# Basic Principles of Programming Language

Abstraction

Avoid requiring something to be stated more than once; factor out the recurring pattern.

Automation

Automate mechanical, tedious, or error-prone activities.

Defense in Depth Principle

If an error gets through one line of defense, then it should be caught by the next line of defense.

Elegance

Confine your attention to design that look good because they are good.

Impossible Error

Making errors impossible to commit is preferable to detecting them after their commission.

Information Hiding

Modules should be designed so that

1. The users have all the information needed to use the module correctly, and nothing more.
2. The implementor has all the information needed to implement the module, correctly, and nothing more.

Labelling

Avoid arbitrary sequence more than a few items long; do not require the user to know the absolute position of an item in a list. Instead, associate a meaningful label with each item and allow the items to occur in any order.

Localized Cost

Users should only pay for what they use; avoid distributed costs.

Orthogonality

Independent functions should be controlled by independent mechanisms.

Portability

Avoid features or facilities that are dependent on a particular computer or a small class of computers.

Regularity

Regular rules, without exceptions, are easier to learn, use, describe, and implement.

Syntactic Consistency

Things that look similar should be similar and things that look different should be different.

Zero-One-Infinity

The one reasonable number in programming language design are zero, one, and infinity.

**Challenges of Programming Education**

Acquiring and developing knowledge about programming are a highly complex process. Novice programmers have to overcome a wide range of difficulties. Programming courses are regarded as difficult, and often have very low passing rates. In this section we discuss some of the goals and problems of programming education.

**Goals, Objectives, and Outcomes**: There are five overlapping domains that students should acquire in an introductory course:

• **General orientation** — the capabilities and applications of programs;

• **The notional machine** — an abstract model of the computer used for executing programs;

• **Notation** — the syntax and semantics of a particular programming language;

• **Structures** — the structuring of basic operations into schemas and plans;

•**Pragmatics** — the skills of planning, developing, testing, debugging, documenting, etc.

None of these issues are entirely separable from the others. That is the main source of difficulties for students since they attempt to overcome all these different kinds of difficulties at once .With respect to the above domains, goals and objectives of an introductory programming course can be summarized as follows:

• **Main goals:**

– Become familiar with the fundamental concepts of computer science.

– Develop proficiency in an engineering problem solving and design methodology.

– Understand the importance of advanced information technologies.

**• Main objectives:**

– Use computers and application software as tools to solve problems.

– Analyze, design, build and test operational solutions.

– Acquire the foundation of algorithmic processes.

– Learn to exploit the educational and professional resources available on the Internet and World Wide Web.

– Develop a framework for considering the ethical implications of advanced information technology.

While goals and objectives of any particular programming course can be stated clearly, the desired characteristics of the resulting programmer are still a bit fuzzy. It is generally accepted that it takes about ten years of experience to turn a novice into an expert programmer, but there seems to be very few metrics for what constitutes a good programmer. Some of the relevant questions are:

• Is it more important to write efficient code or readable code?

• Is it more important to have a clumsy bug-free code or to have an elegant algorithm?

• Does it matter if students’ programming is not entirely pure, as long as it meets all the functional requirements for the task?

In order to answer these questions, it is first necessary to define terms such as “readable” and “elegant”, which have never had clearly defined operational definitions. With or without these nontrivial definitions, it is still very difficult to give precise answers that can guide teachers through the process of teaching programming languages.

**Programming Paradigms**

In computer science, several programming paradigms can be recognized. Moreover, the four main problem-solving approaches, i.e., programming paradigms, are recognized as fundamental. Each of these approaches involves a distinct way of thinking and each is supported by a range of programming languages. These paradigms are:

* Imperative Paradigm
* Procedural Programming
* Object Oriented Programming
* Declarative Paradigm
* Functional paradigm,
* Logical paradigm.
* Mathematical Programming

**Imperative Programming**

The imperative programming paradigm is based on the Von Neumann architecture of computers, introduced in 1940’s. Von Neumann architecture is the dominant computer hardware architecture which consists of a single sequential CPU separate from memory, and with data piped between CPU and memory. This is reflected in the design of the imperative languages, with

• States — representing memory cells with changing values,

• Sequential orders — reflecting the single sequential CPU, and

• Assignment statements — reflecting piping.

Imperative programs are sequences of directions (or orders) for performing an action. Therefore, imperative programming is characterized by programming with states and commands which modify these states. Imperative programming languages provide a variety of commands in order to structure the code and to manipulate the states. Usually, in imperative programming languages, a sequence of commands can be named and the name can be used to invoke the sequence of commands. Named sequence of commands is called subprogram, procedure or function. When imperative programming is combined with subprograms it is called procedural programming.

Imperative paradigm is supported by languages such as FORTRAN (introduced in 1954), Cobol (1959), Pascal (1970), C (1971), and Ada (1979), . . .

**Object-Oriented Programming**:

The object-oriented programming is a generalization of imperative programming. The conceptual model of this paradigm is developed from simulation of events. The main underlying idea of this model is: the structure of the simulation should reflect the environment that is being simulated. If real world phenomena are simulated, then there should be an object for each entity involved in the phenomena. Object is an entity encapsulating data and related operations. As in the real world, objects interact—so, object-oriented programming uses message passing to capture interactions between objects.

A programming language supporting this concept and using objects is called object-based. Object-oriented programming languages support additional features, with the following most important ones:

• Abstract data type definitions are used to define properties of classes of objects;

• Inheritance is a mechanism that allows definition of one abstract data type by deriving it from an existing abstract data type—the newly defined type inherits the properties of the parent type;

• Inclusion polymorphism allows a variable to refer to an object of a class or an object of any of its derived classes;

• Dynamic binding of function calls supports the use of polymorphic functions; the identity of a function applied to a polymorphic variable is resolved dynamically based on the type of the object referred to by the variable.

Object-oriented paradigm is supported by languages such as Smalltalk (1969), C++ (1983), and Java (1995).

**Functional Programming**:

The functional programming paradigm is based on the theory of mathematical functions, more precisely on the lambda-calculus. It allows the programmer to think about the problem at a higher level of abstraction—it encourages thinking about the nature of the problem rather than about sequential nature of the underlying computing engine. Functional languages are motivated and developed by the following questions: what is the proper unit of program decomposition and how can a language best support program composition from independent components.

A functional programming language usually has three main sets of components:

• data objects — such as a list or an array;

• built-in functions — for manipulating the basic data objects;

• functional forms — also called high-order functions, for building new functions (such as composition and reduction).

Functional programming languages are called applicative since the functions are applied to their arguments, and non-procedural or declarative since the definitions specify what is computed and not how it is computed.

Functional paradigm is supported by languages such as LISP (1958), ML (1973), Scheme (1975), Miranda (1982), and Haskell (1987).

**Logic Programming**

The logic programming paradigm is based on first-order predicate calculus. This programming style emphasizes the declarative description of a problem rather than the decomposition of the problem into an algorithmic implementation. A logic program is a collection of logical declarations describing the problem to be solved. As such, logic programs are close to specifications. The problem description is used by an inference engine to find a solution. More precisely, a logic program consists of:

• axioms — defining facts about objects,

• rules — defining ways for inferencing new facts,

• a goal statement — defining a theorem, potentially provable by given axioms and rules.

Logic programming is characterized by programming with relations and inference. The programmer is responsible for specifying the basic logical relationships and does not specify the manner in which the inference rules are applied. Logic languages are usually more demanding in computational resources than procedural and object-oriented languages. Logic paradigm is supported by languages such as Prolog (1970), and G¨odel (1994). Curry (1997) is a multiparadigm programming language merging elements of functional and logic programming.

**Importance of Study of programming languages:**

**-**Important for students in all disciplines of computer science because they are the primary tools of the central activity of computer science: programming.

- The progress of computer science can be traced in the progress of programming languages, and many issues of computer science can be traced in the progress of programming language issues.

-Programming language remains the central tool for problem solving in computer science.

Some importance of knowing evolution of programming language is highlighted below:

**1) Influence of languages on problem solving:**

The Sapir-Whorf hypothesis (still controversial): The structure of language defines the boundaries of thought. Although no programming language can prevent us from finding certain solution to a problem, a given language can influence the class of solution we are likely to see and the frame of mind with which we approach programming, thus influencing the quality of our programs.

**2) Benefits for all computer scientists**

The study of programing language is important to all who uses them. The reason is that from this study you will learn the motivation for and the use of the most important facilities found in modern programming languages. This will provide you with a basis for evaluating languages, which will aid you in choosing the best language for your application.

**3) Benefits for language designers**

All engineering design is a cumulative process; we learn from the success and failures of the designs of the past.

**4) Benefits for language implementers**

If you are interested in language implementations, you will gain insight into the motivations for various facilities, thus allowing you to make reasonable implementation trade-offs.

**5) Benefits for hardware architects**

By understanding the requirements of programming language implementation, hardware architects will gain insight into the ways machines may better support languages.

**6) Benefits for system designers**

Designers of all sorts of s/w system (e.g. OS & DBMS) will learn principles and techniques applicable to all human interfaces. Many s/w tolls including OS command languages, DBMS, editors, text formatters and debuggers, have many of the characteristics of programming languages, and so the principles will be applicable to much of your future s/w design.

**7) Benefits for software managers**

Finally if you manage s/w development efforts, then you will benefit in several ways from the study of programming languages. You will be better able to make the decisions if you know the costs of designing or extending a language, the costs of implementing a language, and the benefits of various language facilities.

**History and Motivation**

**Why study a primitive language??**

-We may benefit by beginning the study of language design in a context in which the issues are salient. Having honed (sharpen) our skills here, we will better able to apply them to more modern, sophisticated, and complex languages.

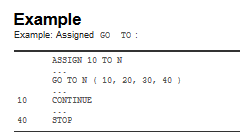
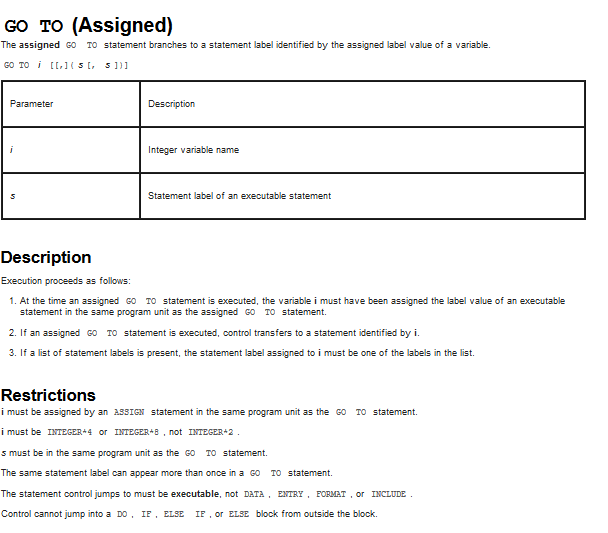
**Programming is Difficult**

Almost as soon as the first computers were built, it become obvious that programming was very difficult; this fact has not changed. Indeed, the tasks we have attempted to accomplish with computers have grown rapidly in ambitiousness and size.

**Programming early computers was especially difficult**

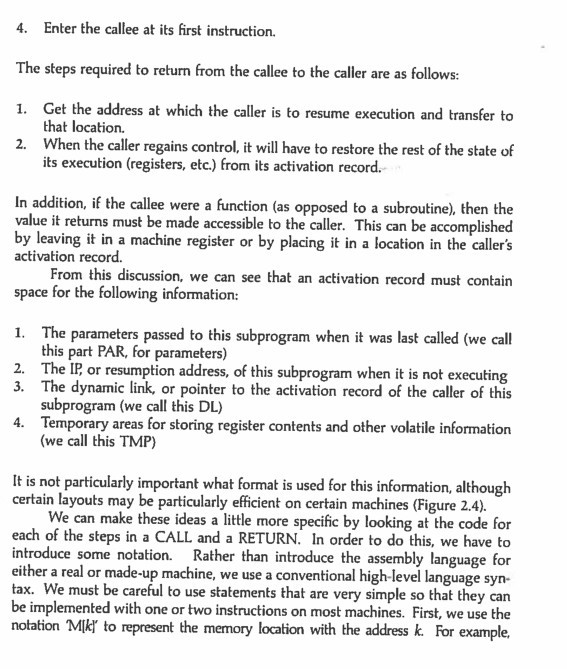
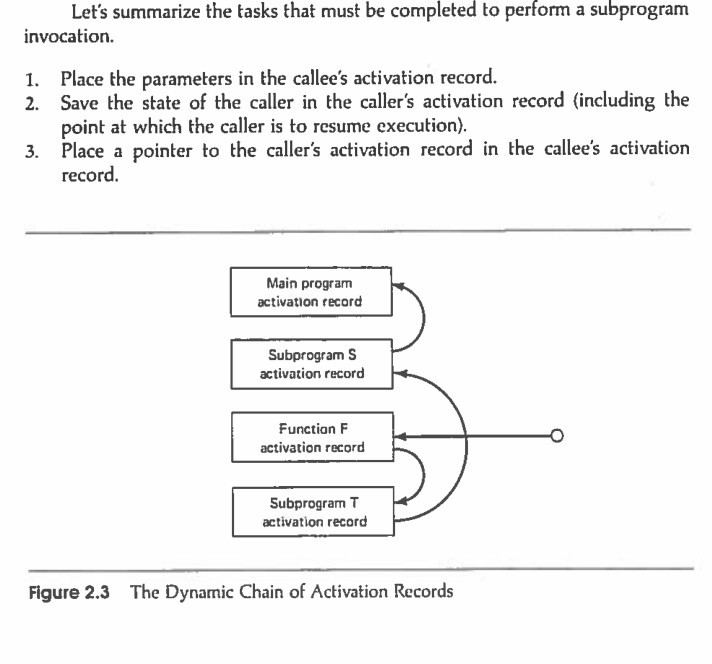
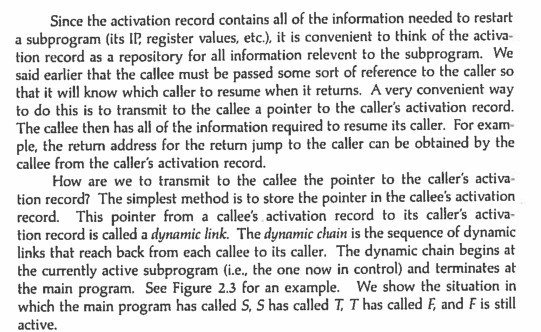
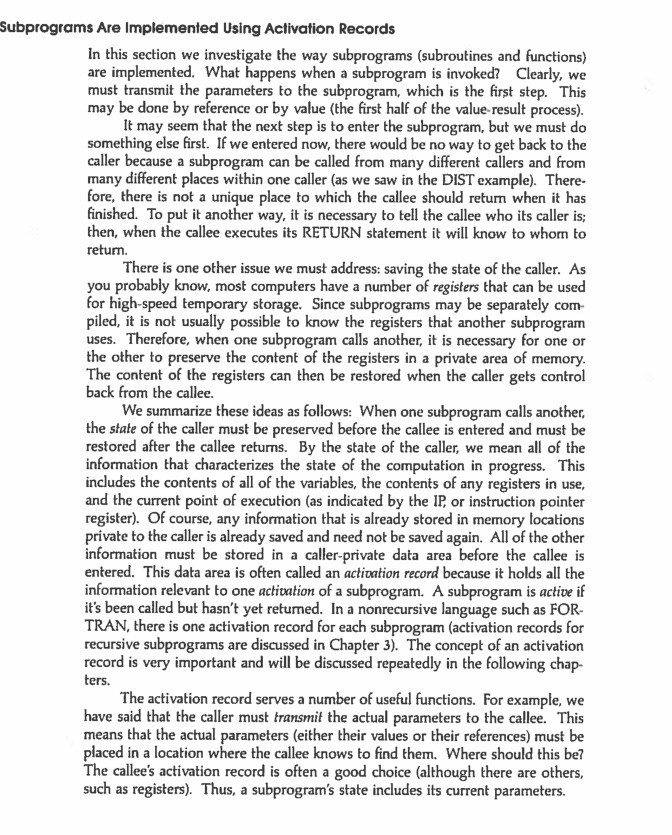
Although the problems addressed on early computers were smaller than many of those now addressed, programming was still very difficult. Part of the reason was that early computers had very little storage; a few thousand words were considered a large memory. Thus, compact code was a necessity. Finally, early computers were more complicated to program than the ones with which we are now familiar.

GOTO Statements in Fortran



# 

# Dynamic Chain of Activation Record



# 

# 

# 

# 

# 

# 

# 

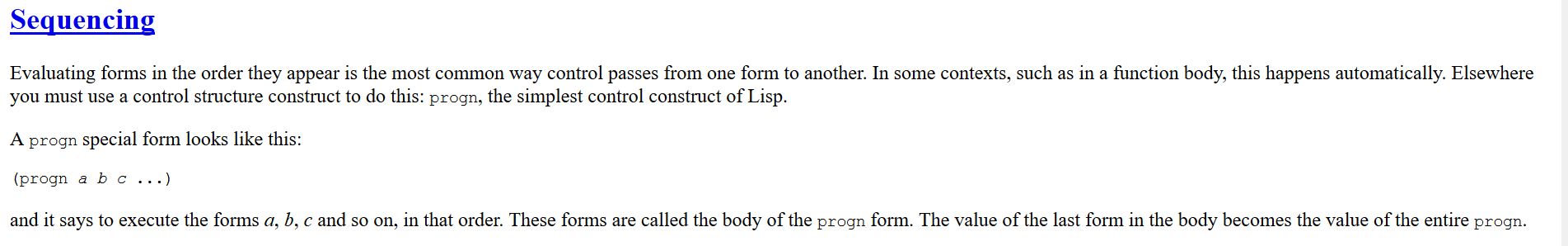
# 

# 

# S-Expression In Common Lisp, a s-exp would be like: s-exp : (op s-exp1 s-exp2 ...) op: a function | a macro | a special... S-Expression A s-exp looks like a linked list but actually a tree. (car (list ‘+ 1 2)) => ‘+ (cdr (list ‘+ 1 2)) => (1 ... S-Expression: Benefits 1. There will be no lexical analysis because all the codes already are the AST. 2. There will be...

LISP Control Structures

Lisp originally had very few control structures, but many more were added during the language's evolution. (Lisp's original conditional operator, cond, is the precursor to later if-then-else structures.)



# 

# 

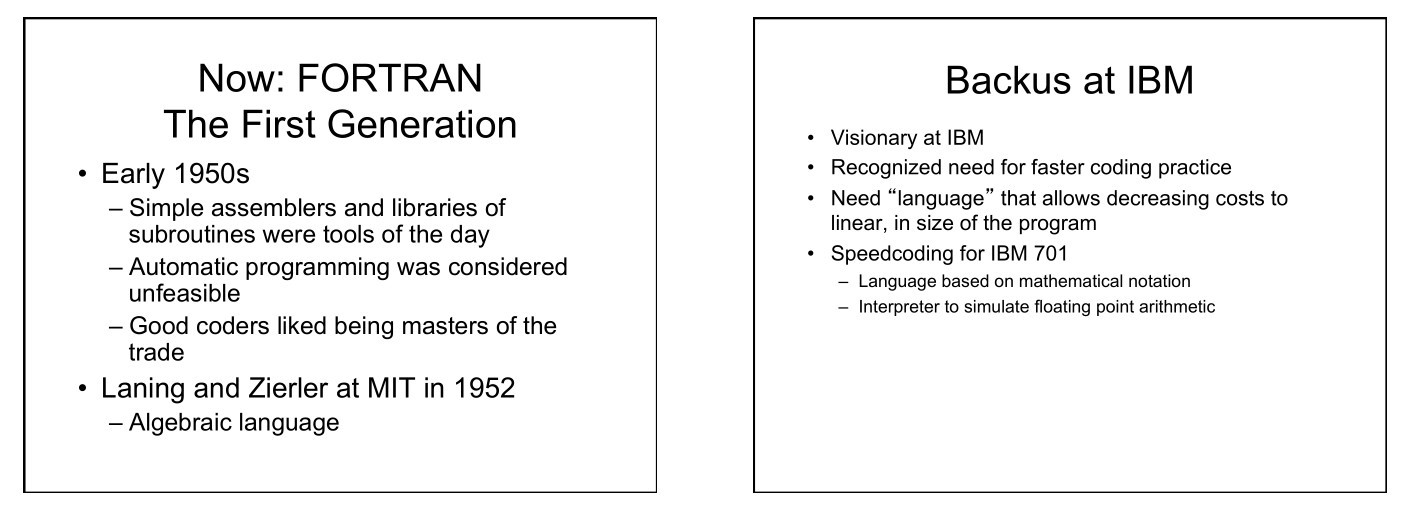
# 

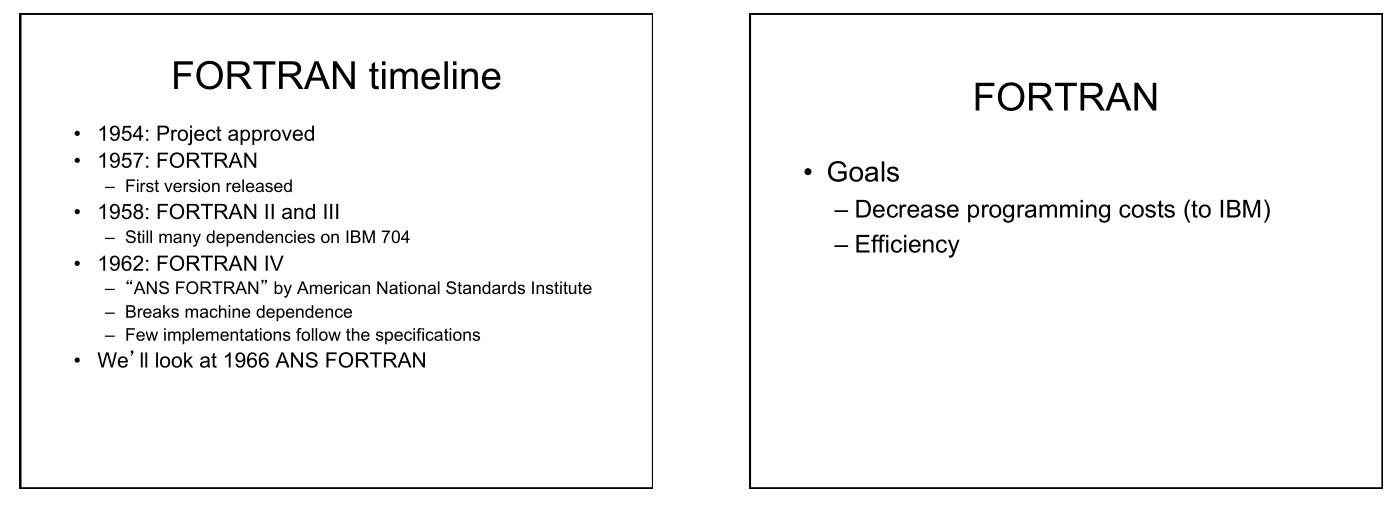
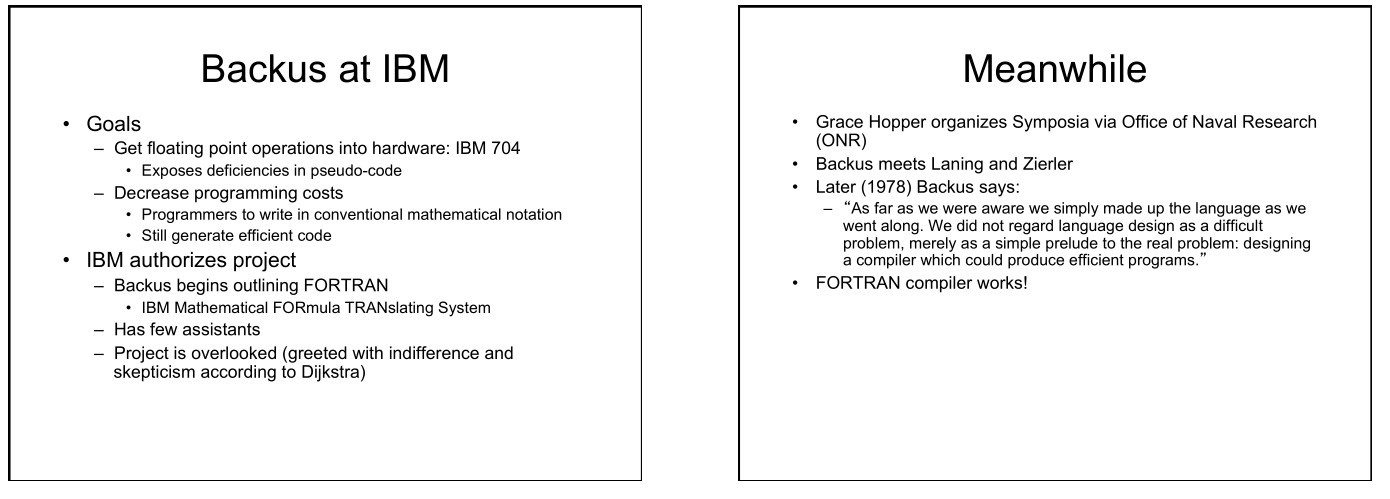
# 

# 

# 

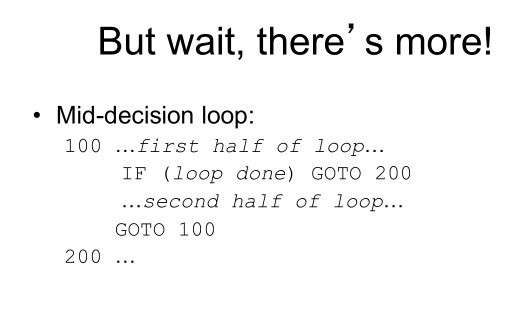
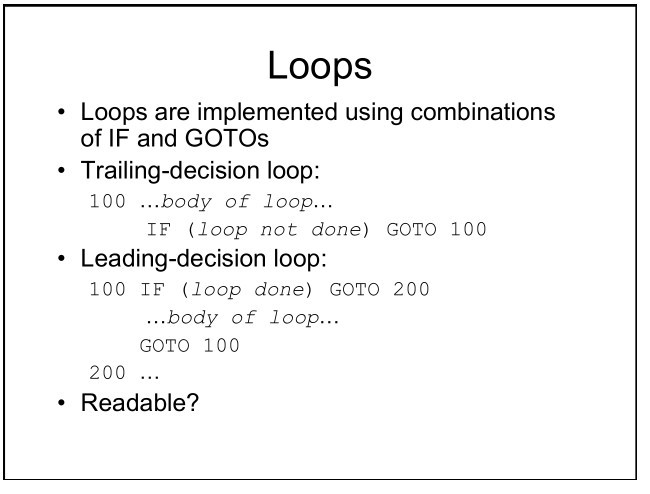
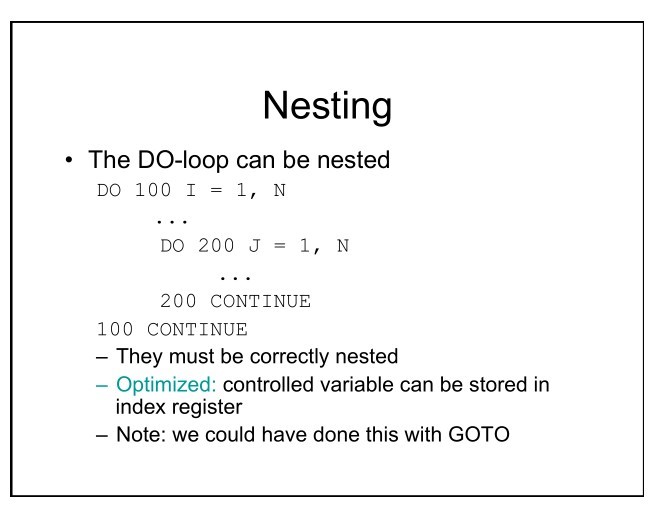
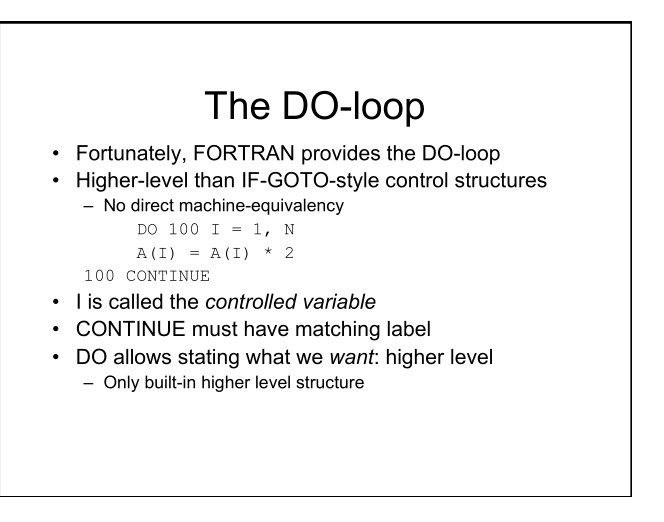
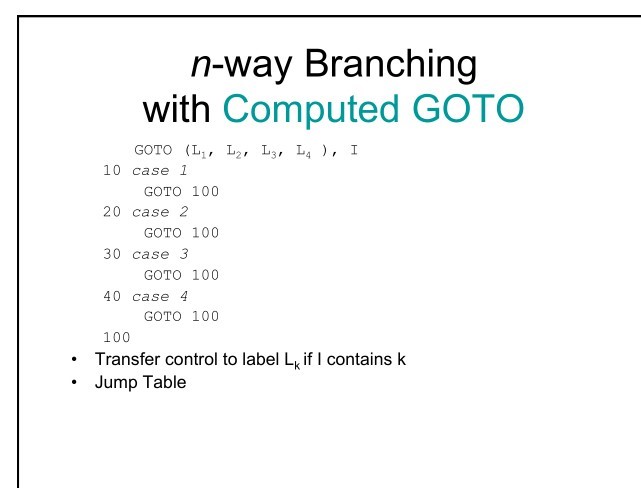
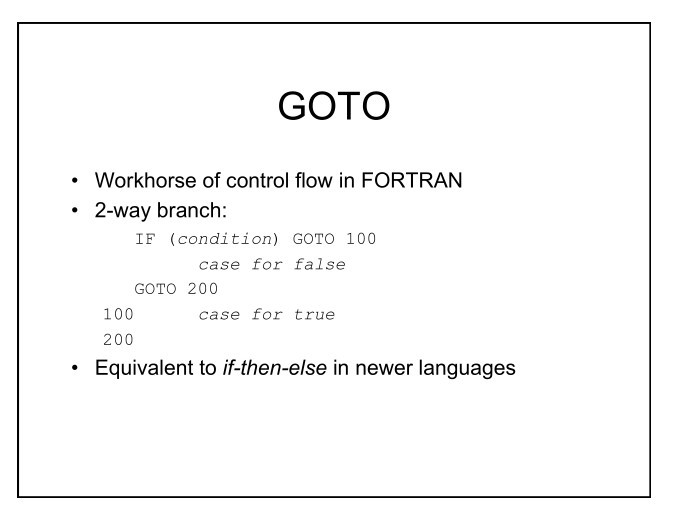
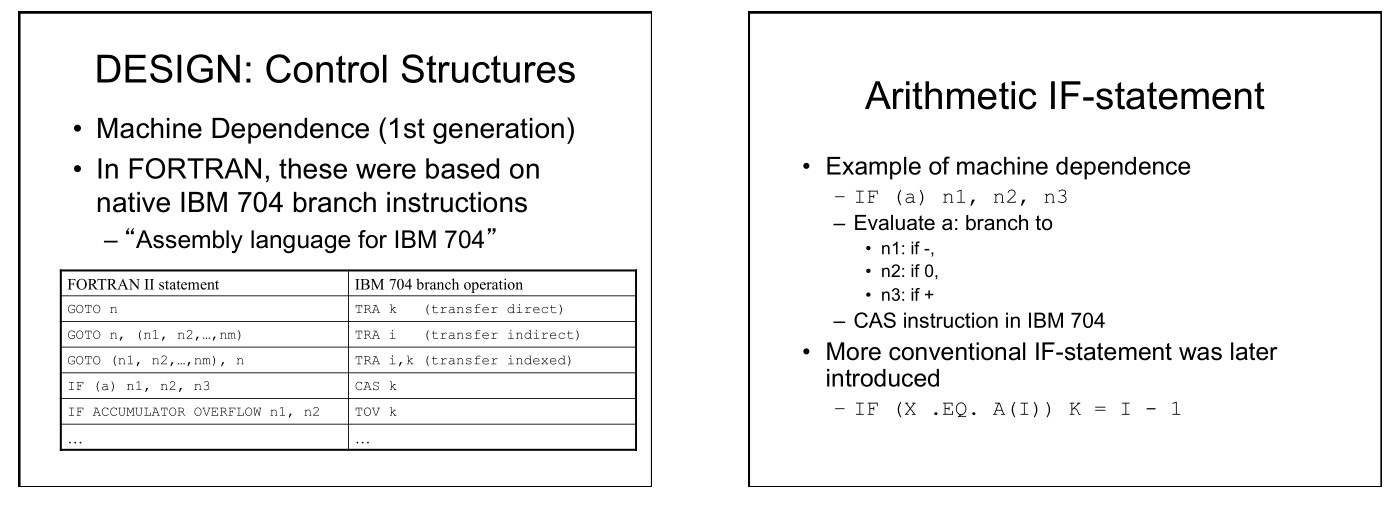
# History of FORTRAN



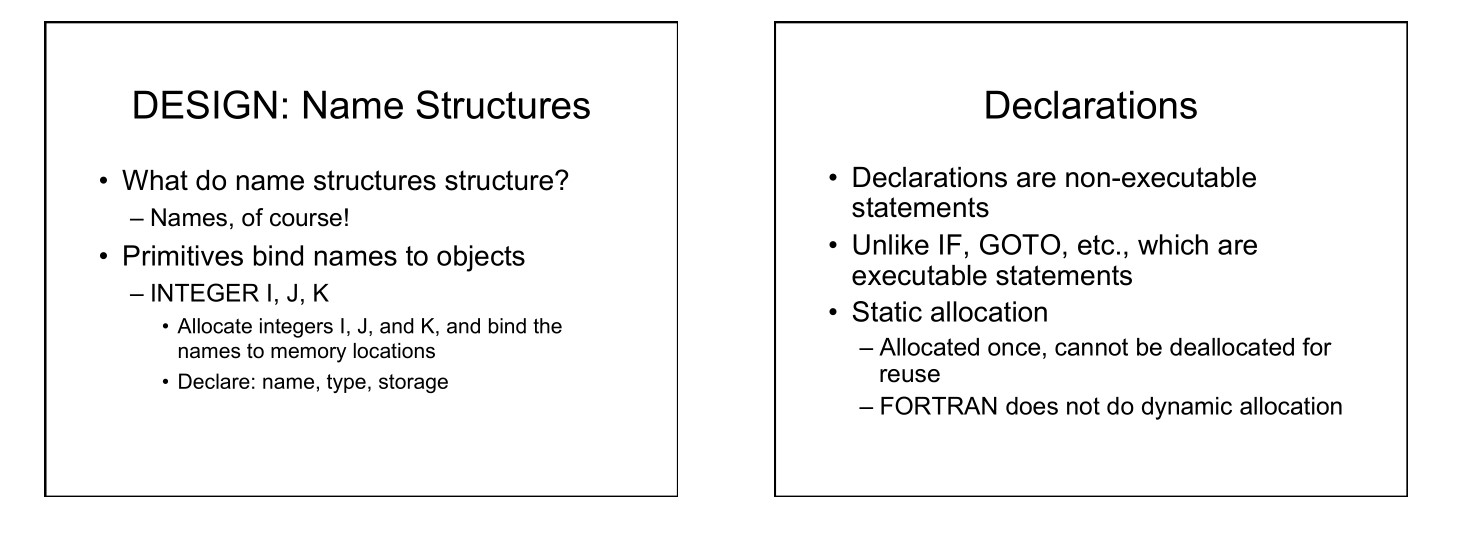


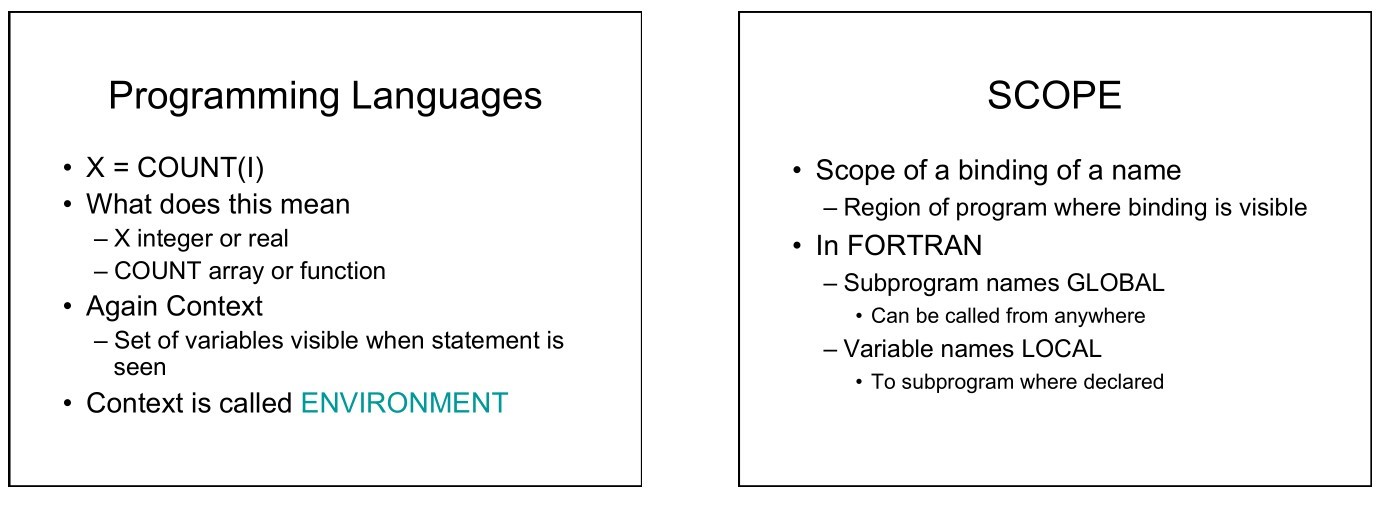
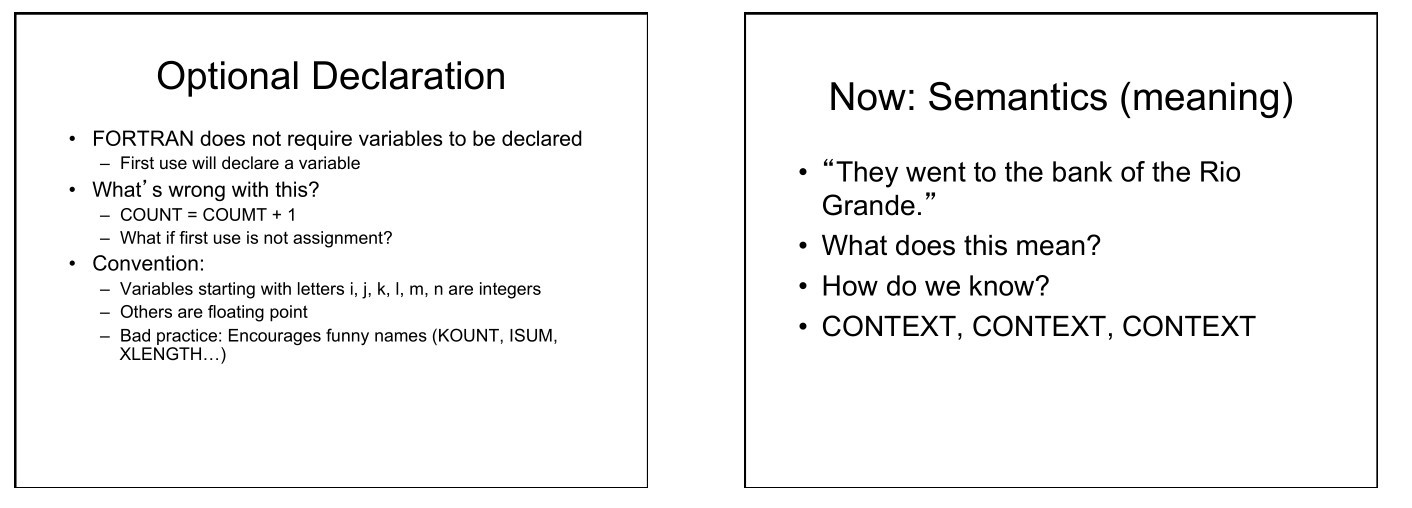
# Control Structure of FORTRAN

A control structure is a block of programming that analyses variables and chooses a direction in which to go based on given parameters. The term *flow control* details the direction the program takes (which way program control "flows"). Hence it is the basic decision-making process in computing; It is a prediction.

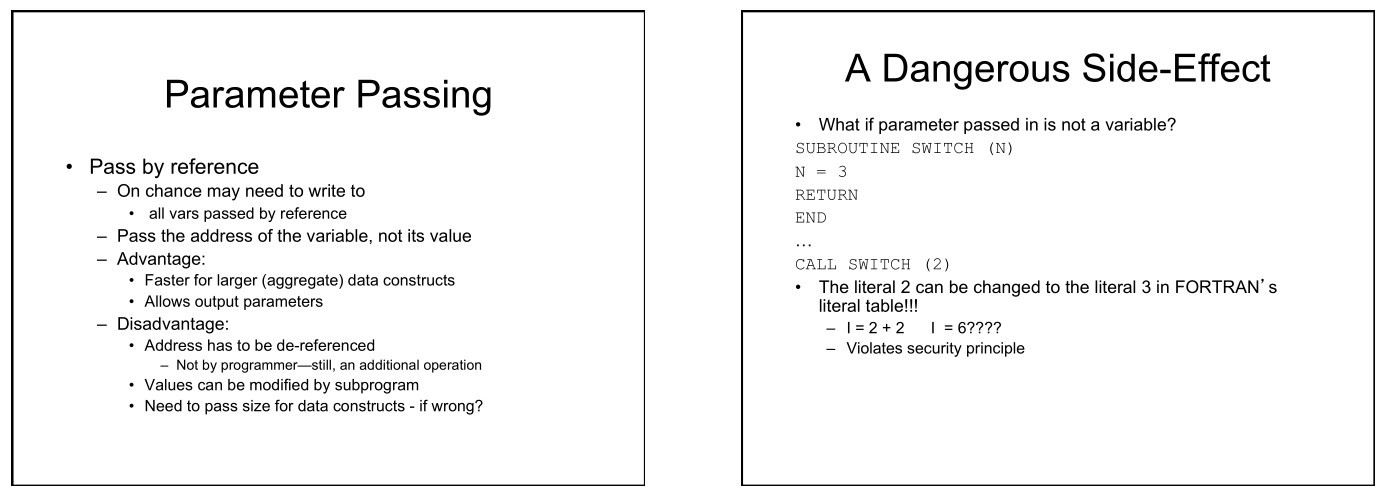
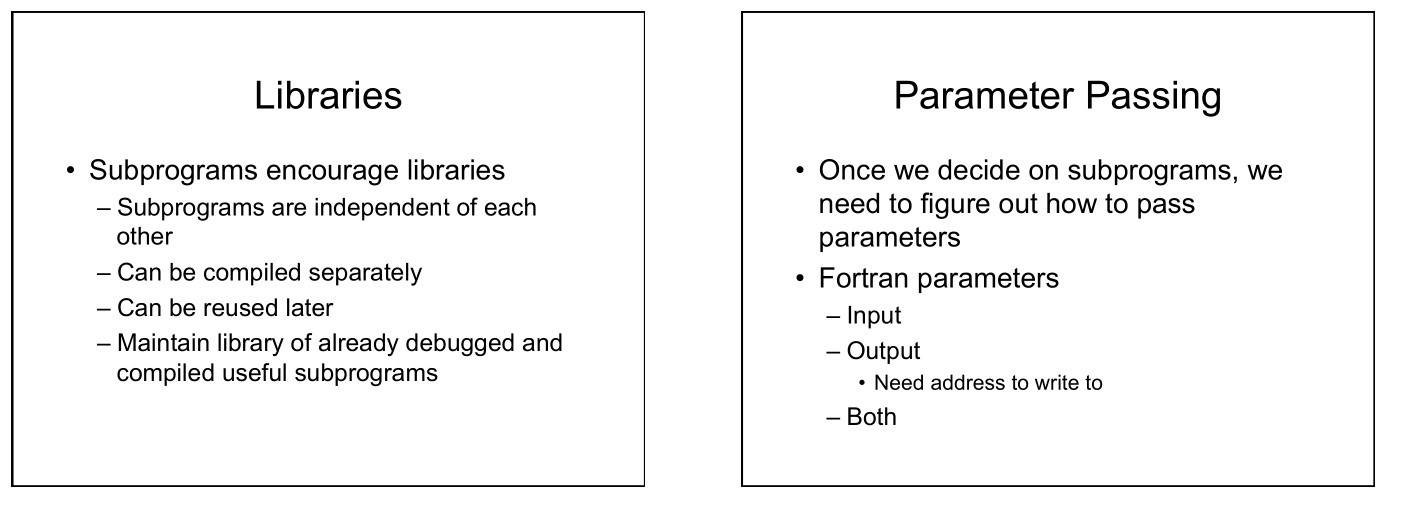


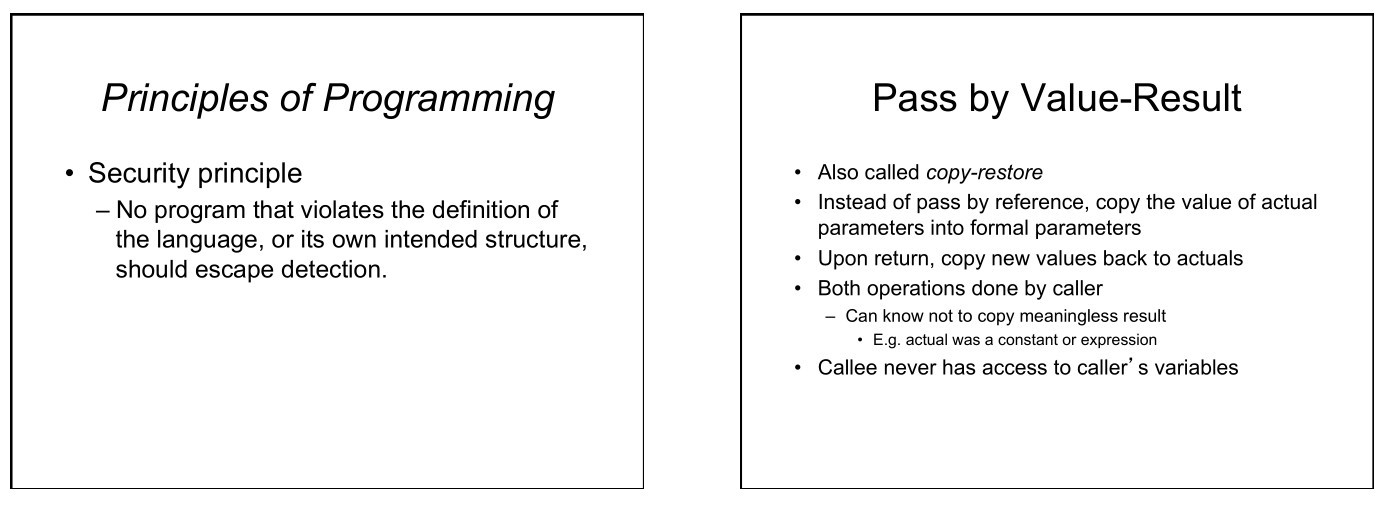
# Name Structures





# Parameter Passing





***FORFTRAN PROGRAM***

## PROGRAM INOUT

C

C This program reads in and prints out a name

C

CHARACTER NAME\*20

PRINT \*,’ Type in your name, up to 20 characters’

PRINT \*,’ enclosed in quotes’

READ \*,NAME

PRINT \*,NAME

END

## PROGRAM AVERAGE

C C THIS PROGRAM READS IN THREE NUMBERS AND SUMS AND AVERAGES THEM.

C

REAL NUMBR1,NUMBR2,NUMBR3,AVRAGE,TOTAL

INTEGER N

N = 3

TOTAL = 0.0

PRINT \*,’TYPE IN THREE NUMBERS’

PRINT \*,’SEPARATED BY SPACES OR COMMAS’

READ \*,NUMBR1,NUMBR2,NUMBR3

TOTAL= NUMBR1+NUMBR2+NUMBR3

AVRAGE=TOTAL/N

PRINT \*,’TOTAL OF NUMBERS IS’,TOTAL

PRINT \*,’AVERAGE OF THE NUMBERS IS’,AVRAGE END