Session 2

I. Questions (about LISP and homework)

- Assignment 0 is due on Thursday, September 8th.
- Examples to illustrate the concepts needed for Assignment 0
  - Fraction handling? Doing exponents and logarithms with integers.

II. LISP II – things I didn't cover last time

Functions

- **Lambda Function –** allows you to define an anonymous function with no name.
  (lambda (x) (+ x 5))
  - Why would we want "nameless" functions?
    - Sometimes you want to make a simple function without the rigor of defining a whole new function.
    - This case comes up often when passing functions as arguments.
  - Passing Functions as arguments can be done by referring to the function name with the #'<function-name>. e.g. (apply #'+ '(1 2 3 4))
- **Keyword Arguments –** some functions take keyword arguments, arguments that come in name, value pairs where the name is proceeded by a colon.
  - e.g. the function *member* allows you to specify equality: :test eq
- **Recursion –** one of the building blocks of functional programming is recursion; the idea of a function calling itself → Factorial.
  - **Base case** – covers the "easy" case where the answer is simple.
  - **Recursive case –** covers the cases where we don't know how to solve the big problem directly, but we know how to break the problem into smaller parts.

Special Functions

- *sharp quote* (#') – an abbreviation for *function*, which returns the function associated with a given name. This is typically used to pass functions as arguments.
- *backquote* (`) – used like *quote* except we can force evaluations within the quoted expression:
  - To evaluate within a backquote, use a comma. e.g. for x=1,
    `(1 ,x) → (1 1)
  - To get elements of a list use ,@. e.g. for x = (a b c)
    `(1 ,@x) → (1 a b c)
- *apply* – applies a function to a list of arguments. e.g. (apply #'+ '(1 2 3)) → 6
- *funcall* – applies a function to arguments. e.g. (funcall #'+ 1 2 3) → 6
- *mapcar* – applies a function to consecutive elements of lists it received as arguments:
  e.g. (mapcar #'list '(1 2 3) '(4 5 6)) → ((1 4) (2 5) (3 6))

Objects

- The primary way we will associate data into a “class” is through the **defstruct** function that creates a new type with members.

  (defstruct group x y z)

  - We can define objects to be a given structure by using the <u>constructor</u> for that structure, **make-<name-of-struct>**. This constructor is automatically created when we create a structure. e.g.

    (make-group g)

    - this constructor can also take keyword arguments to specify the initial value for its members. These are named by the member name: (make-group g :z 1 :x 3) → makes a “group” *g* with x-part 3, y-part nil, and z-part 1.
    - Any arguments not passed to the constructor are set to nil.
  - defstruct also creates a member <u>accessor function</u> to refer to the members of a structure. e.g. (group-x g) → returns the “x” part of *g*.
  - defstruct also creates a <u>member-predicate</u> to check if a variable is of the type of that structure: e.g. (group-p g) → returns *t*
  - defstruct also creates a <u>copier function</u> to copy an instance of that structure: e.g. (copy-group g) → returns a copy of *g*.
- **Slot options**
  - When defining each member of a struct, we can give a default value, a type, and define if it is read-only.

    (defstruct thing (height 0.0 :type double-float)
    (weight 0.0 :type double-float :read-only t))
- **Inheritance & overriding default methods**
  - When defining a structure, we can cause it to inherit:

    (defstruct (person (:include thing)) name)

    which causes person to inherit from our thing class.
  - In addition, we can also override constructor, predicate, copier, etc.
- To create new methods for a structure, we use the **defmethod** function.
  - This is similar to a *defun* declaration, but we need to refer to the class the method operates on. In specifying our method, we therefore pass arguments of the form (<arg-name> <class-name>):

    e.g. (defmethod setx ((p group) x) (setf (group-x p) x))
  - While we will be using *defmethod* to build methods for our structures, *defmethod*, it is really just a specialized version of *defun* that allows overloading of a function name – a generic function
    - defmethod allows several functions to have the <u>same</u> name as long as they have different argument types.
    - To specify argument types, we have argument lists of the form:

      ((arg1 type1) … (argn typen))
    - Thus, we are able to create several functions of the same name:

      (defun generic (x y) …)
      (defun generic ((x integer) y) …)
    - LISP applies the most *specific* method that matches the arg types.

Compiling

- Unlike Scheme, LISP *does not* automatically compile your programs.
    - A typical user-defined function (defun, defmethod, etc.) is interpreted by the LISP interpreter, a program that converts LISP functions into machine language as the functions run.
    - For speed, we want to compile our programs → create the "machine language" version of it so that no interpretation occurs at run-time.

      (*compile* '<function-name>)

Blocks

- There are a number of ways to make program *blocks* – a sequence of instructions.
    - In pure functional programming, blocks are not necessary since only the last line is returned and previous lines cannot affect the final one since there are no side-effects.
- Common blocks in LISP: let, progn, block, tagbody
    - **let** – allows us to create new variables for use in the block – a *new lexical context*.

      ```
      (let ((x 0) (y 1))
           (f x y)
           (g x y))
      ```
    - **progn** – a simple block that evaluates its arguments in order and returns the final statement. (progn *expr1 expr2* … *exprN*)
    - **block** – a block with a name and emergency exits. The *return-from* (just *return* if the name is nil) function allows us to exit the block with a value before evaluating all statements.

      ```
      (block A
          (if (< x 0) (return-from A nil))
          (exp x))
      ```
    - **tagbody –** a block that allows *tags* and *goto*s.

Closure

- One of the interesting things about LISP is that we can pass functions as arguments to another function and manipulate them on argument sets. We can also *return* arguments from functions, we simply quote them:

  ```
  (defun sum () #'+ )
  ```
- However, when the function returned requires a variable ***outside of its context*** this is called a closure. For instance:

  ```
  (defun addn (n) #'(lambda (x) (+ x n)))
  ```

  where the returned function depends on *n* which is defined outside the *lambda*.
    - Since the returned function is dependent on the external environment, a new *independent* function is created each time our "addn" is called.
- We can also create functions that depend on the *same external* variable:

  e.g.
  ```
  (let ((count 0))
      (defun reset () (setf count 0))
      (defun inc() (setf count (+ count 1))))
  ```

AIMA

- What is the AIMA library.
- How can you use it.

III. AI Topics

task environment – the problem the agent is solving as characterized by

1) <u>Performance Measure</u> 2) <u>Environment</u> 3) <u>Actuators</u> 4) <u>Sensors</u> – PEAS.
    - In groups, discuss how to formulate the following problems:
        i. Checkers
            1. Win/Lose. Percentage of your pieces on the board.
            2. The checker board
            3. Moving a piece (turn-based).
            4. Observations of board (visual perhaps).
        ii. Rubik's Cube
            1. Number of uni-colored sides / some measure of uniformity.
            2. A Rubik's cube.
            3. Rotations of the cube.
            4. Observations of cube state (visual perhaps).
        iii. Elevator Dispatching
            1. Maximal waiting/system time; average squared weighting time.
            2. Elevator world with people coming and going.
            3. Elevator controls.
            4. Elevator buttons.
        iv. Engine Optimization
            1. Balance between max output and not exploding
            2. set of variables regulating fuel flow, etc.
            3. various parts to adjust
            4. temperature gauges, speed/work measure
        v. French to English Translator
            1. Percentage of words wrong. Precision/Recall.
            2. Stream of voice/noise being produced in a real world.
            3. Speaker/Screen output device.
            4. Microphone/keyboard.

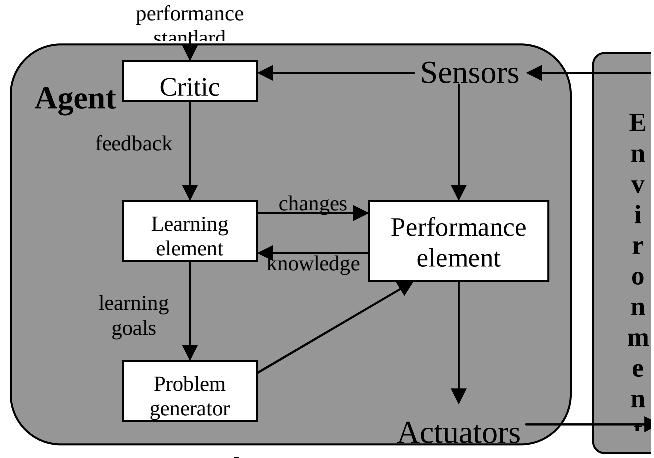
What are the properties: **Observable, Deterministic, Episodic, Static, Discrete, & Agents**

Randomization → partial information is planning against worst case scenarios.

- A game we are playing against the environment.
- In general, a deterministic strategy is flawed.

Learning – the process of modification of each component of an agent to make the components agree closer with the available feedback thereby improving the agent's performance.

- **learning element –** responsible for making improvements
- **performance element –** responsible for selecting external actions… the agent being modified.
- **critic –** provides feedback on the agent's performance and suggests improvements.
    - **performance standard –** a *fixed* measure of agent's performance.
        - distinguishes the *reward* in the percept by providing direct feedback on quality of agent's performance.
- **problem generator –** suggests actions that will lead to exploration.



learning agent