Principles
of
Programming Language

[BE SE-6th Semester]

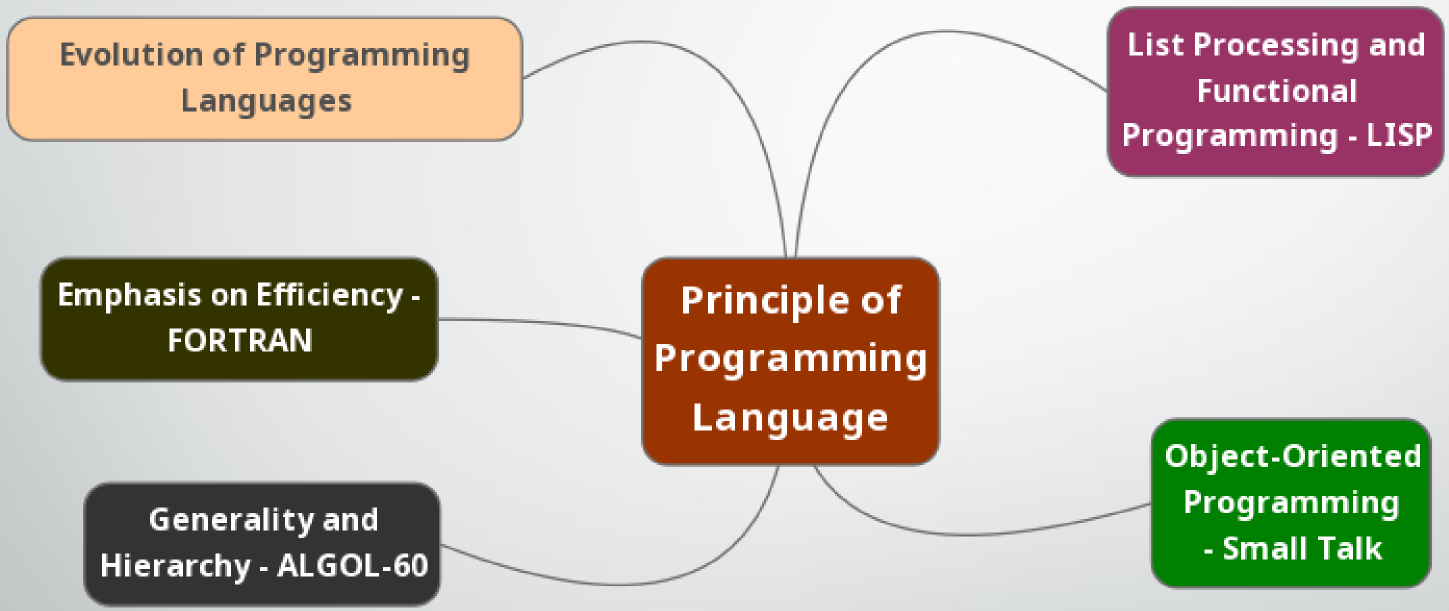
Rishi K. Marseni

Textbook:

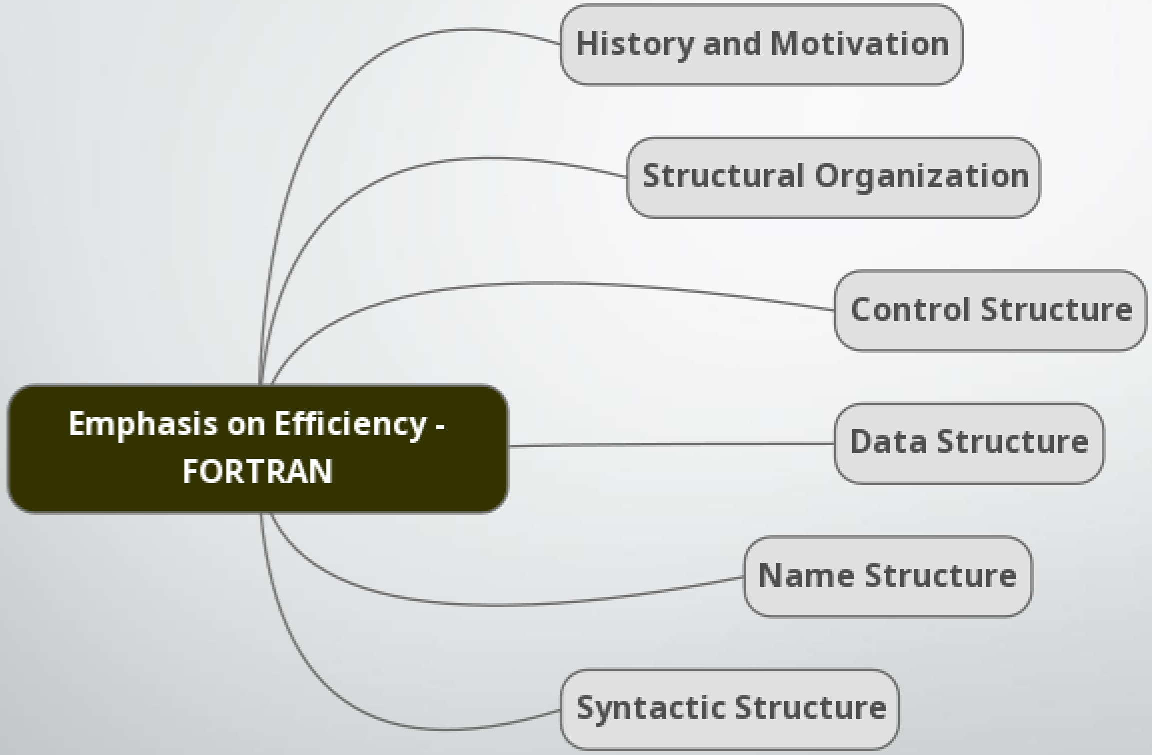
Principles of programming languages: design, evaluation, and implementation.

Author: Bruce J. MacLennan

Principle of Programming Language



2. Emphasis on Efficiency: FORTRAN



2.1. History and Motivation

Highlights of Psuedo-Code

- Virtual computer
  - More regularity
  - Higher level
- Decreased chance of errors
  - Automate tedious and error-prone tasks
- Increased security
  - Error checking
- Simplify debugging
  - trace

FORTRAN: Overview

- Before Fortran, programs were written in assembly language (very tedious to say the least)
- Fortran was the first widely-used high-level computer language
- a general purpose programming language
- mainly intended for engineering & scientific computation
- Interpreted language(MATLAB, Python, Java)
- Compiled language(FORTRAN, C, C++)

History and Motivation(1)

- 1953: John W. Backus (of IBM)
  - Programming cost can be decreased by a system
    - That allowed the programmer to write in the conventional mathematical notation,
    - And generated code's efficiency comparable to that produced by a good programmer
- 1954: a report on
  - The IBM Mathematical ***FOR***mula ***TRAN***slating system.

History and Motivation(2)

- In 1958 FORTRAN is a successful language.
- FORTRAN has been revised several times.
  - FORTRAN I
  - FORTRAN II, 1957
  - FORTRAN III, 1958
  - FORTRAN IV, 1962
  - ANSI FORTRAN , 1966
  - FORTRAN 77, 1977
  - **FORTRAN 90, 1990**
  - FORTRAN 2000
  - Fortran 2003 (OOP)
  - Fortran 2008 (Parallel Processing, Bit)

2.3. Design: Structural Organization

A small FORTRAN I program

```
      DIMENSION DTA(900)
         SUM = 0.0
         READ 10, N
   10 FORMAT(I3)
        DO 20 I = 1, N
        READ 30, DTA(I)
   30 FORMAT(F10.6)
        IF (DTA(I)) 25, 20, 20
   25 DTA(I) =  -DTA(I)
   20 CONTINUE
         DO 40 I=1,N
         SUM = SUM + DTA(I)
   40  CONTINUE
         AVG = SUM/FLOAT(N)
         PRINT 50, AVG
   50  FORMAT(1H, F10.6)
          STOP
```

A small FORTRAN 90 program

```fortran
program add
    implicit none
    integer a,b,s
    print *, ' This program compute the sum of 2 integer numbers'
    print *, ' Enter numbers(separated by a comma/space)'
    read *, a,b
    s = a + b
    print *,  'The sum of ', a,' and ' , b
    print *, ' is ' , s
    stop
end
```

Overall structure of FORTRAN

- A Main program and zero or more subprograms.
- Communicate using
  - Parameters,
  - Shared data areas called COMMON blocks.

Main program

Subprogram 1

⋮

Subprogram n

A FORTRAN 90 code with subprograms

```fortran
module global
implicit none
real a,b,s
end module global
program add2
implicit none
call input
call add
call output
stop
end
```

```fortran
subroutine add
use global
implicit none
s = a + b
return
end
```

```fortran
subroutine input
use global
implicit none
print *, ' This program adds 2 real numbers'
print *, ' Enter numbers with comma/space'
read *, a,b
return
end
```

```fortran
subroutine output
use global
implicit none
print *,  ' The sum of ', a,' and ' , b
print *, ' is ' , s
return
end
```

Constructs

Declarative constructs

– (First part in pseudo-code: data initialization)

– Declare facts about the program, to be used at compile-time

Imperative constructs

– (Second part in pseudo-code: program)

– Commands to be executed during run-time

Declarative Constructs

- Constructs are either declarative or imperative
- Declarations perform three functions:
  - *Allocate* an area of memory of a specified size
  - *Bind* a name to an area of memory
  - *Initialize* the contents of that memory

```
DIMENSION DTA(900)
DATA  DTA, SUM / 900*0.0, 0.0
```

Imperative Constructs

- Imperatives are
  - *Computational* statements(arithmetic, move)
    [AVG = SUM / FLOAT(N)]
  - *Control-flow* statements (comparison, loop)

    [IF-statements, DO loop, GOTO]
  - *Input-output* statements [read, print]

Building a FORTRAN Program

1. Compilation (relocatable object code)

exact addresses of variables and statements have a later *binding time*

- Syntactic analysis
- Optimization
- Code synthesis

2. Linking (libraries, external references)
3. Loading (relocatable to absolute format)
4. Execution

Compilation

Compilation has 3 phases

- **Syntactic analysis**
  - Classify statements, constructs and extract their parts
- **Optimization**
  - FORTRAN has considerable optimizations, since that was the selling point
- **Code synthesis**
  - Put together parts of object code instructions in relocatable format

2.3. Design: Control Structures

Control structures and primitive statements

- Govern the flow of control of the program
- The purpose of control structures are to control various *primitive* computational and input-output instructions.
- Primitive operation: one that is not expressed in terms of more fundamental ideas in the language.
- Common to all imperative languages.

Machine Dependence

- In FORTRAN, control structures were based on IBM 704 branch instruction.
  - The arithmetic IF-statement in FORTRAN II
    - IF(e)n1,n2,n3

      it means If e<0 goto n1, e=0 goto n2, e>0 goto n3
  - The logical IF-statement in FORTRAN IV
    - IF (X .EQ. A(I)) K=I-1

The Portability Principle

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

GOTO as a workhorse of control flow(1)

- Selection statements:
- A two way branch

```
        IF (condition) GOTO 100
        …case for condition false…
        GOTO 200
100   …case for condition true…
200   …
```

GOTO as a workhorse of control flow(2)

- More than two cases (a computed GOTO)

```
      GOTO (10, 20, 30, 40) , I
10      ... handle case 1 ...
      GOTO 100
20      ... handle case 2 ...
      GOTO 100
30      ... handle case 3 ...
      GOTO 100
40      ... handle case 4 ...
100     ...
```

- Much like a case-statement

Reversing TRUE and FALSE

- To get if-then-else –style if:

```
   IF (.NOT. (condition)) GOTO 100
        case for true
   GOTO 200
100   case for false
200
```

- IF and GOTO are selection statements

The concept of iteration(1)

- Loops, by combinations of If-stat and GOTO
  - **Trailing-decision loop (while-do)**

```
100  …body of loop …
     IF (loop not done) GOTO 100
```

  - **Leading-decision loop (repeat-until)**

```
100  IF (loop done) GOTO 200
     …body of loop …
     GOTO 100
200  …
```

The concept of iteration(2)

- We can also use GOTO and IF statements to make the following loop:
  - *Mid-decision* loop

```
100   ...first half of loop...
       IF (loop done) GOTO 200
      ...second half of loop...
      GOTO 100
200  ...
```

And also more complicated control statements.

The concept of iteration(3)

- GOTO : a primitive and powerful control statement.
- It is possible to implement almost any control structure with it,
  - Those that are good,
  - Those that are bad.
- What makes a control structure ***good***?
  - Mainly it is *understand-ability*

The Structure Principle

- The static structure of a program should correspond in a simple way to the dynamic structure of the corresponding computations.

E. W. Dijkstra (1968)

  - To visualize the behavior of the program easily from its written form.

Syntactic Consistency Principle

- Things that look similar should be similar and things that look different should be different.
  - For example
    - *Computed* GOTO,
      - GOTO(L_1,L_2,…,L_n), I

      Transfers to L_k if I contains k
    - *Assigned* GOTO
      - GOTO N,(L_1,L_2,…,L_n)

      go to a statement which its address is in N.

Computed and Assigned GOTO

- We just saw the Computed GOTO:

  GOTO (L 1 , L 2 , ..., L n), I

  – Jumps to label 1, 2, ...
- Now consider the Assigned GOTO:

  GOTO N, (L 1 , L 2 , ..., L n)

  – Jumps to ADDRESS in N

  – List of labels not necessary

  – Must be used with ASSIGN-statement
- ASSIGN 20 TO N

  – Put address of statement 20 into N

  – Not the same as N = 20 !!!!

Review: GOTO

- Very powerful
- Can be used for good or for evil
- But seriously is GOTO good or bad?

– Good: very flexible, can implement elaborate control structures

– Bad: hard to know what is intended

– Violates the structure principle

Problem with GOTO

- ASSIGN 20 TO N
- GOTO (20, 30, 40, 50), N

- N has address of stmt 20, say it is 347
- Look for 347 in jump table - out of range
- Not checked
- Fetch value at 347 and use as destination for jump
- Problem– Computed should have been Assigned

Problem with GOTO

- A problem caused by confusing GOTOs:

I = 3

...

GOTO I, (20, 30, 40, 50)

Control will transfer to address 3!

Probably in area used by system, i.e. not a stmt

– Assigned should have been computed

Defense in Depth Principle

- If an error gets through one line of defense (syntactic checking, in following case), then it should be caught by the next line of defense (type checking, in following case)

FORTRAN's ***weak type checking***:

  - Using integer variables to hold a number of things besides integers, such as the addresses of statements.
  - If we have a ***label type***, then confusing two kinds of GOTOs would lead to an easy-to-find compile-time error, and not a run-time one.

Interaction of features

- interaction between:
  - Syntax of GOTOs and
  - Using integer variables to hold addresses of statements.
- It is one of the hardest problems in language design.

Do-Loop versus GOTO

Do-Loop is more structured than GOTO

```
      Do 100 I=1, N
      A(I) = A(I)*2
100   CONTINUE
```

is *higher-level*, says

what they *want* (execute the body N times)

not *How* to do (initialize I, inc I, test it…)

Do-Loop Nesting

The DO-loop can be nested

```
DO 100 I = 1, N
...
DO 200 J = 1, N
...
200 CONTINUE
100 CONTINUE
```

– They must be correctly nested

– Optimized: controlled variable can be stored in index register

– Note: we could have done this with GOTO

Do-Loop illustrates

- The Impossible Error Principle
  - *Making errors impossible to commit is preferable to detecting them after their commission.*

- The Automation Principle
  - *Automate mechanical, tedious, or error-prone activities.*

The Do-Loop is highly optimized

- We can put the loop index in an index register

- Because the controlled variable and its initial and final values are all stated *EXPLICITLY* along with the extent of the loop.

- Higher-level programming language constructs are easier to optimize.

Subprograms were added in FORTRAN II

- The Abstraction Principle
  - Avoid something to be stated more than once; factor out the recurring pattern.

- Subprograms define procedural abstraction.
- Subprograms allow large programs to be modularized.
- Subprograms encourage libraries.
- Parameters are passed by **reference.**

Subprograms encourage libraries

- Subprograms are independent of each other
- Can be compiled separately
- Can be reused later
- Maintain library of already debugged and compiled useful subprograms

Parameter Passing

- Once we decide on subprograms, we need to figure out how to pass parameters
- Fortran parameters

  Input

  Output

  Need address to write to

  Both

Pass By Reference

- On chance may need to write to - all vars passed by reference
- Pass the address of the variable, not its value
- Advantage:

  Faster for larger (aggregate) data constructs

  Allows output parameters
- Disadvantage:

  Address has to be de-referenced

  Not by programmer—still, an additional operation

  Values can be modified by subprogram

  Need to pass size for data constructs - if wrong?

Pass By Reference

- Always efficient.
- It has dangerous consequences.

```
SUBROUTINE SWITCH (N)
N=3
RETURN
END

CALL SWITCH(2)
```

- The compiler has a *literal table*.
- Then ***I=2+2*** caused ***I*** to be ***6 !!!***
- ***(the security principle: escape detection!)***

Pass by Value-result

- Instead of pass by reference, copy the value of actual parameters into formal parameters
- Upon return, copy new values back to actuals
- Both operations done by caller
- Can know not to copy meaningless result
- E.g. actual was a constant or expression
- Callee never has access to caller's variables

Pass by Value-result

- Another way of implementing FORTRAN's parameter passing, (also called *copy-restore*)
- At subprogram entry:
  - Value of actual par. → formal par.
- At subprogram exit:
  - Result (final value of formal par.) → actual par.

- Both are done by the *caller.*
- It preserves the security of implementation (when the actual is a constant or expression)

Activation Records(1)

What happens when a subprogram is called?

- Transmit parameters
- Save caller’s status
- Enter the subprogram
- Restore caller’s state
- Return to caller

Activation Records(2)

Before subprogram invocation:

- Place parameters into callee’s activation record
- Save caller’s status
- Save content of registers
- Save instruction pointer (IP)
- Save pointer to caller’s activation record in callee’s activation record
- Enter the subprogram

Activation Records(3)

Returning from subprogram:

- Restore instruction pointer to caller’s
- Return to caller
- Caller needs to restore its state (registers)
- If subprogram is a function, return value must be made accessible

Activation Records(4)

Contents of Activation Record

- Parameters passed to subprogram
- P (resumption address)
- Dynamic link (address of caller's activation record)
- Temporary areas for storing registers

2.4. Design: Data Structures

Design: Data Structures

- Primitive data: numbers
- Arrays

Primitives

Primitives are scalars only

- Integers
- Floating point numbers
- Double-precision floating point
- Complex numbers
- No text (string) processing

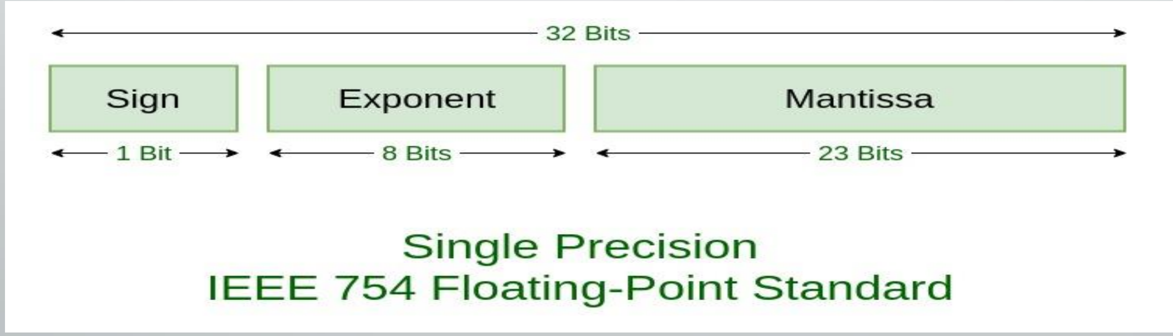
Data Structures were suggested by mathematics

- Primitive data: numbers
  - Integer, float, complex, logical(Boolean), double-precision
  - The numeric operations in FORTRAN are Representation independent: they depend on the logical or abstract properties of the data values and not the details of their representation on a particular machine.

Data Structures representation

- Word-oriented { Most commonly 32 bits }
- Integer– Represented on 31 bits + 1 sign bit
- Floating point

– Using scientific notation: characteristic +mantissa



The arithmetic operators are overloaded

- Representations of integer, real and complex variables are different:
  - Overloading the meaning of operations onto each arithmetic operation is necessary.
  - The meaning of '+' depends on its context.
- FORTRAN allowed mixed-mode expression:
  - Expressions of more than one mode or type.
  - Type conversion is necessary.
  - Coercion: an implicit, context-dependent type conversion.

The arithmetic operations

- **2 + 3.1 = ?**

– 2 is integer, 3.1 is floating point

- How do we handle this situation?

– Explicit type-casting: FLOAT(2) + 3.1

- Type-casting is also called “*coercion*”

Automatic type coercion

- Always coerce to encompassing set
- Integer + Float => float addition
- Float * Double => double multiplication
- Integer – Complex => complex subtraction
- Types *dominate* their subsets

The Data constructor is the Array

- Data structure: *array*.
  - Static, limited to 3 dimensions
  - Column-major

  DIMENSION DTA, COORD(10,10)

- Constructor: linguistic methods used to build complex data structures from primitives.

Array Representation

Element	Address
A(1,1)	A
A(2,1)	A + 1
…	
A(m,1)	A + m - 1
A(1,2)	A + m
…	
A(m,2)	A + 2m - 1
…	
A(m,n)	A + nm - 1

- Data structure: *array.*
- Column-major order
- Most languages do row-major order
- Addressing equation:
  - $\alpha\{A(2)\} = \alpha\{A(1)\} + 1 = \alpha\{A(1)\} – 1 + 2$
  - $\alpha\{A(i)\} = \alpha\{A(1)\} – 1 + i$
  - $\alpha\{A(i,j)\} = \alpha\{A(1,1)\} + (j – 1)m + i – 1$

FORTRAN arrays allow many optimizations

- Using index register in Do-loops, working on array elements.

Optimizations

- Arrays are mostly associated with loops
- Most programmers initialize controlled variable to 1, and reference array A(i)
- Optimization:
- Initialize controlled variable to address of array element
- Therefore, we'll increment address itself
- Dereference controlled variable to get array element

Subscripts

- Arrays are mostly associated with loops
- Subscripts can be expressions
- A(i+m*c)
- This defeats above optimization
- Therefore, subscripts are limited to c and c' are integers, v is an integer variable
- [c, v, v+c, v-c, c*v, c*v+c', c*v-c']
- A(J - 1) ok; A(1+J) not ok

2.5. Design: Name Structures

Data, Control and Name Structures

- The purpose of a *data structure* is to organize the primitive data in order to simplify its manipulation by the program.

- The purpose of a *control structure* is to organize the control flow of the program.

- The purpose of *name structures* are to organize the names that appear in the program.

Design: Name Structures

What do name structures structure?

- Names, of course!

Primitives bind names to objects

- INTEGER I, J, K
- Allocate integers I, J, and K, and bind the names to memory locations
- Declare: name, type, storage

Declarations

- Declarations are non-executable.
  - Type → the amount of storage
  - Static allocation vs. Dynamic allocation
- Variable names are local in scope.
  - Information hiding principle
  - Programs will be much more maintainable.
- Subprogram names are global in scope.

Declarations and Allocations

Static allocation

- Allocated once, cannot be deallocated for reuse
- FORTRAN does not do dynamic allocation

FORTRAN does not require variables to be declared

- First use will declare a variable [k = 5]

But the optional declaration is dangerous

- What's wrong with this?
- COUNT = COUNT + 1

-> What if first use is not assignment?

Naming Convention

Static allocation

- Variables starting with letters i, j, k, l, m, n are integers
- Others are floating point
- Bad practice: Encourages funny names (KOUNT, ISUM, XLENGTH…)

The Semantics (meaning)

"They went to the bank of the Rio Grande."

- What does this mean?
- How do we know?
- CONTEXT, CONTEXT, CONTEXT

The Semantics (meaning)

X = COUNT(I)

- What does this mean
- X integer or real
- COUNT array or function

The Context

- Set of variables visible when statement is seen
- Context is called ENVIRONMENT

Structuring with Subprograms

- We need to find a way to share data: Parameters

- Pass by reference
- Pass by value-result
  - Caller copies value of actual to formal variable
  - On return, caller copies result value to actual
    - » Omit for constants or expressions as actuals

Scope of the name

- The scope of a binding of a name is defined as that region of the program over which that binding is visible.
- The scope rules of FORTRAN permit a subprogram to access data from only two sources:
  - variables declared within the subprogram
  - variables passed as parameters.

Scope of the name

- It is hard to share data with just parameters!
  - Cumbersome, and hard to maintain
  - Produces long list of parameters
  - If data structure changes, there are many changes to be made
  - Violates information hiding
- Consider a program work with a data structure like symbol table, which shall be shared among multiple subprograms.

Information hiding principle

- Modules should be designed so that
  - The user has all the information needed to use the module correctly , and nothing more.
  - The implementer has all the information needed to implement the module correctly , and nothing more.

D. L. Parnas

Sharing Data

FORTRAN's solution:

• COMMON blocks allow more flexibility

– Allows sharing data between subprograms

– Scope rules necessitate this

• Consider a symbol table

```
SUBROUTINE ARRAY2 (N, L, C, D1, D2)
COMMON /SYMTAB/ NAMES(100), LOC(100), TYPE(100)
...
SUBROUTINE VAR (N, L, C)

COMMON /SYMTAB/ NAMES(100), LOC(100), TYPE(100)
```

Scope

- Scope of a binding of a name
  – Region of program where binding is visible
- In FORTRAN
  – Subprogram names GLOBAL
  Can be called from anywhere
- Variable names LOCAL

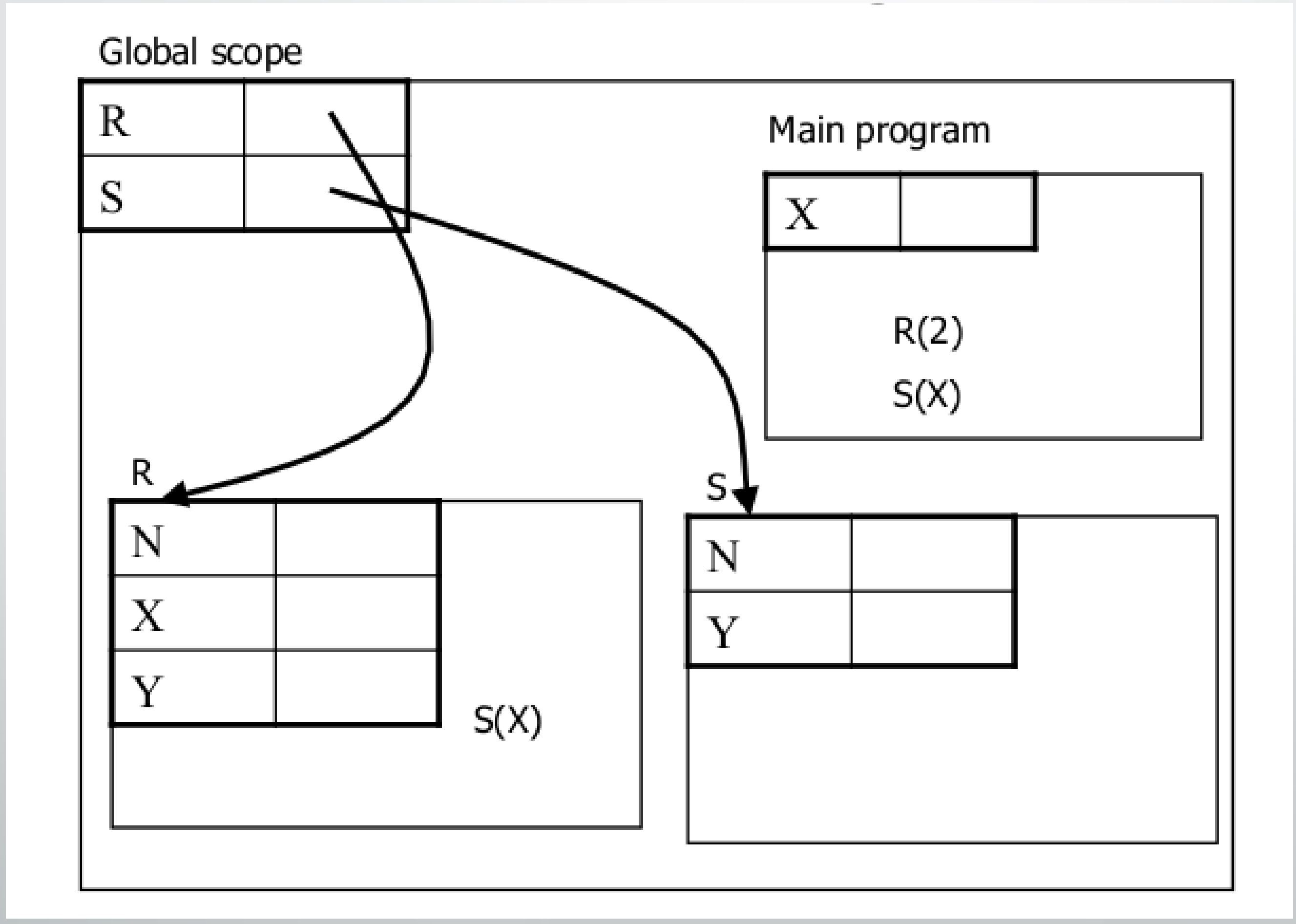
  To subprogram where declared

Common blocks vs. Equivalance

- COMMON blocks allow sharing between subprograms.
- COMMON permits aliasing, which is dangerous.
  - Aliasing: the ability to have more than one name for the same memory location.

- EQUIVALENCE allows sharing within subprograms.
  - Computer memories were extremely small
  - Better use of storage
  - Suffers from all of the problems of aliasing
  - Is no more useful

COMMON Block



COMMON Block

```
SUBROUTINE  A(...)
COMMON / SYMTAB / NAMES(100),LOC(100)
.
.
.
END

SUBROUTINE  B(...)
COMMON / SYMTAB / NAMES(100),LOC(100)
.
.
.
END
```

COMMON Problem

- Tedious to write
- Unreadable
- Virtually impossible to change AND
- COMMON permits aliasing, which is dangerous
- If COMMON specifications don't agree, misuse is possible

Aliasing

- Tedious to write
- The ability to have more than one name for the same memory location
- Very flexible!

COMMON /B/ M, A(100)

COMMON /B/ X, K, C(50), D(50)

EQUIVALENCE

- Since dynamic memory allocation is not supported, and memory is scarce, FORTRAN has EQUIVALENCE
- Allows a way to explicitly alias two arrays to the same memory

EQUIVALENCE

```
DIMENSION      INDATA(10000), RESULT(8000)

EQUIVALENCE  (INDATA(1), RESULT(1))
```

2.6. Design: Syntactic Structures

Syntactic Structures

- Languages are defined by lexics and syntax

Lexics: Way to combine characters to form words or symbols

- E.g. Identifier must begin with a letter, followed by no more than 5 letters or digits

Syntax

- Way to combine symbols into meaningful instructions

Syntactic analysis:

- Lexical analyzer (scanner) Syntactic analyzer (parser)

Fixed Format Lexics

Still using punch-cards!

- Particular columns had particular meanings
- Statements (columns 7-72) were free format

Columns	Purpose
1-5	Statement number
6	Continuation
7-72	Statement
73-90	Sequence number

Languages :: lexics & syntax(1)

- The *syntax* of a language is the way that words and symbols are combined to form the statements and expressions. (syntactic analyzer: parser)

- The *lexics* of a language is the way which characters (I.e., letters, digits, and other signs) are combined to form words and symbols. (lexical analyzer: scanner)

Languages :: lexics & syntax(2)

- A fixed format lexics was inherited from the pseudo-codes.
- Ignoring blanks everywhere is a mistake.
  - Do 20 I = 1, 100
  - DO20I = 1.100
- The lack of reserved words is a mistake.
  - IF(I-1) = 1 2 3
  - IF(I-1) 1, 2, 3

Languages :: lexics & syntax(3)

- Algebraic notation was an important contribution.
- Arithmetic operations have precedence.

- A linear syntactic organization is used.
  - The only nesting in
    - Arithmetic expressions
    - DO-loop

Algebraic Notation

- One of the main goals was to facilitate scientific computing
- Algebraic notation had to look like math
(-B + SQRT(B**2 – 4*AA*C))/(2*A)
- Very good, compared to our pseudo-code
- Problem: How do you parse and execute such a statement?

Operators Precedence

- $b^2 - 4ac == (b^2) - (4ac)$

- $ab^2 == a(b^2)$

- Precedence rules

1. Exponentiation
2. Multiplication and division
3. Addition and subtraction

Operations on the same level are associated to the left (read left to right)

- How about unary operators (-)?

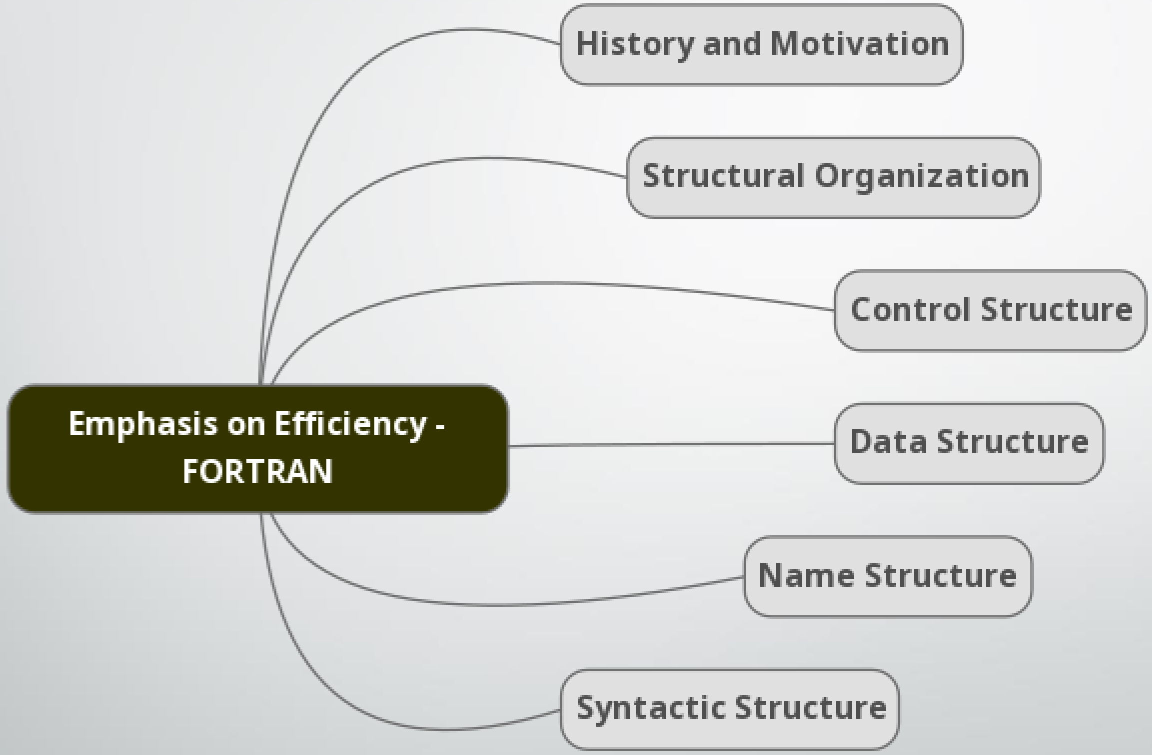
Evaluation and Epilog

- FORTRAN evolved into PL/1
- FORTRAN continues to evolve
- FORTRAN has been very successful.

Characteristics of first-generation programming languages

- Machine dependent
  - Instructions: control structures
  - Data structures
- Linear structure
- no recursive procedures,
- one parameter passing mode,
- weak type system.

2. Emphasis on Efficiency: FORTRAN



Principle of Programming Language