F.G.H.
5/2/85

LispCourse #24: Data Abstraction

Programs as Representations of the Real World

Extensionally, computer programs carry out a sequence of information manipulating actions on a set of computational objects called data.

Intensionally, computer programs the actions and objects are usually designed to be some sort of model of some "real world" actions and objects. In other words, computer programs are ***representations*** of real world actions and objects.

Data is used to represent real world objects. Data structures represent the structure of real world objects.

In talking about computer programs, we often blur the distinction between data structures and the real world objects which they represent.

Data, Compound Data & Data Structures

Atoms are the ***primitive data*** structures in Lisp.

Some objects can be represented by atoms alone; e.g., numbers by NUMBERPs and English words by LITATOMS.

However, most objects are more complicated and can only be represented by a combination of many simpler pieces of data. In Lisp, this combination is usually achieved using lists that "glue" together atoms and other lists. A list is ***compound data***.

A ***data structure*** is a "scheme" for using compound data to represent a complex object.

Example:

A persons name is an object with some structure, i.e., it has a first name, a middle initial, and a last name. In Lisp, we might represent this structure using a list data structure with three elements. where the first element was an atom representing the first name, tyhe second element an atom reprersenting the middle initial, and the third element an atom representing the last name.

Compound data is important because it allows us to deal with the many pieces of simpler data as a single entity, just as we deal with the real world complicated object as a single entity.

Data structures are important because they allow us to take a very simple and straight-forward compounding scheme (i.e., lists) and represent very complex objects. The data structure provides the "rules" by which to interpret a list structure into representation of a complex object.

Data Abstraction: Dealing with Compound Data

Dealing with compound data and data structures can lead to difficult programming if the proper rules are not followed.

The Need for Data Abstraction

Example 1

Consider the following programming problem:

> write a set of functions that fill out tax forms. There will be one function for each tax form. On each tax form, you need to fill in the person’s name at least once and sometimes two or three times . The name is passed to you as a list of three items of the form *(First Middle Last)*.

How should you handle the placing of names on the tax forms?

One solution is the following: every time you need to print the persons name, print the CAR of the name list, then the CADR of the name list and then the CADDR of the name list.

What happens if for some reason you start getting names in the form *(Last First Middle)*.

> You would have to go through each function looking for each place that you print out the person’s name and change the procedure to be CADR of name list followed by CADDR of name

list followed by CAR of name list.   UGH!! This could be a god-awful job.

BUT - what if you had done the following to begin with: Write three functions called **FirstName**, **MiddleName** and **LastName** that take a name and return the indicated part of the name. FirstName would simply take the CAR of the NameList, the MiddleName function the CADR and the LastName the CADDR.

Then every time you want to print the person's name, you would print *(FirstName NameList)* followed by *(MiddleName NameList)* followed by *(LastName NameList)*. Everything would work as in the initial case above.

However, when the name format change can along, you would have to change only the three functions **FirstName**, **MiddleName** and **LastName** to use CADR, CADDR, and CAR respectively.

None of the tax form function would have to change at all!!!!
Thus the change in name formats would be trivial.

Example 2

Consider the following programming problem:

You are writing the program for filling out Schedule G (Income Averaging). You have a record of the person's last 4 years' tax liabilities in the form *(NameList Year-1 Year-2 Year-3 Year-4)*. In your calculations, you need the Year-i tax liability.

Solution: you write four functions called **Year-1** thru **Year-4** that take the liability list and return the Year-I tax liability. You implement these functions by taking the (CAR (NTH LiabilityList N)) for the correct N in each case.

BUT - what if the liability list format changed a bit to be *(LastName Firstname Year-1 Year-2 Year-3 Year-4*). Then you would have to go back and change all four functions.

However, if you had written a function called **ListOfLiabilities** that took the liability record and returned a list of the four liabilites. Then you could have written the **Year-I** functions to use the value returned by **ListOfLiabilities**.

If you did this, then when the liability list format changed, you would have had to change only one function, **ListOfLiabilities**, instead of four.

Not a big deal, but if there were 100 variables instead of 4 it might be a big deal.

Data Abstraction

What do these examples show? The need for data abstraction to improve program maintainability and adaptability.

Data abstraction is a programming design methodology in which:

1) You carefully separate functions that **USE** data from functions that **ACCESS** and manipulate data structures. Changes to the data structures will then affect only the data structure functions and not the data using functions.

2) You carefully separate functions dealing with the various levels of a compound data structure. Changes to one level will then effect only the functions dealing with that level of the compound data and not all other levels.

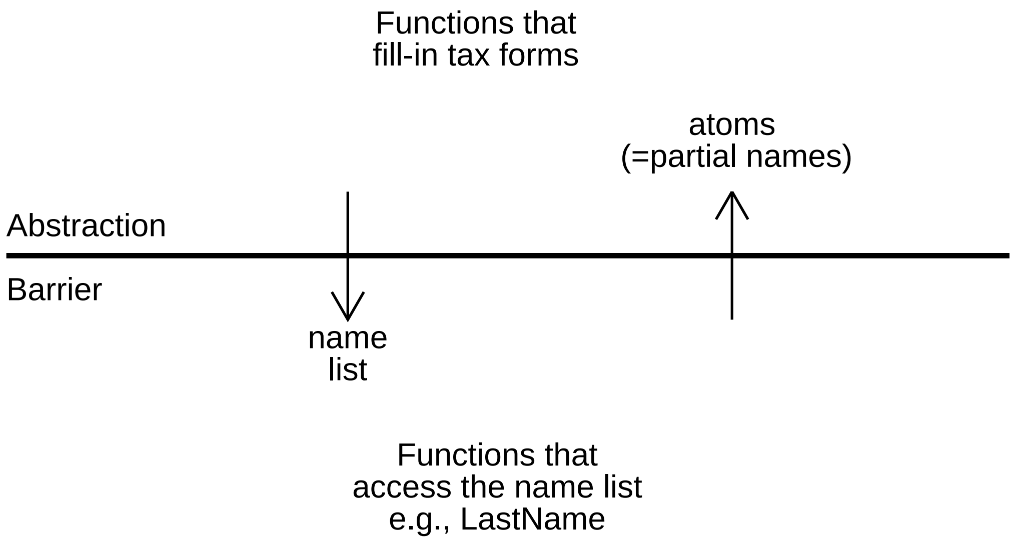
Abstraction Barriers

Data abstraction is accomplished by creating ***abstraction barriers*** between function that use data and functions that access data AND between the functions that access various levels of data from a compound structure.

An abstraction barrier is a line that separates the functions of a program. Functions on one side of the line can make no (unnecessary) assumptions about the data structures used by functions on the other side of the line, except as described in the well-defined interface between the two sets of functions.

Example

In the tax forms/name list example above, an abstraction barrier was required between the form filling programs and the name list data structure.


Functions that fill-in tax forms
atoms (=partial names)
Abstraction
Barrier
name list
Functions that access the name list e.g., LastName

The form filling programs should not make any assumptions about the format of the name list (or even that it is a list instead of an atom or a string or whatever).

Instead, if they need to access the name list structure, they should call the functions (FirstName, LastName, etc) on the other side of the abstraction barrier that deal specifically with accessing the name list structure.

There should be a well defined protocol that states that whenever a data using function (i.e., one that fills in a tax form) calls an access function (e.g., LastName), it passes down a name thing and gets back an atom that represent the first, middle or last name, as indicated.

How this is to be accomplished is entirely up to the access function.

The using function should make no assumptions what-so-ever about the format of the name thing.

Similarly, the access functions should make no assumptions about how the atom they return will be used.

Constructing Access Functions: Constructors, Selectors & Mutators

The trick in writing a program with strong abstraction barriers is to write a good set of access functions for isolating the access to your data structures from the rest of your program (and for isolating the access to low levels of the data structure from functions that deal with higher levels of the data structure).

These access functions serve as the interface between the rest of your program and your data structures. If your program wants access to the data structures, it has to call one of these access functions.

The access functions take data in an agreed upon form and do whatever work is necessary to translate it into a form compatiable with the chosen data structures.

They also take data from the data structures and do whatever work is necessary to translate it into an agreed upon form to pass back to the rest of the program.

Constructors, **Selectors and Mutators**

There are basically three kinds of access functions: **constructors, selectors** and **mutators**.

Constructors take individual pieces of simpler data and build a data structure from them, returning the data structure.

Example: **(MakeName *First Middle Last*)** takes three atoms and returns a name data structure.

Selectors take a compound data structure and return individual pieces of data from that data structure.

Example: **(FirstName *NameList*)** takes a name data structure as an argument and returns the first name portion of that name data structure.

Mutators take a compound data structure and alter individual pieces of data in that data structure.

Example: **(ChangeFirstName *NewFirstName*)** takes a name data structure as an argument and returns the same data structure with the first name portionaltered to be *NewFirstName*.

A complete set of access functions for a name data structure might be the following:

Constructor: **MakeName** takes three atoms representing the first, middle and last names and returns a name object.

Selectors: **FirstName** takes a name object and returns the first name. **LastName** and **MiddleName** do the corrsponding thing.

Mutators: **ChangeFirstName** takes a name object and returns the altered name object. **ChangeLastName** and **ChangeMiddleName** do the corrsponding thing.

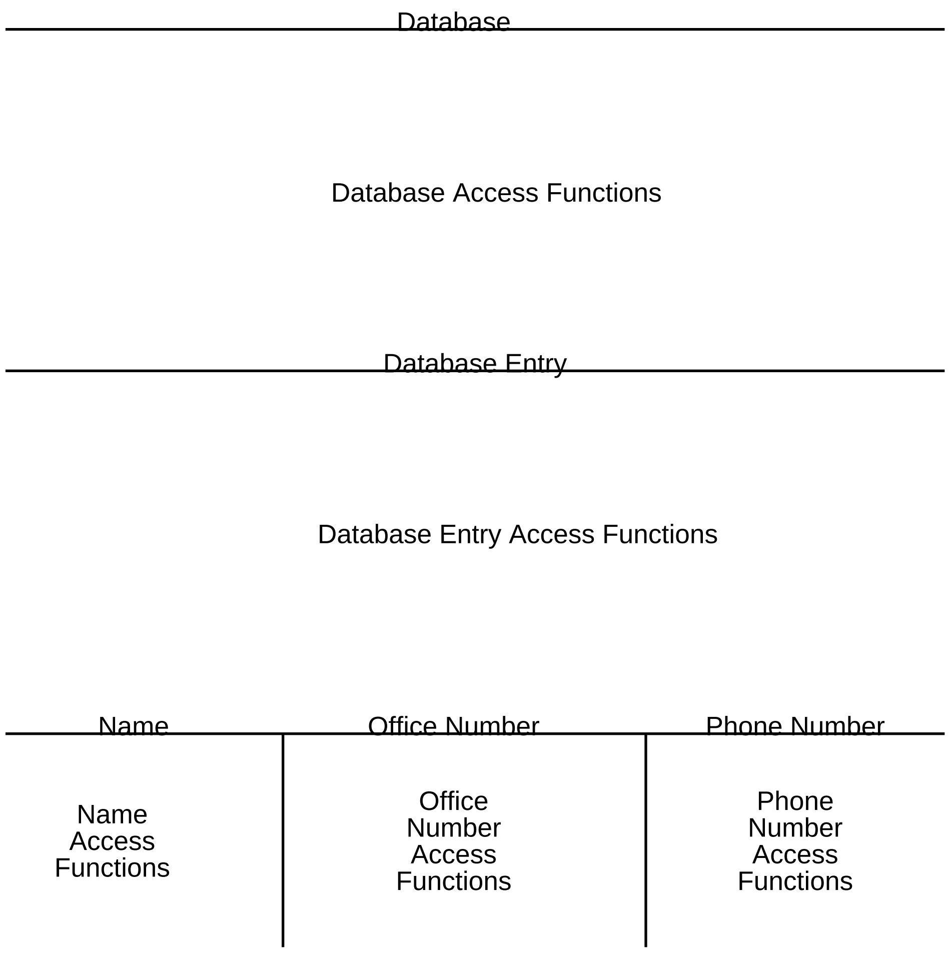
In a program with good data abstraction, any function that wanted to create a new name would have to call **MakeName**. Any function that wanted to access part of the full name would have to call the appropriate selector function. Any function that wanted to change part of the name would have to call the appropriate mutator function.

Other than these seven functions NO other functions in the program could make any assumptions about the format of the name object.

Example: Problem from LispCourse #23

Attached is a solution to the problem at the end of LispCourse#23 that makes strong use of data abstraction, both to separate the use functions from the access functions AND to separate the access functions that deal with different levels of the database data structure.

A diagram of the abstraction barriers in this program is the following:


Database
Database Access Functions
Database Entry
Database Entry Access Functions
Name
Office Number
Phone Number
Name Access Functions
Office Number Access Functions
Phone Number Access Functions

References

Winston & Horn, Chapter 5, pages 97 thru 100

Sussman & Abelson, Chapter 2, pages 71 thru 88 (and beyond)

Exercises

Rewrite your program from LispCourse #23 using good principles of data abstraction. Try changing the format of you database entries and seeing how many functions you have to change to adapt to this format change.

Do this ***before*** you study in detail the attached example program.