6.001 Recitation 1: Basic Scheme

7/2/2'7 (7 Feb 2007)

Introductions

- Who am I?
  - Course 6 grad student
  - CS interests in computer vision, machine learning, software engineering
  - Outside interests/activities: graduate student council, computer games, building stuff!

- Who are you?
  - Future directions in CS?
  - Topics of interest?

Announcements / Key Information

- **Section Staff**
  - **Recitation Instructor:** Gerald Dalley (dalleyg@mit.edu)
  - **TAs:** TBD
- **Collaboration Policy:** Read carefully in the handout
- **Resources**
  - Lectures, recitations, tutorials, lab, course website
  - **Course Web Page:** http://sicp.csail.mit.edu
  - **Section Web Page:** http://people.csail.mit.edu/dalleyg/6.001/SP2007/index.html
    - Section notes, solutions, *etc.* will be posted here.
  - **Lab:** 34-501, outer door combination 94210, inner door combination 04862*.
- **Problem Sets:** "Missing ore than a couple of the homework assignments may result in a failing grade…" Do them early! Log in at the bottom of the course web page.
- **Projects 0:** Due next Friday (16 Feb @ 6pm)
- **InstaQuiz!**

High-Level 6.001

- "Anything you can do, I can do meta." (Charles Simonyi).
- Scheme
- DrScheme

Evaluator Model

- **Read/Eval/Print loop**

- **Taxonomy of expressions**
  - Stupidly follow the rules → build intuition

  - **Self-evaluating**
    - Numbers
    - Strings
    - Booleans

  - **Names**
    - A **name** evaluates to the value associated with that name.
    - Any collection of characters that doesn't start with a number.
    - Built-in procedures
      - `+`, `-`, `*`, `/`, *etc.*

  - **Combinations**
    `(procedure arguments-separated-by-spaces)`
    - Prefix notation
    - **Evaluate** the *subexpressions in any order*
    - **Apply** the *value of the operator subexpression* to the *value of the remaining subexpressions*.

  - **Special forms**
    - Only a few "special forms" do not follow the combination rules
    - **define**
      `(define name expr)`
      - Evaluate the expression
      - Associate the name with the value of the expression

    - **lambda**
      `(lambda (params-list) expr)`
      - Returns a value: *pointer to the executable procedure*
      - Syntactic sugar
        ```
        (define double (lambda (x) (+ x x)))
        (define (double x) (+ x x))
        ```

Simple Examples

To what do the following expressions evaluate (assume they are evaluated in sequence)?

```
7                          Answer: 7
-                          Answer: -
(+ 2 4)                    Answer: 6
(* (- 5 3) (/ 9 3))        Answer: 6
(7 - 4)                    Answer: error (7 does not eval to a
                                   procedure)
```

More Examples

To what do the following expressions evaluate (assume they are evaluated in sequence)?

```
(lambda (x) (* x x))
   Answer: a procedure
((lambda (x) (* x x)) 5)
   Answer: 25
(define double (lambda (x) (* 2 x)))
   Answer: undefined (double is associated with a new procedure)
(double (double 6))
   Answer: 24
(double double)
   Answer: error (cannot multiply two procedures)
(define cube (lambda (x) (* x x x)))
   Answer: undefined (cube is associated with a new procedure)
(cube 3)
   Answer: 27
(define + 3)
   Answer: undefined (the name "+" is associated with the value 3)
(define – 6)
   Answer: undefined (the name "-" is associated with the value 6)
(* + -)
   Answer: 18
```

Writing a Procedure

Define a procedure called `average` that computes the average of its two numeric arguments.

```
> (define average (lambda (a b) (/ (+ a b) 2)))
> ; test:
> (average 5 7)
6
```

Subtleties

Consider the following two definitions below. How are they similar and how do they differ?

```
(define plus +)
(define add
  (lamdba (x y)
    (+ x y)))
```

Answer: Both of them will add two numbers. *plus* creates an alias to the addition procedure. *add* creates a new procedure that calls the addition procedure. More subtle points: the built-in addition procedure can handle variable-length argument lists, but *add* can only

support exactly two arguments (this isn't all that important right now, but it is a difference). After Lecture 2, you should also notice that if we redefine + at some point in the future, *plus* will use the original addition procedure and *add* will use whatever is the currently-associated value of + when *add* is evaluated.

Glossary

Here are a number of terms you'll see introduced over the next few weeks.

- **Program:** collection of procedures and static data that accomplishes a specific task.
- **Procedure:** a piece of code that when called with arguments computes and returns a result; possibly with some side-effects. In Scheme, procedures are normal values like numbers.
- **Function:** see procedure; they're equivalent in scheme. Some other languages make a distinction.
- **Parameter:** An input variable to a procedure. A new version of the variable is created every time the procedure is called.
- **Argument:** The actual value associated with a parameter. For a procedure created via `(define double (lambda (x) (+ x x)))` and evaluated with `(double 5)`, `5` is the argument and `x` is the parameter.
- **Expression:** A single valid scheme statement.
  `5`, `(+ 3 4)`, and `(if (lambda (x) x) 5 (+ 3 4))` are expressions.
- **Value:** The result of a evaluating an expression. 5, 7, and 5 respectively.
- **Type:** Values are classified into types. Some types: numbers, booleans, strings, lists, and procedures. Generally, types are disjoint (any value falls into exactly one type class).
- **Call:** Verb, the action of invoking, jumping to, or using a procedure.
- **Apply:** Calling a procedure. Often used as "apply procedure p to arguments a1 and a2."
- **Pass:** Usage "pass X to Y." When calling procedure Y, supply X as one of the arguments.
- **Side-effect:** In relation to an expression or procedure, some change to the system that does not involve the expression's value.
- **Iterate:** To loop, or "do" the same code multiple times.
- **Variable:** A name that refers to a exactly one value.
- **Binding:** Also verb ``to bind''. The pairing of a name with a value to make a variable.
- **Recurse:** In a procedure, to call that same procedure again.

InstaQuiz #1

Name: _____

1. What programming experience do you have (none is fine)?

2. What do you hope to learn in 6.001 / why have you chosen to take this class?

3. What do the following expressions evaluate to, if evaluated in sequence?

```
1
```
Answer: 1

```
(+ 2 3)
```
Answer: 5

```
(define fred +)
```
Answer: undefined (the name *fred* is associated with the addition procedure, or whatever + was associated with)

```
(fred 4 6)
```
Answer: 10