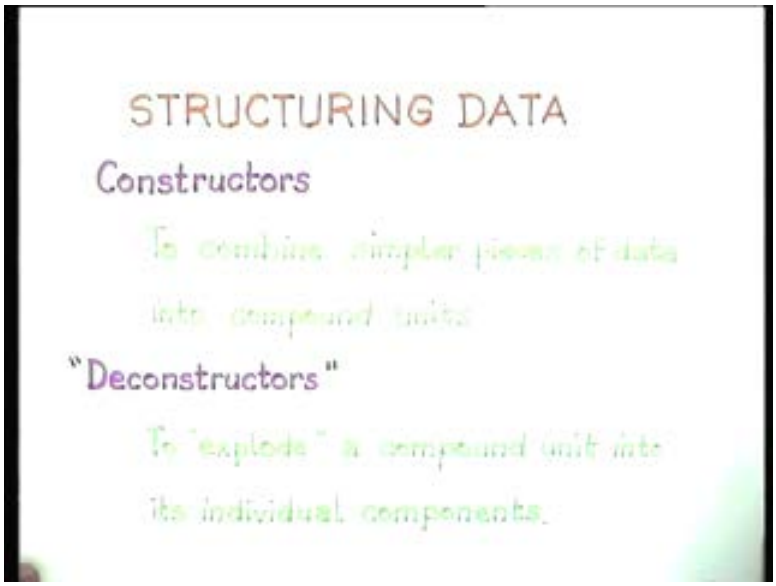
Principles of Programming Languages
Prof: S. Arun Kumar
Department of Computer Science and Engineering
Indian Institute of Technology
Delhi
Lecture no 22
Lecture Title: Sequences

Welcome to lecture 22. Last time we started on structured data and we looked at essentially Cartesian products and their variants. It corresponded approximately to records and variant records. We will look at sequences. Let me just briefly recapitulate what we will have to do while structuring data.

(Refer Slide Time: 01:10)



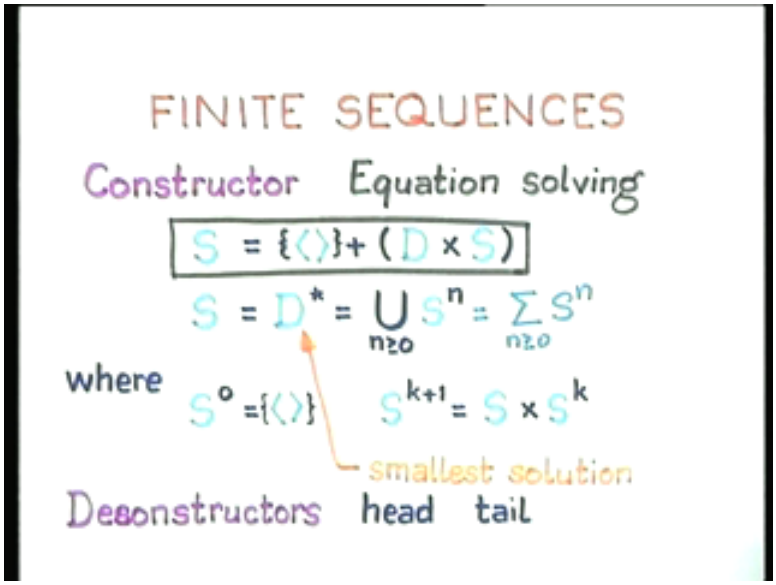
As I said one important point is to find constructors to combine simpler pieces of data into compound units and then find deconstructors which explode a compound unit into its simpler components. We should look at it this way. Long time ago the mathematician and philosopher Rene Descartes said that after addition and multiplication the most important human intellectual activity is equation solving. That is also how school curricula are designed. But of course equation solving also means finding inverses of additions and multiplications so you also have subtractions and divisions. Equation solving is really the most fundamental activity. After these four operations you could think of equation solving as the fifth operation which is part of any branch of mathematics.

The main constructor is that of an equational definition. Given some data domain D, I can define this equation in terms of the unknown S. So S is the unknown and the question that we are asking is what the solution of S is for such an equation. You might be asking actually what the possible solutions of S are which satisfy this equation.

If you look at it that way one solution or what we might call the least solution is that of D * and what is D *? D * is defined as let us say the union of Sn where S 0 is the singleton set consisting of the empty sequence and S^{k+1} is S Cartesian product S^k.

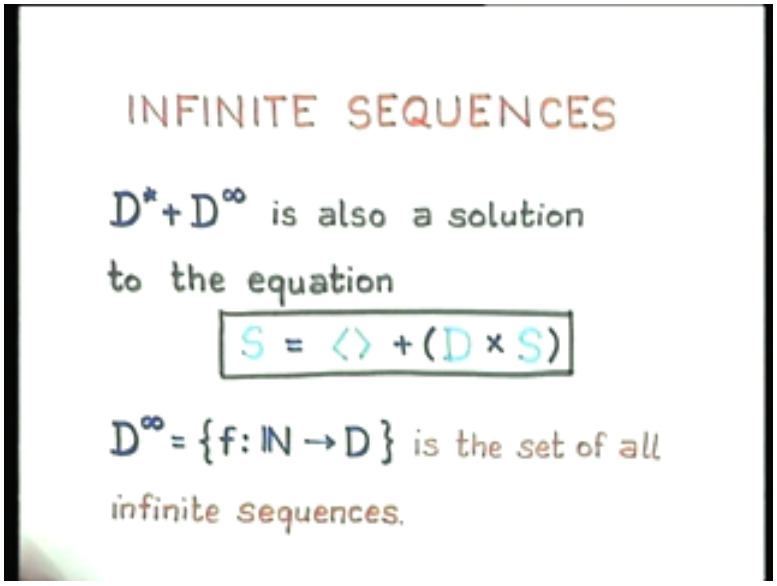
The solution to this equation is actually what might be called the least solution. It is in fact the smallest set S which when you substitute on both sides for S would yield an equality. Since it is actually the construction of ordered pairs, you construct S^{k+1} which is an ordered pair consisting of two elements; the element is an element of D and the second element is an element of S^k. That is the way you should look at it.

(Refer Slide Time: 05:39)



This is in fact the natural motivation why most of these functional languages actually provide some form of head and tail functions because a sequence is really an ordered pair whose second component might be another ordered pair. The deconstructors are just the head, tail etc. The constructor really is the ‘cons’ operation which comes from here and the deconstructors are just the head and tail functions. If you look upon it as just equation solving then this is not the only solution. There are other solutions to this equation for example; you could have infinite sequences also as solutions to this equation.

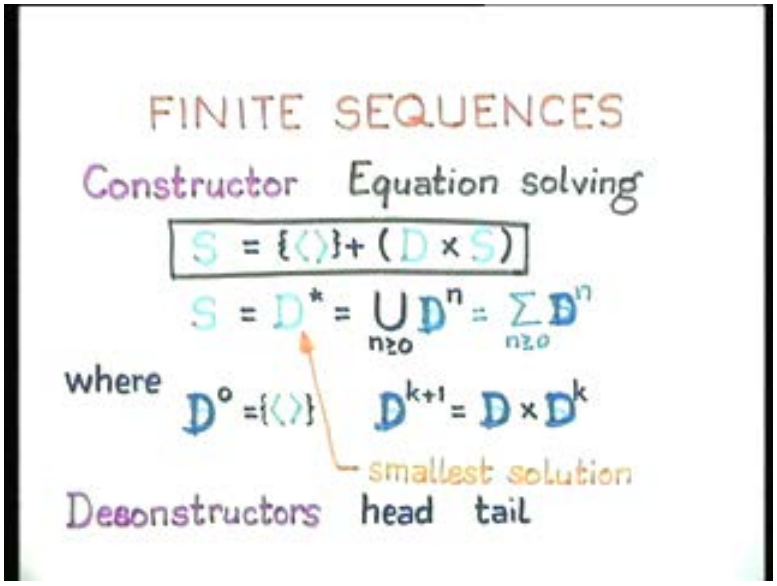
(Refer Slide Time: 06:15)



Consider $D^* + D\infty$ where $D\infty = \{f : IN -> D\}$.
For the moment let us look at it. Then this $D^* + D\infty$ is also a solution to this equation and this solution is in fact the larger solution to this equation and you could regard infinite sequences as an ordered pair whose first element is an element of D and whose second element is another infinite sequence. The constructors and deconstructors that were used for this are equally applicable also for infinite sequences. Although we will not be looking at infinite sequences in some detail they are actually present in some form in ML.

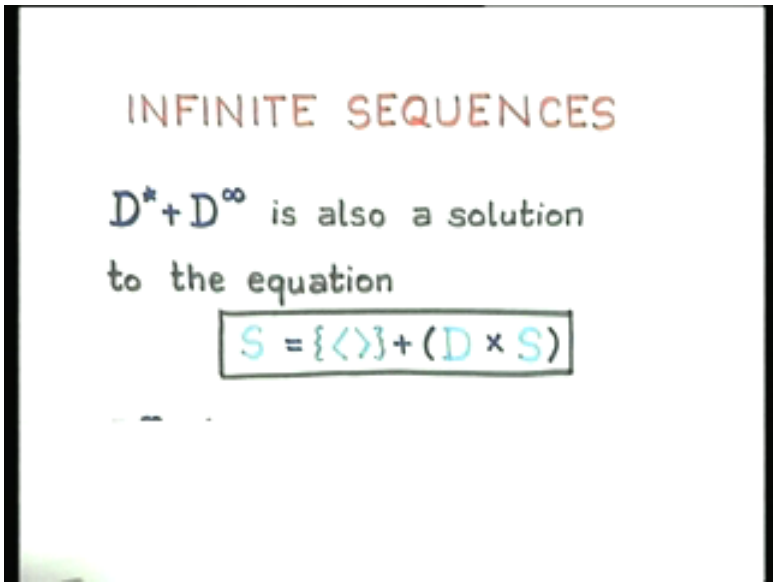
The lazy evaluation mechanism of ML actually has these infinite sequences. S here is the unknown variable. $S = D^*$. This is the empty sequence. This is $D \times S$. A solution for this equation I said was D^n where D Not is the empty sequence. $Dk+1 = D \times Dk$.
I used S through out though what I had written was also not wrong. Let me clarify that that is also equally correct. For any solution if you have done these operations you would have got that also to be a solution of this equation. That is immaterial. That is not the least solution. The least solution is that which is obtained by using D.

(Refer Slide Time: 08:42)



There are other solutions. If you add infinite sequences also then you get another solution which might be called the greater solution. This equation really has only two solutions. There are really no other solutions. Normally you do not see a set definition which is recursive but in fact computer science is full of set definitions which are recursive. If you take a context-free grammar of this form, $S->01/0S1$ the question that you can ask is what is the language generated by this grammar?

(Refer Slide Time: 9:41)



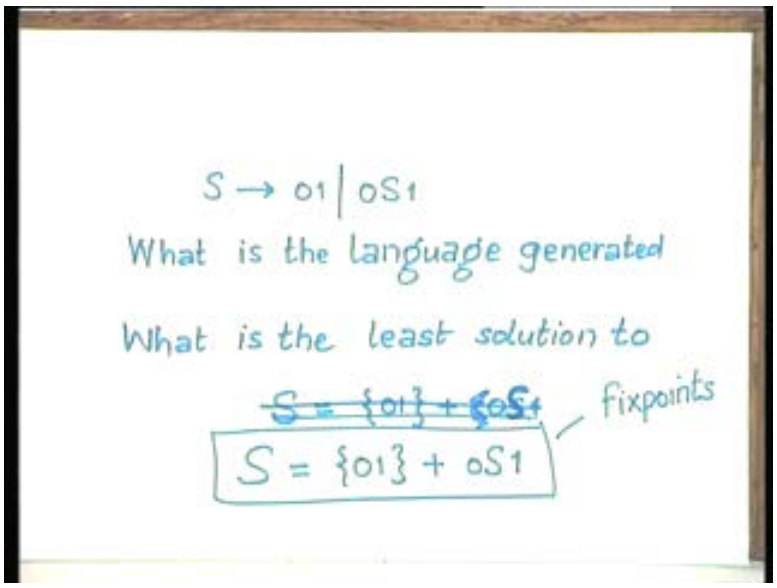
If you ask the question what is the language generated by the grammar you might equally well ask the question what is the least solution to the equation given by this?

$S = \{01\} + 0S1$

That is why equation solving is absolutely basic where of course I have used a very loose notation. This S is presumably supposed to be a set and I have used 0 and 1 around S to denote that every element of S is padded on the left with 0 and on the right with 1. I have just generalized the normal prefixing or suffixing of strings to sets of strings. You can ask this question; what is the least solution to this equation? In fact that is the language generated by the grammar.

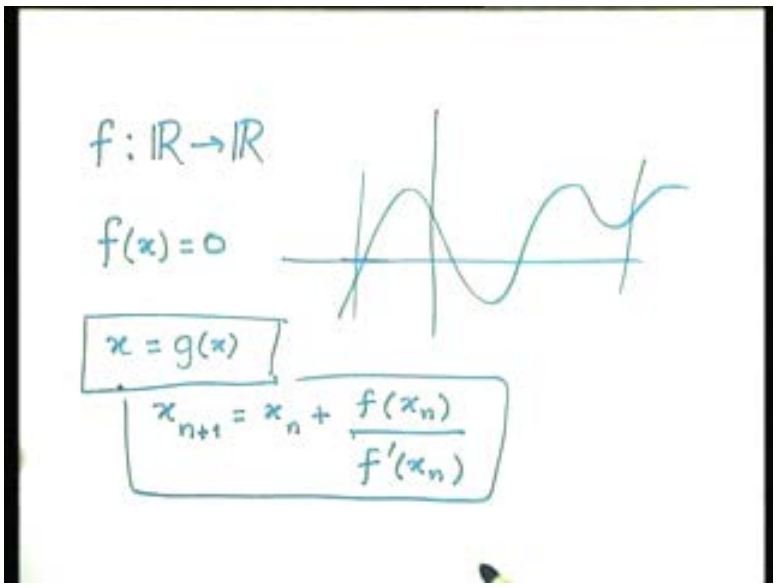
Equation solving is absolutely fundamental and almost all things that we talk about computationally are instances of equation solving. In fact most of the equation solving that we do in computer science is of finding what might be called fixedpoints.
When we talk of recursive definitions in particular the grammar that we have defined is also a recursive definition. This set is defined in terms of itself and so it is a recursive definition.

(Refer Slide Time: 13:39)



All recursive definitions are essentially equations which have to be solved and the meaning of finding a solution to a recursive definition is to find a non recursive definition which will satisfy the equation. That means finding a fixpoint. What is a fixpoint? Take a look at your standard methods. Let f be a continuous differentiable function over this interval on the reals so then $f : IR -> IR$. You want to find where this curve intersects the X axis. You are essentially solving the equation $f(x) = 0$.

(Refer Slide Time: 15:48)



In order to solve this equation $f(x) = 0$ f, which involves finding the root of this function, you often manipulate things so that you get an equation of the form $x = g(x)$. Way back in the 16th century somebody said that the solution to this equation is defined by the recurrence. So, you can find the root by doing some transformations and get an equation of this form which is very much like giving a recursive definition to x.

$$xn+1 = xn + f(xn) / f1(xn)$$

I could look upon this as giving a recursive definition to x and given a recursive definition to x, I give an iterative solution which is a recurrence. So, I can use recurrences in order to give iterative solutions to what might be called fixpoint equations. This is standard on real numbers. There is absolutely no reason why one should not do it on sets. These solutions that I am talking about are in fact also recurrences in that same form. If you look at it analogously this is a recurrence which defines an iterative solution to this fixpoint equation. This is clearly an iterative solution and most iterative solutions will give you solutions to fixpoint equations.

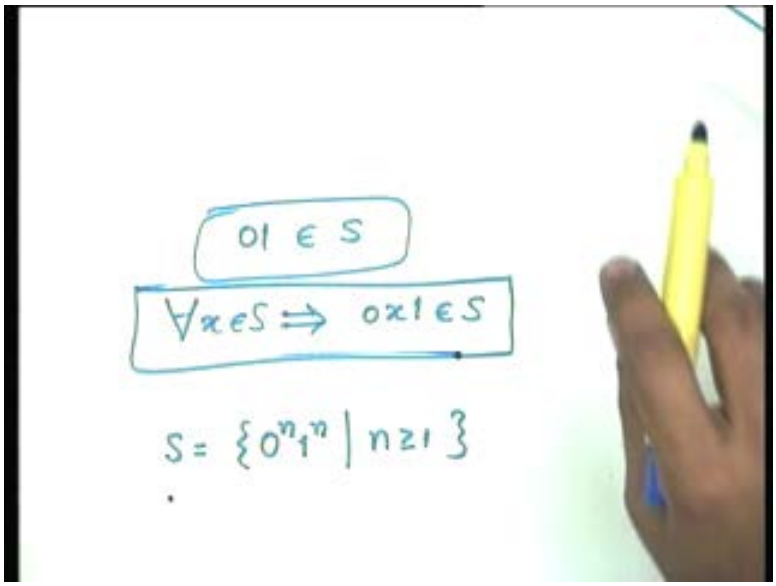
When we are talking about the language generated we are talking about the least solution that will satisfy this equation in the sense that we want the least solution which is closed under the operation of prefixing and suffixing of 0 and 1 respectively.
What is the least set S such that 0 1 belongs to S and if x belongs to S then 0 x 1 also belongs to S. So, there is a closure property that has to be satisfied.

$$01 \in S$$
$$\forall x \in S \Rightarrow 0x1 \in S$$

This is an important closure given by this definition and the least solution that you can think of is all the possible purely nested bracket matchings of 0 1's. So, if you go through an iterative solution the language generated by this grammar is just 0 raised to n 1 raised

to n where $n \geq 1$. We have the set S. It is the least solution because I am not for example permitting infinite sequences.
If you permit infinite sequences there is a possibility of getting cardinality that is greater than a left Not. You are actually finding accumulation points or limit points of sequences. I do not know whether you have done all these sequences but it is a topology which is completed. This is the least solution in the sense that it has to satisfy these two closure properties. You take any set smaller than this and I can find that there is an element in that set which does not satisfy one of these properties. So, we are usually talking always of least solutions though not necessarily always now but we are always talking of least solutions which satisfy such closure properties.

(Refer Slide Time: 20:34)



If you were to take this constructor for example; such a domain equation, then no set smaller than D * will actually do for you. If you take any set smaller than D * either by removing some elements or just by putting some bound n then I can find a sequence which does not satisfy these closure properties.

For example; I can take an n element sequence. For any n element sequence it does not satisfy the property that if I take that n element sequence and an element of D as an ordered pair, then that ordered pair belongs to the set S because it is a $n+1$sequence.
The smallest solution that you can get is only the set of all finite sequences of elements of D. You cannot get anything smaller than this because there is a closure property to be satisfied.

Very much as Newton gave a solution to this fixpoint equation you can give solutions to such fixpoint equations and in fact most of our recursive definitions in computer science are really equations to be solved and they are fixpoint equations to be solved. What is

most surprising is that if you did a course on semantics of programming languages, you will see that in fact the way you obtain solutions to those fixpoint equations is also very much like the way Newton did it for continuous differentiable functions.

The analogy is absolutely striking. So, you can have also infinite sequences as solutions to this equation. You might just look upon an infinite sequence loosely but then more particularly when we talk about an infinite sequence or at least a sequence that is infinite on one side and not necessarily infinite on both sides we can look upon that as a function from the natural numbers to the domain.

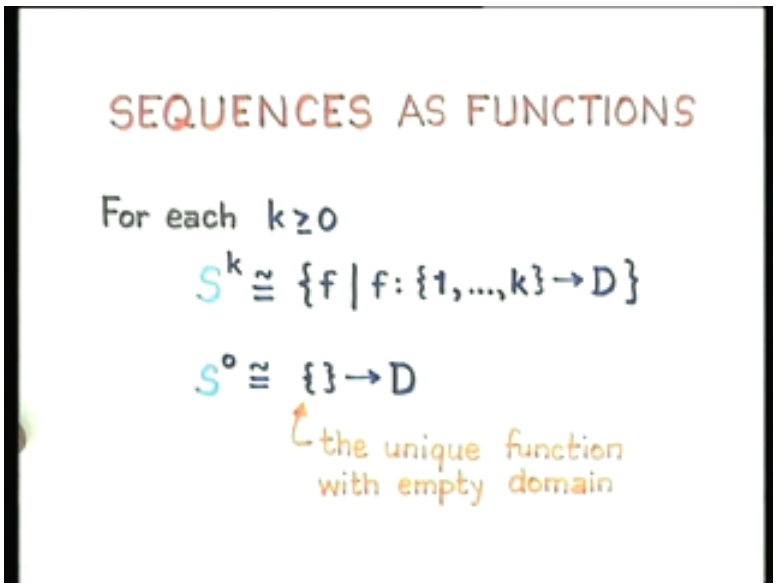
After all when you consider a sequence, it is most natural to write it as a1 a2 a3 a4 a5 etc. This essentially means that you are drawing a mapping from the natural numbers to elements of the domain. You are saying that the first element is a1 and the second element is a2 etc.

D ∞(infinity) itself is a solution to this equation. In fact it is not the least solution because the empty set is the least solution for this equation but it is the least solution other than the empty set. So, whenever you think of recursion think of equation solving. Whenever you think of equation solving think of equation solving by solving a fixpoint equation and when you think of a fixpoint equation solve it by a recurrence. Find a suitable recurrence which will satisfy it and that is the basic idea.

As I said we could look upon infinite sequences as just a structured data element containing an infinite number of components in some order or I can look upon that merely as a function from the natural numbers to my data domain. This function basically tells you what is the first element, the second element, the third element etc. You can think of that infinite sequence as a function defined by an infinite enumeration. We could also think of other sequences as functions. Even though we defined those sequences as ordered pairs we defined a $k+1$ length sequence as an ordered pair consisting of an element of D and a 'k' length sequence as a second component.

However, we could just collapse all that and we can look upon every finite sequence as being isomorphic to a function defined by enumeration. You can look upon that either as just a k length sequence of data elements or as a function from the set 1 to k to the elements to D where of course the empty sequence is the unique function from the empty set to D. The domain is empty so it does not matter really what the co domain is when there is only one possible function from the empty set to any domain D. You can look upon the empty sequence as just such a function and you can look upon any k length sequence for $K>1$ as a function from 1 to k to D.

(Refer Slide Time: 25:30)



The set of all k length sequences is just the set of all such functions. It is isomorphic and this is isomorphism. There is a one to one correspondence between the definitions of S k as we have given before and this set of functions. This is the basic idea behind arrays.

Let us take a k length sequence of integers. I have the sequence $<8,3,4,8,2,1>$. The first element is 8. This is a function; $f(1)$ is 8, $f(2)$ is 3, $f(4)$ is also 8 and $f(6)$ in which is the sixth length sequence is 1.

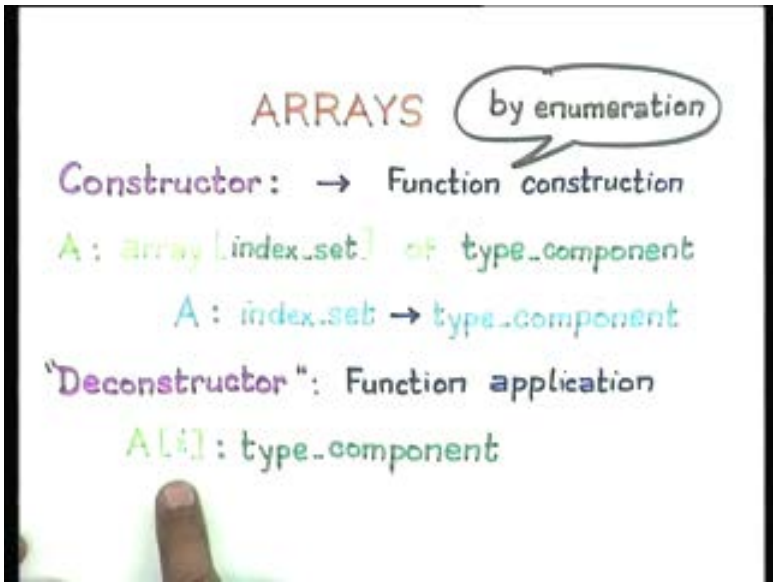
You can actually write a function by enumeration. You can look at the graph of the function. A sequence is really a function in that sense. It is a function not defined in this form $f(x) = x2$ (square). This is a definition which is not by enumeration. It is a definition by a predicate. But you can also have functions defined. After all the definition of a function just says that for each element in the domain there exists a unique element in the co-domain which means that you cannot have an arrow of this form. You cannot have two arrows from the same element of the domain and that is all. Otherwise you could write a function out by enumeration that is as a good a function as any other.
So, Sequences could be regarded as functions.

It is important to know that there is no essential difference between data and control. Control is really a function. When we do the lambda calculus you will see that there is really no essential difference between data and control. You already know that. In the case of your computer architecture course both data and control are represented as data.
The fact that you got opcodes for every instruction means you are not really doing much control; you are just representing program as data. There is an alternate view in which we can look upon all data as functions and all controls also as functions.

Meanwhile let us delve into finite sequences. This set is isomorphic to the set of all functions from this closed interval 1 k to D. This also gives us the reason why we might look upon arrays in programming languages as functions. Arrays are really functions defined by enumeration. It is important for us to look upon sequences in their generalities so that we understand that there is an underlined unified whole called the sequences. We should not be misled by the fact that arrays look different from files and lists but fundamentally they are all sequences.

The only difference is that arrays are functions by enumeration and they are functions over a finite index set of some component type. This is of type D and so, your very basic declaration in a Pascal like language is really that of a function specification. A is an array …something index set of type component which is really specifying a domain and a co-domain and is specifying the name of the function to be A.

(Refer Slide Time: 31:59)



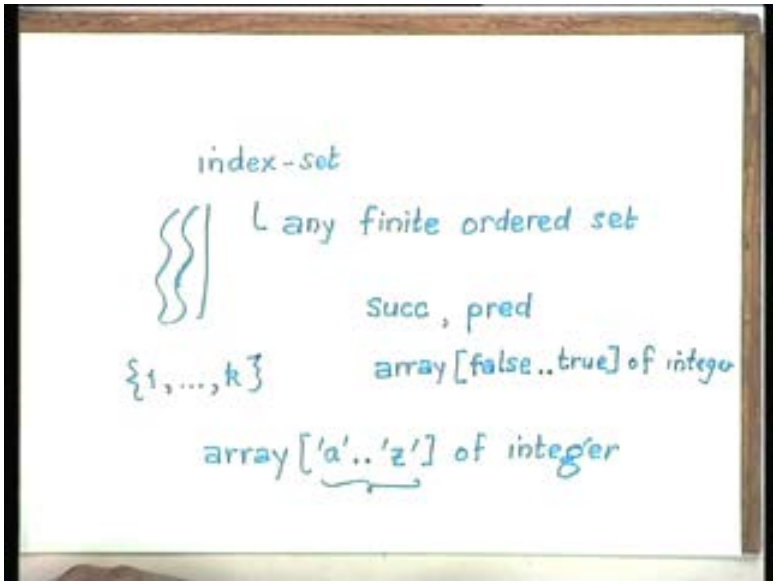
It is a function defined by enumeration as opposed to a function defined by a closed form and the deconstructor for such a function is just function application. How do you get a component out of the array? By applying the function A to the component I, you get the ith component. It is as simple as that. This is the deconstructor. Many languages for example, do not distinguish between array-component reference and function application at all. They use the standard parenthesis for array-component reference or subscripting. The subscripting operation on arrays is really a function application.

Let us look at arrays. The one important reason why arrays are useful in programming is that they provide a form of direct access which is what is known as random access. If you look at the pragmatics of the array allocation, you would find that firstly, since these arrays are bounds, in a language like Pascal the bounds are known at translation time and the type of the components is also known at compile time, you can give an accurate calculation of the space that is needed to store the entire array.

If you can give an accurate calculation then all your address calculations can be based on the stack and you do not need to go into the heap. I have to be careful about the index set. So, an array is a function from the index set to a component type. This index set is by definition any finite ordered set. It is ordered either by a predicate or more usually it is a finite set with the operations of successor and predecessor defined for all elements except the extremes. That is also the reason why you can take any enumeration type or a sub range type in Pascal and call that the index set of the array.

The successor and predecessor functions are defined for all the basic data types except the reals. For example; a dense set is one in which between any two distinct elements there is guaranteed to be another element. Any dense set is going to be infinite. Dense sets cannot be index sets. This means logically that the reals and the rationales are out. If you are talking about any finite set which is ordered by a successor and predecessor relation then this index set is isomorphic to some ordered set of natural numbers for some k. In fact if you look at *array*['*a*'..'*z*']*of* int*eger* then you are looking at a function that is a composition between this isomorphism from index set to the set where k happens to be 26 and integers. This is a finite ordered set with a successor and a predecessor defined on it. You can also have for example *array*[*false..true*]*of* int*eger* . You could have arrays of this form too.

(Refer Slide Time: 36:57)

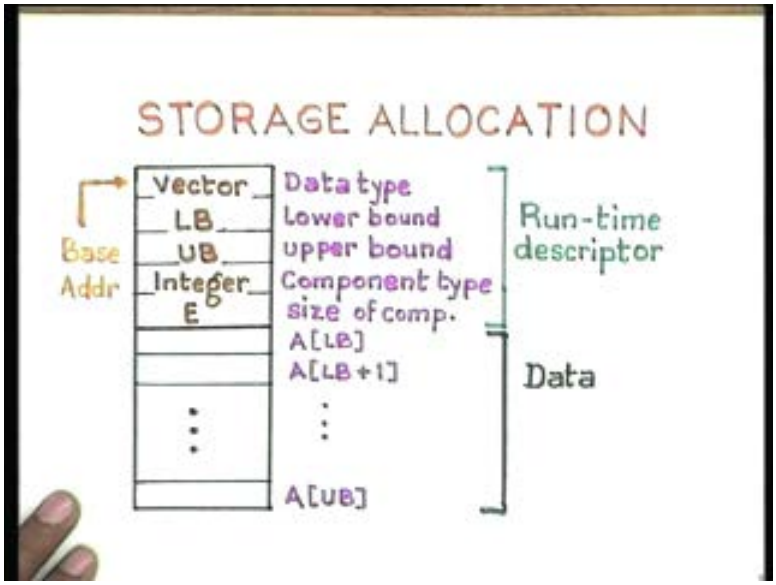


As long as successor and predecessor functions are defined on the index set and it is ordered in some way, the index set can be placed in one to one correspondence with the closed natural interval 1 to k for some k and therefore any array could be defined in terms of any such index set. In fact the sub range types and the enumerated types in Pascal are actually implemented by a one to one correspondence. You actually create an array and put the elements in that order so that you have already performed the mapping by an

enumeration mechanism. We are looking at storage allocation. The only other problem that you have to worry about is really that of checking bounds at runtime and for that you require what is known as a runtime descriptor.

Here I have given a picture of a single dimensional array. It is called a vector. In order to be able to perform runtime checks on indexes you require to know the lower bound, upper bound and the type of the component and the amount of space each component is going to occupy. Since you have to do runtime checks on expressions which are going to access components of the array you require to store this descriptor along with the array on the stack as and when you enter a block which declares this array. The data of the array is actually stored down here.

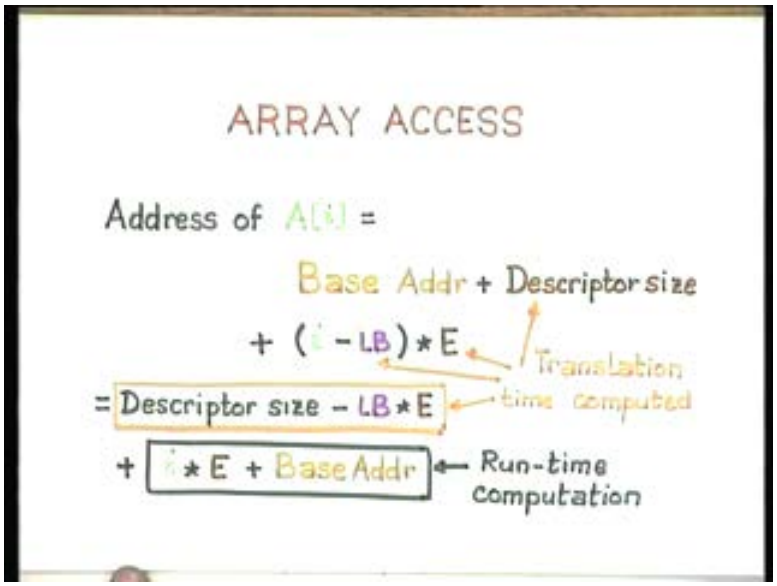
(Refer Slide Time: 39:30)



Most of these languages which involve runtime checks on array bounds actually perform these bounds because they keep a descriptor which is created. This descriptor template is created at compile time and there is enough information in the declaration of the array to have this stored somewhere and each time on entry into a block during execution time you first store this descriptor on the stack and then store the actual array elements after that. So, you can do a direct address calculation which is quite trivial.

If you look at array access whether for L value or R value you still have to get to that location of the ‘i’. Then there is a base address which is not compile-time determinable. Relative to this base address there is a descriptor size which is compile-time determinable. There is of course an ‘i’ which need not be an index. It might be an expression on the data type of the index set and therefore ‘i’ is not compile-time determinable. However, the lower bound and the size of the elements are compile-time determinable. You have to do a certain amount of compile-time computation as part of your array creation mechanism and the rest can be a runtime computation.

(Refer Slide Time: 40:50)



The actual direct access is more often called random access. But what you mean is direct access and the direct access means that the direct access does not have to go through the structure in sequence to find an appropriate element but that you can do an address computation outside that structure and hit upon the absolute address of that component directly and that of course involves this very trivial runtime computation.

Most machine architectures usually specify some byte length or some word length. Very many architectures starting from the IBM 360 actually specified that all integers should be in half word boundaries and all floating points should only be in full word boundaries where a word consisted of 4 bytes. Some machines like Deck System 10 had a word consisting of 5 bytes and then they had all these complicated conditions and one was that you can store data only starting from a word address or a half word address or some such condition.

The whole point was that the underlying assembly language provided operations which worked fastest on word boundaries or half word boundaries and worked much slower when you had to pull apart the individual bits of a byte one at a time. They could do these parallel Boolean operations on words; they could do parallel Boolean operations on half words but sequentialyzing all those operations bit by bit would slow down the machine extremely. They always recommended that you store data in such a way that they occupied word boundaries so that they might exploit the parallel Boolean operations to the fullest extent to give you the maximum throughput.

As a result you will be storing arrays mostly according to some architecture-specified boundary so that the computations are fastest. However, if storage is at a premium then you might want to use a packed representation.

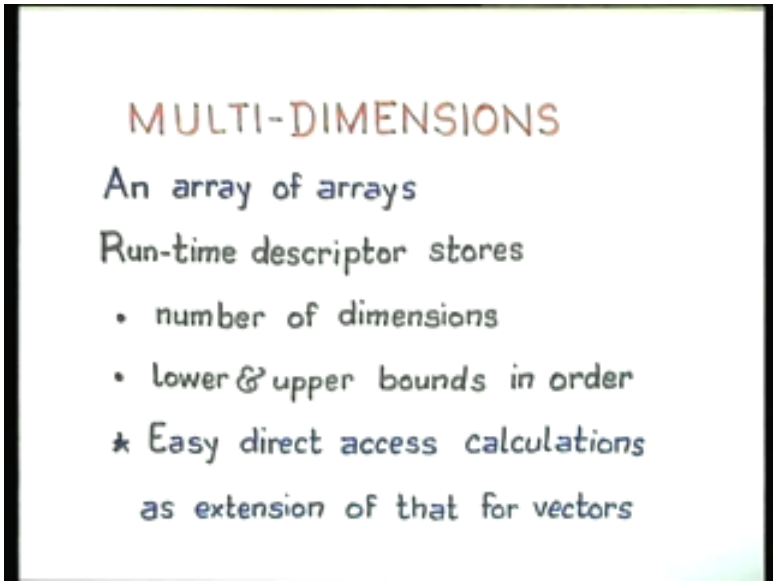
If you are going to use a packed representation it means that storage is more important to you than actual computation time and so you are disregarding all the advice that the architect of the machine gave you regarding fast operations.

It is more important for you to able to pack as much data as possible into a single word rather than do faster computations because you have got tremendous amount of data. When you have packed representation then usually you are not going to stick to the discipline of word boundaries or byte boundaries or half word boundaries. You are going to move things to occupy as small a place as possible. So, if you have integers only from 1 to 100 then it is not necessary to allocate one word to each integer. You could allocate one word to 4 integers. If the integers are all going to be less than 2 raised to 8 or 2 raised to 4 then you are going to pack as many integers into one word as possible and if you are going to pack that many then you cannot exploit the parallel application of those logical operations that the machine architecture provides you. This implies that in order to do them fast, you have to unpack these integers and store them afresh in some place to do the computation and be stored back in this pack back fashion.

(Refer Slide Time: 46:13)

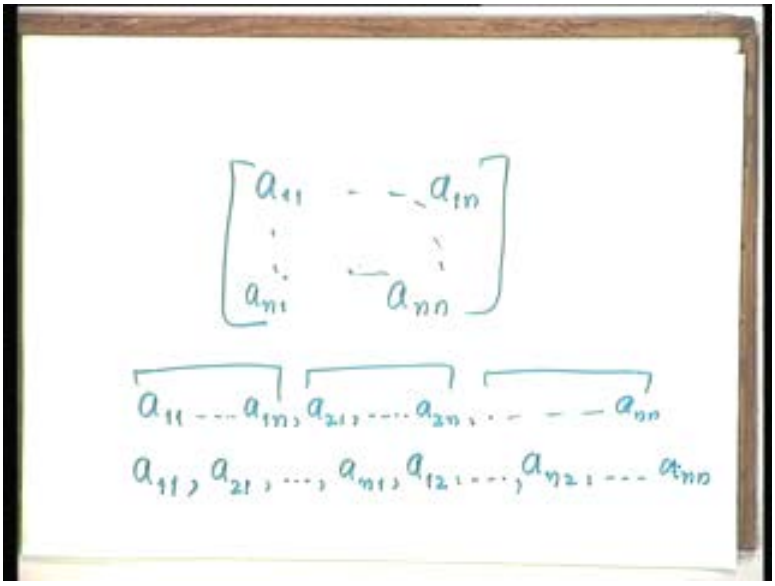
Any kind of packing mechanism means that your access to individual components will no longer be direct and secondly if you want faster computation you will have to unpack that representation. So, you might use more intermediate storage in order to gain faster computation but you are going to lose some time because of unpacking and packing again because that is an extra loading and storing. Unpacking and packing also means accessing individual components whose addresses are integral multiples of whatever is the basic unit in the machine memory. Packed representations are likely to be slower and more expensive and are going to be used only when the data is so huge and it is possible to save on storage spaces. When storage space is at a premium it is possible to save on storage space but at the expense of slower access and slower computations.

(Refer Slide Time: 46:59)



This notion of arrays can be easily generalized to multidimensional arrays. All it means is that your runtime descriptor should also include information about the dimensions and for each dimension you have a lower bound and an upper bound specification. You have to specify them in a particular order which is also going to be reflected in the representation of the entire array for example; a two dimensional array could be represented either in row major order or column major order. To explain row major order if you have a matrix of this form a 11 to 'a nn', a two dimensional matrix is going to be represented linearly in memory. If it is represented in row major order then you are going to have 11 to a 1n followed by a 21 to a 2n…..up to 'a_{nn} '.

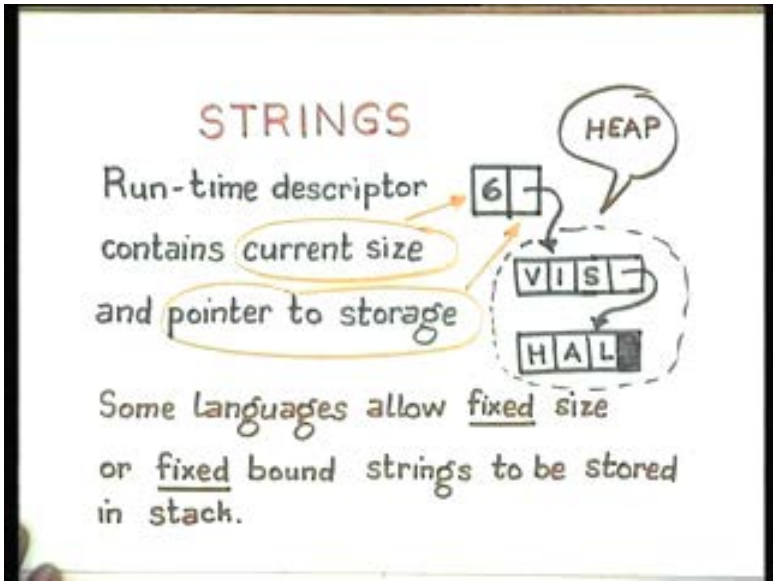
(Refer Slide Time: 48:48)



This is row major that means the matrix is regarded as a vector of vector of rows or as an array of rows. The other alternative is a column major order, a_{11} a_{21} ….up to a_{n1} and then a_{12} and then up to a_{n2} and up to a_{nn}. Many of the initial FORTRAN compilers and even now many FORTRAN compilers actually use this column major ordering. That means they represent the two dimensional matrix as a vector of columns whereas most languages now a days such as Pascal and Ada would represent a two dimensional array in row major order. That means they would represent a matrix as a vector of rows.

In the case of multi-dimensions again you can write an easy formula to do a direct access calculation of the address, separate it out into translation time computable term and runtime determinable term. The other kinds of sequences that we use are strings. Normally the strings are stored in heap. Sometimes the heap is divided into a separate string space where all strings are stored contiguously with just a single descriptor which gives the length of the string. Very often they are stored contiguously because each character in a string does not require more than 7 bits.

(Refer Slide Time: 49:41)

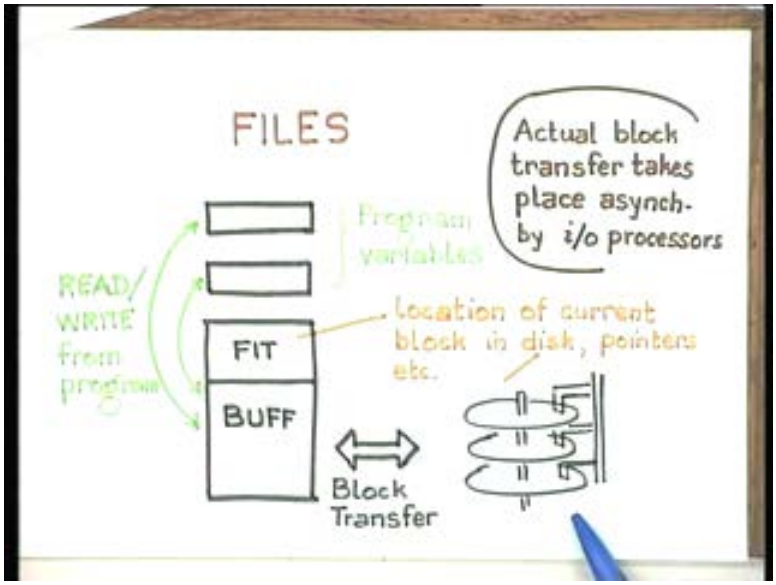


You could afford to store them in an 8 byte word contiguously for example or you could actually use the heap for dynamic data structures and you could use it as part of the dynamic data structuring. This means that a pointer can make things slow. A language like SNOBOL4 which is meant specifically for string processing would try to store all the strings contiguously without having to use too many pointers. There are languages like PL1 which allow fixed size strings which means that they could be stored in the stack because their length is calculate-able or fixed bound but variable-sized strings which means you specify an upper bound of the length of the string but you do not guarantee to use all of it.

This means that the runtime descriptor for that kind of a string would contain both the upper bound and the current length of the string so that you know exactly how much you are using and you are not reading more and more garbage that is there in the end of that allocated amount. Then in such cases these strings could be stored directly on the runtime stack since their sizes are compile-time determinable. Lastly, you have files. This is actually the disk drive or some secondary storage device. In the case of files there is a logical difference between the program 'read' and 'write' operations and actual transfer to and from the secondary storage.

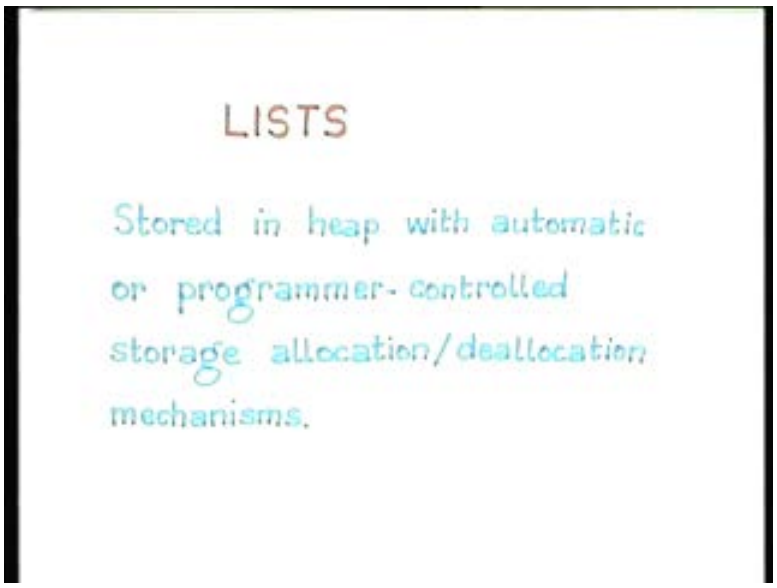
Usually the transfer is by a block and there is a file information table which usually contains information such as firstly, what area of the disk a piece came from or what area it should go to; secondly there should be a pointer to the actual area in the buffer where the current pointer is. Very often the actual transfer of data takes place at an idle time for the processor or it is just handled directly by the IO processors. This means that you have to have interaction with the operating system on which you are implementing your language.

(Refer Slide Time: 51:35)



These are only sequential files. Languages like COBOL allow direct access and index sequential access files which means that essentially the structure of directories has to be pulled down into data components within the file and you will have store disk addresses and cylinder sector addresses in some files so that you can do a direct access on to the disk for the block of storage which contains the data you are looking for. But then that means it involves a lot more intensive computation.

(Refer Slide Time: 53:25)



Then we come to lists. They are just stored in the heap and they might be controlled. They are usually programmer controlled. They are automatic in functional languages and the allocation and de-allocation depend upon the kind of list it is going to be.