6.090, Building Programming Experience
Ben Vandiver

Lecture 1: Introduction and Basic Scheme

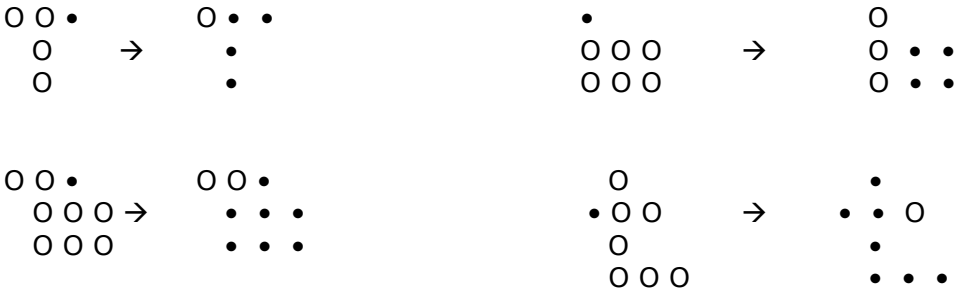
We were handed a piece of cardboard with 32 thumbtacks arranged in a cross pattern with one hole left empty in the very center. The object was to have only one thumbtack left in the middle of the cardboard after executing thumbtack jumps (similar in rules to jumping pieces on a checkerboard; no diagonal jumps, only in the cardinal directions, and the jumped tacks get removed).

```
    O O O
    O O O
O O O O O O O
O O O   O O O
O O O O O O O
    O O O
    O O O
```

Below is a list of supplied smaller moves to aid us in the bigger puzzle. The circles represent the thumbtacks, while the dots represent holes in the board. Jumps may only be executed using the spaces shown.

Macros

The "T"



Find patterns: which pattern when; **composition of patterns simplifies problems.**

There was a similar game on the back of the board. We were instructed to use the "T" macro and another macro listed below the image of the puzzle:

```
    •
   O O
  O O O
 O O O O
O O O O O
```

Additional Macro:

Emacs
The mode line is the black highlighted line that has the characters **Edwin: *scheme*
The minibuffer is the region where emacs asks questions and gets answers. The first in-class assignment was saved under class1.scm
Some commands not listed in the handout (Edwin-cheat.txt), also commands that were mentioned out loud and may still be listed in handout:

Saving file C-x, C-s
Asterisks indicate that the file has been modified since it was last saved
Undo C–x, u
Switching buffers C–x, b
Evaluate C-x, C-e
Escape Whatever C-g
Emacs Tutorial C-h, t <= worth looking try out

Value
1. Numbers - written as 4, .75, ¾
    Operations performed on numbers:
        Add/Subtract/Mult./Div.
        Equality
2. Booleans – "true" and "false", written #t and #f
    Operations performed on Booleans:
        Not (e.g., not #t indicates "false")
        And (e.g., #t and #t indicates "true")
        Or (e.g., #t or #f indicates "true")
3. Strings – sequence of characters such as "yay" → strings are listed in quotes
    Operations performed on strings:
        Append strings
        Split strings
        Length of string
        Characters of strings
        Compare two strings
4. Procedures – in scheme, macros are considered "things" (a procedure is considered a "thing")
    Example of procedure: recipe that's written, can physically be held in hand
    Questions asked of a recipe:
        What are inputs of recipe?
        Is it a recipe?
5. Lists – e.g., (1 3 4). It is possible to have lists of lists.

Scheme

Expression – text input by the programmer and sent to the evaluator (with C-x, C-e)
Value – result computed by the evaluator when it works on an expression

For every expression, there is a rule to evaluate it and produce a value. Almost every expression has a value (excepting cases covered by 6.001).

Two types of expressions: simple and compound.

1. Simple Expressions

(a) *self-evaluating* – expression whose value is the same as the expression
Self-evaluating expression is the simplest type of expression
There are three types of self-evaluating expression: number, Boolean, string
7, 5, 5.5 - all numbers
#t, #f - Booleans
"foo", "yay" - Strings
Expressions follow the path: expression → EVALUATOR (where the caps represent a "cloud")

(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.
This is where the name/value table comes in:

N	V
foo	4
+	PROCEDURE +

2. Compound Expressions

(a) Combination
(*procedure arguments-separated-by-spaces*)
When you see open parens, this tells you that you have a combination. The procedure is the thing you want to do. The arguments are the things you want to do it to.

(+ 3 4)

There are three sub-expressions in this combination. To evaluate the combination, we must first evaluate all of the sub-expressions and then combine them. Evaluating the + returns a PROCEDURE "cloud". Evaluating the self-evaluating expressions 3 and 4 return 3 and 4, respectively. Applying the procedure "cloud" to the numbers 3 and 4 returns 7.

(b) Special Forms
When one of several key words appears where the procedure would be in a combination, it means that you have a special form, not a combination. The key words we'll start with are *define* and *if*.

(i) *define* – (define *name value*)
The name is bound to the result of evaluating the value. Return value is *unspecified*. Exactly like a combination, except the first word is *define*
(define x 5)
The evaluation rule for the define special form is: first evaluate the value (5), and then binds the name (x) to the value in a name/value table.

(ii) *if* (if *test consequent alternative*)
If the value of the test is <u>not false (#f)</u>, evaluate the consequent, otherwise evaluate the alternative. (#f is read "hash f")

(if #t 1 2)
if true-ish, then 1, else 2
(not #f)

(if 3 1 2) => 1
(if 0 1 2) => 1
(if + 1 2) => 1

Experiments with names:

Name value
foo 4

foo→ 4
bar →
fo →error
Not in table, produces error
D debug, q quit