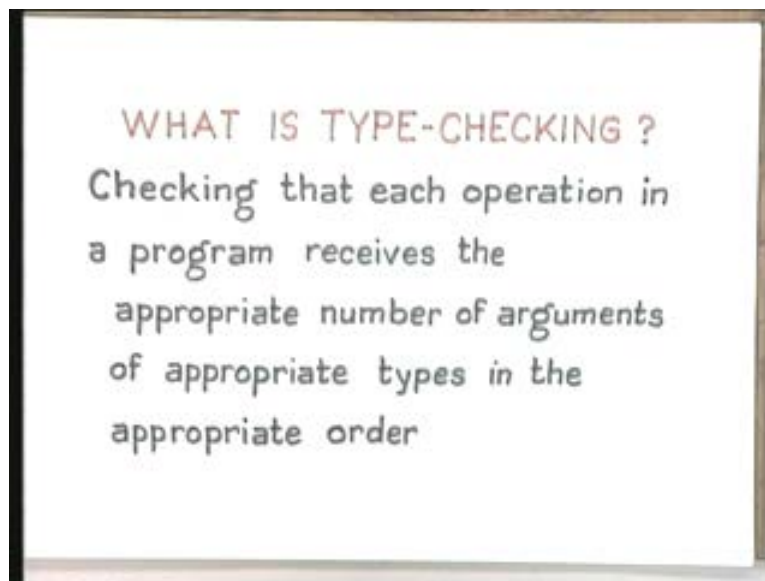


**Principles of Programming Languages**  
**Dr. S. Arun Kumar**  
**Department of Computer Science & Engineering**  
**Indian Institute of Technology, Delhi**  
**Lecture - 34**  
**Type Checking**

Welcome to lecture 34. So today we will do some backward integration. We have already done the hardest parts of type checking. So we will just try to integrate it with whatever we already know about programming with our simple, functional and imperative programming languages. So always keep in mind that it is possible often to do some static type checking that is at compile time. When you come to general programming languages the question asked is what type checking is.

In the case of lambda calculus our type checking was entirely governed by the fact that all lambda abstractions are unary functions therefore you had to do the type checking essentially only for unary functions. Since all other kinds of functions were carried forms in the lambda calculus certain issues were overlooked. So essentially in its most general form in an applied lambda calculus or in a programming language type checking is just the fact that you have to check that each operation in the program receives the appropriate number of arguments of the appropriate types and in the appropriate order.

(Refer Slide Time: 02:12)

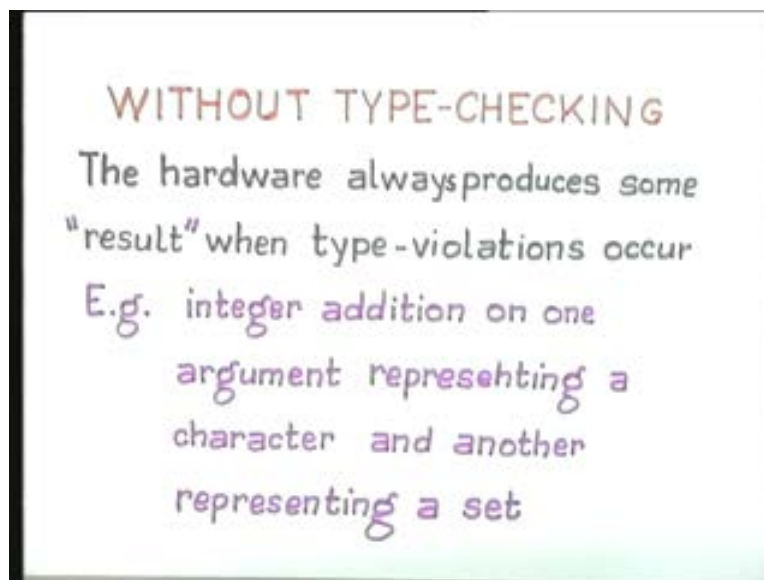


Appropriate number of arguments was automatically taken care of in the lambda calculus by the fact that it took arguments one at a time each and as and when an argument was obtained it would have to type check. Appropriate types were automatically taken care of in the rules for type checking in the lambda calculus. In the appropriate order was also automatically taken care of.

However, in any applied lambda calculus where there could be n-ary functions n-ary operations what it means is that you will have to check all these things. For example if the operation is not commutative or it is over a multi sorted structure then it all the arguments should come at appropriate times and should come in the appropriate order and they should be all of appropriate types. And the question is of course why you require type checking. Probably the most direct answer is to answer the question what actually happens without type checking.

We already have an idea of what happens in the untyped lambda calculus. What happens there is that any lambda term might be applied to any other lambda term and the result is actually a lambda term but you may not always be able to interpret it. And essentially the same thing happens in hardware. Whatever may be the operation and whatever may be the arguments the underlying hardware which is totally untyped, of course the difference with the lambda calculus is that the hardware is of course all of untyped data, bit strings if you like, the hardware always produces some result when types are not with respect to.

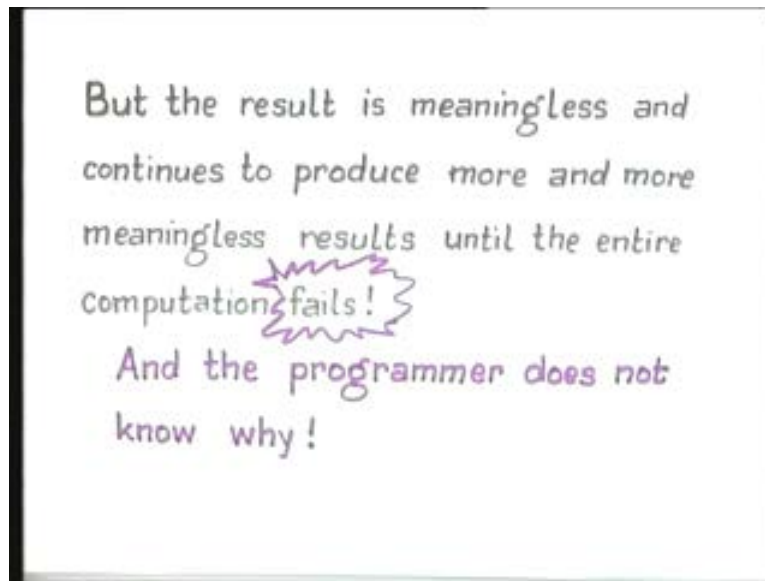
(Refer Slide Time: 04:32)



So, for example you could do integer addition on two arguments one of them representing a character and another representing a set. And what actually happens is since the hardware is untyped rather memory locations are untyped, registers are untyped they are just bit strings so what can happen is that you will get something but the problem is that you won't be able to interpret the result. In fact the result of one type violation might actually be carried forward as an argument for another operation and so on and so forth. It will continue to produce more and more meaningless results till the entire computation actually fails. and of course this "fails" is a judgmental word the hardware does not know anything as "failed" and question of whether it has failed or not really depends upon your interpretation. So it will produce something and it will be very hard for the programmer to actually detect what exactly went wrong.

Therefore the question of failure is not something that might be immediately detectable too. It is something that might come out years and years after the software has been sold for a phenomenal price to some unsuspecting customer who actually believed that it worked till something happened.

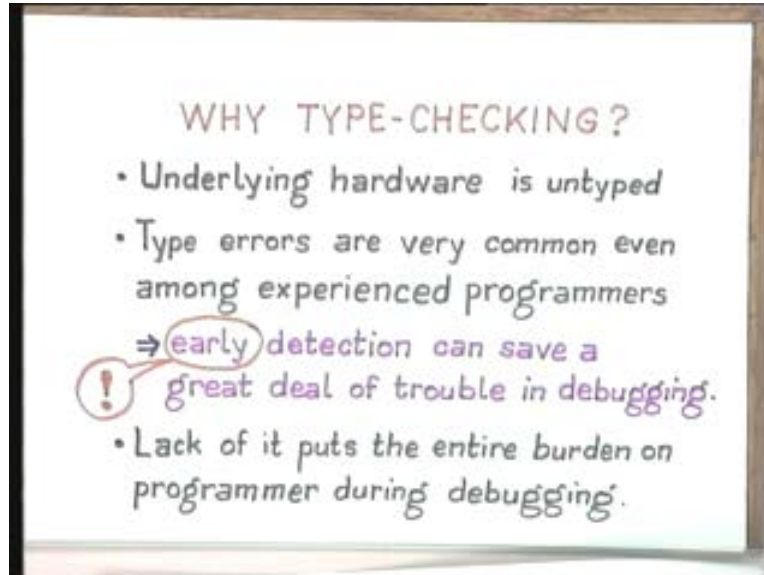
(Refer Slide Time: 06:03)



It is true that you cannot get rid of all programming errors in any full proof fashion ever. What you can do at most is to introduce checks and one important check wherefore is type checking. Since the underlying hardware is untyped it is a good idea to introduce type checking among other things. So, type checking actually is a much more general term. For example, when you look at array bounds checking in programming languages that is also a part of type checking because the index set for the array is a sub range type and therefore you have to do type checking. So type checking is a fairly general word to catch a whole lot of issues not necessarily all.

Another unfortunate thing is that type errors are very common even among experienced programmers which means that early detection if you can detect it early then you can save a great deal of time and effort in wasted execution and in debugging. Even though what it means is that there is going to be a greater overhead on the compilation. it is often better to detect it early so that the production runs are not compromised.

(Refer Slide Time: 08:03)



The entire burden of detecting type errors or type violations and type violations include things like array, bounds, checks and so on and so forth is entirely on the programmer during its debugging phase. So early detection implies that what we would like to do is actually detect it at compile time and this is called static type checking. So static type checking means that you do the detection of part violations as far as possible during compilation or translation time. So what this means is that compilation is slowed down but it also puts in other things. For the typed lambda calculi you can do the type checking during translation time always but it is not in general possible for all features of programming languages.

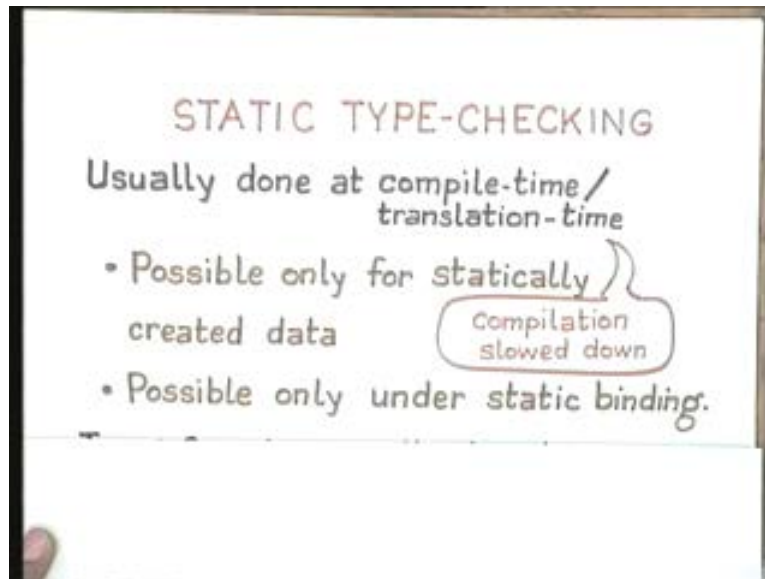
For example, it is only possible where you have statically created data that means through the mechanism of explicit declarations, declarations either before or after use but declarations which clearly create places for data. So dynamically created data cannot be for example statically type checked. So any kinds of pointer mechanism, list mechanisms the data of which changes during run time require dynamic type checking.

For example, for most Pascal data structures which are static they have explicit declarations which give the full size of each piece of data. Whatever is stored in the activation stack is something that can be statically type checked. Whatever is going to be stored in the heap is something that will have to be dynamically type checked. So an early type checking strategy means that as far as possible whatever declarations are there they can be type checked. You do the type checking at compile time to say one execution time.

For dynamic data structures of course you require dynamic or run time type checking and this is something for example which Pascal actually has in the run time descriptor for heap data. Of course all these implies that you can do static type checking only for statically created data and it is possible only where bindings are static where bindings are

determined at compile time. If there are bindings which are determined at run time then for example in languages like LISP and APL and SNOBOL what happens is that you cannot do type checking at translation time.

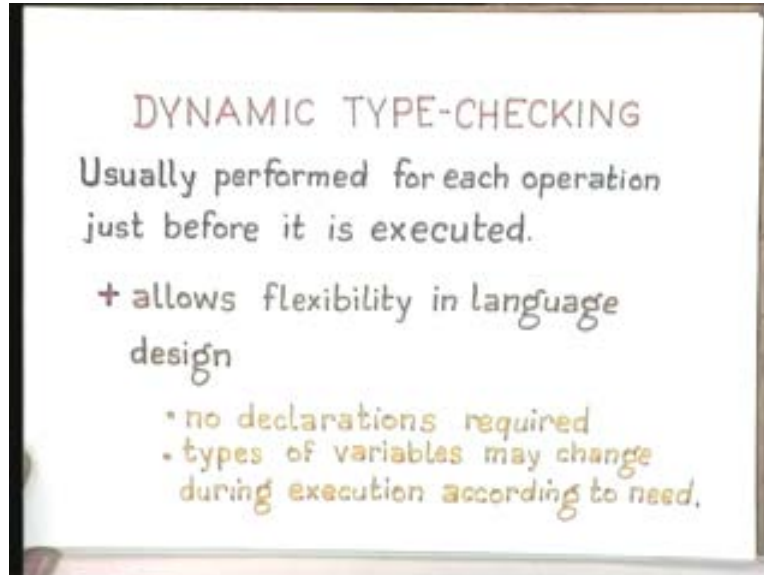
(Refer Slide Time: 11:25)



So normally what happens is that all this type information is really the information that is stored in the symbol table during the process of translation. Thus, type information is usually some sort of attribute which is part of the compilation process and that is how it is implemented and so on. On the other hand, if you look at dynamic type checking it is really something that has to be performed for each operation just before it is executed.

As I said from the point of view of the hardware an operation can always be executed and it will always give you some result. But if you want to ensure that type violations do not occur then what it means is that you will have to perform this type checking before each operation is actually executed. So there has to be a code generated for checking. So what it means is before you execute an operation just check that the number of arguments that are required for that operation available, each argument is of the right type and they are available in the right order. That is really all that it means. But the point is that a program is full of little, little, little operations and what it means is that during execution you will have to do this checking before executing each operation and that can slow down executions considered.

(Refer Slide Time: 13:39)

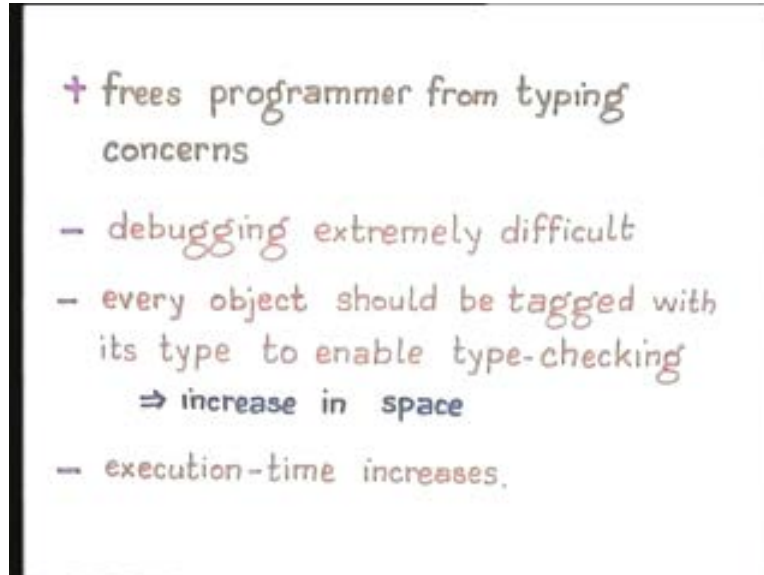


However, what it does is it allows a sort of flexibility in language design. It does not force you to do early binding, static binding etc. The kind of flexibility also means possible abuse that is an important thing to realize. Any kind of flexibility could also imply potential abuse which means that no declarations may be required in some of these languages which allow for a flexible typing mechanism whose type checking is done only at run time.

More importantly what can happen is that the types of variables may never remain static throughout the scope. Actually the types of variables may change during execution according to some perceived need. But there is a price to pay for these things and what it means is that you cannot take a print out of the program and expect to try to debug it to see what is wrong. If such programs fail for some reason or give you unexpected results what it means is that you will have to sit constantly with the debugger and try to determine what exactly is wrong with the program.

So especially the type variables changing during execution can be very confusing for very large software and even for software which is just as large as compiler can be very confusing and debugging can be very hard. Of course the other kind of flexibility abuse that it allows is that it frees the programmer from typing concerns but it makes debugging extremely difficult.

(Refer Slide Time: 15:36)



The extra overhead that it adds when you have dynamic type checking is that of course you could take the attitude that I do not require any type checking it is too slow forget about it no type checking but that means that there could be strange type violations that might occur and they might all actually produce results which on the surface look meaningful. But if you want to do some type violation detection and if you decide to delay it to run time then what it means is that you will have to also tag every object with some type information to enable this kind of type checking attractor. So one thing of course is that it often means a certain amount of increase in space which may or may not be negligible but what it more often does is that it actually slows down execution.

Therefore, most languages do some form of type checking because even when you look at them as just forms of the untyped lambda calculus still they use operations of the underlying hardware and there are representational differences and so on and so forth so in the underlying domain  $t$  to which they are applied they do some elementary type checking. But when you do these things dynamically what it means is that you are going to considerably slow down execution so that is an extra overhead. Thus type checking really is something that is not an absolute essential for programming but it is a desirable thing to have in your translator or in your run time system.

The attitude towards types has actually varied tremendously from the early programming languages. But more and more the feeling has come that really type checking is something that should be performed at sometime before execution which means either static or dynamic mainly to detect errors in large software. So what has happened historically is that the early languages like FORTRAN had really no type except integers and reals. But more and more typing information has been gathered. So for example with Pascal you get a fairly rigid form of static type checking with certain amount of dynamic type checking to ensure that array accesses are not violated, array boundaries are not



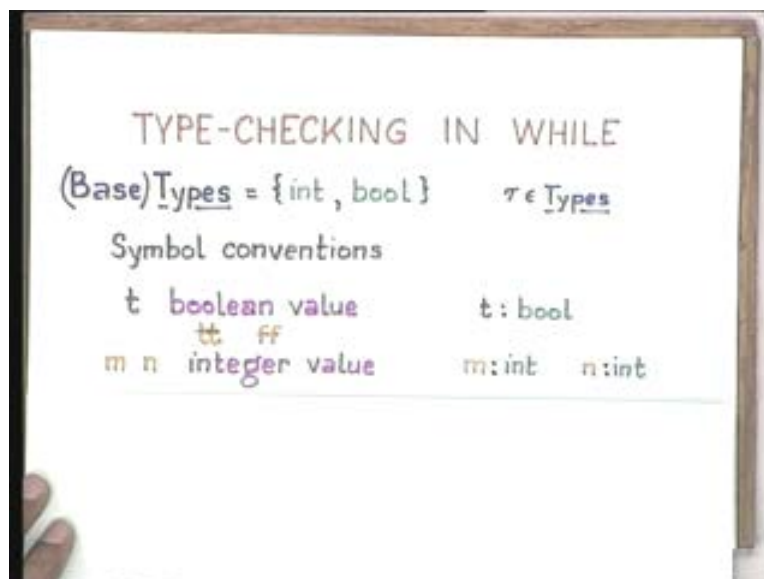
violated and to ensure that heap data is whenever is created and destroyed is of the appropriate type.

In C for example there is an automatic possibility of casting one type into another which is a form of type cohesion which is quite acceptable in many cases. You explicitly want to **cohere** certain types. But if you look at C++ again the rigidity of a Pascal typing system has come which means that we have realized somehow that an early detection of type errors is somehow important to reduce debugging time to reduce overheads of run time type checking. Hence a large amount of C++ type checking is actually static.

However, because C++ is a super set of C and every C program should be compilable and executable in a C++ environment is the basic principle. This means that the various kinds of abuse that you can do in C are also possible in C++ without any problems. So type abuse that you can do in C is also possible in C++. But if you were to restrict your C++ usage to just the standard libraries that are available with the C++ and just the C++ constructs that are new then what you will find is that there is a tremendous amount of static type checking. This is one of the reasons why compilation in C++ is considerably slower than that in C. They actually do a tremendous amount of static type checking to ensure somehow that type violations do not occur.

But the typing overhead that C++ carries with it is the fact that it has got a very type flexible language sub language which is open to enormous amount of abuse. So let us carry forward our type inferencing mechanisms or rather carry backward type inferencing mechanisms because we have done the hardest part of type checking. We know how to type check higher order functions. What we do not know is how to type check the underlined data. **Hence let us finish that.**

(Refer Slide Time: 23:36)





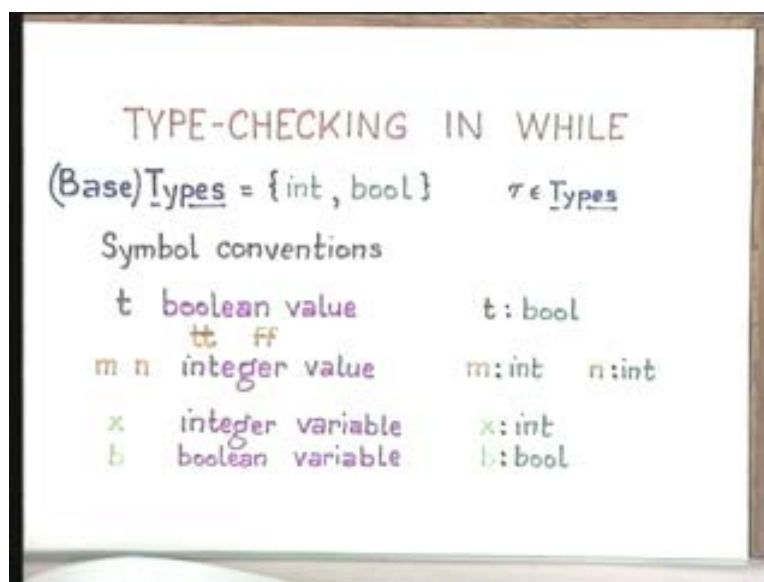
So let us assume some elementary base types for the present. I will just take the while programming language just to give you a flavor of how type checking can be defined in a structural inductive fashion so that what it means is that you can actually do a early type checking in many cases especially for a statically scoped language.

A statically scoped language means that with the binding is early, the bindings are also static and so type checking can also be done static, memory allocation can be done statically for all except the dynamic data structures so the type checking can also be done statically with relocatable addresses and so on. Therefore we will just assume that our collection of types which is actually if you go back to the lambda calculus what it means is that we are talking about the collection of base types. So it is just integers and Booleans and then we will use some symbol conventions.

I am changing one thing, I am using I am using this dark brown t to denote one of the Boolean values true or false and t is of type bool. Then of course I am using m and n as integer values.

Even though I claim that the underlying hardware is largely untyped very many architectures came up during the 70s and 80s which actually did a tagging an automatic hardware tagging of memory locations to do elementary type checking in a speedy fashion. What it means is that they would actually tag the memory locations to be either floating point or integer or Boolean or character, they would just use these four basic tags and all other type checking especially for example of higher order functions and so on was the compiler's responsibility. So they did use some form of either tagging or a separation of the elementary data types somehow to enable at least basic type checking to be performed, it is not a very sophisticated type checking. Of course we will have things like integer variables may be we will also have Boolean variables etc.

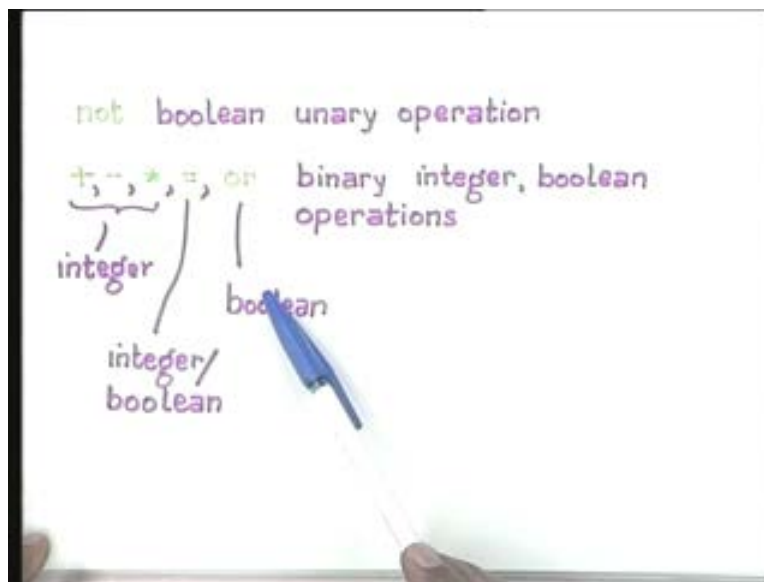
(Refer Slide Time: 24:52)



For the moment we will just assume that there is some way of determining types for the most basic constructs in the language which means variables. I am not actually explicitly introducing declarations here but ideally what you would do is you would introduce declarations and that would be the explicit way of determining the types of variables at compile time and using them somehow to do the type checking for the rest of the program.

Now let us just see a structurally inductive definition of type checking without worrying too much how the base types are determined as they could be determined in several ways. I previously defined two expression languages an integer expression language and a Boolean expression language and then a language of commands. In fact what I did was I did the type separation in the syntax so what I will do now is I will mix up the syntax of expressions, one cannot mix up the syntax of expressions with commands but one can mix up the syntax of Boolean and integer and expressions and see how type checking can be done.

(Refer Slide Time: 26:44)



We will assume that these binary operators are available and I will go ahead and also look for equality as being either between integers or Booleans. These operations are integer operations, (Refer Slide Time: 2:45) this is the only other Boolean binary operation. It is enough to have one representative of each kind actually otherwise more than anything else it becomes boring and repetitive. So we will just assume that this is a sweet of operations and we will define the language and we will define what is known as a static semantics.

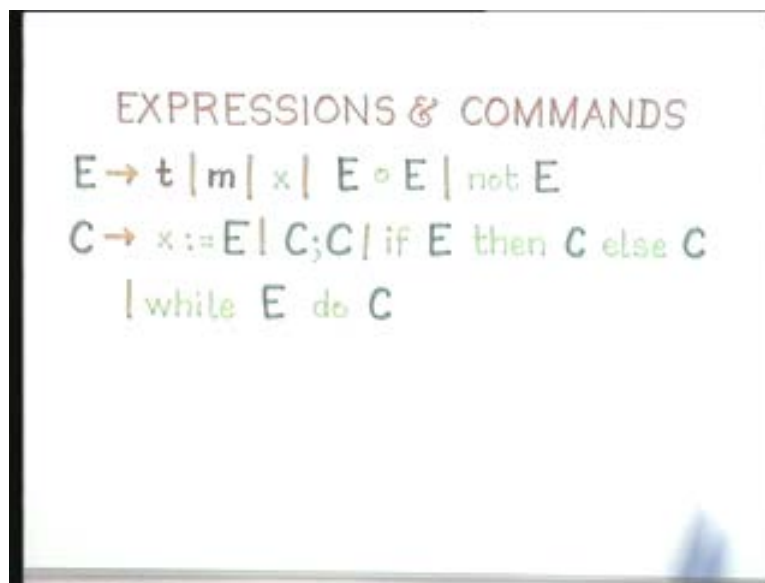
So our expression language somehow consists of all these expressions. So essentially  $t$  is the truth value,  $m$  is an integer may be so somehow let us assume that we can distinguish between  $t$  and  $m$ ,  $x$  is a variable it could be a Boolean variable or an integer variable then (Refer Slide Time: 27:52) this is a binary operation on expressions it could be any of the

binary operations we have seen and there is a unary operation the only unary operation in expressions. The commands remain as they are, there is nothing much to do with commands.

The only thing that I have done now is from the previous incarnation of the “while language” is that I have collapsed the Boolean expression language and the arithmetic expression language into a single language which is just one set of production rules partly to ensure that we can actually do the type checking and it is not necessary to separate them very early at the level of syntax. It is possible to do the type checking and separate out what is well typed from what is not well typed.

As far as commands are concerned expressions denote values so they have a type. Hence depending on what value it is supposed to be written it will have a type whereas when you are looking at commands the commands just change state. So commands really do not have a type except that we should ensure that the underlying expressions are well typed and we should ensure for example here since there is no separation between Boolean and arithmetic expressions you should ensure that you only have a Boolean expression here and similarly for the “while”. So commands themselves will not have a type, we will just assume that there is a unary predicate of well **formed-ness** which allows us to check whether a command is properly formed expressions of the appropriate type.

(Refer Slide Time: 29:51)



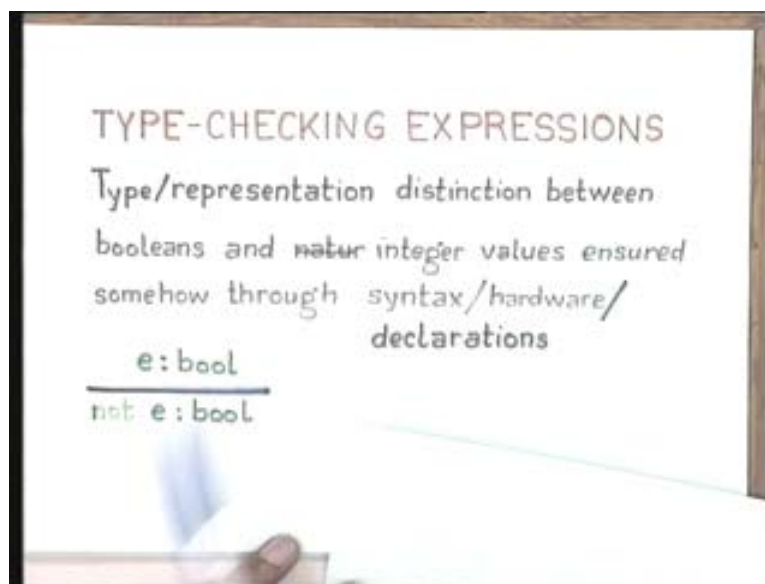
And of course a command at the top does not type check or rather it is not well typed. A standard thing is to say that when something is well typed you say that it type checks. A command does not type check if any of these arms do not type check or if this is for example not a Boolean expression, this might type check to an integer expression in which case of course the command is not well formed. So these are the elementary considerations which we will use for type checking. And principally we have to somehow

be able to extract information about when an expression is a Boolean expression and when it is an integer expression.

Now we will assume some of the base types or either representation distinction or actually may be through some declarations somewhere. There is somehow some way of distinguishing the base types which may not be entirely true on a **bare** machine. So how does one type check expressions?

Let us take the unary operation.  $E$  is a Boolean expression where those type axioms like  $m$  is of type `int` and  $t$  is of type `bool` and so on that is already available and if  $E$  is of type `bool` then `not e` must be of type `bool`.

(Refer Slide Time: 31:38)



The point is that this can be done in a recursive descent fashion. And after the kind of complicated things we have done all this actually looks really trivial except that it seems more and more essential in programming languages to do this kind of type checking as early as possible even though it appears to be totally trivial.

(Refer Slide Time: 32:26)

$$\frac{e_1:\tau_1 \quad e_2:\tau_2}{e_1 \circ e_2:\tau_3}$$

$+ - *$	bool	int
bool	err	err
int	err	int

So, for any binary operation what I am going to have is that if  $e_1$  is of type  $\tau_1$  and  $e_2$  is of type  $\tau_2$  then  $e_1$  binary operation  $e_2$  is of type  $\tau_3$  except that now I have to worry about what  $\tau_1$ ,  $\tau_2$  and  $\tau_3$  are. I have circled  $\tau_1$  in brown,  $\tau_2$  in sky blue and  $\tau_3$  in red and now this rule is subject to these tables which actually give you the type rules.

(Refer Slide Time: 32:39)

$$\frac{e_1:\tau_1 \quad e_2:\tau_2}{e_1 \circ e_2:\tau_3}$$

$+ - *$	bool	int
bool	err	err
int	err	int

or	bool	int
bool	bool	err
int	err	err

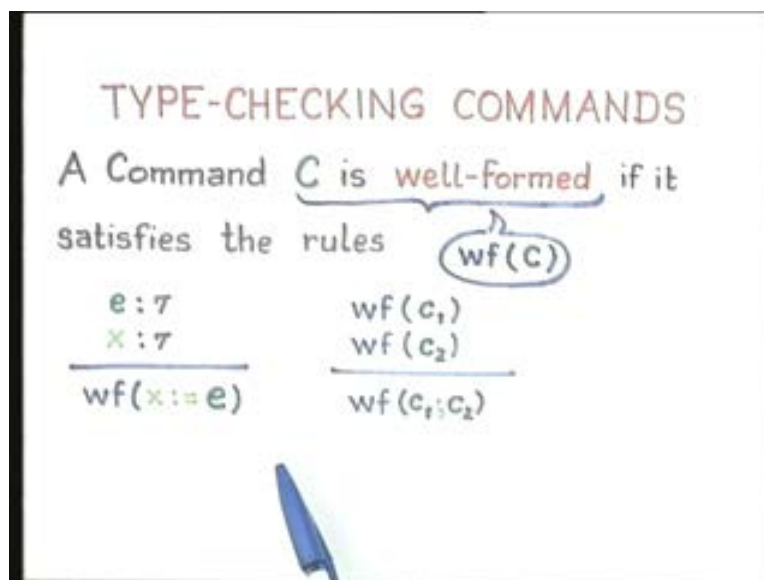
$=$	bool	int
bool	bool	err
int	err	int

So if you take the arithmetic operations plus minus and star so what happens is an arithmetic operation type checks so  $\tau_3$  has to be integer only if each of  $e_1$  and  $e_2$  is of type integer. So here is where the type checks in all other cases there is a type error. So

this is the only non error type otherwise there is always some type error and this is compile time detector. Therefore, if you look at equality of course either you are comparing two Booleans or you are comparing two integers in all other cases there is a type error. And in the case of or it is only when the two operations are Boolean that you actually type check in all other cases you have, or is meant to be a Boolean operation so that is all. Hence all those are there is to type check once you have dealt with polymorphism and so on and so forth. The hardest parts of type checking are really type checking higher order functions for which you require the kind of heavy inference rules that we had for the typed lambda calculus.

The red entries here are  $\text{tou}_3$ , here this has to be a Boolean so this has to be a Boolean and otherwise it is actually extremely trivial. The really hard part is type checking higher order functions. And once you have declarations available to you, you carry the carry forward the type information in a static environment and you perform the type checking just like you had a dynamic environment in which you wrote rules for the execution. So you can look at PL0 compiler it essentially does similar forms of elementary type checking and in the case of commands all that we require is that we just have to go down deep into the command and in a recursive descent fashion we should just ensure that the commands are well formed. So I would say that command C is well formed, I will use the predicate  $\text{wf}(c)$  to indicate that a command type checks if it satisfies these rules. So the basic thing is that the assignment should be between things of compatible types. Now, in most programming languages you have type cohesion mechanisms. That means casting mechanisms so that you can either explicitly or implicitly coerce an integer value to be real value or truncate a real value in order to give you an integer value or whatever.

(Refer Slide Time: 36:34)

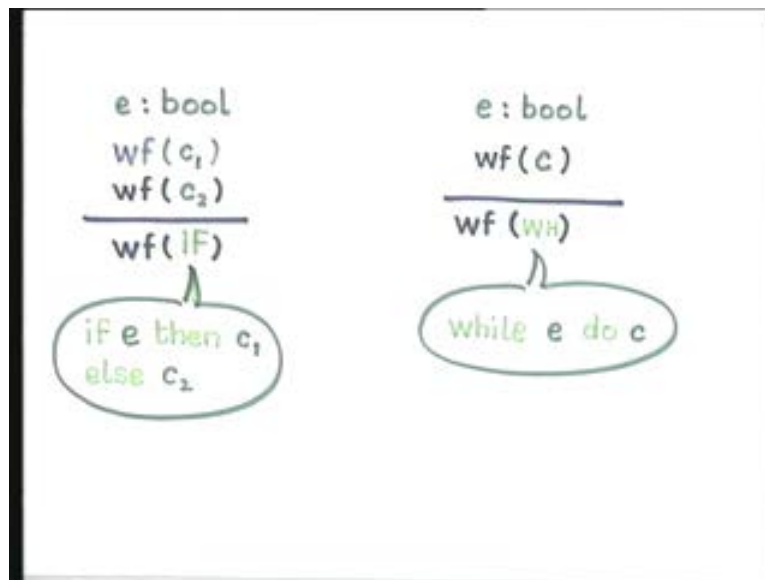


In many programming languages including Pascal actually the integer to real conversion is implicit. If the left hand side is real variable and there are integers on the right hand side then there is an implicit function which converts the integers to reals and then does

the assignment. In languages like ML they have actually made it explicit which is actually a good idea in the sense that if it is explicit then you know that the programmer actually knows what he is doing whereas if it is implicit it is not clear whether the programmer is aware of what he is doing and extreme case is in the case of FORTRAN where you have the same symbol for division and the same symbol is used for integer division and real division and it is not clear when the programmer will make a mistake, when we will get a quotient which is a real number and especially with variables not necessarily being declared with symbolic conventions and so on.

In FORTRAN it is actually a complete mess in terms of readability whereas the later languages actually allow for some form of implicit or explicit type cohesion. The type cohesion function let us assume its explicit is really a function from lets say integers to reals or reals to integers, truncation is a function from reals to integers and given an argument by the standard application rules of the lambda calculus you get an answer which is an integer and you can write out those rules for both implicit and explicit type cohesions.

(Refer Slide Time: 39:16)



And in the case of other commands you just follow the normal forms of structural induction. If these two commands individually type check then their sequencing also type checks. The “if then else” and the “while” just require this kind of checking their individual bodies have to type check and it should be ensured that the expression is a Boolean then the individual commands also type check.

Therefore, type checking rules are usually quite trivial and it is because they are so trivial and can be done so simply you always recommend early preferably static type checking that will isolate errors early in the program, it is possible to do it most of the time except for dynamically created data, if you have a language in which you insist some

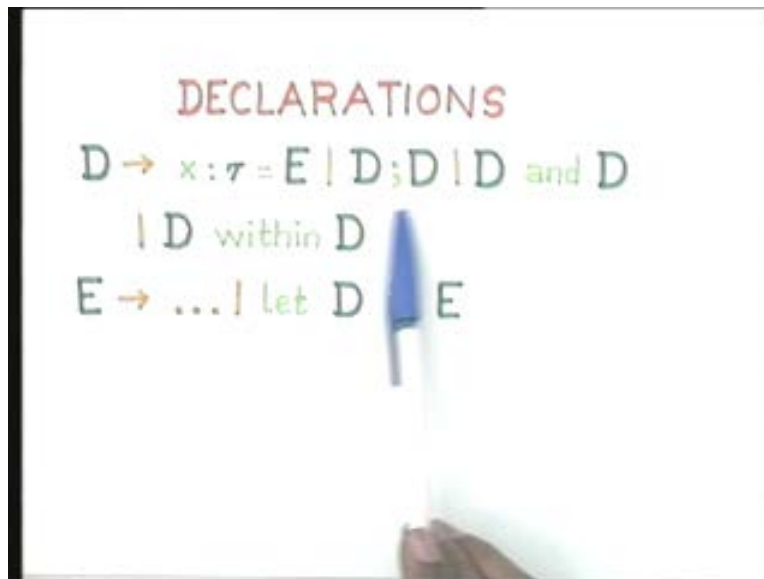


declarations before use a large amount of the type checking can be done at compile time and errors can be pointed out.

Hence it saves on execution time, it saves on having large run time descriptors, it saves on generating code to do the type checking at run time and it saves on execution time so more and more languages actually try to do type checking early. So the most recent languages actually do all the type checking and so type checking is one important reason why many languages have moved from dynamic binding mechanisms to static binding mechanisms because then you can do the type checking at compile time, at translation time and you do not need to wait for run time type checking.

So the introduction of declarations brings in a certain amount of complication. Now let us look at a purely functional fragment an ML like functional fragment then what it means is that you will somehow have to make a distinction between what has been declared now and what was free, what could be re-declared now for example in a fresh declaration. So a typical typed declaration of a value could be like this and let us assume that the other expressions in the language are the same and there is this one extra construct a let construct.

(Refer Slide Time: 42:03)



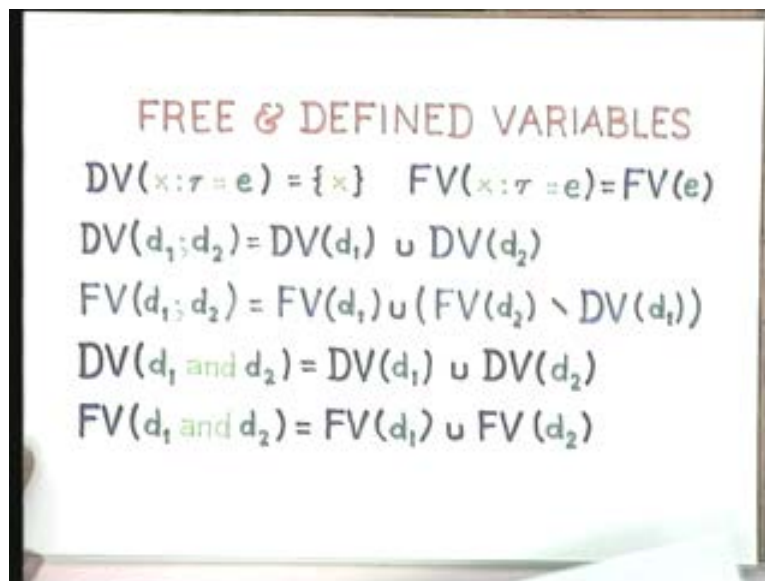
Then of course we have the normal declaration mechanisms, the sequencing of declarations or rather if you like a pipelining of declarations, parallel evaluation of declarations or rather independence of declarations and nesting on the declarations. So now the only sort of complication it raises is to actually know what the free variables in an expression are, what are the bound variables etc. The moment you introduce declarations means that you have a concept of free variables and bound variables. You have a concept of a variable being re-declared to be of a different type therefore creating a whole in the scope and therefore it has to be taken into account during the type checking process.

Hence with a language with declarations like this, what it means is that you have to give some structurally inductive definitions for what exactly is being declared and what exactly is free.

So let us look at free and defined variables in a functional language. Somehow they do not use the word declaration they say it is a definition so I will use these two terms interchangeably. In an imperative language it is called a declaration. So the declared variable in an elementary value expression of this form is just  $x$ . The free variables in this definition are all the free variables of  $(e)$ . So  $x$  itself could occur in  $e$  but we know from our run time semantics that, that  $x$  which might occur in  $e$  refers to an  $x$  that was previously declared and this is a re-declaration of  $x$ . So  $e$  may or may not contain  $x$  but whatever  $x$  it contains it is not the same  $x$  as this which is being freshly declared.

And in the case of sequencing of declarations the declared variables are just the unions of the declared variables and free variables of  $d_1; d_2$  are just the free variables of  $d_1$  union, of course you could use the variables that were declared in  $d_1$  in order to have the declarations in  $d_2$  that is in expressions within the declarations in  $d_2$  which means you just exclude those variables which were declared in  $d_1$  from the free variables of  $d_2$ . Then in  $d_1$  and  $d_2$  there is a condition of disjointness that the same variable cannot be declared in both  $d_1$  and  $d_2$  and cannot be used. So the declared variables or the defined variables are just the unions of the individual defined variables in the declarations and the free variables are also just the union.

(Refer Slide Time: 45:48)



FREE & DEFINED VARIABLES

$$\begin{aligned} DV(x:\tau = e) &= \{x\} & FV(x:\tau = e) &= FV(e) \\ DV(d_1; d_2) &= DV(d_1) \cup DV(d_2) \\ FV(d_1; d_2) &= FV(d_1) \cup (FV(d_2) \setminus DV(d_1)) \\ DV(d_1 \text{ and } d_2) &= DV(d_1) \cup DV(d_2) \\ FV(d_1 \text{ and } d_2) &= FV(d_1) \cup FV(d_2) \end{aligned}$$

And in the case of nested declarations what you have is that in a nested declaration of the form  $d_1$  within  $d_2$  what you are actually declaring is only whatever is in  $d_2$ . If you remember I said that since the declarations in  $d_2$  could be very complex you abstract out some of the sub expressions give them new names in  $d_1$  and use them but essentially at the end of this declaration only the environment created by  $d_2$  is available. So the

declared variables of this declaration are just the variables of  $d_2$  and the free variables here of course are all the free variables of  $d_1$  union all the free variables of  $d_2$  which are not free in  $d_1$ .

All this of course is meant to define the notion of free variables in the expression language. So, for all other expressions the notion of free variables remains unchanged and what I mean by all other expressions is the expressions of the form  $e_1$  binary operation  $e_2$  or  $\text{let } d \text{ in } e$  and so on and so forth.

(Refer Slide Time: 47:27)

$$\begin{aligned} DV(d_1 \text{ within } d_2) &= DV(d_2) \\ FV(d_1 \text{ within } d_2) &= FV(d_1) \cup (FV(d_2) \setminus FV(d_1)) \\ \hline FV(e) &\text{ remains unchanged} \\ FV(\text{let } d \text{ in } e) &= (FV(e) \setminus DV(d)) \cup \end{aligned}$$

And in the case of a let construct what you have is just that the free variables of this (Refer Slide Time: 47:37) are just the free variables of  $e$  excluding all the free variables of  $e$  which have been declared in  $d$  and including all the free variables that are already in  $d$ .

(Refer Slide Time: 48:12)  $d_1$  within  $d_2$   $d$  minus declared variable  $d_1$ .

Now essentially we require just this much and this notion of free and declared variables is what is going to be used in processing declarations as a context.

(Refer Slide Time: 48:43)

$$\begin{aligned} DV(d_1 \text{ within } d_2) &= DV(d_2) \\ FV(d_1 \text{ within } d_2) &= FV(d_1) \cup \\ &\quad (FV(d_2) \setminus BV(d_1)) \\ \hline FV(e) &\text{ remains unchanged} \\ FV(\text{let } d \text{ in } e) &= (FV(e) \setminus DV(d)) \cup \\ &\quad FV(d) \end{aligned}$$

So, if you remember what I said about contexts a context is just a collection of variable to type bindings. So, given a program for which you want to do static type checking all the free variables in the program should have types available in the context otherwise you cannot do type checking of the program. Of course a complete program does not have any free variables but inductively speaking if you go through a program segment within a larger program, let us say an expression is a particular program segment in a functional programming language then all the free variables in that expression should have bindings in the context and the bindings defined by that context are exactly what you require in order to do the type checking for these expressions.

I have already pointed out that the notion of context is very similar to the notion of a run time environment, you use the same updation, you use the fact that you have to look up the run time environment each time in order to find values during run time but in the case of a context you have to look at the context in order to determine types, you cannot determine values necessarily because you are not running the program yet but you can determine types and it has exactly the same structure and you can reproduce using the notion of whatever is free and whatever is declared rules for type checking statically which are very similar to rules for type checking in the lambda calculus which are very similar in some sense to the run time environment mechanisms which we use. So this whole type checking since it is going to be done statically whatever we have done is supposed to be static this whole kind of semantics which deals not with values but with only types is called the static semantics for a programming language which is something I have delayed a lot in coming to but essentially if you were to look at in any modern treatment of programming languages there is syntax, pragmatics, static semantics and dynamic semantics. So whatever we have done using the environments row and so on represents the dynamic or run time environment and all that semantics is the dynamic semantics and a similar semantics that we define for type checking is static semantics. So

a complete language document is really complete only if it defines the syntax, the static semantics and the dynamic semantics completely.