WWSP)

```
((lambda (x) (x x)) (lambda (y) (if (equal? y 4) y (x 4))))
```

Quick Quiz #2 (WWSP)

```
((lambda (x) (x x)) (lambda (y) (if (equal? y 4) y (x 4))))
```

Error (unknown identifier: x)… but it doesn't have to be when we get to mu procedures

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" p2, Q4-5

Scheme lists

They're like linked lists in Python (they're made up of "pairs").

Scheme lists	**Python** linked lists
`cons`	`Link(...)`
`car`	`.first`
`cdr`	`.rest`
`'(), nil`	`Link.empty`

Pairs intro

`(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**.

Suggestion: with pairs it sometimes helps to draw them out as box-and-pointer diagrams (esp. for car/cdr predictions).

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`.

The dot

The dot delimits the **first** and **second** element of a pair. Since we’re talking pairs, you’ll never have multiple elements after a dot!

Well-formed lists, cont.

```
scm> (cons 2 3)
(2 . 3)
scm> (cons 2 (cons 3 nil))
(2 3)
scm> (cdr (cons 2 3))
3
scm> (cdr (cons 2 (cons 3 nil)))
(3)
```

^difference between **well-formed** and **malformed** lists:
well-formed lists don’t have dots in final interpreter output

Well-formed lists, cont.

Rule for displaying a pair in the interpreter:

- 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 parenthesis pair**.

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.

List operators

```
(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 very 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)
```

p3-4, Q1-4

If time:
p5–6, Q1–3

CLOSING STUFF

Discussion Quiz 7

```
5 minutes; get as far as you can
(maybe make 2b a WWSP)
```

Quiz solutions

Q1a. Parentheses either denote procedure calls or special forms. Importantly, note that unlike in the case of Python every set of parentheses counts: you can never leave them out and you can never add more.

In Python, you can do this [if you hate yourself]:

```
(((3))) + ((((4)))) # evaluates to 7
```

Scheme won't let that fly.

[If you try to run (+ (((3))) ((((4))))), it will think that (3) is a function call and immediately error.]

Quiz solutions (Q1 cont.)

Q1c. (See earlier slides.) A symbol is like a variable name. It’s like the code itself. Symbols will come in handy when we deal with interpreters. For now you don’t really need to worry about them, though.

Quiz solutions (Q2)

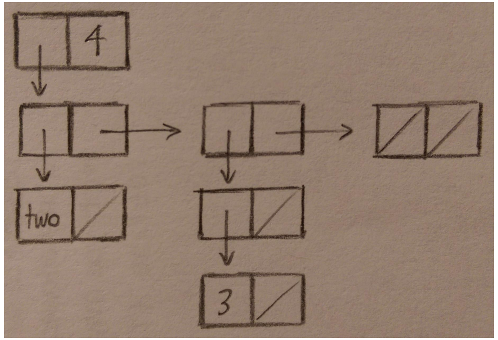
```
Q2a. ‘((list 2 3)) → ((list 2 3))
Q2b. (list ‘(2 3)) → ((2 3))
Q2c. (x 3 4) → Error: cannot call: 0
Q2d. (y 3 4) → 7
```

Quiz solutions (Q3)

(I’ll LaTeX these up nicely later. For now...)

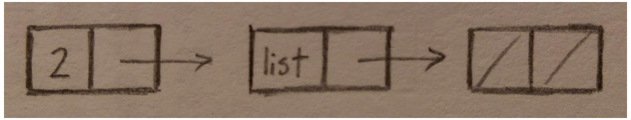
Q3a. ‘(2 . 3 4) → Error; you can only have a single element after a dot

Q3b. (cons (list '(two) '((3)) nil) 4)

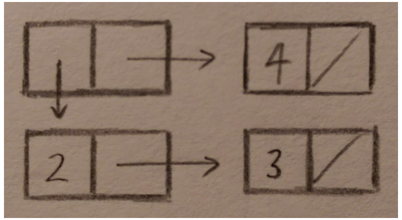


Quiz solutions (Q3 cont.)

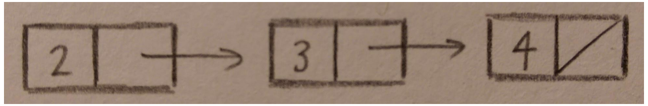
`Q3c. (cons 2 '(list nil))`



`Q3d. (list (append '(2) '(3) nil) 4)`



`Q3e. '(2 . (3 . (4)))`



Quiz solutions (Q4)

```
(define (cadr lst) (car (cdr lst)))
(define (cddr lst) (cdr (cdr lst)))
(define (finish-sort lst)
  (cond ((or (null? lst) (null? (cdr lst))) lst)
        ((> (car lst) (cadr lst)) (append (list (cadr lst))
                                          (list (car lst))
                                          (cddr lst)))
        (else (let ((rest (finish-sort (cdr lst))))
                   (if (< (car lst) (car rest))
                       (cons (car lst) rest)
                       (append (list (car rest))
                               (list (car lst))
                               (cdr rest)))))
  )
)
```

ATTENDANCE

- tiny.cc/ilovecs
- Comes with a survey; 2/3rds-semester feedback? :)

CLOSING REMARK

Enjoy the rest of your Thursday!