MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.090—Building Programming Experience
IAP 2005

Lecture 1

Glossary

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.

argument An input variable to a procedure. A new version of the variable is created every time the procedure is called.

parameter see argument.

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.

Values

1. **Numbers**

2. **Booleans**

3. **Strings**

4. **Procedures**

5. **Lists**

Scheme

1. **Basic Elements**
   (a) *self-evaluating* - expressions whose value is the same as the expression.

   (b) *names* - Name is looked up in the symbol table to find the value associated with it. Names may be made of any collection of characters that doesn't start with a number.

2. **Combination**
   `(` *procedure arguments-separated-by-spaces* `)`
   Value is determined by evaluating the expression for the procedure and applying the resulting value to the value of the arguments.

3. **Special Forms**

   (a) *define* - `(define` *name value*`)`
   The name is bound to the result of evaluating the the value. Return value is *unspecified*.

   (b) *if* - `(if` *test consequent alternative*`)`
   If the value of the test is not false (#f), evaluate the consequent, otherwise evaluate the alternative.

Problems

Run scheme, open a new file "class1.scm". All of your answers for the following problems should end up in this file.

1. *Evaluation* - For each expression:

   (a) Write the type of the expression

   (b) Write your guess as to the expression's return value. If the expression is erroneous simply indicate "error" for the value. If the expression returns an unspecified value, write whatever you want!

   (c) Evaluate the expression, and copy the response from the *scheme* buffer.

```
4

5.5

4.2e1

(+ 1 2)

(7)

(* (+ 7 8) (   -   5 6))
```

```
(define one 1)

(define two (+ 1 one))

(define five 3)

(+ five two)

(define biggie-size *)

(define hamburger 1)

(biggie-size hamburger five)

(= 7 (+ 3 4))

(= #t #f)

((+ 5 6))

biggie-size

(if #t 1 (+ 3 0))

(if (if #f #t #f) #f #t)

(if (if (= hamburger five) 3 7)
    (+ (if (= (+ 1 one) two)
           3
           5)
       7)
    "yay")

(/ 256
   (if (> five (+ two 1))
       (/ 7 (- hamburger 1))
       2))
```

2. *Five* - write an expression that evaluates to 3.

3. *Five* - write a *more interesting* expression that evaluates to 3.

4. *Define X* - for each of the following expressions:

   (a) Identify the variables are are *unbound.*

   (b) Supply definitions (ie `(define x ...)`) for each of the variables that make the expression evaluate to the target value.

   (c) Type in the expressions and verify that your solution gives the correct result.

```
(+ x (* y 3))
;Value: 13

(if q (if r 3 4) 7)
;Value: 4

(+ 7 (if z (+ a z) 3))
;Value: 11

(= yum (* -1 (+ yum 2)))
;Value: #t

(< (if (not thirty-four) 34 thirty-four)
   (+ thirty-four (* seventy-seven (if seventy-seven -1 1))))
;Value: #f
```

5. *Primitive Procedures* - for each of the following expressions:
   (a) Identify the primitive procedures which you don't already know
   (b) Write down a guess as to what the primitive procedure does.
   (c) Look it up in the MITScheme reference manual (you may find the index handy).
   (d) Write an example usage of the procedure and test it to see that it works as you suspect it does.
   (e) Fill in the blanks in the original expression such that it evaluates to the target value.

```
(+ 3 (abs ________))
;Value: 5

(if (> (expt 2 ____) 15)
    (sqrt 2)
    (sqrt 4))
;Value: 1.4142135623730951

(string-append "foo" ___ "baz")
;Value: "foobarbaz"

(string-append
  (number->string
    (gcd 35 _____))
  " rules!")
;Value: "7 rules!"

(define s _______)
(if (string-suffix? "yowzah" s)
    (+ 7 (string-search-forward "ow" s))
    #f)
;Value: 14
```