Efficiency and lisp

- Now we’ll look at a C implementation of lisp, and discuss the runtime efficiency of different aspects
- How does the implementation of different data types impact speed and storage?
- What kind of optimizations are performed for recursion and iteration?
- Can we establish some general guidelines for writing more efficient lisp

C implementation of data types

- See a sample .h file for common lisp at

  csci.viu.ca/~wesselsd/courses/csci330/code/lisp/cmpinclude.h
- a `lispunion` is just a C union of all possible types, each of which is a struct (with fields for the data and relevant properties)
- a lisp `object` is simply a pointer to an item of type `lispunion` (i.e. a pointer to something of any defined lisp type)
- a cons is represented by two structs: a `c_car object` and a `c_cdr object` (i.e. car and cdr are really both pointers, and can point to any type of item)

Function implementations

- we can see parts of lisp implemented as C macros directly accessing the right field in the right struct (e.g. `char_code(x)` becomes `((object(x))->ch.ch_code`
- in the latter half of the header we can see that all function prototypes return `object` and take parameters of type `object`
- functions can thus take and return parameters of any type, and what is passed/returned is always just a pointer
- In the last quarter of the header we can see many functions are implemented as inlines, using C functions/the math library

car, cdr, cons

- cons represents a data type: a struct with two fields, for car and cdr, each an object (pointer to something of any type)
- car, cdr, cadr, cddr, etc are each implemented as C macros directly accessing the right pointer in the chain, e.g.
  - `car(x) becomes (x)->c.car`
  - `cdr(x) becomes (x)->c.cdr`
  - `cadr(x) becomes (x)->c.c_cdr->c.c)car`
  - `etc`
- (hence crashes when you try to access through a null pointer)

Data types

- (ignoring the many additional properties of each type)
- rationals, complex numbers etc simply use the necessary fields of other types to store the needed data
- string and bit vector data stored as a `char*`
- vector and array data stored as an object* whereas the various fixed arrays are stored as `fixnum*`, `int*`, etc (i.e. actual C-style arrays of integers)
- functions to `make_int`, `make_list` etc, returning the object
- `make_list` takes size of list as param, note that list function itself takes variable number of arguments (we’ll talk about those in C macros later)

Efficiency: data type access

- data types in order of decreasing efficiency:
  - string, access with (char str i)
  - vector, access with svref
  - array, access with aref
  - list, access with nth
  - sequence, access with elt
- for lists of constant size, struct more efficient

Repetition/efficiency

- Iteration constructs more reliably efficient than tail recursion, simply because some compilers may not perform good optimization of tail recursion (iteration macros do the rewrite so the compiler doesn’t have to)
- dotimes, dolist provide more specialized/efficient implementations than the generic loop types
- Tail recursive functions should avoid the use of dynamically-scoped (special/defvar) variables

Maintainability

- another aspect of efficiency: how much work is it to maintain the code?
- Mostly common sense programming tips for clarity/ease of comprehension:
  - Avoid side effects where possible
  - Avoid runtime use of eval, #, intern
  - Avoid dynamic scope where possible
  - Use (function f) or #’f rather than just ‘f