Principles of Programming Languages

Prof: S. Arun Kumar
Department of Computer Science and Engineering
Indian Institute of Technology
Delhi

Lecture no 23
Lecture Title: Control

Welcome to lecture 23. We will start on control. We have gone through what might be called usual data structures that are normally available in programming languages. It is now the turn of control. In control it is to actually look at some important features and important variations and look at some pragmatics of the implementation of certain features. The interesting aspect about both data and control is that they are not very different from each other in many ways. Firstly, if you look at the Von Neumann architecture, both data and control have the same representation which is part of the fundamentals of such architecture. The stored program concept so to speak essentially envisages representing data and control with the same kind of representation. That is what also makes self modifying programs possible.

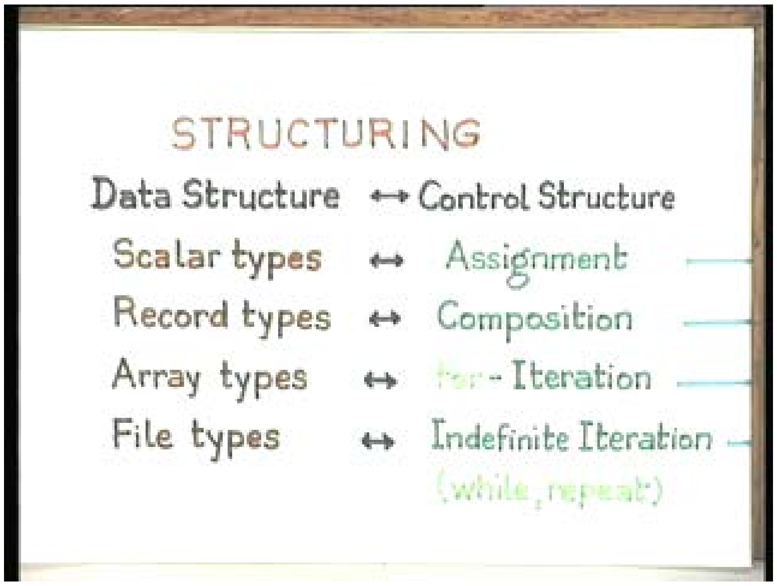
There is also another way of looking at it. You can look upon that Von Neumann architecture of an actual machine as one which essentially says that control is really nothing but data. All control is really data. There is an alternative view which we will look at in the lambda calculus and what that says is really that all data is just control. So, you can look upon all data as just functions and that is what the lambda calculus purports to do and it actually does it quite successfully.

However, in our mental mind set there is a clear distinction between data and control. But the fact that these two are actually mutually convertible is like what Polya said once. True mathematical genius is not in finding differences between structures but in finding similarities between widely different structures. What led to the design of Pascal is exactly the similarity between data structures and control structures. There is an interesting analogy which Wirth points out which motivated the way he designed both the data structures of Pascal and the control structures.

First you have scalar types and scalar types correspond to the assignment in the sense that they are both atomic. They formed the basis for other structured commands. Then he looks at record types and essentially he says that record types that consist of enumerations of types in sequence correspond to a compound statement which may be bracketed by begin and an end. The fact that a record is a sequential enumeration of heterogeneous data the compound statement which is really represented by a sequential composition really represents the fact that you can sequentially compose heterogeneous kinds of commands and you can sequentially enumerate them.

Let us look at array types. The array types actually are a fixed number of repetitions of a homogenous type. Each array type has a structure which is a fixed number of repetitions or replications of a homogeneous type of a single type and if you look at a 'for' loop iteration, whether it is for counting up or counting down which does not really matter, it is really a fixed number of repetitions of the same underlying command.

(Refer Slide Time: 6:50)



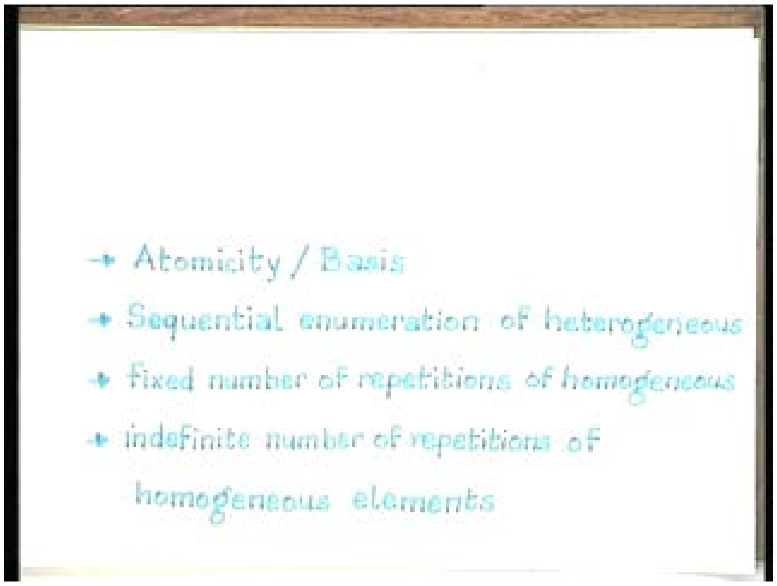
STRUCTURING

Data Structure	↔	Control Structure
Scalar types	↔	Assignment
Record types	↔	Composition
Array types	↔	for - Iteration
File types	↔	Indefinite Iteration (while, repeat)

He goes further and says file types really correspond to indefinite iteration. A file type is just a sequence. Remember that a file type is homogeneous in the sense that it is a sequence of elements of the same type of unbounded size and an indefinite iteration in the form of a 'while' loop or a repeat until is really a form of indefinite number of repetitions of this same kind of command.

The 'while' and the 'repeat' are actually like the file type and they represent essentially indefinite number of repetitions of some homogeneous kind of element be it a command or a type. You can actually carry it further and you could look at variant records. Variant records correspond more or less to conditionals. There is a case analysis and a particular case of case analysis when there are only two possibilities. 'If' or 'if then else' and all kinds of conditional statements really are like variants in a record. So, there is a very close analogy between variant records and conditionals and that also motivates why he has used similar reserve words in all these data and control structures that he has designed.

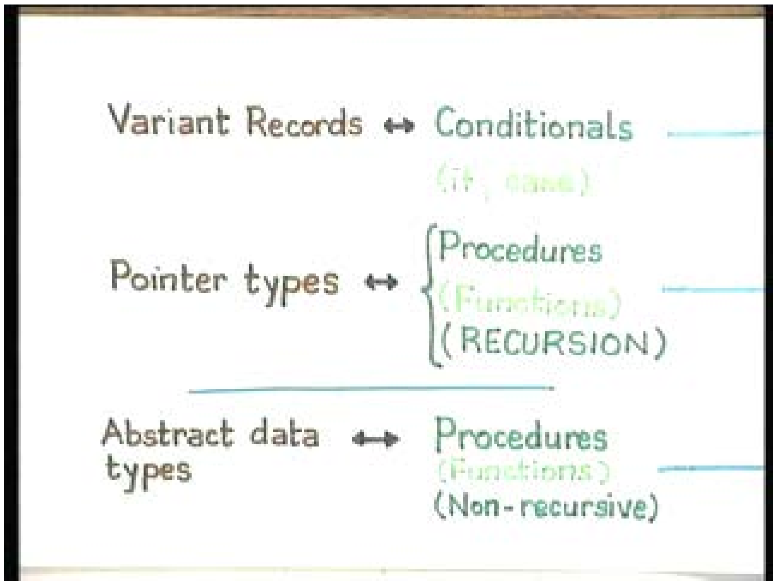
(Refer Slide Time: 07:16)



- Atomicity / Basis
- Sequential enumeration of heterogeneous
- fixed number of repetitions of homogeneous
- indefinite number of repetitions of homogeneous elements

Lastly, we can look upon procedures and functions. Mainly if you look at recursion then you also have a corresponding recursive data type which is a pointer type. The main point about recursive procedures and functions is that recursion is an important control structure and it is also an important data structure in facilitating Pascal. Wirth actually stops there but one could go further. If you look at non recursive data types and non recursive procedures and functions they essentially give you an abstraction of control and corresponding to that you have abstract data types which are an abstraction of data.

(Refer Slide Time: 9:11)

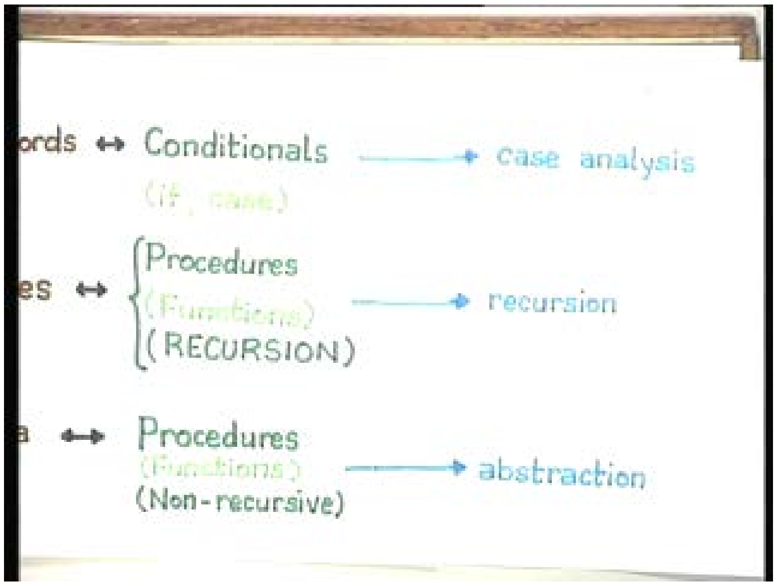


Variant Records ↔ Conditionals (if, case)

Pointer types ↔ Procedures (Functions) (RECURSION)

Abstract data types ↔ Procedures (Functions) (Non-recursive)

(Refer Slide Time: 9:33)



ords ↔ Conditionals (if, case) → case analysis

es ↔ Procedures (Functions) (RECURSION) → recursion

a ↔ Procedures (Functions) (Non-recursive) → abstraction

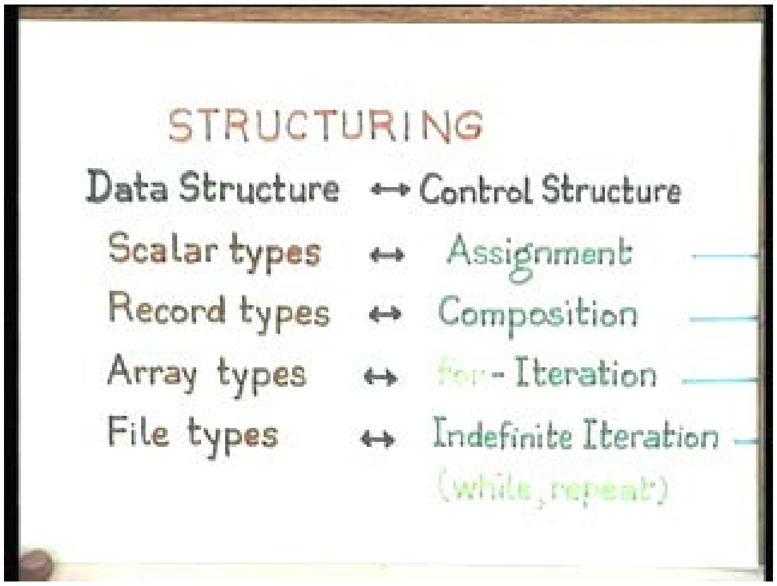
Unlike previously when procedures and functions were considered to be forms of avoiding repetition of the same piece of code or may be parameterized forms of the same piece of code the fundamental reason for having procedures is that they give you a level of control abstraction so that you do not need to look deeper into the procedure. You just need to know a specification of what the procedure does and you can use it. They essentially provide a form of control abstraction and corresponding to this control abstraction there is also a data abstraction that is possible which in fact in Modula 2 Wirth actually implemented this analogy too. So, his analogy in Pascal stops here. But in Modula 2 it goes further into abstract data types. Of course it is not his original idea. It is an analogy that has been pointed out some time after the design of Pascal and it was developed in various other languages like Clue, Alpha and Modula 2.

There is a form of modules also in ML. In both the Moscow ML and in standard ML which is actually a very highly abstract entity you could actually carry this analogy right after this portion. How does a typical pointer type look like? What is the most important characteristic of a pointer type? You would declare;

$$\begin{array}{l} ptr = \wedge node; \\ node = record \\ next : ptr \\ end \end{array}$$

If you look at this, this pointer is defined in terms of node and node is defined in terms of pointer. This is a case of mutual recursion. If you unfold the recursion you can express essentially structurally at least a node in terms of itself. That is the fundamental purpose. You want to define a data structure recursively so that you do not have to fix any bounds.

(Refer Slide Time: 13:32)



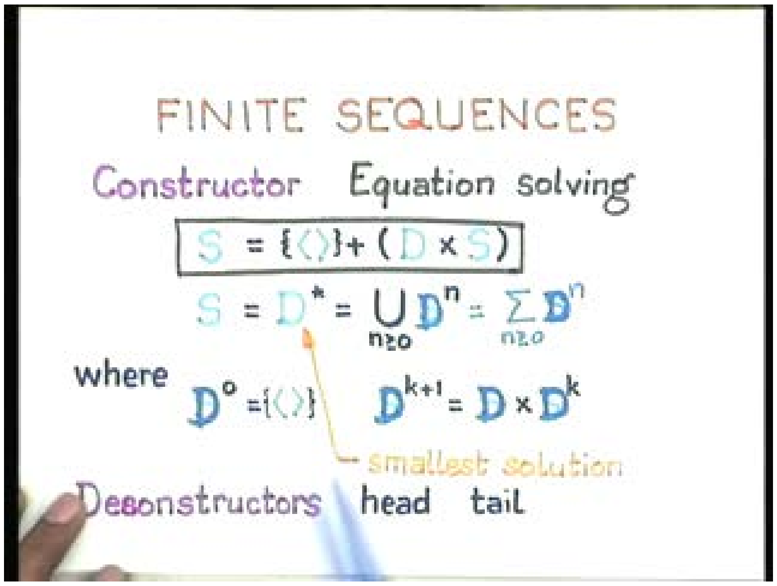
STRUCTURING

Data Structure	↔	Control Structure
Scalar types	↔	Assignment
Record types	↔	Composition
Array types	↔	for - Iteration
File types	↔	Indefinite Iteration (while, repeat)

This is actually a case of mutual recursion. You can pull it apart, unfold it, merge these definitions together and you would get a definition of a node in terms of itself. That is really how a singly linked list works or any other kind of recursive data structure. Lots of these dynamic structures are recursive in that sense. Tree structure and list structures are recursive data structures. Let us look at one particular form of the analogy which has to do with the file types and indefinite iteration.

Let us consider file types. A file is a finite but unbounded sequence. If it is a finite but unbounded sequence essentially it is a solution of an equation. If you look at the equation and look at the solution, the solution has been defined inductively and that forms a part of an iteration solution.

(Refer Slide Time: 14:12)



FINITE SEQUENCES

Constructor Equation solving

$$\boxed{S = \{\langle\rangle\} + (D \times S)}$$

$$S = D^* = \bigcup_{n \geq 0} D^n = \sum_{n \geq 0} D^n$$

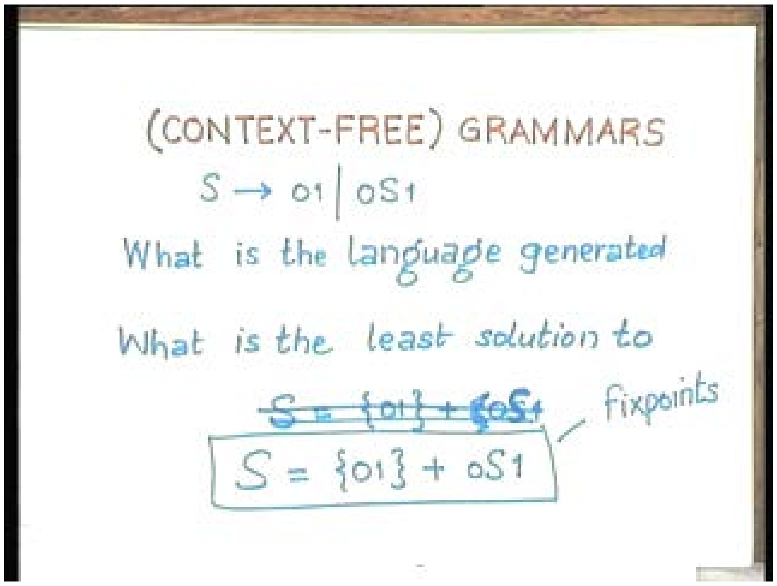
where $D^0 = \{\langle\rangle\}$ $D^{k+1} = D \times D^k$

smallest solution

Deconstructors head tail

You have an inductive solution for essentially a recursive definition or a fixpoint equation. I had mentioned that the solutions to such equations are actually very similar in all branches of mathematics. What I also showed was that though I took a context-free grammar it is not necessarily any grammar which has a set of production rules; the production rules of any grammar can be looked upon as defining a set of mutually recursive definitions of the non terminals of which one non terminal has to be solved and that is usually the start symbol.

(Refer Slide Time: 15:07)



(CONTEXT-FREE) GRAMMARS

$S \rightarrow 01 \mid 0S1$

What is the language generated

What is the least solution to

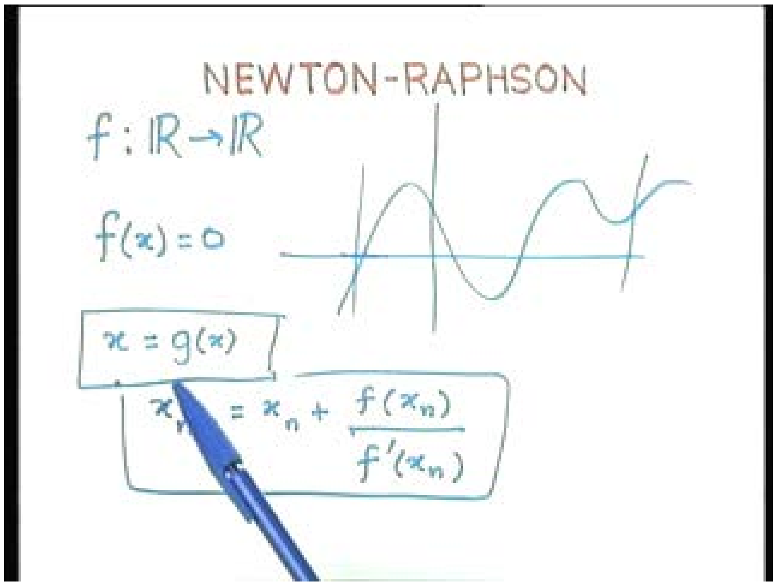
~~$S = \{01\} + \{0S1$~~

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

fixpoints

You can take those definitions and go through an iterative process and obtain a solution which is a fixpoint of that equation. A similar method was what was adopted for example in the NEWTON-RAPHSON method or in any kind of fixpoint equation solution even on real numbers. If you look at the numerical methods for solving fixpoint equations you actually have iteration or a recurrence that gives you a solution which gives you closer and closer approximation to this solution, provided, certain conditions are satisfied. The derivative should nowhere be 0 if you follow standard convergence criteria and smoothness of functions. Inductively or through a recurrence it defines closer and closer approximations to at least one of the actual roots that you might be interested in so that only in the limit you actually get that root.

(Refer Slide Time: 15:52)


NEWTON-RAPHSON
$f: \mathbb{R} \to \mathbb{R}$
$f(x) = 0$
$\boxed{x = g(x)}$
$x_{n+1} = x_n + \frac{f(x_n)}{f'(x_n)}$

You can look upon this union for example as also computing a limit. Take this sequence and look at it this way. Take the definition of a real number. If you go from rationals to reals, you are adding irrational numbers also to a set. If you look at how we can define a real number in general, it means first, how we can define an irrational number in general and how we can ensure that all the reals which are also rationals satisfy that definition. Two centuries ago Dedekind solved that problem by declaring that an irrational number is not just a number. It is actually the set of all rationals that are less than it. Any irrational number is actually the set of all rationals. Let us take $\sqrt{2}$. It is actually the set of all rationals such that it is less than $\sqrt{2}$. You can express this $\sqrt{2} = \{p/q \mid p/q < \sqrt{2}\}$ otherwise by writing p square by q square < 2.

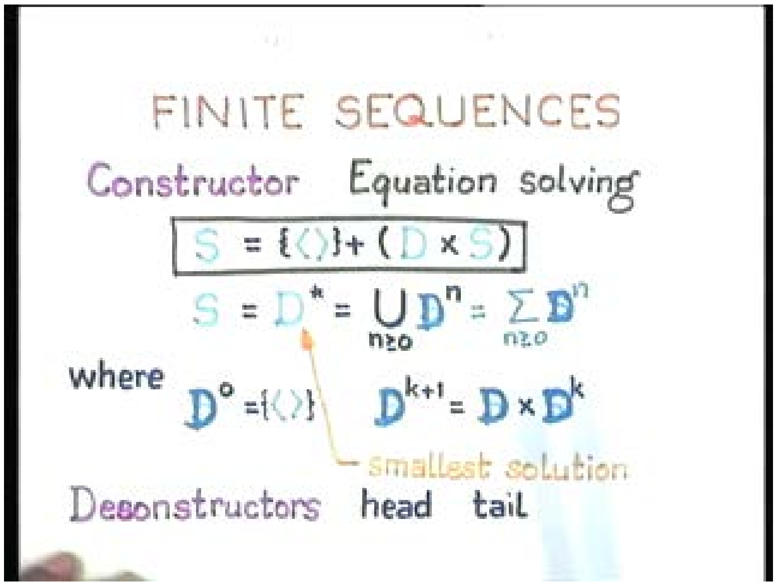
He said that an irrational number is really a set in the sense that if you take all the rationals less than it then in the limit you approach that irrational number. So, the notion of a limit itself is that of a set. Take this and apply this definition let us say to the rationals themselves. The rationals are also reals, so, a similar definition should apply. You can take every rational to be a set of all rationals.

Every rational number can be defined as being isomorphic to the set of all rationals that are less than or equal to it so that the limit of that set is the rational itself. Now you have a completion of the rationals into the reals.

Every real number is really a set of all the rationals that are less than or equal to it. It does not matter whether the real number is rational or irrational. It is still a set. Further if you look upon a real number as $r \cong \{p/q \mid p/q \le r\}$ you can also look upon the real number as being isomorphic to the union of all the reals,
$r \cong \{s \mid s \le r\}$.
What is the concept of a limit? The concept of a limit is just union or least upper bounds. A similar process is actually what happens in this kind of equation solving. If you look at it you can look at successive approximations.

Let me define a new Dn as $D_n = \bigcup_{m \le n} D^m$
Then I have a sequence of approximations to D * in this fashion; $D_0 = D^0$ $D_1 = D^0 + D^1 D_{k+1} = D^0 + D^1 \ldots Dk$

(Refer Slide Time: 21:16)



FINITE SEQUENCES

Constructor Equation solving

$$\boxed{S = \{\langle\rangle\} + (D \times S)}$$

$$S = D^* = \bigcup_{n \ge 0} D^n = \sum_{n \ge 0} D^n$$

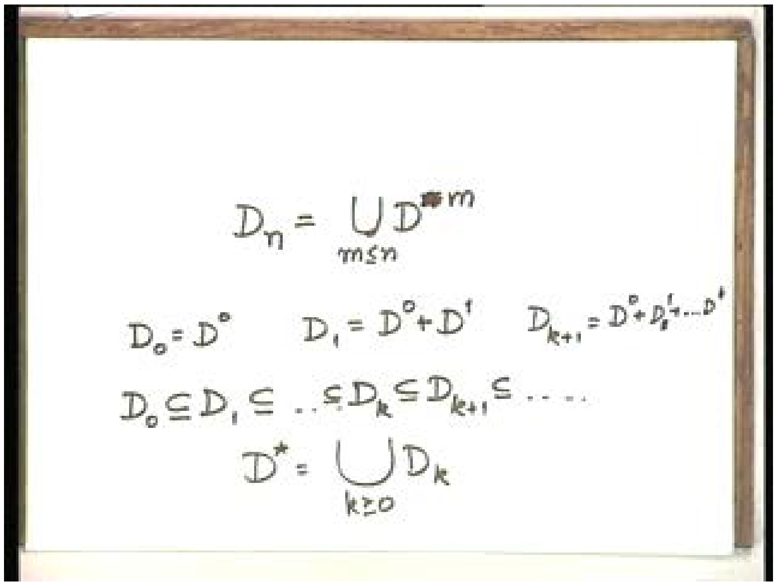
where $D^0 = \{\langle\rangle\}$ $D^{k+1} = D \times D^k$

smallest solution

Deconstructors head tail

So, what we get is a chain of increasing approximations $D_0 \subseteq D_1 \subseteq D_k \subseteq D_{k+1} \subseteq \ldots\ldots$.
In the limit you get the union of all this which is the least upper bound of this chain. If you take the least upper bound of this chain you are really computing the limit of an infinite sequence and $D * UD_k$.
I took a short cut here by writing D* to be the unions of these Dn's but I could equally well regard D* to be the union of these Dk's.
D* is a limit that is reached by successive approximation. We are going to have a similar piece of reasoning for while loops.

(Refer Slide Time: 23:23)



$$D_n = \bigcup_{m \le n} D^{*m}$$

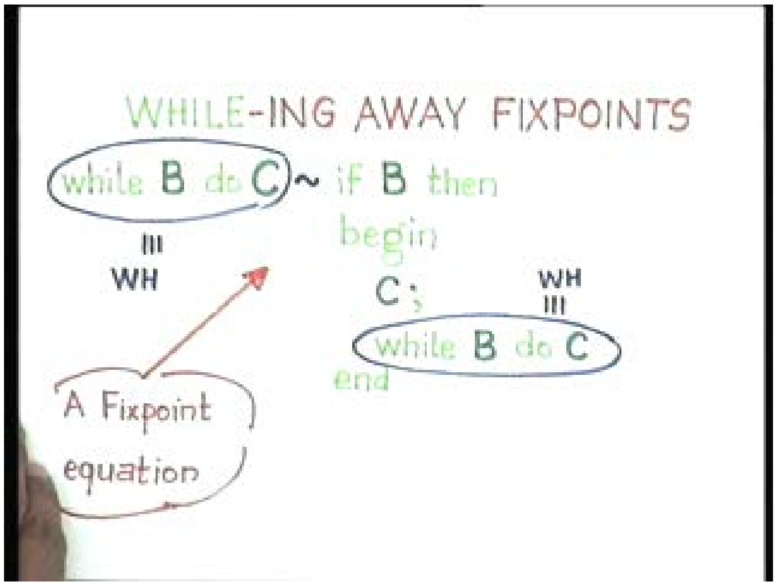
$D_0 = D^0$ $\quad$ $D_1 = D^0 + D^1$ $\quad$ $D_{k+1} = D^0 + D^1 + \dots D^k$

$D_0 \subseteq D_1 \subseteq \dots \subseteq D_k \subseteq D_{k+1} \subseteq \dots$

$$D^* = \bigcup_{k \ge 0} D_k$$

One of the things that everybody gives in an operational semantics essentially defines operational semantics of the while loop in terms of itself. If you look at our operational semantics in terms of those two rules it essentially says nothing more than this. If you ask the question what does the while loop do then you are asking for a solution for this equation which is a fixpoint equation. You are looking for a fixpoint of this equation and we would like to define the notion of successive approximations to obtain a solution to this fixpoint equation. Just as in the case of finite sequences I can define the successive approximations like this.

(Refer Slide Time: 24:35)

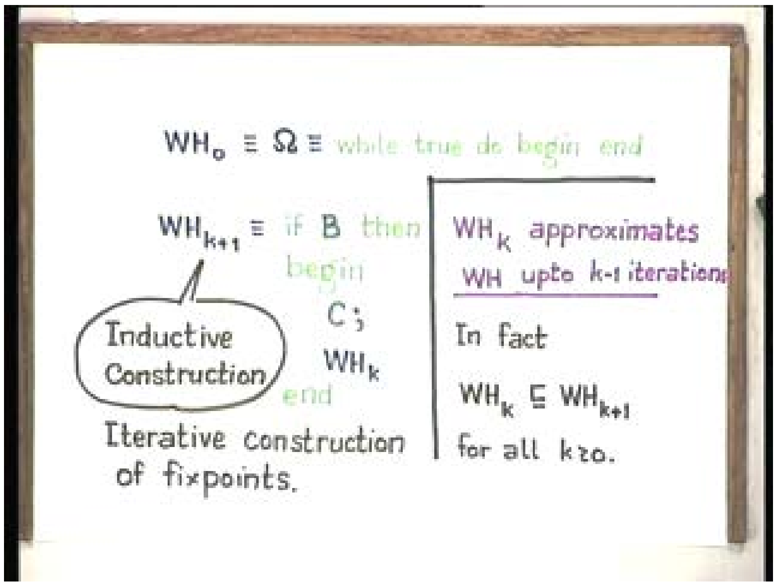


WHILE-ING AWAY FIXPOINTS

```
while B do C  ~  if B then
                 begin
     |||             C ;          WH
     WH                           |||
                     while B do C
                 end
```

Firstly, just like we have an undefined value in expressions, I can define an undefined command in the sense that any looping command is an undefined command. The notion of approximation that I am going to have is going to be relative to this notion of undefinedness. As in the case of sequences, as in the case of NEWTON RAPHSON or in context- free grammars we define a solution to the fixpoint equation in terms of successive iterations in terms of a recurrence and here is a recurrence.

(Refer Slide Time: 26:06)



$WH_0 \equiv \Omega \equiv$ while true do begin end

$WH_{k+1} \equiv$ if B then begin C; WH_k end

Inductive Construction

Iterative construction of fixpoints.

WH_k approximates WH upto k-1 iterations

In fact

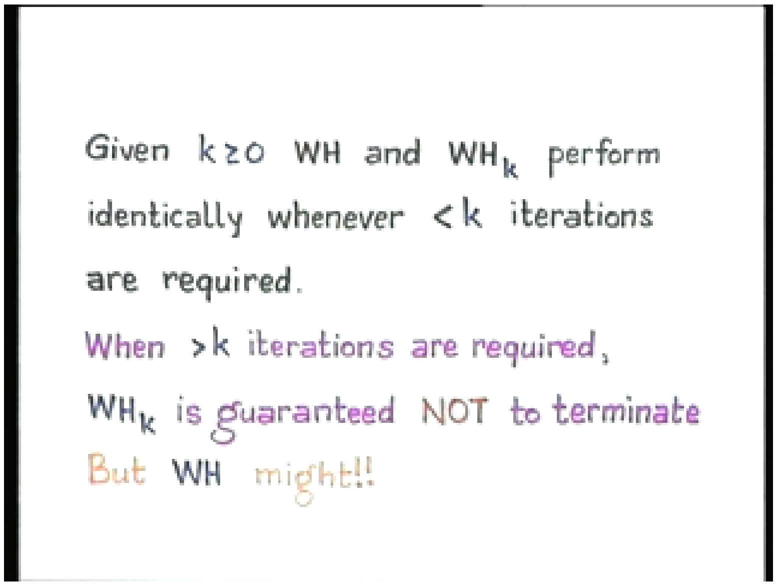
$WH_k \sqsubseteq WH_{k+1}$

for all $k \geq 0$.

Let me define WH_0 to be this looping construct and WH_{k+1} is defined in terms of WH_k. So it is no longer a recursion but it is inductive. This is like a recurrence. Essentially what we are saying is that I will define WH_{k+1} plus to be this construct if B then begin C; WH_k. Supposing I have a while loop and supposing in the given state it requires some k iterations to terminate then its behavior is exactly captured by WH_{k+1}. If it requires k iterations before it terminates then WH_{k+1} is also equivalent to doing k iterations of that same while loop. My construction of WH_k and WH_{k+1} really depends upon the two components of this while loop; the same Boolean expression and the same command which forms the body of the while loop. If you look at it this way then WH_{k+1} just represents the same behavior as the while loop for all states from which you require to do at most k iterations in order to terminate and get an answer.

The moment the number of iterations is k+1 or more then this WH_{k+1} goes into an infinite loop because it will hit this ω(omega) here. Note that this is inductive which means after k iterations if the Boolean is checked out to be true then it will execute this body and then go into ω which means it will never come out of the loop. The main difference between the actual while loop and this WH_{k+1} is that that actual while loop might actually terminate sometime after k plus some iterations. This WH_{k+1} will never terminate.

(Refer Slide Time: 28:41)


Given $k \geq 0$ WH and WH_k perform identically whenever $< k$ iterations are required.

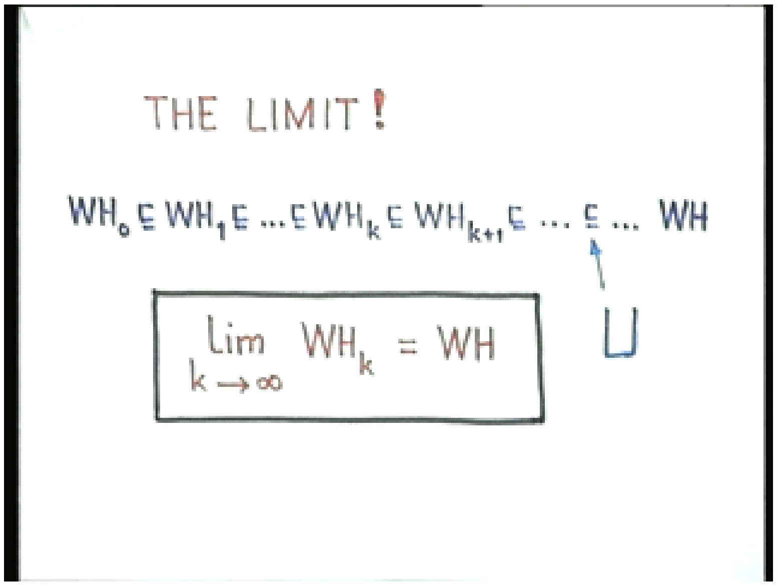
When $> k$ iterations are required, WH_k is guaranteed NOT to terminate

But WH might!!

The sequence WH_0 WH_1 $WH_{2...}$ etc is an increasing sequence of approximations to the actual while loop in the sense that WH_0 will never terminate; WH_1 will do all the computations of the original while loop which require no iterations. WH_k will do all the computations of the while loop which require less than k iterations but no more. Each of these therefore is an approximation along a definedness ordering and this definedness ordering is quite similar to this sequence. It is actually a sequence of approximations.

You can define the while loop iteratively as an infinite chain of approximations that come closer and closer to the behavior of the original 'while'. In this limit k tends to infinity. This is a loose way but corresponding to this less than or equal to, it is possible to show that this ordering on programs is a partial order and it is not just a partial order but that the least upper bounds are well defined. There is a binary and the least upper bounds are commutative and associative. So, there is an operation of finding least upper bounds which looks like a square in the form of the big set union and that least upper bounds always exist.

(Refer Slide Time: 31:38)



THE LIMIT!

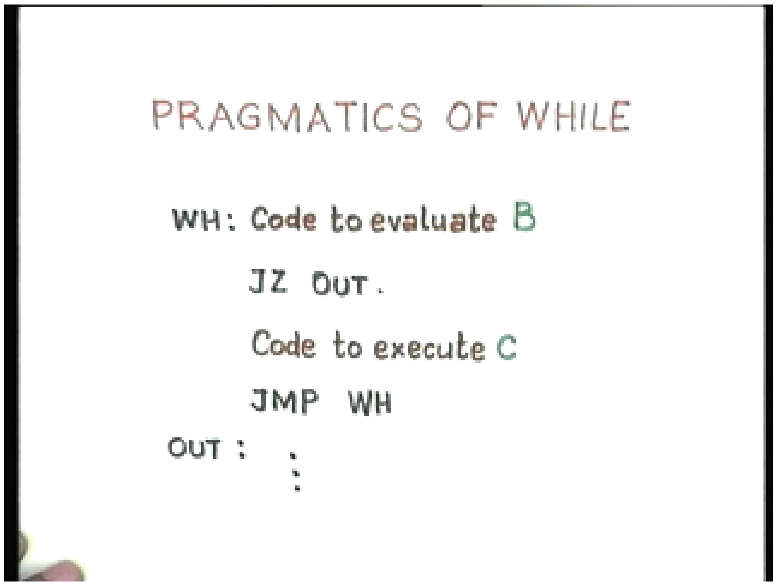
$$\begin{tikzcd}[column sep=tiny] WH_0 \sqsubseteq WH_1 \sqsubseteq \ldots \sqsubseteq WH_k \sqsubseteq WH_{k+1} \sqsubseteq \ldots & \sqsubseteq & \ldots\ WH \\ & \bigsqcup \arrow[u] & \end{tikzcd}$$

$$\boxed{\lim_{k \to \infty} WH_k = WH}$$

Once those least upper bounds always exist, you have a limit which is defined as the least upper bounds very much like the limit of an increasing chain of sets which is defined as a union of those sets. For details you will have to do a course on semantics of programming languages but this ‘while’ loop is really expressed in a form very similar to the NEWTON RAPHSON method of computations. It is exactly of the same kind. You have a fixpoint equation and you define recurrences. You define a recurrence in such a way that you get closer and closer approximants to the solution. The limit of these closer and closer approximants is the solution to a fixpoint equation. It unifies the whole of mathematics, computer science and logic.

It is enough to know that there are Polya style similarities between programs and other branches of mathematics. Finally after all the interesting aspects about fixpoints, we come to the pragmatics of ‘while’ loops and we find that it is too simple. Essentially, you implement that fixpoint equation and you replace the left hand side by the right hand side. If you are replacing the left hand side by the right hand side then you implement it as an ‘if then’ with the ‘while’ loop again in between. In basic architecture the loops are just jump statements at some point. You evaluate the code; evaluate the Boolean expression and this is the code to evaluate. Let us say this is a block of code to evaluate the Boolean expression and of course the result is stored somewhere in a condition code or in a register. You look at that register and if it is 0 then you get out of the loop. If it is not 0 then you execute the code corresponding to the body of the ‘while’ loop and after having executed it you jump back to the beginning of the loop.

(Refer Slide Time: 34:08)



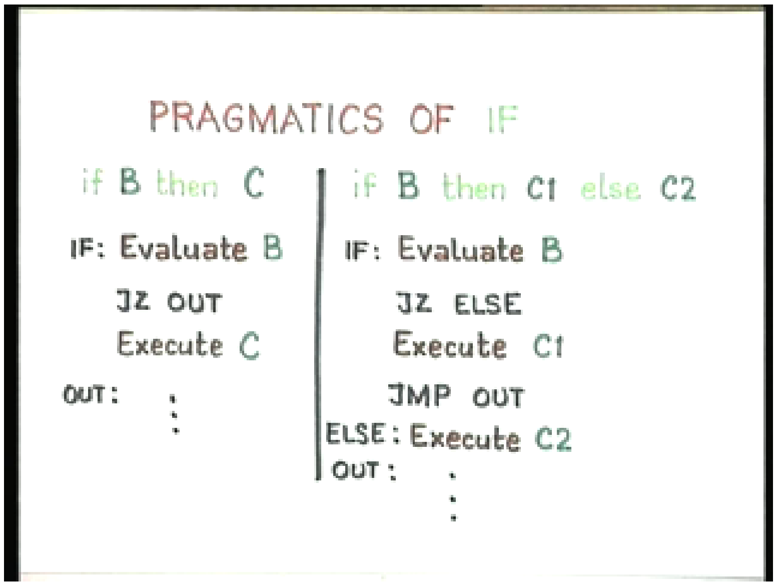
PRAGMATICS OF WHILE

```
WH:  Code to evaluate B
     JZ  OUT.
     Code to execute C
     JMP  WH
OUT:  .
      :
```

That is the analogy between data structures and the control structures and the while loop is very close to recursion but it is easily the most interesting control structure. The variations in the while loop are easily the most interesting control structure and you can see the similarity between that and the data structuring mechanism of sequences. If you look further at our approximations actually you also see that Wirth's analogies get carried. For example; this approximation uses the sequential composition whose analogy in the case of sequences was the record which is a tuple formation.

If you look at the corresponding solution for sequences you have Wirth's analogy exactly being carried through. Then we have a case analysis here that corresponds to variant records which really correspond to a summation disjoint unions and so Wirth's analogy carries exactly from data structures to control structures even right after the solution of these equations. I will not go too much into other control structures. You can do a similar analysis of the 'repeat until' statement and you can do analysis of the 'for' statement. But we have defined the semantics of the 'for' statements and other forms along the line. So, there is no problem really with talking about the pragmatics of these control structures.

(Refer Slide Time: 36:49)



PRAGMATICS OF IF

```
if B then C            | if B then C1 else C2
IF: Evaluate B         | IF: Evaluate B
    JZ OUT             |     JZ ELSE
    Execute C          |     Execute C1
OUT:  .                |     JMP OUT
      :                | ELSE: Execute C2
                       | OUT:  .
                       |       :
```

Let us quickly go through the alternation. You will find other variations of the control structures in the next tutorial sheet. ‘If then else’ has this straightforward syntax-directed translation. The ‘if then’ has a similar straightforward syntax directed translation. You can look at any compiler like the PL 0 to see that. But if you use a recursive descent parsing technique then there is really no problem in the code generation of the translation because as part of the parsing the moment you have recognized ‘if then’ or an ‘if then else’ statement you would generate the code for it. The code that is generated actually by the ‘if’ statement procedure is really what is given in black. The code that is generated by procedures are called from the ‘if’ because after all the whole parsing technique is mutually recursive.

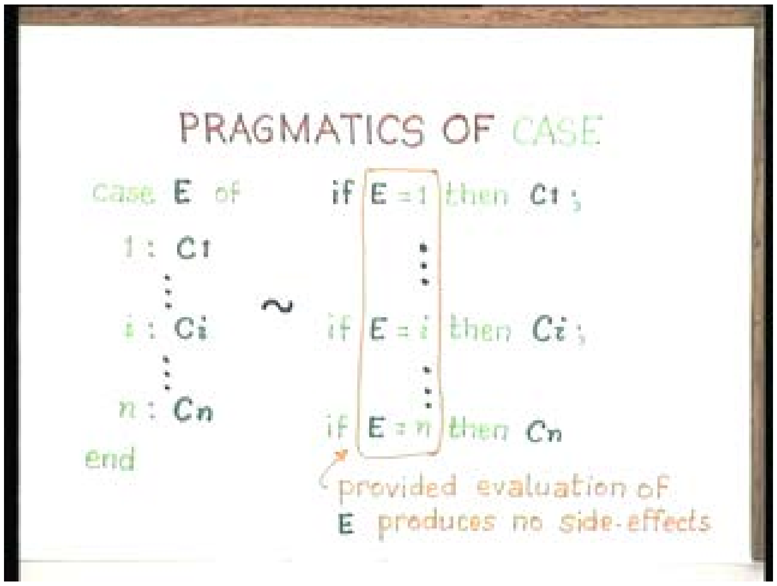
When you see a Boolean condition or when you see an ‘if then’ you transfer control to the Boolean condition recognizer and that also generates the code for evaluating Boolean expressions. Then since it is recursive you return to the ‘if’ statement and then you generate this instruction and then you have to call command again to generate the code for the body of the ‘if then’ and then you return to it. Recursive descent is a simple parsing technique but it requires recursion to be available in the programming language. If you are using something other than recursion in your parsing technique then you normally back patch.

What is back patching? There is a forward reference to this label; you do not know what is going to be the instruction number corresponding to this ‘out’. When you generate this you do not really know what is going to be the instruction number corresponding to this. Only after generating the code for C you know what is going to be the instruction corresponding to this. You keep this blank and make a parse backwards and fill up this address at all the appropriate occurrences. You actually maintain a list of blanks which have to be filled up. For each different label you actually maintain that list and keep filling it up backwards as you go.

This is useful for example when you are programming an assembly language. You have to know it otherwise you are going to get into a major difficulty. You would chain all references especially forward references to the same label. As far as backward references are concerned there is no problem because you can always see the backward reference. You know the number of the instruction which is labeled with the backward reference. But as far as forward references are concerned you create a chain of all the forward references and when the forward reference address has finally got resolved you traverse backward on the chain and fill up all the gaps with that address. It is called back patching or forward chaining. Similarly, in the case of an ‘if then else’ you have to generate two jumps after some outer portion and you could generate these codes recursively. The most important point as far as conditionals are concerned is that there is the case statement that is often used which one should look at.

One way of looking at a standard Pascal like case statement is that it is actually semantically equivalent to this sequence of statements. I can look upon this either as a sequence of Booleans which equate E with each of these integers and then executes the corresponding block. Either I can look upon it as this piece of code or I could look upon it as a huge nested ‘if then else’ which checks for each of these cases in sequence but the whole point about the case statement is that it is a very highly symmetrical statement and the order of occurrence should not matter. For example; the alternative should commute and associate in any order. But the most serious pragmatic problem is that if the evaluation of E itself can create side effects then this semantic equivalence is no longer valid.

(Refer Slide Time: 42:57)



PRAGMATICS OF CASE

```
case E of           if E = 1 then C1;
  1 : C1                  :
    :          ~          :
  i : Ci            if E = i then Ci;
    :                     :
  n : Cn            if E = n then Cn
end
                    provided evaluation of
                    E produces no side-effects
```

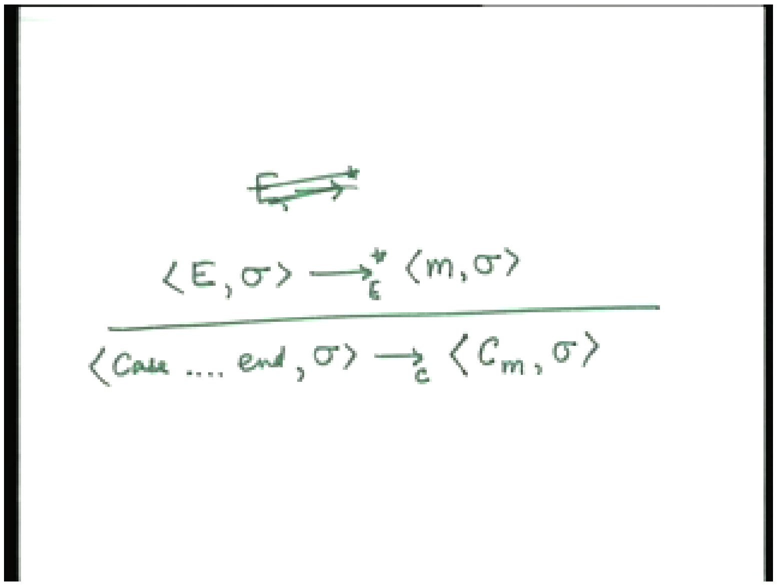
So, the case statement is where you evaluate E. Look at the semantics of the case statement.

$<E,\sigma> \to_e^* <m, \sigma>$

$<\text{case….end}, \sigma> \to_c <C_m, \sigma>$

This would be my rendering of the semantics of the case statement and this rendering means that the expression E is evaluated exactly once. Only if you specify this as a meaning of the case statement can you ensure somehow that there are no arbitrary numbers of evaluations of the expressions E possible and this is a simple enough rendering. So, we would like to ensure that even in the case of side effects if this rendering has to be true then that E should be evaluated only once.

(Refer Slide Time: 44:02)

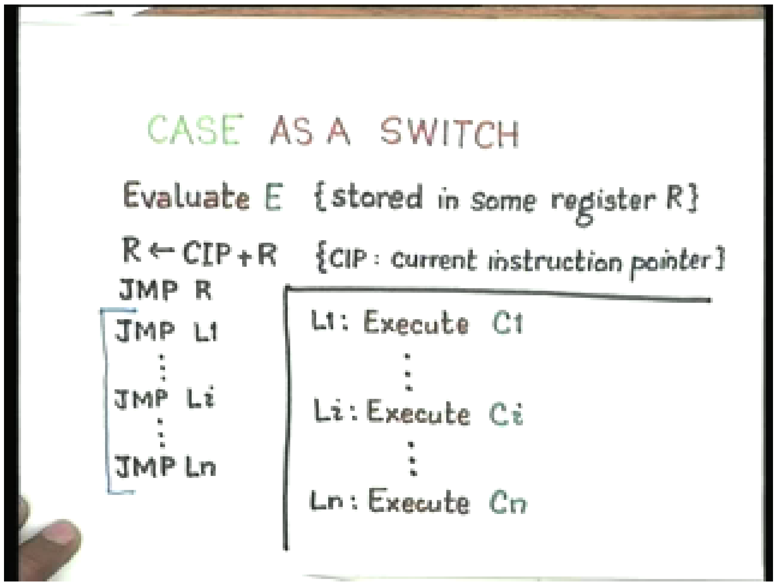


$$\frac{\langle E,\sigma\rangle \longrightarrow_e^* \langle m,\sigma\rangle}{\langle \text{case} \ldots \text{end},\sigma\rangle \longrightarrow_c \langle C_m,\sigma\rangle}$$

Most implementations of the case statement actually look upon the case as a form of a switch. You evaluate E only once and let us assume that the code for evaluating E stores the value of E in some register R. You create what is known as a jump table. If there are n cases to be considered then you generate n jump instructions in sequence and you also generate an instruction with jumps to whatever is the current instruction pointed plus whatever is the value of E.

For the present moment I have taken E to be an integer expression. As long as it is some enumerated data type you can always map that expression to an integer expression from 1 to n. Then I jump directly to an appropriate instruction in this jump table. There is only one possible jump and that itself gives the jump to an appropriate code. If I have the labels L_1 to L_n in successive instructions and if this expression evaluates to some m then I would be jumping to the m**th** instruction in this table. This is the jump table. It is actually like an array of jump statements and you jump to Lm and that in turn is a jump to the actual piece of code which constitutes the body of that case. This will ensure for example that E is evaluated exactly once and it is evaluated very fast.

(Refer Slide Time: 47:02)



CASE AS A SWITCH

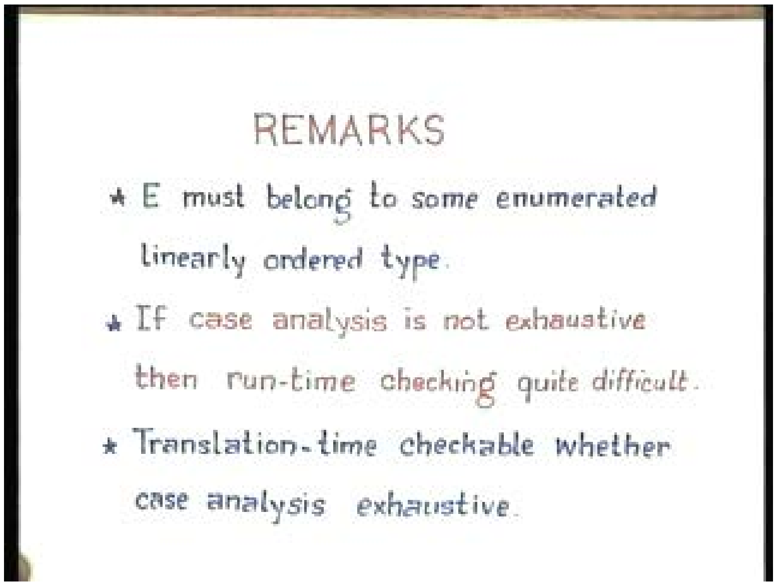
```
Evaluate E   {stored in some register R}
R ← CIP + R  {CIP : current instruction pointer}
JMP R
JMP L1       L1: Execute C1
  :                 :
JMP Li       Li: Execute Ci
  :                 :
JMP Ln       Ln: Execute Cn
```

Since you are not doing repeated evaluations of E, E could be a very complicated expression. It means that you are making a very efficient implementation of the cases by using a switch rather than by using shady equivalents. It is not always equivalent if there are side effects using ‘if then’ or ‘if then else’.

However, if this implementation has to be successful then this E cannot be anything other than enumerated linearly ordered type with a well defined successor predecessor function except for the extremal points. Secondly, if the case analysis is not exhaustive your automatic code generation for a case that is not listed in the case statement will give a ‘no op’. There is going to be a blank instruction. If the case analysis in the statement is not exhaustive then you would have generated this label Lm and you would have also generated a corresponding Lm but there might be nothing in that Lm.

What actually is going to happen is quite unpredictable which is also one reason why the case analysis should be exhaustive and the ISO standard Pascal actually specifies that it should be exhaustive. But however at translation time itself you can actually point out an error for each case statement to determine whether the case analysis is exhaustive or not. Since the runtime checking is going to be very difficult you should do all this analysis at translation time which of course most Pascal compilers do not do and further most implementations of Pascal also assume that you are going to have such a large number of cases that you may not want to exhaustively do case analysis.

(Refer Slide Time: 48:52)



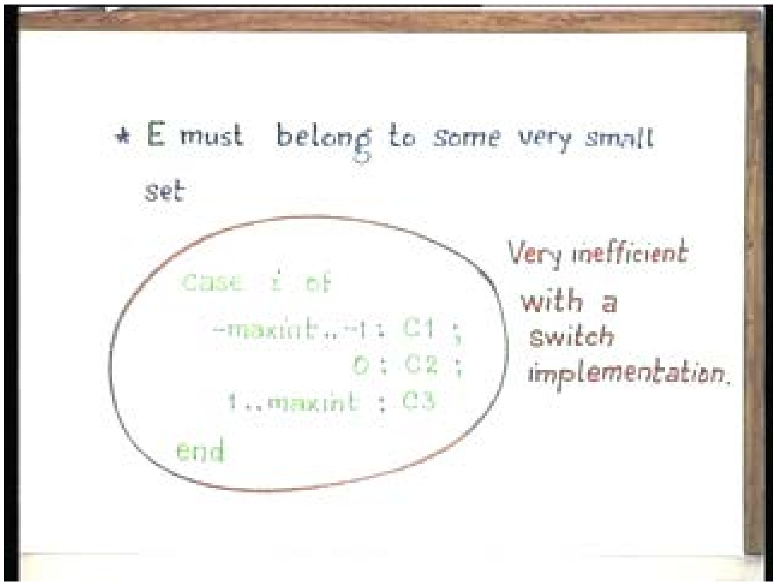
REMARKS

- E must belong to some enumerated linearly ordered type.
- If case analysis is not exhaustive then run-time checking quite difficult.
- Translation-time checkable whether case analysis exhaustive.

What they have is an 'else' clause or an 'other's clause which acts as a default in case none of those cases is true. The problem with that is that I may have just forgotten one case and then without my intending to, it actually goes into the else clause or the 'other's clause and executes something and gives me wrong results. But if the compiler actually did a translation time check and found the case analysis to be non-exhaustive then it could point out a compile time error which will solve the problems. If it is a matter of repeating code that is not a problem because for example Pascal allows you to give lots of case labels for the same block of statement. It is not a matter of repeating code but the fact that you might have forgotten a case is more important and should be pointed out and especially since it is compile time checkable, there is absolutely no reason why you should have an else clause or an other's clause.

It should be some enumerated type but if it is some enumerated type like just an integer which can range from 'minus max int' to 'plus max int' and if this is going to be how your case analysis is going to proceed, then you are going to have two times 'max int' number of 'jump' statements which is a phenomenal explosion in the code length in the code that is generated. But it is in the programmer's own interest in such cases to use an 'if then else' rather than a case statement. If you have a very large number of cases in your enumerated data type but there are a very small number of actually distinct cases then you are better of using a nested 'if then else' than a case statement.

(Refer Slide Time: 52:10)



* E must belong to some very small set

```
case i of
  -maxint..-1 : C1 ;
            0 : C2 ;
   1..maxint : C3
end
```

Very inefficient with a switch implementation.

The programmer has to take a decision if it is better to use a case statement in this kind of a situation which is one extreme. Otherwise supposing E actually belongs to some user defined enumerated data type then there is absolutely no reason why you cannot use a case statement. But if it belongs to a sub range type which is going to be huge then it is a programmer’s decision to decide if it is more efficient to use a case statement or to use an if then else nested.

The compiler is not going to take a decision. In fact that is one of the reasons why the case as a statement is introduced. The case as a switch is meant to provide an efficient switching mechanism. So, all implementations of the case are going to provide it as a switch mechanism because implicitly there is an assumption in the semantics that the expression E is evaluated only once. The compiler is not going to take any decision. If it is a case then it is going to implement it as a switch. That is what any compiler writer would ensure. It is a programmer’s decision if you want an inefficient case like this or if you would rather use an ‘if than else’ knowing fully well how the case and the ‘if then else’ are implemented or knowing fully well what the semantics of the case and the ‘if then else’ are. It is the programmer’s decision and not the compiler writer’s decision.

The case as a switch actually is the most common implementation of a case statement because you really want to look upon it as a multi-way tree of ‘n’ cases. You do not want to worry about ‘if then else’. You would like to be faithful to the syntactical structure of the statement itself. It is a programmer’s decision and this decision of a lot of compilers to provide this extra feature of an ‘other’s clause is actually completely misplaced because it can only create more errors due to forgetfulness and carelessness than solve problems. The standard Pascal does not allow for it. But most other implementations of Pascal that I have seen actually allow for an ‘other’s clause for a default clause when the case analysis is not exhaustive.