1 Data Abstraction

So far in this class, we have seen ways in which we can represent complex data as lists. We have also seen how we can combine these basic data items in more complex ways; for example, we have seen how we can represent student projects as association lists, and store several such items in a list of student projects.

Patient records can be structured as lists. Each record contains the patient name, address, telephone numbers, date of birth, and other relevant information grouped together in a complex structure:

```lisp
((simone jones)
  ((1234 boylston street) seattle wa 98102)
  ((home 206-234-5678) (work 425-432-8765))
  (10 11 1978)
  (pruritis-of-the-palm)
  (angela a aardvark md))
```

We could then create a simple database of these records and perform operations such as searching for a patient, adding a new patient, etc. If this were a small system that only a single person was responsible for, it might be feasible to create database operation functions that relied directly on the underlying data structures. For instance, the patient’s name appears first in the record, so that knowledge might be used to create a search function that evaluated only the first element of each record.

Now imagine what would happen if a unique patient ID number was added to each record as the first element. What would happen to our initial name search function? By tying the database operation functions directly to the underlying data structures, changes in the records themselves would result in system-wide problems.

Instead of searching for a match on the first element of a patient record, what we really should do is match on a name or ID number, regardless of how that information is actually stored. Data abstraction is the process by which internal implementation details (such as underlying data structures) are abstracted away and only relevant information is presented. In the patient record example, all we want to access is the patient’s name; we don’t want to concern ourselves with details such as where the name appears in the underlying list construct.

In general terms, abstraction refers to ignoring the details and focusing on the essentials. Abstraction gives us the ability to encapsulate and isolate design and execution information.

*Student Question:* What is the difference between procedural and data abstraction?
Procedural abstraction hides operational details, such as the exact method needed to get something out of a complex data structure. Data abstraction is a method of hiding the details of the way data is structured. Both are two faces of the same idea.

2 Accessor Functions

We want to hide specific implementation details of the underlying structures that make up our patient records, yet still be able to access the data. Accessor functions allow you to do just that; they define a way to access data while abstracting away implementation details.

Here is an example of an accessor function for retrieving a name from a patient record:

```lisp
(defun name (pt)
  (first patient))
```

If we wanted to get the name of a specific patient, all that is required now is to call `name` for that patient; there is no need to be concerned with underlying record organization.

Data and procedural abstraction allow us to create abstraction layers. Abstraction layers are created when information about underlying structures and operations are hidden behind abstraction boundaries. For example, an abstraction layer is created by the accessor function `name`. An abstraction boundary exists between the outside environment and the patient record at this point; if something needs access to the name field in the record, it goes through the `name` function instead of directly accessing the record itself. So, instead of needing to know the details of the record, we only need to understand the `name` interface.

Abstraction layers can be built up, with each relying only on the one immediately below it. For example, we can build on our initial `name` function to access a patient’s first name:

```lisp
(defun first-name (pt)
  (first (name pt)))
```

Note that `first-name` only uses `name`; it never directly accesses the underlying patient record.

2.1 Why use abstraction?

Once you create abstraction layers, you can change the underlying data structures as long as you keep functional interfaces consistent. For example, we could change how names are stored in our patient records, and as long as we changed the `name` function accordingly, nothing beyond that abstraction boundary would be affected. `first-name` would work just as smoothly, because it only relies on `name`, not the records themselves. Multiple levels of abstraction mean that you can make changes at each level relatively independent of all others.

Abstraction layers allow us to build very complex systems while keeping design simple. This is one of the most important concepts to take away from this class.

2.2 Accessor functions can do some work

Accessor functions can be used to provide useful data transformations as well as basic retrieval. For example, we can define a function called `print-nice-date` which transforms basic information retrieved from a
patient record into formatted output.

**phone-number** is another example of an accessor function which can do additional work on patient data:

```lisp
(defun phone-number (pt &optional type)
  (let ((numbers (third pt)))
    (second (if type
               (assoc type numbers))
             (first numbers))))
```

*phone-number* takes a patient record and an optional type specifier (for work or home number), and returns the requested type of phone number if available.

Accessor functions can have varying degrees of complexity, but all are related in that they access underlying data structures while hiding unnecessary organizational details.

## 3 Constructors

Constructors hide details of creating new data objects. For example, here is a constructor to create patient records:

```lisp
(defun make-patient (&key name address
city state zip
home-phone
work-phone
birthdate
diagnosis
physician)
  (list name address city state zip
       (list (list home home-phone)
             (list work work-phone)))
birthdate diagnosis physician))
```

Note that to create a new patient record only requires that **make-patient** be passed the appropriate information. This is far more convenient than having to explicitly build new lists and remember how things are organized every time a new patient is added. Only the internal workings of **make-patient** are concerned with constructing a list in a way that reflects the decisions made regarding the underlying data structures.

**make-patient** includes the additional convenience of representing patient information as optional keyword arguments. This way, you can create a patient record by listing information in any order, or omitting any unknown facts.

## 4 Transforming procedures

Transforming procedures are used to change data into some other form. For example, it is useful to have a function that takes a list and applies a transformation to each element. We don’t want to be concerned with how the list elements are organized; we just want to have a function that allows us to make changes, not worry about the details of how it does so.
4.1 Transforming procedures for lists are recursive

Imagine we had a list that represented a single DNA strand, and wanted to transform it to represent the complimentary strand. It is easy to see how to do this recursively:

- Solve a small part of the problem: take the first base off the DNA strand (get the first element of the list) and transform it into its complementary base. Construct a new list/strand of the transformed bases.
- Apply the transformation process to the rest of the strand/list.
- Stop when the list is empty, and all bases have been transformed.

We can generalize this example into a basic template that all list transformation procedures take:

```lisp
(defun <transform-fn> (s)
  (if (null s) nil
    (cons (<elt-transformer> (first s))
      (<transform-fn> (rest s))))
)
```

In all cases, we want to stop the transformation process when a list is empty. Otherwise, we start with the simplest base case: apply the transformation to the first element in the list. This is represented in the above template as `<elt-transformer>`. `<elt-transformer>` is a function that takes a single element and changes it in some way. In our DNA example, the element transformer would take a single base and return the complementary one. We put the result of the single transformation on the front of a new list, and continue on with the complete transformation by recursively applying the procedure to the remaining elements of the list.

Note that instead of needing to store temporary results of single transformations, recursion allows you to easily change a single element and then stick it on the front of a new list.

**Student Question:** Why do we need recursion in transformation procedures?

We use recursion to solve the problem of transforming an entire list because it is very easy to transform a single element and then apply the function to the rest of the list. For example, imagine that we want to transform the `(a b c d)` into `(1 2 3 4)`.

Here is a diagram of how the transformation would work, using the template given above. Note that at each level, we transform the first element then recurse on the remaining sublist:

```
( a b c d )
  ↓
( 1 ( b c d ) )
    ↓
( 1 2 ( c d ) )
      ↓
( 1 2 3 ( d ) )
        ↓
( 1 2 3 4 )
```
4.2 Transforming procedure for DNA replication

Applying the template above, we can get a transformation procedure for DNA replication:

```lisp
(defun dna-copy (input)
  (if (null input) nil
   (cons (dna-comp (first input))
        (dna-copy (rest input))))
)
```

dna-comp works on a single list element (a base in the strand) by changing A to T and G to C, and vice versa. It knows how to deal with a single element only, and is not concerned with transforming the list as a whole:

```lisp
(defun dna-comp (x)
  (cond ((eql x g) c)
        ((eql x c) g)
        ((eql x a) t)
        ((eql x t) a)
        (t (error
            "your dna is screwy"))))
)
```

**Student Question:** What does the `error` function do?

`error` is included in Common Lisp. It displays an error message and causes the interpreter to enter into a break loop. At this point, the user can issue debugging commands. Here is an example of how the above function responds to normal vs. error conditions:

```
> (dna-comp 'a)
T
> (dna-comp 'g)
C
> (dna-comp 'h)
Error: your dna is screwy
[break-1]>
```

Note that `error` requires an error message string as its single input.

4.2.1 Generalizing the DNA transformation procedure

Note that defining `dna-comp` external to `dna-copy` makes it very easy to generalize its applicability. For instance, if we wanted to transform strands of RNA instead of DNA, we could modify the single element transformer to work with RNA bases instead, and wouldn’t have to change `dna-copy` itself. By including the single element transformation function as an input to `dna-copy`, we can extend this idea of generalizability even farther. An example call would thus be:

```
> (dna-copy '(g a t t a c a) #'dna-comp)
```
And here is an example of a more generalized dna-copy:

```lisp
(defun better-copy (input fn)
  (if (null input) nil
   (cons (funcall fn (first input))
        (better-copy (rest input) fn))))
```

Now we have a function that takes any kind of single element transformer and applies it to any kind of list. This kind of function generalization is good programming practice because it ensures code re-use for commonly seen problems.

**Student Question:** What if people using our dna-copy have no idea whether they want to transform DNA or RNA? Can we generalize the function to account for that?

This is a very important question. What we are really asking is “how can we second-guess the user?”. The short answer is, we can’t. No amount of documentation, optional arguments, and default values can help when you are dealing with an end user who simply does not know what they want.

When you write a function, you can’t guess about what people might want to happen, or intend to happen. You need to have concrete, well-defined functions. Computers are not sentient things, and programs cannot guess what is intended. This is why it is very important that you clarify specifications and not rely on assumptions.

### 5 Transform functions are generalized in mapcar

The idea of transforming functions is such a common and useful one that a completely generalized one was included in lisp: mapcar. Here is an example of using mapcar to transform a DNA strand:

```
> (mapcar #'dna-comp '(g g c a))
(C C G T)
```

mapcar maps a transformation function onto each element in a list. Note that it returns a new list; it does not actually modify the original one. mapcar has no side effects.

Here is another example using mapcar to calculate square roots:

```
> (mapcar #'sqrt '(1 2 3 4))
(1 1.4142135 1.7320508 2)
```

**Student Question:** Can mapcar be used on nested lists?

mapcar itself works on top level elements only. For instance, the following counts the number of elements in each sublist:

```
> (mapcar #'length '((a b) nil (c g w) (e) (f)))
(2 0 3 1 1)
```
If you want to operate on the sublists themselves, you can do so by specifying an appropriate element transformation procedure that handled underlying organizational details. It can then be passed to `mapcar` which will then apply it to each top-level sublist. The important question to ask when working with `mapcar` is, “does my element transformation procedure actually do what I need it to?”.

5.1 You can write your own version of `mapcar`

Even though `mapcar` is included in Common Lisp, you could write your own version if necessary:

```lisp
(defun fake-mapcar (fn s)
  (if (null s) nil
      (cons (funcall fn (first s))
            (fake-mapcar (rest s)))))
```

`mapcar` is an example of a programming cliché; transformation procedures for lists are so common they become clichés. As you write your own programs, you may find some patterns continuously reappearing. Instead of re-implementing them from scratch each time, it is better to create a template function like `mapcar` which can be reused.

Relying on template functions, whether built-in or created, are an example of procedural abstraction. We don’t care about the low-level details of how `mapcar` transforms list elements; we just want to be able to tell it what to do by specifying how it should change each element.

6 Functions as arguments

We have seen how functions may be passed as inputs to other functions. For instance, we were able to pass the function `sqrt` to `mapcar` in an earlier example.

The function `funcall` is used to call another function. It is most often used inside a function to call a function passed in as input.

6.1 `funcall` vs. `apply`

`funcall` requires you to know the exact number of arguments a function takes. For something like the `+` function which takes a variable number of arguments, specifying exact inputs is difficult. `apply`, on the other hand, takes a list of arguments and applies a function to them, so you do not need to consider exact inputs.

7 Anonymous functions

Anywhere a named function is used, an anonymous function can be used instead. Anonymous functions are implemented by lambda expressions, which allow you to specify what operations need to be performed without having to explicitly `defun` a named function.

Anonymous functions can use information from the surrounding environment in a way that named functions cannot. We saw an example of how this feature is used by the Prism pen plotter output code. In it, graphic
primitives are sorted by color before drawing, to minimize the time spent on changing the pens within the plotter. Each plotter has different pens in different orders, so it became important to determine which plotter was being used. This was done by accessing the environment variable plt. Anonymous functions had to be used in order to do so.

Another example of using anonymous functions to access environment data was demonstrated in the bank-account function, which uses a lambda expression to access the local captured variable balance. Lambda functions have the benefit of showing exactly what needs to be done at a certain code point. Thus, if you’re only going to use something once, it can be convenient and make code easier to read if you use an anonymous function. However, if you find that you are continuously re-using the same lambda expression, it is better to instead use a named function.

8 fibonacci and efficiency

Being able to break down problems and solve them using recursion is an extremely important concept. Recursion is a powerful tool, but we have seen some instances of recursive functions where the computational and system resources required by them make them slow and inefficient.

The most notable example we have seen of this has been the fibonacci function. As originally defined, it repeatedly computed the same values, causing severe slowdowns and high memory requirements. Tracing the original fibonacci function demonstrates exactly where these repeat calculations occur.

In class we saw a way to dramatically increase fibonacci’s efficiency by having it cache intermittent results instead of continuously recalculating them. The memoize function was used to produce a fibonacci version that did just that.

Intermediate values are stored in a hash table in the improved fibonacci function. Hash tables can be thought of as extensions of association lists, where each key is associated with a value. Hash table utilities are already built into lisp, so they are easily included.

The memo function uses a lambda expression to create a function that contains a hash table. Because the hash table needs to be accessed after it is created, it has to be “kept around” by creating reference to it, just as if it were a piece of data. This is what the memoize function does.

Thus when fibonacci is memoized, it gets a new function definition. This new definition includes caching abilities, and thus causes fibonacci to run much more efficiently.