CS 61A DISCUSSION 7

SCHEME

Raymond Chan
Discussion 134
UC Berkeley Fall 16

AGENDA

- Announcements
- Scheme
  - Primitives
  - Call Expressions
  - Special Forms
  - Pairs and Lists

ANNOUNCEMENTS

- Midterm 2
- Homework 9 extended to Monday 10/31
- Map composition revision Sunday 11/06

SCHEME

- It's a "clean", functional programming language. (Dialect of Lisp)
  - http://scheme.cs61a.org/
- 4 main points:
  - **Everything is an expression.**
  - **All functions are hidden lambdas.**
  - **Everything is a symbol unless evaluated.**
  - **Non symbols are values (no objects).**

PRIMITIVES

- Atomic primitive expressions cannot be divided up and evaluate to themselves.
- Numbers and booleans.
- The only false-y value in scheme is false (#f, False).
- Use **nil** instead of **None.**

PRIMITIVES

- Atomic primitive expressions cannot be divided up and evaluate to themselves.
- Numbers and booleans.
- The only false-y value in scheme is false (#f, False).
- Use **nil** instead of **None.**

```
scm> 123
123
scm> 123.123
123.123
```

```
scm> #t
True
scm> #f
False
```

```
scm> nil
scm> ()
```

VARIABLES & PROCEDURES

- **define** is a special form that defines **symbols** and **procedures** (functions).
- The equivalent of both assignment and def statements in Python. (no `a = 3` in Scheme)
- **Define** binds a value to a symbol.
- When a symbol / function is defined, returns the symbol.
    - In the function cause, the symbol is the procedure name.
    - The symbol has a value of a procedure.

VARIABLES & PROCEDURES

- (**define** <variable name> <value>)
- (**define** (<function name> <parameters>) <function body>)
- <parameters> are split up by at least one space.

VARIABLES & PROCEDURES

- (**define** <variable name> <value>)
- (**define** (<function name> <parameters>) <function body>)
- <parameters> are split up by at least one space.

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

VARIABLES & PROCEDURES

- (**define** <variable name> <value>)
- (**define** (<function name> <parameters>) <function body>)
- <parameters> are split up by at least one space.

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

```
scm> (define (bar x y) (* x y))
bar
scm> (bar 4 5)
20
```

SYMBOLS

- Any expression that is quoted is not evaluated. (Use single quote)
- They become symbols.
- Below, a is bound to the symbol of b

WWSP

scm> (**define** a 1)

scm> a

scm> (**define** b a)

scm > b

scm> (**define** c 'a)

scm> c

WWSP

```
scm> (define a 1)
a
scm> a

scm> (define b a)

scm > b

scm> (define c 'a)

scm> c
```

WWSP

```
scm> (define a 1)
a
scm> a
1
scm> (define b a)

scm > b

scm> (define c 'a)

scm> c
```

WWSP

scm> (**define** a 1)

a

scm> a

1

scm> (**define** b a)

b

scm > b

scm> (**define** c 'a)

scm> c

WWSP

```
scm> (define a 1)
a
scm> a
1
scm> (define b a)
b
scm > b
1
scm> (define c 'a)

scm> c
```

When we define b, we eval a to be 1.
Thus symbol b has value of 1.

WWSP

```
scm> (define a 1)
a
scm> a
1
scm> (define b a)
b
scm > b
1
scm> (define c 'a)
c
scm> c
```

WWSP

```
scm> (define a 1)
a
scm> a
1
scm> (define b a)
b
scm > b
1
scm> (define c 'a)
c
scm> c
a
```

Evaluate ‘a as symbol a.
c is has value symbol a.

CALL EXPRESSIONS

- Use prefix notation.
- Call expressions starts off with an **operator** that is followed by zero or more **operand** subexpressions.
- Procedures (function) are called with parenthesis.
    - (<operator> <operand1> <operand2> …)
    - Open parenthesis “(” always starts a function call.
    - Spaces matter.

CALL EXPRESSIONS

- (<operator> <operand1> <operand2> …)
- Operators can be symbols (+, *, …) or more complex expressions.
- Operators need to evaluate to procedure values.
- The first expression after "(" is the operator.
- Evaluate the operator and then each of the operands.
- Apply the operator to those evaluated operands.

CALL EXPRESSIONS

```
scm> (- 1 1)                    ; 1 - 1
0
scm> (/ 8 4 2)                  ; 8 / 4 / 2
1
scm> (* (+ 1 2) (+ 1 2))        ; (1 + 2) * (1 + 2)
9
```

CALL EXPRESSIONS

- Built-in functions:
- +, -, *, /
- >, <, >=, <=
- =   Checks for number equality
- eq?   Checks equality for everything else
- null? Checks if the expression is nil

WWSP

```
scm> (+ 1)

scm> (* 3)

scm> (+ (* 3 3) (* 4 4))

scm> (define a (define b 3))

scm> a

scm> b
```

WWSP

```
scm> (+ 1)
1
scm> (* 3)

scm> (+ (* 3 3) (* 4 4))

scm> (define a (define b 3))

scm> a

scm> b
```

Default start value for + is 0

WWSP

```
scm> (+ 1)
1
scm> (* 3)
3
scm> (+ (* 3 3) (* 4 4))

scm> (define a (define b 3))

scm> a

scm> b
```

Default start value for + is 1

WWSP

```
scm> (+ 1)
1
scm> (* 3)
3
scm> (+ (* 3 3) (* 4 4))
25
scm> (define a (define b 3))

scm> a

scm> b
```

WWSP

```
scm> (+ 1)
1
scm> (* 3)
3
scm> (+ (* 3 3) (* 4 4))
25
scm> (define a (define b 3))
a
scm> a

scm> b
```

WWSP

```
scm> (+ 1)
1
scm> (* 3)
3
scm> (+ (* 3 3) (* 4 4))
25
scm> (define a (define b 3))
a
scm> a
b
scm> b
```

(define b 3) returns symbol b
a defined to have value symbol b

WWSP

```
scm> (+ 1)
1
scm> (* 3)
3
scm> (+ (* 3 3) (* 4 4))
25
scm> (define a (define b 3))
a
scm> a
b
scm> b
3
```

SPECIAL FORMS

IF STATEMENTS

- Expressions that look like function calls but don't follow the rules of evolution are called special forms (ex. **define**).
- (**if** <condition> <then> <**else**>)
  - Only #f is false-y. Everything else is truth-y.
  - To replicate Python's if, elif, else, we need to nest **if** expressions.

```
scm> (if (< 4 5) 1 2)
1
```

```
scm> (if #f (/ 1 0) 42)
42
```

SPECIAL FORMS

IF STATEMENTS

- Expressions that look like function calls but don't follow the rules of evaluation are called special forms (ex. **define**).
- (**if** <condition> <then> <**else**>)
  - Only #f is false-y. Everything else is truth-y.
  - To replicate Python's if, elif, else, we need to nest **if** expressions.

```
scm> (if #f (* 1 100)
            (if (= 4 5) 8 10))
10
```

SPECIAL FORMS

BOOLEAN OPERATORS

- **and, or,** and **not** work like the same in Python.
- **and** and **or** are special forms as they short-circuit.

```
scm> (and 1 2 3)
3
scm> (or 1 2 3)
1
scm> (or True (/ 1 0))
True
```

```
scm> (and False (/1 0))
False
scm> (not 3)
False
scm> (not True)
False
```

WWSP

```
scm> (if (or #t (/ 1 0)) 1 (/ 1 0))

scm> (if (> 4 3) (+ 1 2 3 4) (+ 3 4 (* 3 2)))

scm> ((if (< 4 3) + -) 4 100)

scm> (if 0 1 2)
```

WWSP

```
scm> (if (or #t (/ 1 0)) 1 (/ 1 0))
1
scm> (if (> 4 3) (+ 1 2 3 4) (+ 3 4 (* 3 2)))

scm> ((if (< 4 3) + -) 4 100)

scm> (if 0 1 2)
```

WWSP

```
scm> (if (or #t (/ 1 0)) 1 (/ 1 0))
1
scm> (if (> 4 3) (+ 1 2 3 4) (+ 3 4 (* 3 2)))
10
scm> ((if (< 4 3) + -) 4 100)

scm> (if 0 1 2)
```

WWSP

```
scm> (if (or #t (/ 1 0)) 1 (/ 1 0))
1
scm> (if (> 4 3) (+ 1 2 3 4) (+ 3 4 (* 3 2)))
10
scm> ((if (< 4 3) + -) 4 100)
-96
scm> (if 0 1 2)
```

Can return symbols.
Evaluate the returned symbols to be procedures.

WWSP

```
scm> (if (or #t (/ 1 0)) 1 (/ 1 0))
1
scm> (if (> 4 3) (+ 1 2 3 4) (+ 3 4 (* 3 2)))
10
scm> ((if (< 4 3) + -) 4 100)
-96
scm> (if 0 1 2)
1
```

SPECIAL FORMS

LAMBDAS & DEFINE

- All functions are secretly lambda expressions.
- When a lambda expression is called, a new frame is created.
- (**lambda** (<parameters>) <expr>)
- To call the lambda procedure:
- ((**lambda** (<parameters>) <expr>) <arguments>)

SPECIAL FORMS

LAMBDAS & DEFINE

- (**define** (<func name> <parameters>) <expr>)
- Can be translated as.
- (**define** <func name> (lambda (<parameters>) <expr>)
- This is why procedure name is returned for **define**

SPECIAL FORMS

LAMBDAS & DEFINE

```
scm> (define x 3)
x
scm> (define y 4)
y
scm> ((lambda (x y) (+ x y)) 6 7)
13
```

SPECIAL FORMS

LAMBDAS & DEFINE

scm> (**define** x 3)
x
scm> (**define** y 4)
y
scm> ((**lambda** (x y) (+ x y)) 6 7)
13

6 and 7 are passed in as arguments and bound to x and y in the lambda's local frame

SPECIAL FORMS

LAMBDAS & DEFINE

```
scm> (define square (lambda (x) (* x x)))
square
scm> (square 4)
16
```

Lambda functions also values.

SPECIAL FORMS

LET

```
(let ( (<symbol_1> <expr_1>)
          ...
       (<symbol_n> <expr_n>) )
       <body> )
```

- Let binds symbol to expressions locally and then evaluates the body.
- Useful if you want to reuse values multiple times.
- Make code more readable. (Composition)

SPECIAL FORMS

LET

```
(let ( (<symbol_1> <expr_1>)
          …
       (<symbol_n> <expr_n>) )
       <body> )
```

```
( (lambda (<symbol_1> … <symbol_n>) <BODY>)
          <expr_n> … <expr_n>)
```

SPECIAL FORMS

LET

```
(let ( (<symbol_1> <expr_1>)
           …
       (<symbol_n> <expr_n>) )
       <body> )
```

```
(define (sin x)
    (if (< x 0.000001)
        x
        (let ( (recursive-step (sin (/ x 3))) )
              (- (* 3 recursive-step)
                 (* 4 (expt recursive-step 3))))))
```

SPECIAL FORM

COND

```
(cond (<p_1> <e_1>)
      (<p_2> <e_2>)
           ...
      (<p_n> <e_n>)
      (else <else-expr>))
```

- Nested **if** statements are complicated and hard to read.
- The **cond** forms checks each predicate expression pair.
- If the predicate is true, we evaluate the corresponding expression. Otherwise we continue to check the next pair.
- The else expression is evaluated if no predicate is true.

SPECIAL FORM

BEGIN

- Begin is a special form that takes in subexpressions.
- It evaluates all subexpressions in order.
- The value of a begin form is the value from evaluating the last subexpressions.

scm> (**begin** (factorial 4) (square 5))
25
scm> (**begin** (/ 1 0) (factorial 4))
Error

PAIRS & LISTS

- The only data structure in scheme is list.
- Caveat: They are linked lists!
- We call each “link” a pair with a first value and a rest value.

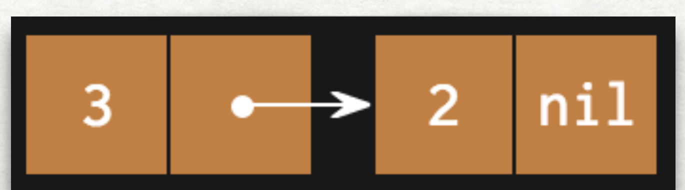
PAIRS & LISTS

- Constructor: (cons 2 nil) -> (2)
  - nil is an empty list.

- Obtain first element: (car (cons 2 nil)) -> 2
- Obtain second element: (cdr (cons 2 (cons 3 nil)) -> (3)
  - The second element is a list!

PAIRS & LISTS

```
scm> nil
()
scm> (null? nil)
#t
scm> (cons 2 nil)
(2)
scm> (cons 3 (cons 2 nil))
(3 2)
```




```
scm> (define a (cons 3 (cons 2 nil)))
a
scm> (car a)
3
scm> (cdr a)
(2)
scm> (car (cdr a))
2
scm> (define (len a)
        (if (null? a)
            0
            (+ 1 (len (cdr a)))))
len
scm> (len a)
2
```

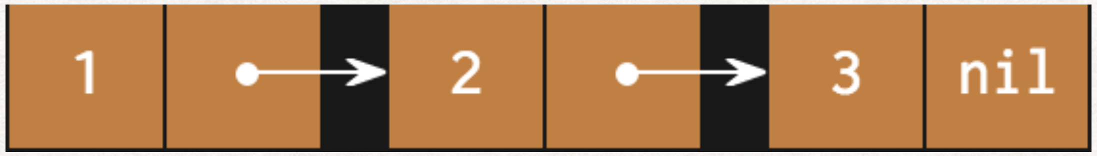
PAIRS & LISTS

- Well formed lists are those where the second element is nil or another linked list.

PAIRS & LISTS

- Well formed lists are those where the second element is nil or another linked list.

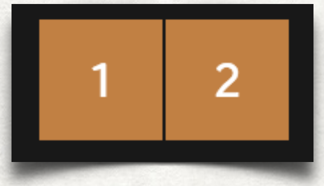
```
scm> (cons 1 (cons 2 (cons 3 nil)))
(1 2 3)
scm> nil
()
```



PAIRS & LISTS

- Malformed list occurs when the second element is a value.
- A dot *separates* the first value and the second value.

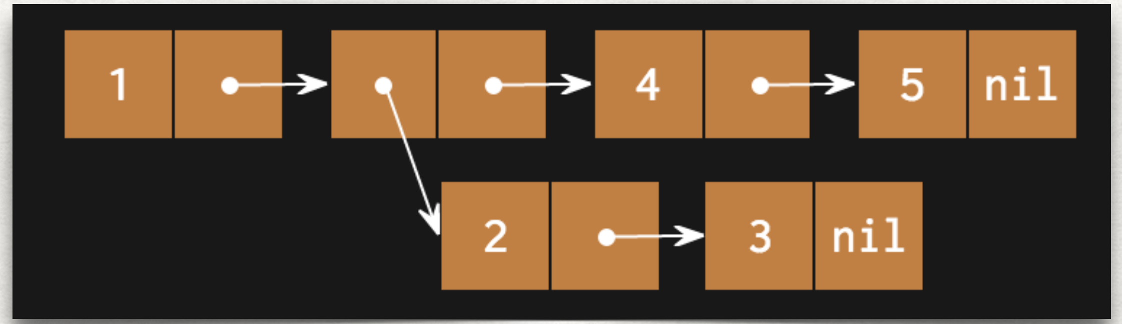
```
scm> (cons 1 2)
(1 . 2)
```



PAIRS & LISTS

- Deep list occurs when the first element is another list!

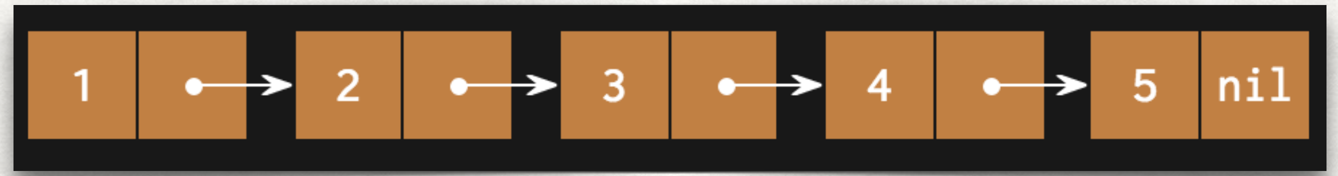
```
scm> (define lst (cons 1 (cons (cons 2 (cons 3 nil)) (cons 4 (cons 5 nil))))
(1 (2 3) 4 5)
scm> (car lst)
(2 3)
scm> (car (cdr (cdr lst)))
4
scm> (car (cdr (car (cdr lst))))
3
```



PAIRS & LISTS

- We can also construct well-formed lists with the **list** operator.

```
scm> (list 1 2 3 4 5)
(1 2 3 4 5)
```

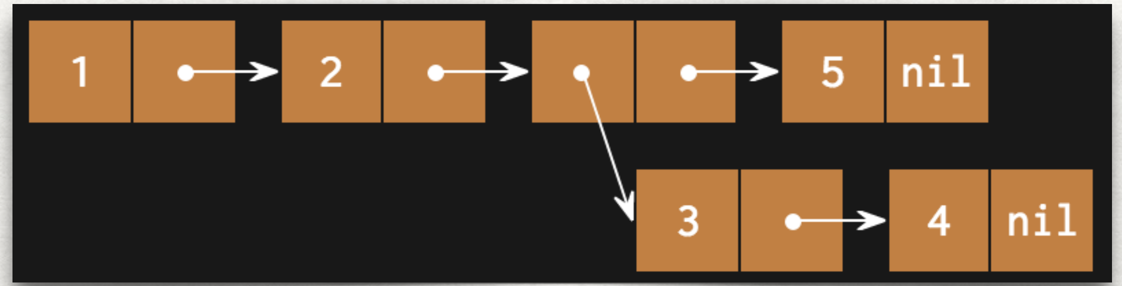


PAIRS & LISTS

- We can also construct well-formed lists with the **list** operator.

```
scm> (list 1 2 3 4 5)
(1 2 3 4 5)
```


```
scm> (list 1 2 (list 3 4) 5)
(1 2 (3 4) 5)
```



List creates same # of pairs as the # of operands.
Each operand will go into the **first** value of each pair.

PAIRS & LISTS

- Or we can use quote form.

```
scm> ‘(1 2 3 4)
(1 2 3 4)
scm> ‘(3 . (2 1))
(3 2 1)
scm> ‘(define (foo x) x)
(define (foo x ) x)
```

```
scm> '(3 . (2 . (1 . nil)))
(3 2 1)
```

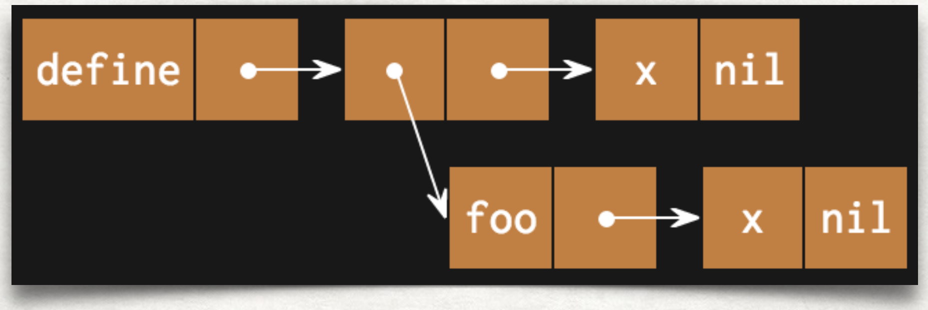
Note: open “(” and closing “)” parenthesis as **symbols** represent lists.

PAIRS & LISTS

- Or we can use quote form.

scm> ‘(1 2 3 4)
(1 2 3 4)
scm> ‘(3 . (2 1))
(1 2 3)
scm> ‘(define (foo x) x)
(define (foo x) x)

scm> '(3 . (2 . (1 . nil)))
(3 2 1)



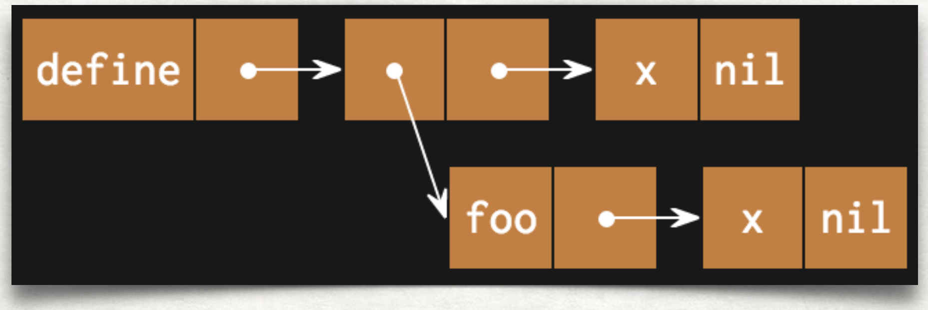
The quote form is propagated through the list.
define and foo are symbols.

Note: open “(“ and closing “)” parenthesis as **symbols** represent lists.

PAIRS & LISTS

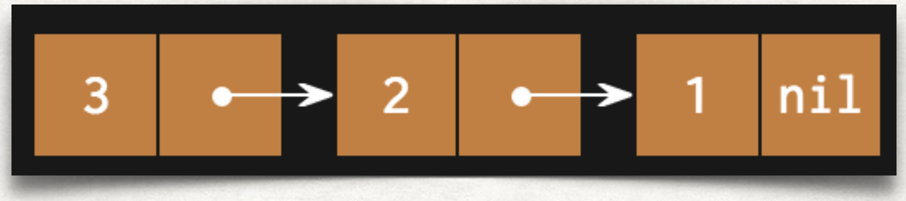
- Or we can use quote form.

scm> ‘(1 2 3 4)
(1 2 3 4)
scm> ‘(3 . (2 1))
(1 2 3)
scm> ‘(define (foo x) x)
(define (foo x) x)



The quote form is propagated through the list.
define and foo are symbols.

scm> '(3 . (2 . (1 . nil)))
(3 2 1)



The expression after the dot is the second element.
Since it is another list, the list is well-formed.

Note: open “(“ and closing “)” parenthesis as **symbols** represent lists.

HINTS

- Scheme has no iteration or objects. Only recursion and functions.
- For list code writing questions, it may seem easier to use iteration sometimes.
- We can turn recursion into iteration by defining a helper function that has an additional parameter **so-far**.
- This parameter is the list we have built thus far in our recursive calls.
- When we reach the base case, we can just return this **so-far** list.

RECAP

- Scheme is a functional programming language.
- We can define variables and procedures with **define**
- Symbols have values that can be obtained if you evaluate the symbols.
- Scheme lists are linked lists.