The current topic: Scheme

- ✓ Introduction
- ✓ Object-oriented programming: Python
- Functional programming: Scheme
  - ✓ Introduction
  - Next up: Numeric operators, REPL, quotes, functions, conditionals
- Types and values
- Syntax and semantics
- Exceptions
- Logic programming: Prolog

Announcements

- Reminder: Term Test 1 is on Monday in **GB405**, *not in the regular lecture room*.
  - 50 minutes (11:10 – 12:00).
  - You're allowed to have one double-sided aid sheet for the test. You must use standard letter-sized (that is, 8.5" x 11") paper. The aid sheet can be produced however you like (typed or handwritten).
  - **Bring your TCard.**
  - What's covered?
    - Everything from the first seven lectures (that is, everything up to and including September 26th).
    - Lab 1.
  - An old Term Test 1 has been posted.
  - The exercises at the end of each lecture are also good practice.
  - Solutions to Lab 1 are available from the directory `~ajuma/share/326/lab1` on ECF.

Review: car, cdr, and cons

- `car` returns the first element of a list:

```
> (car '((a) b (c d)))
(a)
```

- `cdr` returns the rest of a list:

```
> (cdr '((a) b (c d)))
(b (c d))
```

- `cons` adds an element to the front of a list:

```
> (cons '(a b) '((c d)))
((a b) (c d))
```


Numeric operators

- The numeric operators `+, -, *, /` are used just like any function, in parenthesized prefix form.

```
> (+ 5 3)
8

> (- 5 3)
2

> (* 5 3)
15

> (/ 5 3)  ; returns a fraction!
5/3
```

Numeric operators

- More examples:

```
> (* (+ 5 3) (/ 3 2))
12

> (+ 5 6 7 8)
26

> (- 5 3 4 2)
-4

> (/ 16 2 4)
2

> (* (+ 1 2 3) (* (+ 1 2) (- 4 2)) 2)
72
```


Number comparisons

- Number comparisons are performed using the functions `=`, `<`, `>`, `<=`, and `>=`.

```
> (= 4 5)
#f

> (>= 4 4)
#t

> (= (+ 2 1) (- 4 1) (/ 6 2))
#t

> (< 5 6 7)  ; checks if 5 < 6 < 7
#t
```


Type-checking functions

- The functions `number?`, `symbol?`, and `list?` check the type of the given argument and return boolean values (`#t` or `#f`).

```
> (number? 5)
#t
> (number? 'sam)
#f

> (symbol? 'sam)
#t
> (symbol? 5)
#f

> (list? '(a b))
#t
> (list? (+ 3 4))
#f
> (list? '(+ 3 4))
#t
> (list? 7)
#f
```


Other useful functions

- The function `zero?` returns true iff given the number 0.

```
> (zero? 0)
#t

> (zero? (- 3 3))
#t

> (zero? (* 3 1))
#f

> (zero? '(- 3 3))
zero?: expects argument of type <number>; given (- 3 3)
```

Other useful functions

- The function `null?` returns true iff given an empty list.

```
> (null? '())
#t

> (null? '(a))
#f

> (null? (car '(a)))
#f

> (null? (cdr '(a)))
#t
```


The empty list, or nil

- Some Schemes treat `()` as equivalent to `#f`.
  - mzscheme does not.
- The Scheme standard says you should quote `(): '()`
  - mzscheme is more relaxed.

```
> ()
()

> '()
()
```


Boolean operations: and, or, not

- Like the `and` and `or` operators in Python, the `and` and `or` operators in Scheme are short-circuited: they evaluate only as much as needed.
  - `and` stops at the first false condition.
  - `or` stops at the first true condition.

```
> (and (zero? 0) (number? 2) (eq? 1 1))
#t

> (and (zero? 0) (number? 'x) (eq? 1 2))
#f

> (or (symbol? 'x) (symbol? 3))
#t
```


Boolean operations: and, or, not

- As in Python, the `and` and `or` operators in Scheme return the last thing they evaluate.
- Everything except `#f` is treated as "true".

```
> (or (symbol? 1) "no" "other")
"no"

> (or #f 3 #t)
3

> (and 3 4 "hi" #f "bye")
#f
```

Boolean operations: and, or, not

- The `not` operation always returns `#t` or `#f`.

```
> (not (null? '(1 2)))
#t

> (not 3)
#f
```

- Since Scheme doesn't have a numeric "not equals" operator (like the `!=` operator in C/Java/Python), we have to combine `not` and = in order to evaluate "not equals".

```
> (not (= 3 4))
#t

> (not (= (+ 4 5) (* 3 3)))
#f
```


READ-EVAL-PRINT Loop (REPL)

- The Scheme interpreter runs in a Read-Evaluate-Print Loop (REPL).
- READ: Read input from user.
  - e.g. The user enters a function application
- EVAL: Evaluate input: `(f arg1 arg2 … argn)`.
  1. Evaluate `f` to obtain a function.
  2. Evaluate each `argi` to obtain a value.
  3. Apply function to argument values.
- PRINT: Print resulting value.
  - e.g. Print the result of the function application
- And then READ the next input from the user.

READ-EVAL-PRINT Loop Example

- Let's go through the REPL for the following interaction.

```
> (cons 'a (cons 'b '(c d)))
(a b c d)
```

1. Read the function application:
   `(cons 'a (cons 'b '(c d)))`
2. Evaluate `cons` to obtain a function
   - `cons` evaluates to a built-in function
3. Evaluate `'a` to obtain `a` itself.

READ-EVAL-PRINT Loop Example

4. Evaluate `(cons 'b '(c d))`:
   (a) Evaluate `cons` to obtain a function.
   (b) Evaluate `'b` to obtain `b` itself
   (c) Evaluate `'(c d)` to obtain `(c d)` itself
   (d) Apply the `cons` function to `b` and `(c d)` to obtain
   `(b c d)`
5. Apply the `cons` function to `a` and `(b c d)` to obtain
   `(a b c d)`
6. Print the result of the application:
   `(a b c d)`
7. Display the prompt in order to read the next input from the user.
   `>`

The quote issue

```
(list? (a b))
undefined identifier: a

> (list? (quote (a b)))
#t
> (list? '(a b))
#t

> (hi)
undefined identifier: hi
> (quote hi)
hi
> 'hi
hi
```

- Observe that `'x` means the same thing as `(quote x)`.

The quote issue

- In the context of REPL, `quote` inhibits evaluation.

```
> (cons 'a (cons 'b '(c d)))
(a b c d)

> (cons 'a '(cons 'b '(c d))) ; quote before 2nd arg
(a cons 'b '(c d))

> (cons a (cons 'b '(c d))) ; no quote before 1st arg
reference to undefined identifier: a
```


The quote issue

- Some things evaluate to themselves. For these things, quoting has no effect (and is unnecessary).

```
> (list '1 '42 '#t '#f '())
(1 42 #t #f ())

> (list 1 42 #t #f ())
(1 42 #t #f ())
```

- Note that the `list` function constructs a list that consists of the arguments it's given.

REPL and function definition

- The basics are still the same when you're defining a function, but the interpretation is a little different.

- READ: Read input from user.
  - This time the input is a symbol definition rather than a function application.
- EVAL: Evaluate input.
  - Store the function definition
- PRINT: Print resulting value.
  - The symbol defined, or perhaps nothing.

- mzscheme prints nothing for a function definition.

```
> (define (square x) (* x x))
>
```

Function definition syntax 1

(define (<function-name> <function-parameters>)
  <expression>)

```
> (define (square x)
    (* x x))

> (square 2)
4

> (define (mean x y)
    (/ (+ x y) 2))
> (mean 5 11)
8
> (mean 1 2 3)
procedure mean: expects 2 arguments, given 3: 1 2 3
```


Function definition syntax 2

(define <function-name> <function-value>)

- The "function value" is a created using a *lambda expression*.
  - A lambda expression evaluates to an unnamed function.
  - "evaluates to an unnamed function": just like an arithmetic expression (like `(+ 5 3)`) evaluates to an unnamed number.

```
> (define square
    (lambda (x) (* x x))  )

> (define mean
    (lambda (x y) (/ (+ x y) 2))  )
```

- In the examples above, we use `define` to give a name to the unnamed functions produced by the lambda expressions.

Lambda calculus

- A formal system for defining functions and their properties.
- Equivalent to Turing machines.
  - That is, equivalent to any general computing machine.
- Since a lambda expression evaluates to a function, we can (if we want to) evaluate this function immediately, without ever giving it a name.

```
> ((lambda (x) x) 'a)
a

> ((lambda (x) (* x x)) 2)
4

> ((lambda (x y) (+ x y)) 3 4)
7
```


Conditional Execution: if

(if <condition> <result1> <result2>)

- Semantics:

1. Evaluate <condition>.
2. If result is true (non-`#f`), then evaluate and return <result1>.
3. Otherwise, evaluate and return <result2>.

```
> (define (abs-val x)
   (if (>= x 0) x (- x)))

> (abs-val -4)
4
> (abs-val (- (* 3 4) 4))
8
```

Conditional Execution: cond

(cond (<condition1> <result1>)
(<condition2> <resultt2>) …
(<conditionN> <resultN>)
(else <else-result>) ;optional else clause
)

- Semantics:

1. Evaluate conditions in order until one of them returns a true value.
2. Evaluate and return the corresponding result.
3. If none of the conditions return a true value, evaluate and return <else-result>.
4. If none of the conditions return a true vale and there is no else clause, the result of the `cond` expression is unspecified.

Examples with cond

```
> (define (abs-val x)
     (cond ((>= x 0) x)
           (else (- x)))

> (define (grade n)
     (cond ((>= n 80) 'A)
           ((>= n 70) 'B)
           ((>= n 60) 'C)
           ((>= n 50) 'D)
           (else 'F)))

> (grade 75)
b
> (grade 45)
f
```


if vs cond

- Let's define `atom?`, a function that takes a parameter `x` and returns `#t` if `x` is an atom, and `#f` otherwise. First, we use `cond`:

```
(define (atom? x)
    (cond ((symbol? x) #t)
          ((number? x) #t)
          ((char? x) #t)
          ((string? x) #t)
          ((null? x) #t)
          (else #f)
          )
    )
```


if vs cond

- Now we write `atom?` using `if`:

```
(define (atom? x)
    (if (symbol? x) #t
        (if (number? x) #t
            (if (char? x) #t
                (if (string? x) #t
                    (if (null? x) #t #f
                        )
                    )
                )
            )
        )
    )
```

Better atom? functions

- An atom is anything except a non-empty list.

```
(define (atom? x)
    (if (null? x) #t
          (if (list? x) #f #t
              )
          )
    )

(define (atom? x)
    (cond ((null? x) #t)
          ((list? x) #f)
          (else #t)
          )
    )
```


Even better atom? functions

```
(define (atom? x)
    (if (list? x) (null? x) #t
        )
    )

(define (atom? x)
   (or (null? x) (not (list? x))))
```


Exercises

- Define a function `f` that takes two numbers `x` and `y` as input, and returns a list containing the numbers `x+y`, `x-y`, `x/y`, and `x*y`.
- Define a function `g` that takes two lists `L1` and `L2` as input, and returns a new list formed by adding the first two elements of `L2` to the beginning of `L1`.
- Define a function `everyOtherSum` that takes a list `L` of numbers as input, and returns the sum of every second number in the list, starting with the first number.