The current topic: Scheme

- ✓ Introduction
- ✓ Object-oriented programming: Python
  - ✓ Features, variables, numbers, strings, Booleans, while loops
  - ✓ If statements, sequences, functions, modules
  - ✓ Dictionaries, command-line arguments, files, classes, inheritance, polymorphism
  - ✓ Exceptions, operator overloading, privacy
  - ✓ Multiple inheritance, parameters and arguments, list comprehensions
  - ✓ Regular expressions, doc strings
- Functional programming: Scheme
  - Next up: Introduction
- Types and values
- Syntax and semantics
- Exceptions
- Logic programming: Prolog

Announcements

- Reminder: Lab 1 was due at **10:30am** today.
  - Late penalty is 20% per day.
  - Not accepted after 10:30am on Wednesday.
- Reminder: Term Test 1 is on October 6th in **GB405**, *not in the regular lecture room*.
  - 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.

Basic functional-programming concepts

- Reference: Sebesta, chapter 15.
- *Referential transparency*: The value of a function application is independent of the context in which it occurs.
  - The value of f(a, b, c) depends only on the values of f, a, b and c.
  - It does not depend on the global state of computation.
  - That is, *the point in time* at which the call is made has no effect on the value that is returned.
  - This makes it easier to understand what's happening in a program.
- All variables that appear inside a function must be parameters.
  - No variables can be declared or first introduced within a function body.
  - A function has no access to global variables.

Basic functional-programming concepts

- *No assignment*: The concept of assigning a new value to a variable is not part of functional programming.
  - There are no explicit assignment statements.
  - Variables are bound to values only through the association of arguments to parameters in function calls.
  - Function calls have no side effects.
    - They do not alter the value of their arguments, or make any other changes to the global state.
- Conclusion: There is no need to consider global state.

Control flow

- Control is through function calls and conditional expressions **only**.
  - There are no loops of any kind.
- Recursion is used to achieve repetition.

Functional programming: An example

- Recall how we computed the n-th Fibonacci number in Python:

```
def fib(n):
   # We'll assume n >= 1.
   prev = 1   # first Fibonacci number
   cur = 1     # second Fibonacci number
   for i in range(2, n):
      prev, cur = cur, prev+cur

   return cur
```

- To re-write this function using a functional programming approach, we need to get rid of the assignment statements and the loop.
  - How can we do this?

Functional programming: An example

- The same function, using a functional programming approach (but still in Python):

```
def fib(n):
   if n <= 2:
      return 1
   else:
      return fib(n-1) + fib(n-2)
```

- Observe that the functional version uses an if statement and recursion.

Storage management

- All variables are function parameters, so structures that have a lifetime beyond a function call can't be referred to.
  - A structure returned by a function is never assigned to a variable, so there's no way to explicitly free it.
  - e.g. Suppose (for the sake of this discussion) that in the example on the previous slide, `fib()` returned a list. How would we free the lists returned by `fib(n-1)` and `fib(n-2)`?
- *Garbage collection* is needed to dispose of such structures.

Functions as first-class values

- Recall that *first-class* "objects" are things that a language allows programs to :
  - declare
  - assign values to
  - return as function values
  - return as the value of an expression
  - pass as an argument
  - store in a data structure
  - create without a name
- For example, integers are first-class values in most programming languages.
- In functional programming languages, functions are first-class values.
  - Aside: Python also mostly treats functions as first-class values.
    - Why "mostly"? There are some restrictions on creating unnamed functions.

Functions as "values" in C

- C allows *pointers* to functions, like this:

```
double integrate(double (*f)(double x), double a, double b) {
... sum += (*f)(a + k*step); ...
}
```

- Function pointers are often used in tables to associate an action with an index.
- So, C function *pointers* **can be** stored in a data structure as a value.
- But in C, unnamed functions **cannot** exist as values, though there can be unnamed function *pointers*.
- Functions are **not** first-class values in C.
  - And they're not first-class values in Java either, which doesn't even have function pointers.

History: LISP

- "LISt Processing"
- A functional language developed by John McCarthy in the late 1950s for use in AI.
- Semantics (meaning) based on λ-calculus.
  - The λ-calculus is a theoretical model of computation
- Functions operate on lists or atomic symbols.
  - Programs consist of "S-expressions".
  - five basic functions: cons, car, cdr, equal, atom
  - one conditional construct: cond
- Programs and data are both S-expressions.

Common LISP

- LISP implementations began to stray
  - from the original semantics
  - from each other
- Semantics were redefined in Common LISP.
- Common LISP is now the standard version of LISP.
  - Designed by committee in 1980s.
- Simple syntax, but a large language.

Scheme

- Created in 1975 by Gerald Sussman and Guy Steele.
- A version of LISP
  - Simple syntax, small language
  - Close to the initial semantics of LISP
  - Provides basic list-processing tools
  - Allows functions to be first-class objects
- Not the same as LISP!

The Scheme we'll use

- There are many flavours of Scheme, with quite a bit of variation among them.
  - But not as much variation as among the LISPs.
- We'll use MzScheme.
  - Use the command "mzscheme" to launch the MzScheme interpreter on ECF.
  - Version 202 is installed on ECF. This is not the most recent version.
- DrScheme is an IDE that works with MzScheme.
  - Use the command "drscheme" on ECF.
  - Make sure the selected language is MzScheme.

The Scheme we'll use

- DrScheme can be downloaded from http://download.plt-scheme.org/drscheme/v202.html
  - This download also includes MzScheme.
  - Note that the above link is for **version 202**, the version we're using on ECF.
- Your assignments must run using MzScheme as installed on ECF.

Scheme references

- Sebesta, Section 15.5.
- Dybvig, *The Scheme Programming Language*, 2003.
  - This textbook can be read online at: http://www.scheme.com/tspl3/
- "Teach Yourself Scheme in Fixnum Days", a tutorial at: http://www.ccs.neu.edu/home/dorai/t-y-scheme/t-y-scheme.html

A Scheme program

- A program consists of
  - a set of function definitions
  - an expression to be evaluated
- A simple example that defines function `abs-val` and then calls this function (note that ">" is the interpreter's prompt):

```
> (define (abs-val x)
    (if (>= x 0)
        x
        (- x)))
> (abs-val (- 3 5))
2
```


The basic structure of a Scheme (or LISP) program

- A program consists of *S-expressions* ("symbolic expressions") written in parenthesized *prefix form*.
  - "Prefix form": The name of the function appears before the arguments to the function, even for mathematical functions like +, –, *, and /.
  - The general form of an S-expression in prefix-form is:

    '(' <function> <arg_1> <arg_2> … <arg_n> ')'
- Examples of S-expressions in prefix-form:

```
(+ 4 5) ; A simple example.
```

  - Note that ";" denotes a comment in Scheme.

```
(+ (* 3 4 5) (- 5 3))
```


S-expressions

- To evaluate an expression:
  - Evaluate the <function> to a value that is a function.
  - Evaluate each <arg_i> to obtain its value.
  - Apply the function value to these argument values.
- Evaluating our examples:

```
(+ 4 5)
```

  - 4 and 5 evaluate to themselves.
  - Applying "+" to 4 and 5 results (not surprisingly) in the value 9.

```
(+ (* 3 4 5) (- 5 3))
```

  - Evaluate `(* 3 4 5)` to obtain the value 60.
  - Evaluate `(- 5 3)` to obtain value 2.
  - Evaluate `(+ 60 2)` to obtain value 62.

S-expressions

- So far we've seen S-expressions that are lists (that is, enclosed in brackets).
- *Atoms* are also S-expressions.
- An atom is anything that is not a non-empty list. Atoms include:
  - Numbers.
  - Strings.
  - *Symbols.*
  - The empty list.
  - Booleans.

S-expressions

- Examples:

```
#t  ; true
#f  ; false

()  ; the empty list
(a b c)  ; a list containing 3 symbols

(a (b c) d)  ; a nested list
((a b c) (d e (f)))
(1 (b) 2)
```

- Observe that lists can be nested.

Built-in functions

- **car** : Returns the first element of a list.
- **cdr** : Returns the rest of a list.
- **(cons** <element> <list>**)** : Returns a new list formed by adding an element to the front of the given list.
- **quote** or **'** : Produces constants.
- **eq?** : Returns true iff given two identical atoms.
- **null?** : Returns true iff given an empty list.

car

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

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

> (car '(1 2 3 4))
1

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

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

> (car '())  ; Causes an error.
```


cdr

- `cdr` returns the rest of a list (that is, everything except the first element).

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

> (cdr '(1 2 3 4))
(2 3 4)

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

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

> (cdr '(1))
()

> (cdr '())  ; Causes an error.
```

car and cdr

- `car` and `cdr` can break up any list:

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

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

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

> (car '((c d)))
(c d)

> (caddr '((a) b (c d)))  ; caddr = car cdr cdr
(c d)
```


cons

- `cons` returns a new list formed by adding a given element to the front of a given list.

```
> (cons 'a '())
(a)

> (cons 'd '(e))
(d e)

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

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

> (cons (car '(a b c)) (cdr '(a b c)))
(a b c)
```


Exercises

- Run `mzscheme` on ECF.
  - Try evaluating simple expressions like those covered in this lecture.
  - Experiment with `car`, `cdr`, and `cons`.
  - Try out the built-in combinations of `car` and `cdr`, like `cdddr`, `caddr`, etc.
    - How many `car`s and `cdr`s can be combined into a single built-in function in mzscheme?
- Run `drscheme` on ECF.
  - Set the language to `MzScheme`.
  - Repeat the above exercises, this time in DrScheme.