CS 61A
Discussion 8

Scheme

Raymond Chan
Discussion 121
UC Berkeley

Agenda

- Announcements
- Scheme

Announcements

- Quiz due Friday (check course website)
- Ants Project due tonight
- Lab 8 due tomorrow
- HW 5 due Wednesday 3/28
- Midterm 2 Wednesday 3/30
- Submit Midterm 2 Alternative Petition by tomorrow

Scheme

- Introducing this programming language because it is simple.
  - 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).

- 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).

- Use **nil** instead of **None.**

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

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).

- Use **nil** instead of **None.**

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

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

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).

- Use **nil** instead of **None.** Also can use **()**.

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

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

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

Prefix Notation

- Call expressions starts off with an **operator** that is followed by zero or more **operand** subexpressions.

- Functions (procedures) are called with parenthesis.

- (<operator> <operand1> <operand2> …)

- Open parenthesis “**(**” always starts a function call.

- Spaces matter.

Prefix Notation

- (<operator> <operand1> <operand2> …)
- Operators can be symbols (+, *, …) or more complex expressions.
- Evaluate the operator and then each of the operands.
- Apply the operator to those evaluated operands.

Prefix Notation

- (<operator> <operand1> <operand2> …)
- Operators can be symbols (+, *, …) or more complex expressions.
- Evaluate the operator and then each of the operands.
- Apply the operator to those evaluated operands.

```
scm> (+ 4 5)
9
```

Prefix Notation

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

Variables & Procedures

- **define** is a special form that defines **variables** and **procedures** (functions).

- The equivalent of both assignment and def statements in Python. (no `a = 3` in Scheme)

- Define binds a value to a variable.

- When a variable is defined, **define** returns the **variable name**.

- When a function is defined, **define** returns the **function name**.

Variables & Procedures

- (define <variable name> <value>)
- (define (<function name> <parameters>) <function body>)
- <parameters> are split up by 1 space.

Variables & Procedures

- (define <variable name> <value>)
- (define (<function name> <parameters>) <function body>)
- <parameters> are split up by 1 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 1 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

Symbols

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

```
scm> (define b 3)
b
scm> (define a ‘b)
a
scm> a
b
```

Symbols

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

```
scm> (define b 3)
b
scm> (define a ‘b)
a
scm> a
b
```

```
scm> (define c b)
c
scm> c
3
```

Special Forms

- Expressions that look like function calls but don't follow the rules of evolution are called special forms (ex. define).

- **and**, **or**, and **not** work the same as they would in Python.

- (if <condition> <then> <else>)

    - To replicate Python's if, elif, else, we need to nest **if** expressions.

Special Forms

- Expressions that look like function calls but don't follow the rules of evolution are called special forms (ex. define).

- **and**, **or**, and **not** work the same as they would in Python.

  - (if <condition> <then> <else>)

    - To replicate Python's if, elif, else, we need to nest **if** expressions.

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

Lambdas & Define

- When a lambda expression is called, a new frame is created.
- Lookup for variables occurs in local frame before going to the parent frame.

- ((lambda (<parameters>) <body>) <arguments>)
- (define (<func name> <parameters>) <expr>)
- (define <func name> (lambda (<parameters>) <expr>)

Lambdas & Define

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

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

Lambdas & Define

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

Let

(let ((<symbol1> <expr1>)
…
(<symboln> <exprn>))
<body>)

- Let binds symbol to expressions locally and then runs the body.
- Useful if you want to reuse a value multiple times.

Cond

```
(cond (<p1> <e1>)
      (<p2> <e2>)
       …
      (<pn> <en>)
      (else <else-expr>))
```

- Nested if statements are annoying.
- 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.

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 of evaluating the last subexpressions.

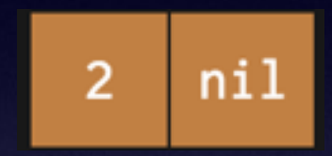
```
scm> (begin (factorial 4) (square 5))
25
scm> (begin (/ 1 0) (factorial 4))
Error
```

Lists

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

Lists

- Constructor: (cons 2 nil) -> (2)
- Obtain first element: (car (cons 2 nil)) -> 2
- Obtain second element: (cdr (cons 2 (cons 3 nil)) -> (3)



Lists

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

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
```

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
()
```

Lists

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

Lists

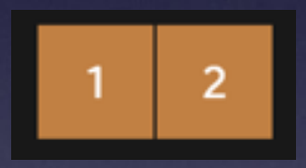
- Malformed list occurs when the second element is a value.
- A dot separates the first value and the second value.

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

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)
```



Lists

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

Lists

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

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

Lists

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

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

- Or we can use the quote form.

Lists

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

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

- Or we can use the quote form.

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

Lists

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

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

- Or we can use the quote form.

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

The dot here means that the second element is another linked list, which makes it well-formed.

Lists

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

Lists

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

scm> (cons 1 (cons (cons 2 (cons 3 nil)) (cons 4 (cons 5 nil))))
(1 (2 3) 4 5)

Lists

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

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

Hints

- For list code writing questions, it may seem easier to use iteration.

- 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.