While this looks complicated for a simple list, it is not, it is just a bit lengthy. XML is flexible enough to represent and communicate very complicated structures in a uniform way.

Implementation of Lists

There are many different *implementations* possible for lists, and which one is best will depend on the primitives offered by the programming language being used.

The programming language *Lisp* and its derivates, for instance, take lists as the most important primitive data structure. In some other languages, it is more natural to implement 14 lists as arrays. However, that can be problematic because lists are conceptually not limited in size, which means array based implementation with fixed-sized arrays can only approximate the general concept. For many applications, this is not a problem because a maximal number of list members can be determined a priori (e.g., the maximum number of students taking one particular module is limited by the total number of students in the University). More general purpose implementations follow a pointer based approach, which is close to the diagrammatic representation given above. We will not go into the details of all the possible implementations of lists here, but such information is readily available in the standard textbooks.

Preview from Notesale.co.uk
Page 4 of 10

3.2 Recursion

We previously saw how iteration based on for-loops was a natural way to process collections of items stored in arrays. When items are stored as linked-lists, there is no index for each item, and *recursion* provides the natural way to process them. The idea is to formulate procedures which involve at least one step that invokes (or calls) the procedure itself. We will now look at how to implement two important *derived procedures* on lists, **last** and **append**, which illustrate how recursion works. To find the last element of a list l we can simply keep removing the first remaining item till there are no more left. This algorithm can be written in *pseudocode* as:

```
last(l) {
   if ( isEmpty(l) )
      error('Error: empty list in last')
   elseif ( isEmpty(rest(l)) )
      return first(l)
   else
      return last(rest(l))
}
```

The running time of this depends on the length of the list, and is proportional to that length, since last is called as often as there are elements in the list. We say that the procedure has *linear time complexity*, that is, if the length of the list is increased by some factor, the execution time is increased by the same factor. Compared to