COMP 200: Elements of Computer Science
Fall 2004
Lecture 12: September 22, 2004

Data Structures & Abstraction

On the Board

Homework 2: back in class today, leftovers outside DH 2065
Homework 3: due next Monday
First Test: out today, due nextWednesday
2 hours, take home, 3 questions + extra credit
Focus on short programs, as in Homeworks 2 & 3

Final Comment on Lists

In class, we have built lists by constructing our own "list" construct using ***define-struct***. As Homework 3 points out, lists are a sufficiently important part of Scheme that Scheme provides a universal list constructor and a set of access functions (or selectors). The constructor is ***cons*** — short for constructor — and the accessors are ***first*** and ***rest*** with the obvious relationship to the ***num*** and ***others*** elements that we built into our list-of-numbers structure. You should use the Scheme list constructs whenever you need a list. You should still write a data definition — just the comment part — to remind yourself of the kinds of data involved and the recursive structure of the list (the red arrow).

Using the Scheme list construct, we could rewrite our program Total as

```
;  a list-of-numbers is a either
;     empty
;  or a structure
;     (cons first rest)
;  where first is a number and rest is a list-of-numbers
;  (We will use Scheme's builtin list construct)

;  Total: list-of-numbers → number
;  Purpose: computes the sum of the values in an input list
(define (Total fee)
    (cond
       ((empty? fee)     0)
       (else  (+  (first fee) (Total (rest fee))) )  ))
```

Talk about termination
— list is finite
— empty case is first
— correctly detect empty implies it halts

Data Structures and Abstraction

Computers don't directly understand Scheme, or the pidgin Algol notations that I use on the board, or English, or flowcharts (as in the book). Instead, they operate on extremely low-level constructs and data. Unfortunately, humans are quite bad at keeping track of low-level data. We have limitations in the way that we think which make it difficult for us to track too many details. Thus, we deal with computers (and with the world) by creating abstractions that let us reason at a higher level.

The control structures and data structures in Chapter 2 are an example of such abstractions. Computers don't really implement **if-then-else** or **bounded iteration** or **recursion**. They do implement **sequencing** and the ability to move from one sequence to another. Our control structures mirror some patterns of logical (or mathematical) thought that people have found make reasonable building blocks for programs.

Similarly, data structures are abstractions that let us organize collections of data in ways that make sense to us. Scheme structures (***define-struct***) do nothing more than group together some values and attach symbolic names to them. This lets us think of them at a higher level — a grade record, a list element, *et cetera*. The *Keys* vector in our bubblesort example from last class gave us some properties that we needed — a list of elements in which we could access arbitrary elements (at constant cost) and could change their values (to rearrange the elements). Change either assumption and the efficacy of Bubblesort goes downhill.

We didn't worry about what it took to implement the *Keys* vector. That part of the abstraction is someone else's worry — the implementers of Dr. Scheme (or whatever programming system we use) must make sure that vectors work correctly and that they have the implied efficiency for random access. In principle, the programming language is a social contract between the programmer and the language implementer that ensures faithful adherence to the original program. (Break the contract and many programs no longer work.)

The reason that we introduce data structures is to allow us to reason about problems and programs at a higher level. By thinking of them in terms of higher level structures, we simplify our approach, play to the strength of the human mind, avoid its weaknesses with regard to detail, and leave the nitty-gritty implementation to automatic tools.

Stacks and Queues

The next part of chapter two introduces us to a series of data structures. We've already talked about variables (objects that hold a value), vectors, and arrays (multidimensional generalizations of a vector such as *Keys*.)

Continuing to think about abstractions, how would we abstract a list? We already have. We have three operations on a list:

cons : object list → list
first: list → object
rest: list → list

We know a lot about the meaning of these three abstract operations from our experience writing programs that manipulate lists. What about an array? We already have two abstract (or not-so-abstract) operations on an array:

Referencing a value of the array — Name[i,j] gives us the value in location *[i,j]* of array *Name*

Updating a value of the array — assigning to Name[i,j] changes the value that a reference to *Name[i,j]* will return

We also need a constructor, some way of creating an array of a given size.

The book presents stacks and queues. We can design abstract interfaces for each of these data structures as well.

Stack needs four operations (perhaps five)

Push(x) → makes x the top element of the stack
Pop → returns the top element of the stack
IsEmpty?→ returns true if the stack contains no objects, otherwise, it returns false

And a constructor. Some implementations find *Swap* useful; it exchanges the top two elements of the stack

Note that *Pop(Push(x)) = x*, *IsEmpty(Push(x)) = false*, and so on… We can build a set of algebraic axioms that completely specify stack behavior. (We call it an *axiomatic specification* for a stack.)

We could implement a *stack* as two programs that operate on a list in Scheme if we just had one more capability— a mechanism to change the value of a define'd object. Imagine the following situation (if that makes sense in a programming language).

```
 ; StackBase will hold our stack
(define StackBase empty)

;  push:  number → Boolean (true or false)
;  Purpose: adds its number to the top of the stack
(define (push x) …)

;  pop:  void → number
;  Purpose: removes top element from stack & returns it
(define (pop ) …)
```

We can write out implementations in our pidgin-Algol pseudocode.

```
push( x )
   StackBase ← (cons x StackBase)
   return true

pop()
   temp ← (first StackBase)
   StackBase ← (rest StackBase)
   return temp
```

Writing this code in Scheme requires us to introduce a feature that we have avoided so far — the equivalent of the assignment arrow in our pidgin-Algol notation. In Scheme, we can assign to any object created with ***define***. To do so, we use the Scheme program ***set!*** We will also need to use ***begin*** to sequence multiple Scheme expressions.

```
; StackBase holds our stack
(define StackBase empty)

; pop needs a temporary home for the top stack element
(define TOS 0)

; push : number -> boolean
; Purpose: adds its parameter to the front of the stack maintained at
StackBase
(define (push x)
  (begin
    (set! StackBase (cons x StackBase))
    true
    ))

; pop : void -> number
; Purpose: removes the first element from the stack at StackBase & returns it
(define (pop )
    (begin
      (set! TOS (first StackBase))
      (set! StackBase (rest StackBase))
      TOS))
```

Queues are quite similar to stacks, except that the *Pop* operation returns the oldest element in the queue rather than the youngest element in a stack. A queue implementation is trickier than that for a stack. (The book discusses a two-list implementation of a queue.)