COMP 200: Elements of Computer Science
Fall 2004
Lecture 11: September 20, 2004

Reconnecting with the Book

On the Board

Homework 3: due next Monday
First Test: out Wednesday, due following Wednesday
2 hours, take home, 3 or 4 questions
Focus on short programs, as in Homeworks 2 & 3

Why did we buy that textbook?

The last three weeks have been an intense short course in writing small programs in Scheme. From this point forward, I expect to spend about 2/3 of the time on matters from the book and 1/3 of the time on programming. Today, we'll try to reconnect with Chapter 2.

Control Structures

We've already talked about this notion. The book mentions sequencing, which each of you used in Homework 1. It mentions conditional execution, in the form of an ***if-then-else*** construct. While Scheme has such a construct, we prefer to use its ***cond*** structure, which has a flat structure rather than a hierarchical structure. The book's third control structure is iteration. In Scheme, we have seen iteration implemented as structural recursion over lists. (We'll look at structural recursion over other inductive domains later this week.) The book also describes bounded iteration in the form of a ***loop*** over some pre-specified range.

for i ← 1 to n
do something of value

We will use this construct in class and in homework. We won't program this way in Scheme, because such loops are not a natural (integral?) part of Scheme. (In particular, Scheme provides poor support for arrays. Most of the examples you will see in the book that use bounded interation also use vectors (1-dimensional arrays) or arrays.)

*An example of bounded iteration — **Bubblesort***

Chapter presents a simple sorting algorithm as its example to illustrate the combined use of sequencing, conditional execution, and bounded iteration. Bubblesort is one of the simplest sorts we can envision — at least in terms of its implementation. It is known as an "exchange sort" because the fundamental operation in bubblesort is swapping two items that are out of order. The concept is simple: given a list (or vector) of items to sort, the algorithm runs over the items and compares adjacent items. It they are out of order, it swaps them and continues. Large items shuffle toward their end of the array; small items shuffle in the other direction.

Assume a preexisting list of N items, stored in a vector "Keys".

The notation Keys[i] refers to the i^{th} element of the vector Keys.

```
for i ← 1 to N-1

   for j ← 1 to N-1
        if Keys[j] > Keys[j+1]
          then swap the contents of
                Keys[j] & Keys[j+1]
```

Describe the inner loop first. Then work outward to the number of tims that it must run.

Work a simple example of eight numbers. Point out that it takes $(N-1)^2$ comparisons and (potentially) swaps. We can make the inner loop shorter each time, cutting the total cost in (roughly) half. We will come back to this point in several weeks when we discuss algorithmic complexity.

Consider a list of numbers that are already in ascending order. The discussion should lead to a version that stops when no swaps occur.

```
Swapped ← true

while (swapped)
swapped  ← false

   for j ← 1 to N-1
        if Keys[j] > Keys[j+1]
          then
                swap  Keys[j] & Keys[j+1]
                swapped ← true
```

Following book's convention that indentation indicates control. Both these execute under the control of the ***then*** clause

Even though it looks as if the *while(swapped)* loop can run indefinitely, we know from our earlier reasoning that swap can be true at most *N-1* times. Thus, the algorithm will halt after at most *N* iterations of the outer loop; in

exchange for that extra iteration (to detect that *Keys* is in order), it will sometimes halt in fewer than *N-1* iterations.

Back to Control Structures — The "Subroutine"

The book contains a lengthy discussion of "subroutines" or "subprocedures" or "functions" and argues that using them makes sense. To a new reader, these ideas appear to come out of nowhere. However, we've already talked about them in class. When discussing Homework 1 (and reading Chapter 1) we talked about the fact that different levels of abstraction are possible and desirable when expressing an algorithm.

Consider our Bubblesort algorithm. What does it mean to "swap" two elements of *Keys*. The book uses destructive assignments to accomplish this action — that is, they write down notation such as

Keys[j] ← *Keys[j+1]* and

j ← *N-1*

In our Scheme programming, I have carefully avoided this explicit representation of state — we have been writing (essentially) stateless programs that have the same behavior on the same input every time. (The goal is to adhere, in our programming exercises, to the clean, crisp world of algebra, where expressions have well-defined and somewhat intuitive meanings).

In the stateful world, assignment views the data object as a box that contains a value. Thus, *j* looks like a box that can hold one value and *Keys* looks like a collection of boxes that can hold *N* values. The assignment operation, ←, replaces the value named on its left side with the value of the expression on its right side. The assignment

Keys[j] ← *Keys[j+1]* and

leaves the value of *Keys[j+1]* intact and copies it into *Keys[j]*. Of necessity, this operation destroys the value that was in *Keys[j]*. Hence, we call this operation a *destructive* assignment or a *destructive* update. We hid all of this baggage behind the statement "*swap Keys[j] and Keys[j+1].*" To implement that operation requires three assignments:

TemporaryHome ← *Keys[j]*
Keys[j] ← *Keys[j+1]*
Keys[j+1] ← *TemporaryHome*

We could write these three lines into the algorithm, indented in an appropriate way (of course) and have a more complete specification. Doing so, however, complicated the expression of the algorithm and makes it harder to read and understand. An alternative approach would be to create a small program (subroutine) called swap that takes two elements of *Keys* and swaps them.

Swap(a,b)
  TemporaryHome ← *Keys[a]*
  Keys[a] ← *Keys[b]*
  Keys[b] ← *TemporaryHome*

Now, we can replace the line in the algorithm that tells us to swap two keys with an invocation of "Swap" — written "*swap(j,j+1)*" — and produce what is clearly a more completely defined algorithm ***without losing the clarity that comes from using a higher level of abstraction to avoid the details.***

The key in choosing a level of abstraction is to ensure that we do not hide algorithmic complexity in one of these abstract operations, such as swap. If the implementation of swap ran over the entire *Keys* array (for some reason) or did it multiple times, then the use of abstraction would reduce our understanding of the algorithm — in particular, its cost or running time.