Session 1

I. *Introduction*

Overview

- Hello, and course intro – no cheating, course website, my website, email addr.
  http://www.cs.berkeley.edu/~russell/classes/cs188/f05/
  http://www.eecs.berkeley.edu/~nelsonb/
- The sections I'm teaching are intended for Cognitive Science majors.
  - These sections will attempt to address the concerns of those students. Since those students haven't had as much programming experience, I will try to focus these sections to meet those concerns.
  - For the time being, I need you to be understanding of three things:
    1. I'm not very good with names, so it may take me some time.
    2. This is my first time teaching, so let me know how I can
    3. I have my Prelim Exams in 2 weeks, so bear with me.
- My office hours will be: Wed: 3-5 pm 751 Soda
  Fri: 10-11 am 751 Soda
- This is an overview course on modern techniques in artificial intelligence.
  - The primary textbook for this course is *Artificial Intelligence: A Modern Approach*, SECOND EDITION by Stuart Russell and Peter Norvig. While the 1st edition covers similar topics, they are quite different so get the 2nd edition – GREEN COVER.
  - This class is taught in LISP – a programming language similar to SCHEME.
    - A good introduction and reference book for LISP is *ANSI Common Lisp* by Paul Graham.
    - An alternative LISP text is available on the web; Common LISP the Language, 2nd edition by Guy Steele:
      http://www.cs.cmu.edu/Groups/AI/html/cltl/cltl2.html

Names

- Let's go around the room and introduce ourselves. Please tell me:
  - your NAME, YEAR, and MAJOR
  - Why are you taking this class?

Notecards

- Finally on the Front of the notecard, please write:
  - Your Name and Major
  - Email Address
  - Programming Languages you are comfortable with.
- On the back,
  - Please write a short description of what you want to get out of this section.

II. LISP

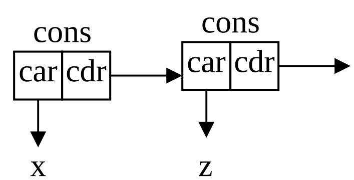
LISP References

Besides the books mentioned above, there are some other

- Differences between Scheme and LISP
  http://dept-info.labri.u-bordeaux.fr/~strandh/Teaching/Langages-Enchasses/Common/Strandh-Tutorial/diff-scheme.html
- Scheme vs. Common Lisp – A table of differences between the languages.
  http://www.cs.utexas.edu/users/novak/schemevscl.html
- Class LISP Tutorial – how to setup, write, and run LISP in the lab.
  http://www.cs.berkeley.edu/~russell/classes/cs188/f05/assignments/a0/lisp-tutorial.html
- Class LISP notes
  http://www.cs.berkeley.edu/~russell/classes/cs188/f05/assignments/a0/lisp-notes.html
- LISP Function Reference – an online reference for LISP functions.
  http://www.cs.cmu.edu/Groups/AI/html/cltl/clm/

Fundamental LISP

- Everything in LISP is a <u>list</u> or an <u>atom</u>, even function calls. Hence, we can,
  - Extend the language.
  - LISP can be written in LISP.
- **Function calls** – even function calls are lists
  - function name is the 1st element of a list
    `(fn arg1 ... argn)`
  - *Prefix Notation* (+ 1 2 3 4 5 6)
  - In normal LISP evaluation, all arguments of a function are evaluated, and the function is applied to the result.
  - **Quote Function** - passes an argument without evaluation. Abbrev. by ‘
    ‘(a b c)
- The fundamental element is the **Atom** – symbols that represent a value.
- LISP stands for *List Processor*. The fundamental data structure is the **list**:
  e.g. (A 1 (1 3))
  - **nil** or () is both an atom and a list. Moreover it is also the symbol for false.
    - However (nil) is not the same as nil
  - Essential LISP list functions:
    - **cons** – operator that builds a list. e.g. (cons x (cons ‘z nil)) makes:
      - Two arguments:
        - car – the first element of the list
        - cdr – the remainder of the list
    - **car** – operator that returns a list’s 1st element.
    - **cdr** – operator that returns the rest of the list after the 1st element.



Functions

- In LISP, functions are defined by the function *defun*, which takes >2 arguments
    - A name
    - A list of argument names
    - All other arguments are evaluated and the last evaluated expression is the return value of the function

      (defun sum (args) (apply #'+ args))
- **Comments** – there are two ways to make comments.
    - semicolon (;) – everything is ignored until the end of the line.
    - #| … |# - everything between the delimiters is ignored.
- **Functional Programming**
    - As designed, LISP is a functional programming language. In this paradigm, programs are defined by the return values of their functions rather than by modifying variables.
    - In pure functional programming, no values are modified → there are no *side effects* such as,
        - printing, incrementing, or setting value.
        - One consequence of pure functional programming is that no operations can modify their arguments, hence, they must copy. e.g. remove

          (remove 'a '(b a n)) returns (b n) but copies those elements leaving the original list unaltered.
    - LISP also has constructs *side effects*.
        - *format* is used to print

          (format t "The number is ~A.~%" x)
        - *incf* is used to increment a variable
        - *setf* is used to set the value of a variable.

          (setf x 5)

          setf is essential as it can set the value of any symbol including members of a structure or elements of an array.
- **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.
- **Lambda Function** – allows you to define an anonymous function with no name.

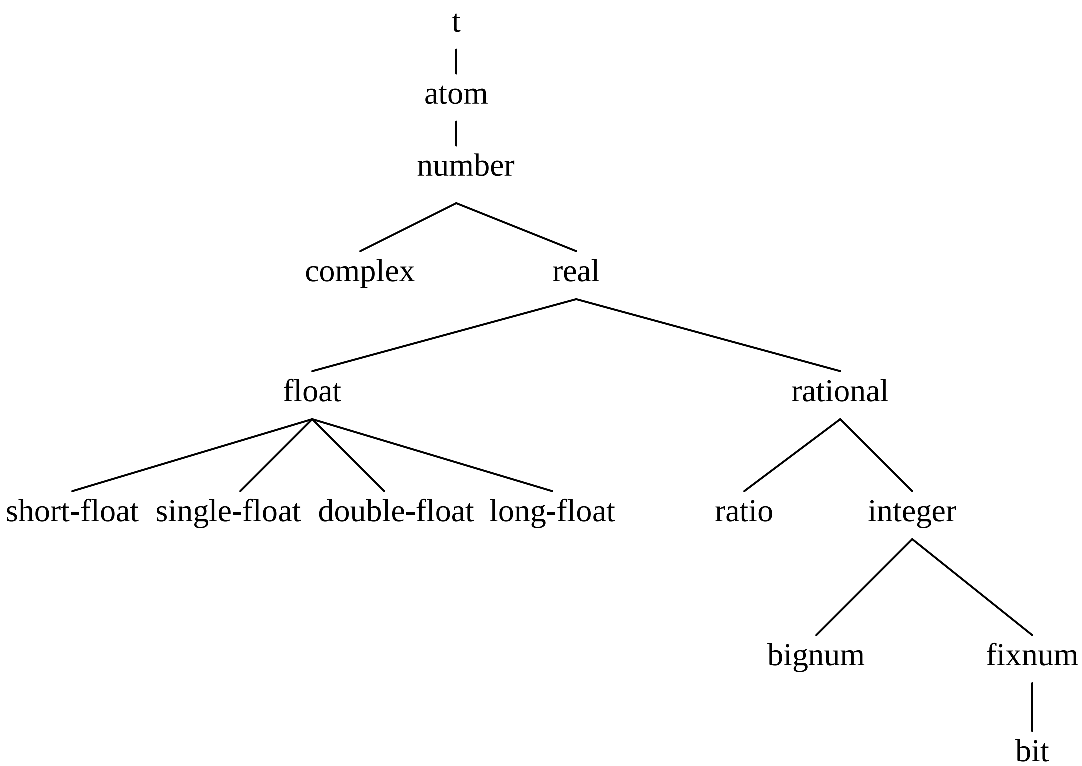
  (lambda (x) (+ x 5))
    - Why would we want "nameless" functions?
        - Sometimes you want to make a simple function without the rigor of defining a whole new function.
        - This case comes up often when passing functions as arguments.
    - Passing Functions as arguments can be done by referring 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. :test eq

Truth & Equality

- Every value in LISP is considered *true* except the special symbol nil, which is considered *false*.
    - This allows functions to return more information than just true/false.
        - e.g. member
          (member ‘b ‘(a b c)) → (b c)
- In LISP, functions that test whether or not a condition holds are called *predicates*.
    - *Predicates* are no different from normal functions, but are often named with a *p* at the end of the word (e.g. listp tests if its argument is a list).
- Equivalences
    - = (= 5 6) – numerical comparison equality.
    - **EQ** (eq x y) – true when x and y point to the same memory location; thus, to numbers may not be equal (compiler dependent)
    - **EQL** (eql x y) – same as EQ, but compares numbers and characters.
    - **EQUAL** (equal x y) – true if x and y have the same list structure (look the same when printed).
    - **EQUALP** (equalp x y) – like equal but recursively compares (arrays, vectors, and structures)
- Many functions use some form of equality to perform their task (e.g. member looks to see if a particular element is present in a list using equality).
    - By default, the equality function used is *eql*.
    - To specify a different equality function, we can use the keyword arg, **:test**
- **Conditionals**
    - In LISP the fundamental conditional is **if**:
        - It takes up to 3 arguments with an optional else:
          (if cond then else)
        - *if* does not use the common evaluation rule of evaluating all its arguments; either the *then* or the *else* is evaluated depending on *cond*, but not both of them.
    - The **case** conditional is a multi-way if:
        - It has a condition, and a number of possible cases each of which is a list.
            - The 1st element of each list are the values handled by the clause.
            - The remaining elements are the statements to be executed conditional on the case.
            - The generic *otherwise* case handles all other cases.

Types Hierarchy

- Every value has a type hierarchy. Every value (except nil) is of type *t*, the generic type. Every subtype of *t* is considered *true* for conditional statements, as discussed earlier.
    - Type hierarchies capture the valid "contexts" of the value.
    - For instance, the type hierarchy of numbers is,



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 constructor 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 accessor function to refer to the members of a structure. e.g. (group-x g) → returns the "x" part of *g*.
  - defstruct also creates a member-predicate to check if a variable is of the type of that structure: e.g. (group-p g) → returns *t*
  - defstruct also creates a copier function 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 same 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.