Principles
of
Programming Language

[BE SE-6th Semester]

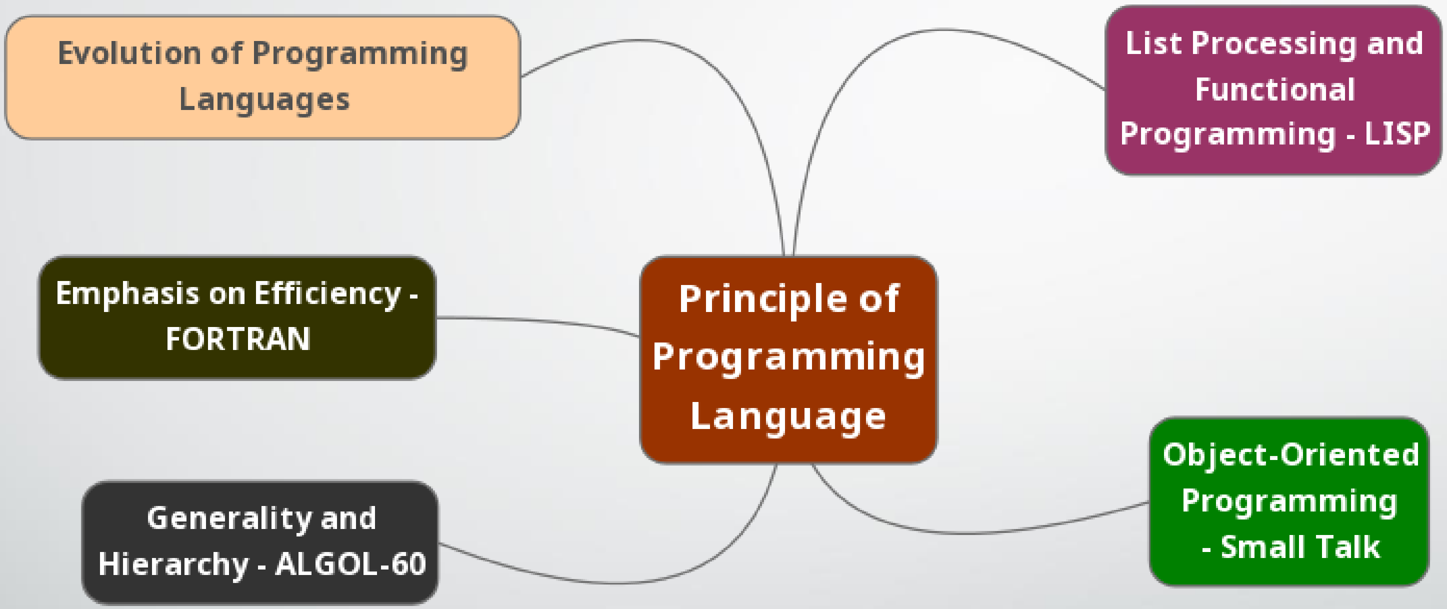
Rishi K. Marseni

Textbook:

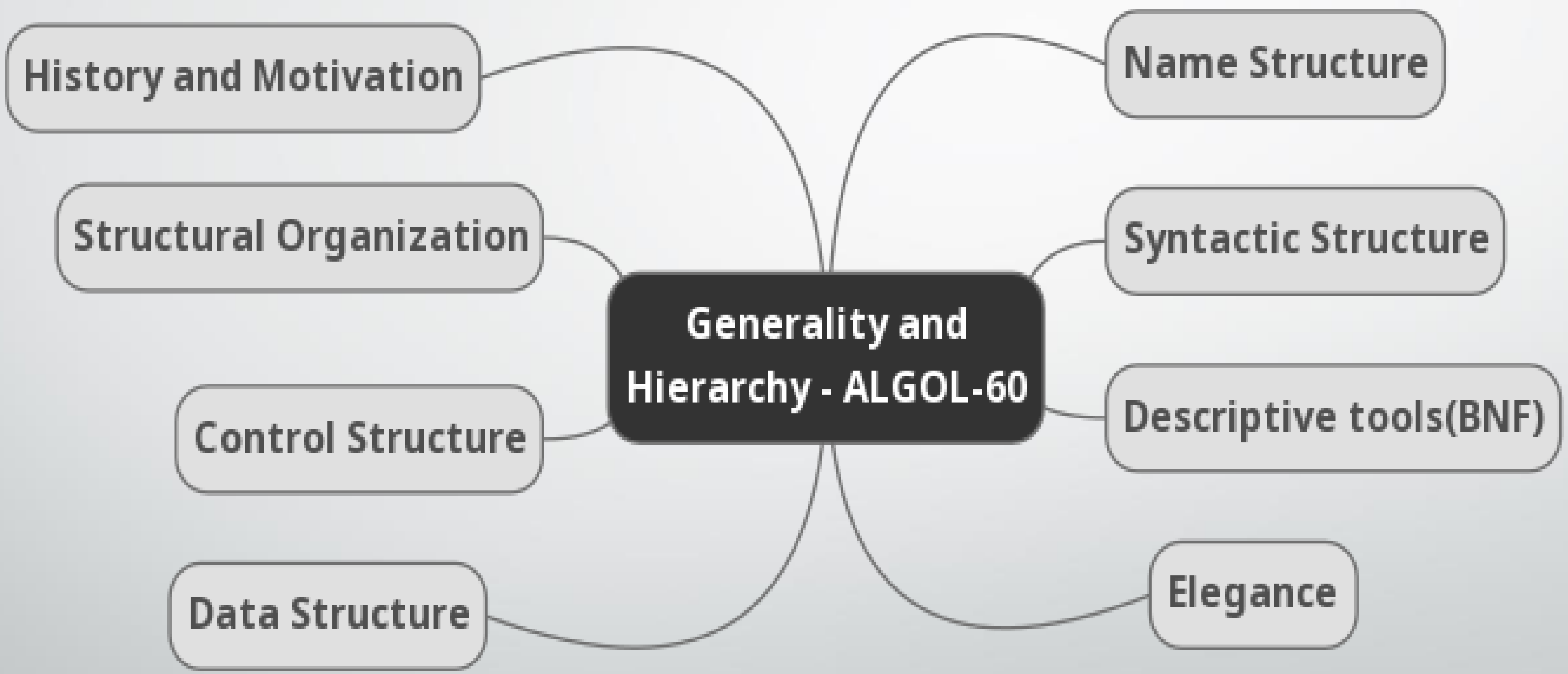
Principles of programming languages: design, evaluation, and implementation.

Author: Bruce J. MacLennan

Principle of Programming Language



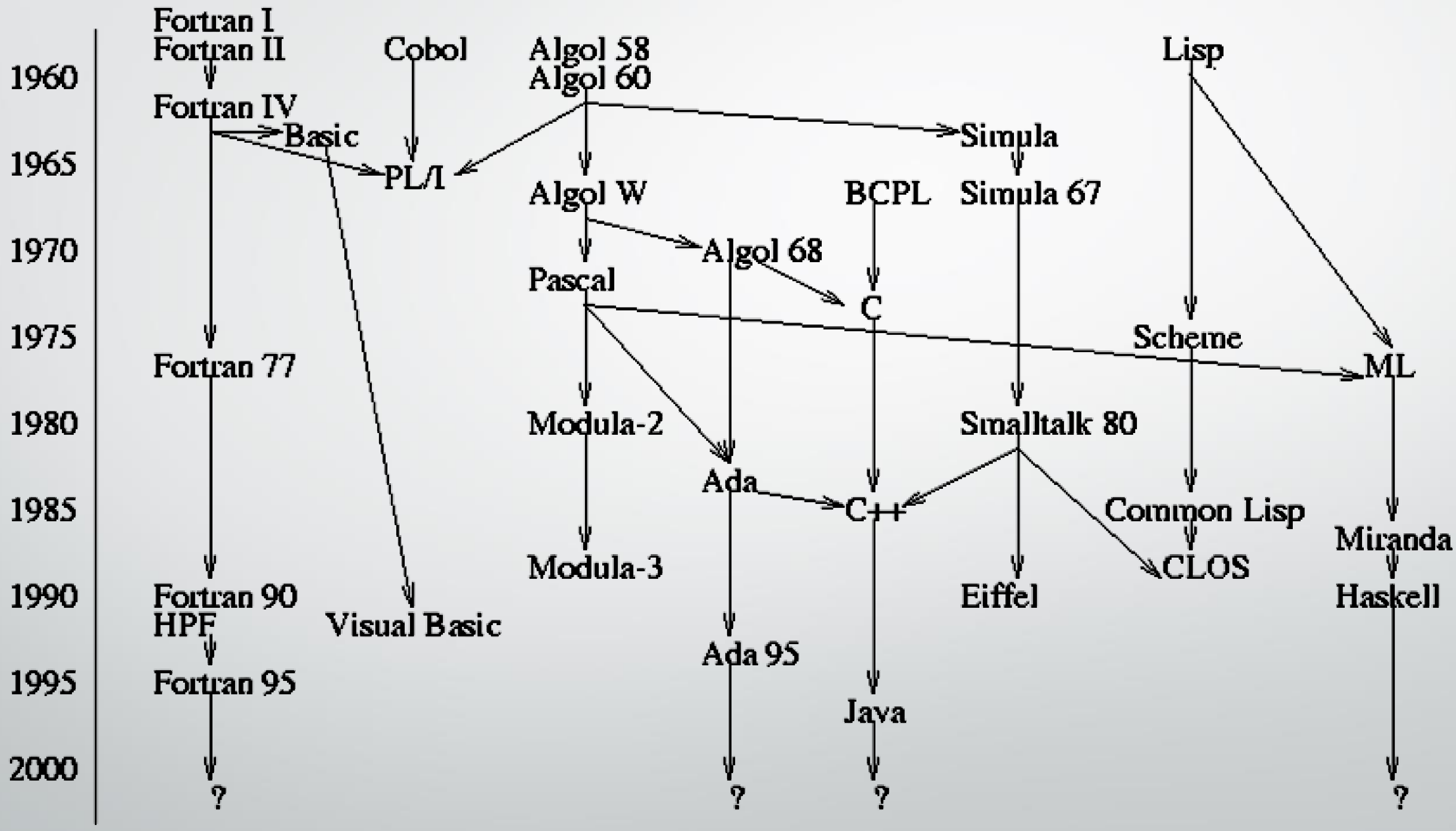
Chapter 3:
Generality and Hierarchy: ALGOL-60



3.1. History and Motivation

- An international language was needed.
- Algol-58 and Algol-60 was designed.
- A formal grammar was used for syntactic description.
- The report is a paradigm of brevity and clarity.

Programming Language Genealogy



History

- Wanted a universal, machine independent language
  - Proposed in 1957
- ALGOL-58
  - Original Name: IAL – International Algebraic Language
  - First version, designed in Zurich (in 8 days)
  - Instant hit, but no standardization

ALGOL-60

- Algol-60 Report in May 1960
- Very different from Algol-58
- Errors correct in Revised Report

Design:
Structural Organization

3.2. Structural Organization

- Algol programs are hierarchically structured.
- Constructs are either declarative or imperative.
  - Declarations: variables, procedures, switches.
    - Variables are integer, real, Boolean.
    - Procedures are typed (functions) and untyped.
    - Switches serves as computed GOTO.

Hierarchical Structure(1)

```
for i := 1 step 1 until N do
          sum := sum + Data[i]
          begin
                    integer N;
                    read int (N);
                    begin
                              real array Data[1:N];
                              integer i;
                              sum :=  0;
                              for i := 1 step 1 until N do
                                        begin
                                        end
                              …
                    end
          end
```

Hierarchical Structure (2)

- Also allowed:

```
If N > 0 then
   for i := 1 step 1 until N do
       sum := sum + Data(i)
```

Hierarchical Structure(3)

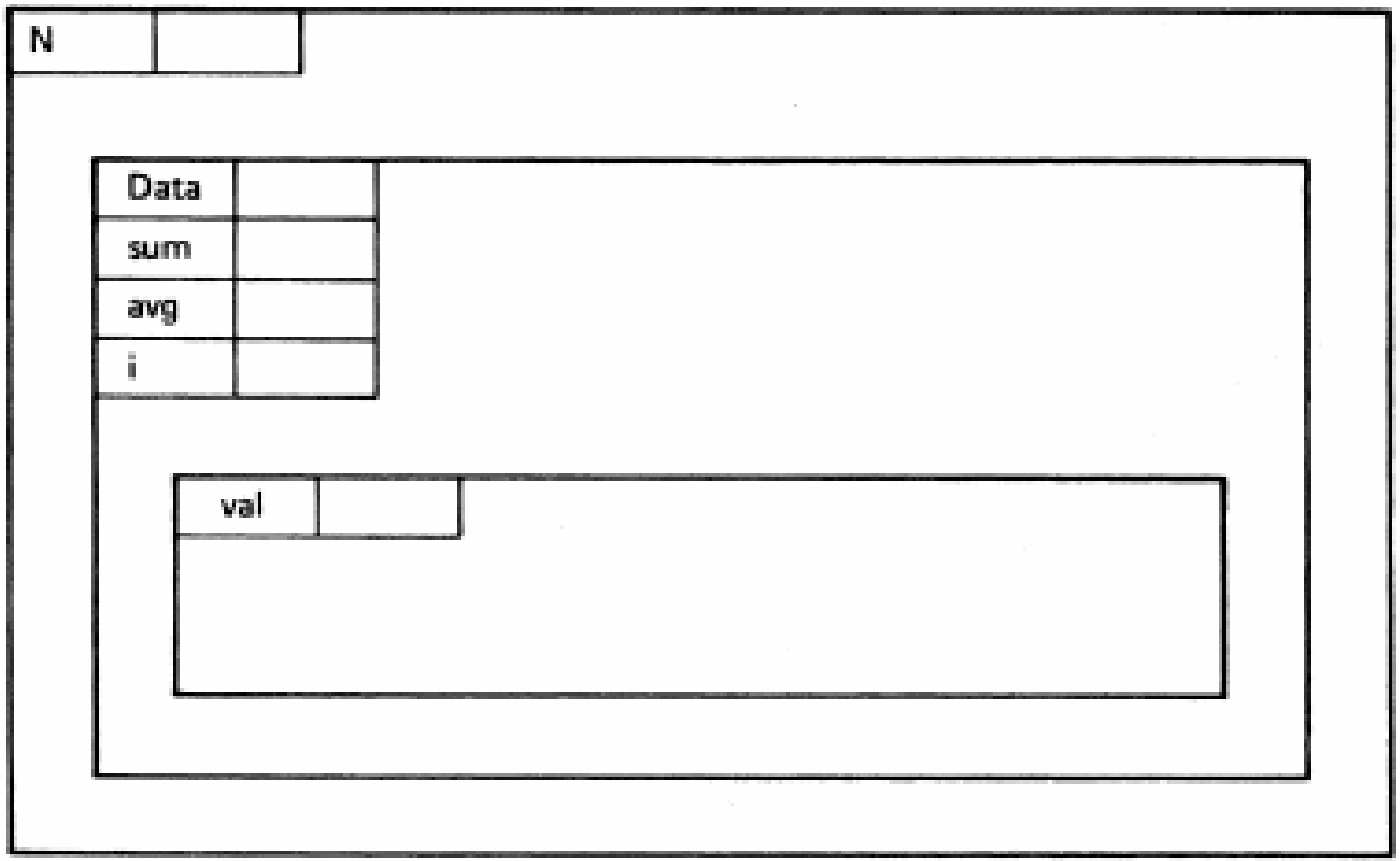


Figure 3.2 Contour Diagram of an Algol Program

Constructs

Like FORTRAN

- Declarative
- Imperative

Declarations

- Variables
  - Integer
  - Real
  - Boolean
  - Arrays can be static or dynamic. Lower bound need not be 1.
- Procedures
  - Typed (really a function)
  - Untyped
- Switch

Imperatives

- Imperatives are computational and control-flow: (no input-output inst.)
  - Computational
    - assignment (:=)
  - Control-Flow
    - Goto
    - If-Then-Else
    - For loop

Compile-Time, Run-Time

- Algol data structures have a later *binding time* than FORTRAN data structures.
  - Various data areas are allocated and de-allocated at run-time by program.
    - E.g. dynamic arrays, recursive procedures
    - The name is bound to its memory location at run-time rather than compile-time.
    - As in FORTRAN, it is bound to its type at compile-time.
- The stack is the central run-time data structure.

Design:

Name Structures

3.3. Name Structures

- The primitives name structures are the declarations that define names by binding them to objects.
- The constructor is the block.
- A group of statements can be used anywhere that one statement is expected: *regularity.*

Blocks

- Blocks define nested scopes.

begin declarations; statements; **end**

- Blocks simplify constructing large programs.

- Shared data structures in Algol are defined once, so there is no possibility of inconsistency. *(impossible error principle)*

Remind: Impossible Error Principle

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

Scope

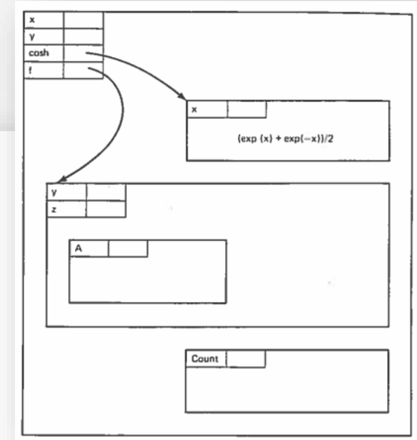
- FORTRAN
  - Global scope (subprogram names)
  - Local scope (variables, COMMON)
- ALGOL
  - Scopes can be nested.
  - Any enclosing scope can be accessed
  - Can cause serious confusion!

Nested Scopes

```
real x,y;

        begin
                real y;
        end

    begin
            real z;
            x := 3;
            y := 4
    end
```

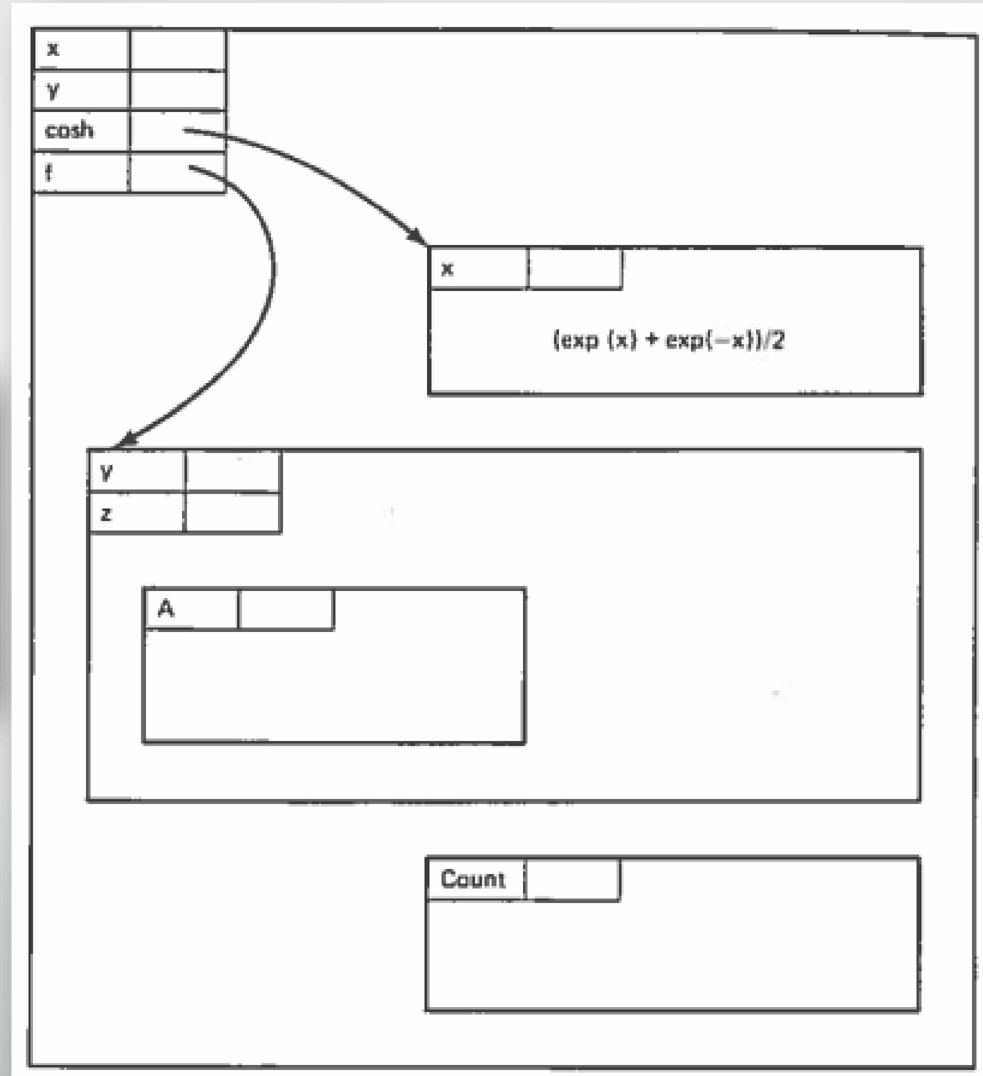
Nested Scopes(2)

```
begin
  real x, y;
  real procedure cosh(x); real x;
    cosh := (exp(x) + exp(−x))/2;

  procedure f(y,z);
    integer y, z;
    begin real array A[1:y];
        ⋮
    end
  ⋮
  begin integer array Count [0:99];
      ⋮
  end
  ⋮
end
```



Nested Scopes(2)

```
begin
   real x, y;
   real procedure cosh(x); real x;
      cosh := (exp(x) + exp(-x))/2;

   procedure f(y,z);
      integer y, z;
      begin real array A[1:y];
         ⋮
      end
   ⋮
   begin integer array Count [0:99];
      ⋮
   end
   ⋮
end
```

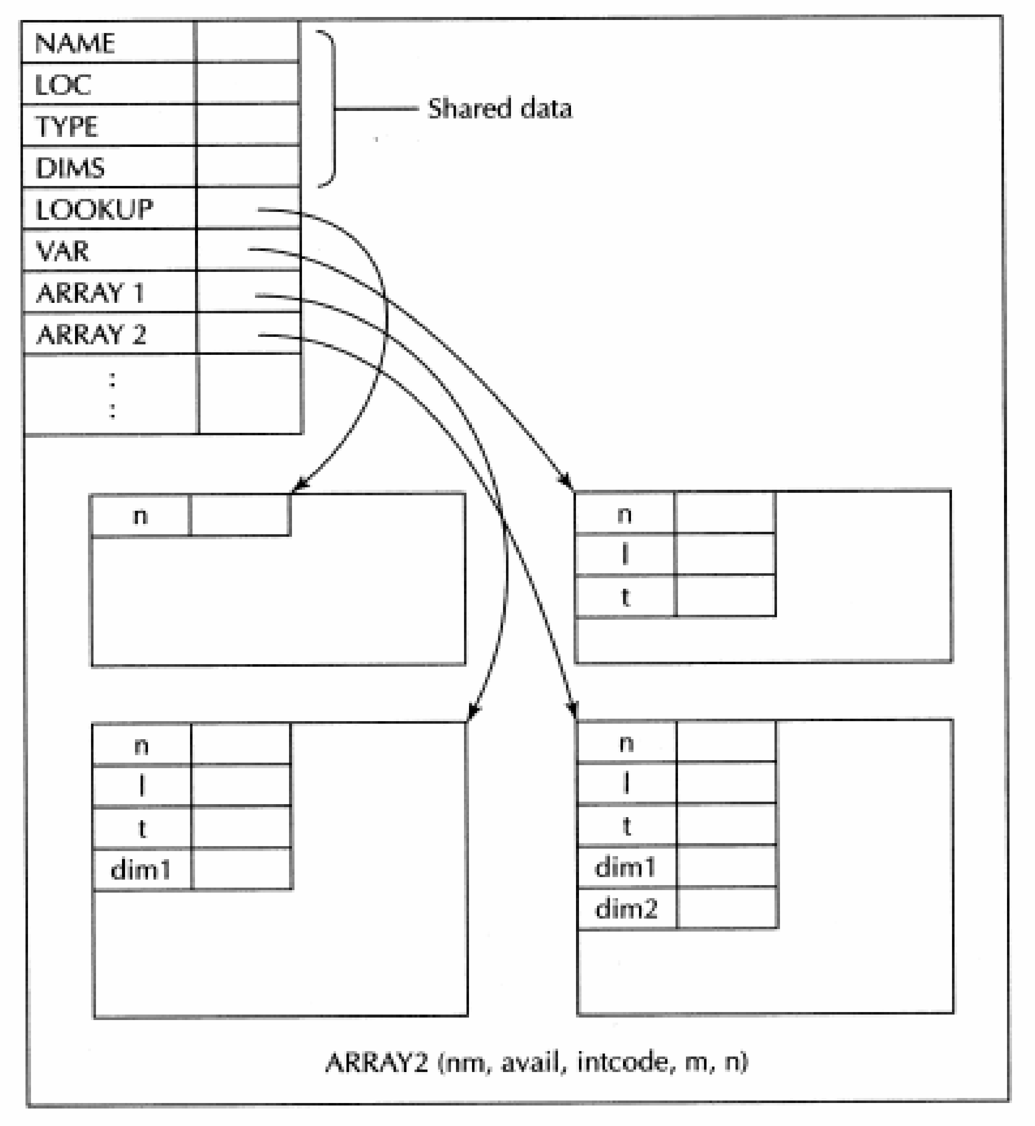


Shared Data and Block Structure

```
begin
   integer array Name, Loc, Type, Dims [1:100];

   procedure Lookup (n);
      ... Lookup procedure ...

   procedure Var (n, l, t);
      ... Enter variable procedure ...

   procedure Array1 (n, l, t, dim1);
      ... Enter 1-dimensional Array procedure ...

   ... other symbol table procedures ...

   ... uses of the symbol table procedures, e.g.,
   Array2 (nm, avail, intcode, m, n);

   ...
end
```

Shared Data and Block Structure

```
begin
  integer array Name, Loc, Type, Dims [1:100];
  procedure Lookup (n);
    ... Lookup procedure ...
  procedure Var (n, l, t);
    ... Enter variable procedure ...
  procedure Array1 (n, l, t, dim1);
    ... Enter 1-dimensional Array procedure ...
  ... other symbol table procedures ...
  ... uses of the symbol table procedures, e.g.,
  Array2 (nm, avail, intcode, m, n);
  ...
end
```



Blocks

- Simplified construction
  - Encourages abstraction of shared structures
  - Permits shared data structures between blocks
  - No need to repeat declarations, as in FORTRAN COMMON blocks
- Allowed indiscriminate access (violates the Information Hiding Principle)

Blocks (2)

- Storage is managed on a stack
- Blocks are delimited by BEGIN…END
- Entry to a block pushes local variables on stack
- Exit from a block pops them from the stack
- Blocks that are not nested are disjoint
- Blocks may not overlap
- Obviates the need for a FORTRAN-like EQUVALENCE statement

Blocks (3)

- Blocks permit efficient storage management on stack.
  - Instead of using EQUIVALENCE in FORTRAN, we can have blocks.
- Responsible Design Principle:
  - Do not ask users what they want; find out what they need.

Static and Dynamic scoping

- In *dynamic scoping* the meanings of statements and expressions are determined by the *dynamic structure* of the computations evolving in time.

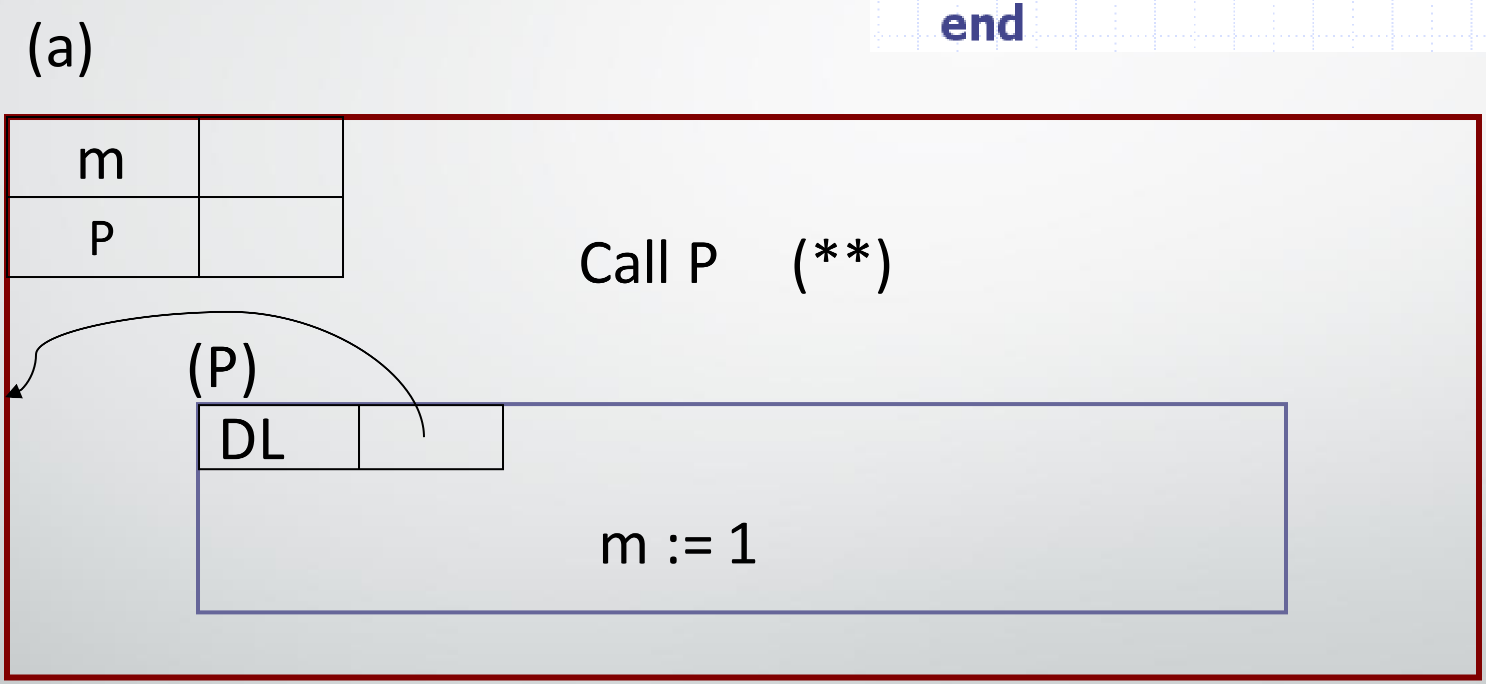
- In *static scoping* the meanings of statements and expressions are determined by the *static structure* of the computations evolving in time.

An Example

```
a: begin integer m;
        procedure p;
                m := 1;
        b: begin integer m;
                P                        (*)
           end;
        P                                (**)
  end
```

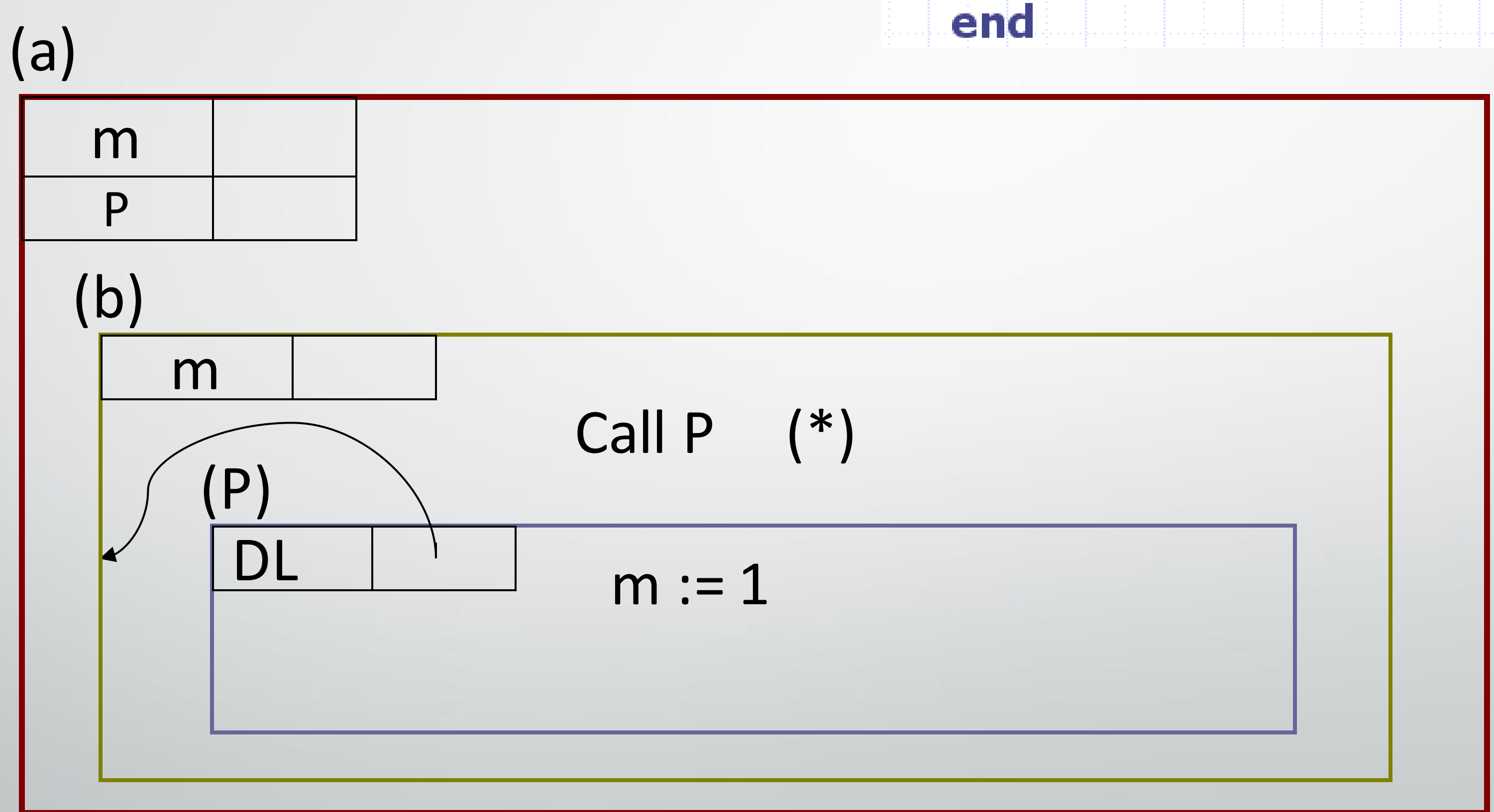
Invocation of P from Outer Block

```
a: begin integer m;
     procedure p;
          m := 1;
     b: begin integer m;
          P                    (*)
       end;
     P                         (**)
  end
```



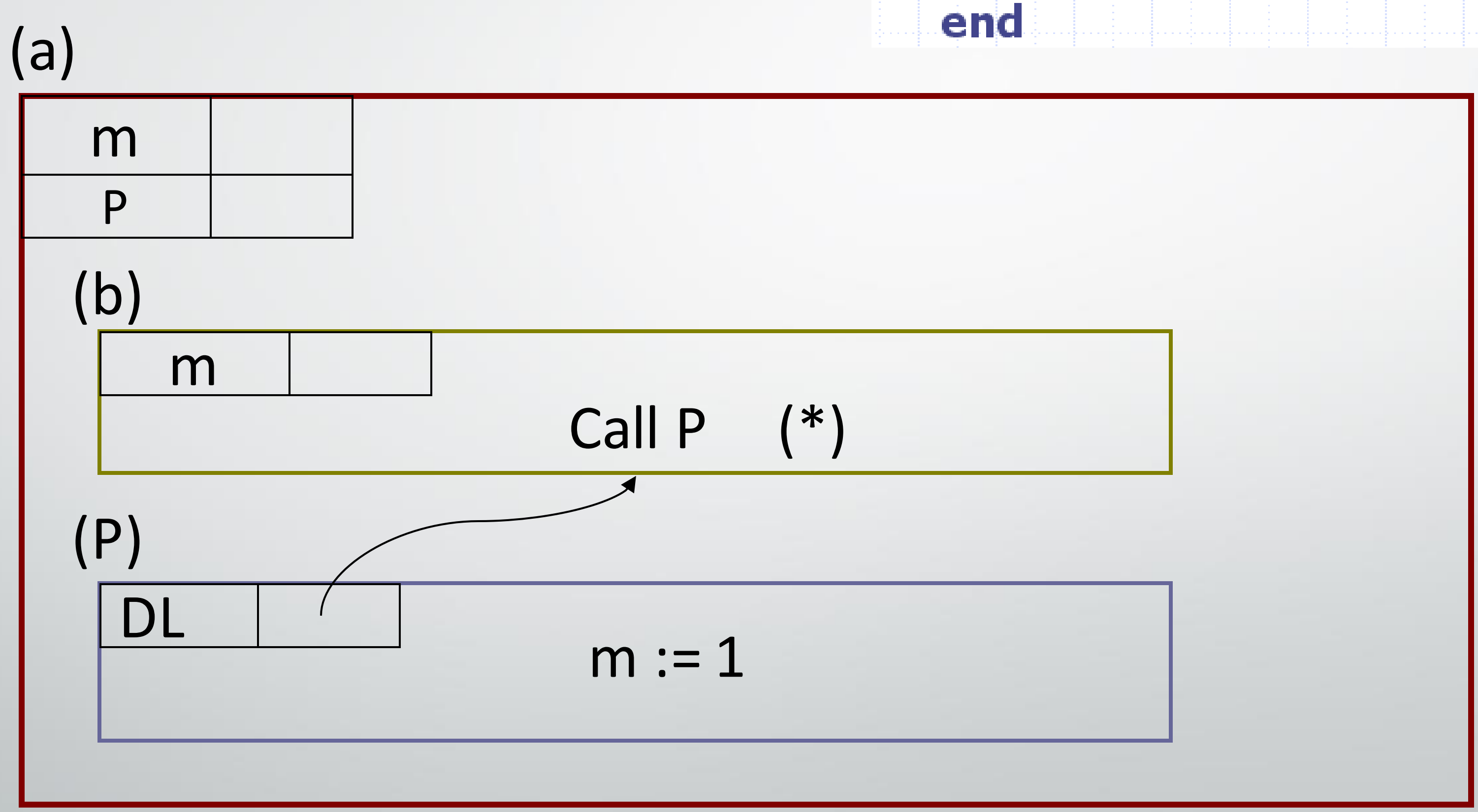
Invocation of P from Inner Block

```
a: begin integer m;
     procedure p;
          m := 1;
     b: begin integer m;
          P                    (*)
       end;
     P                         (**)
  end
```



Invocation of P When Called in Environment of Definition

```
a: begin integer m;
      procedure p;
           m := 1;
      b: begin integer m;
           P                    (*)
         end;
      P                         (**)
   end
```



Static and Dynamic scoping

- Static Scoping
  - Aids reliable programming (Structure Principle).
  - Scope defined at compile time
  - Procedures are called in the environment defined at the time of definition
  - Used by ALGOL

- Dynamic Scoping
  - Determined at run-time
  - Procedures are called in the environment of their caller.
  - Used by Lisp

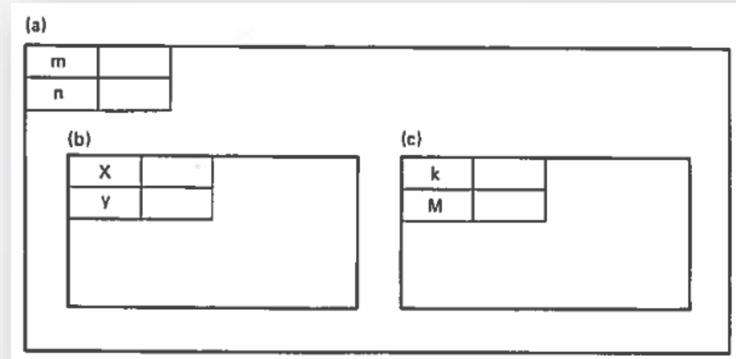
Dynamic Scoping

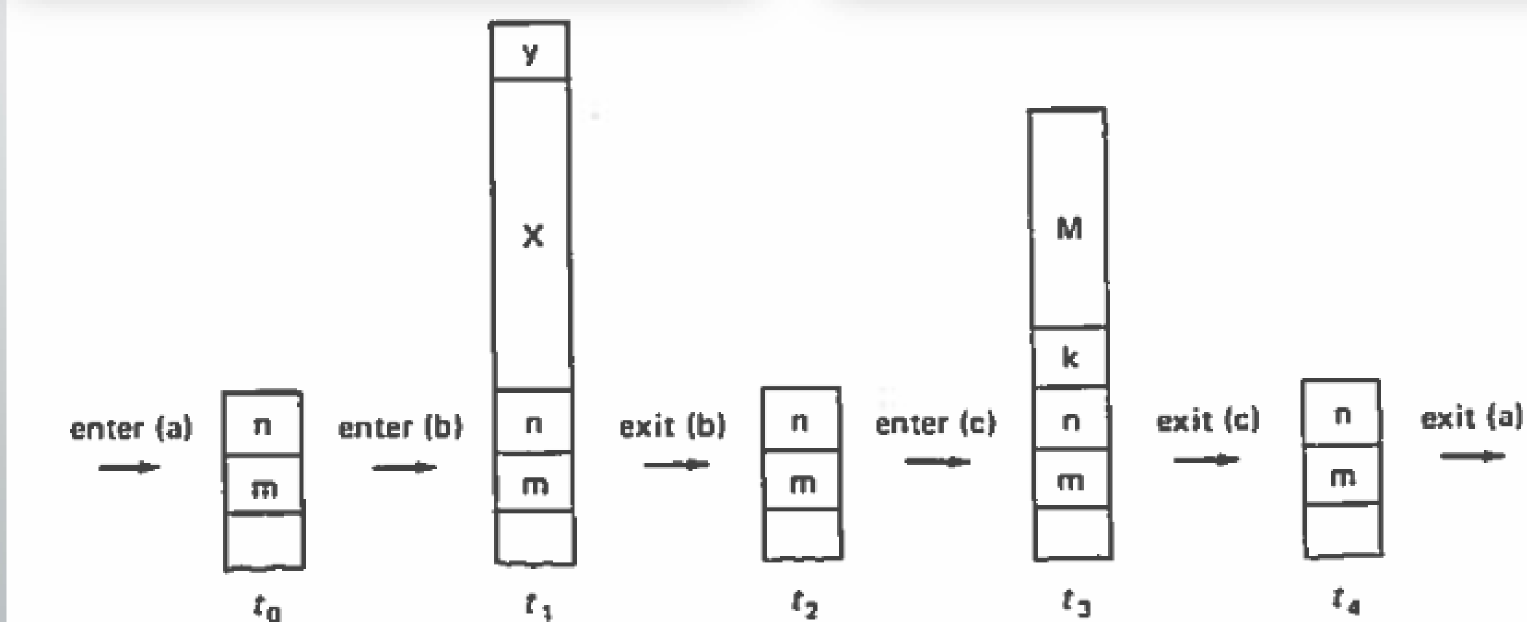
- Generally rejected in recent years

- Even newer versions of Lisp have gone to static scoping.

Block permits an efficient storage management on stack

```
a:begin integer m, n;
    b:begin real array X[1:100]; real y;
          :
      end
    :
    c:begin integer k; integer array M[0:50];
          :
      end
end
```





Design:
Data Structures

3.4. Data Structures

- The primitives are mathematical Scalars.
  - Integers, real, Boolean

Data Structure Primitives

- Mathematical scalars
  - Integer
  - Real
  - Boolean
- No double precision (machine dependent)
  - Causes portability issues
- No complex
  - Can be implemented using real

In FORTRAN

- In FORTRAN we have
  - At most 19 continuation cards
  - At most 6 characters for each identifier
  - At most 3 dimensions for arrays

Zero-One-Infinity Principle

The only reasonable numbers in a programming language design are zero, one, and infinity.

Dynamic Arrays

- Stack allocation permits dynamic arrays.
- The Algol-60 design is a good trade-off between flexibility and efficiency.
  - Dynamic arrays are very simple to implement on a stack.

Arrays

- Lower bounds follow the 0,1, ∞ rule
  - integer array number of days [100:200]
- Arrays are allocated dynamically

```
begin
integer i, j
i:=-35
j:=68
        begin
        integer array x[i:j]
        end
end
```

Strong Typing

- Algol has strong typing: the type abstractions of the language are enforced → prevent meaningless operations by programmer.
- *Conversions* and *coercions* are allowed.

Design:
Control Structures

3.5. Control Structures

- Control structures are generalizations of FORTRAN's.
- Nested statements are very important.
- Compound statements are hierarchical structures.
- Nesting led to structured programming.
- Procedures are recursive.
    - Local may have several instantiations in several activation records.

If Statement

- if *exp* then *stmt1* else *stmt2*;
- Can be use in an assignment statement
- C := if A > 0 then 1 else 2;
  - Equivalent to:
    if A > 0 then C := 1
    else C := 2
  - Similar to C's conditional expression

Compound Statements

- A begin block is a statement
- A begin block can be used anywhere a single statement is legal
- Solves the FORTRAN IV IF statement problem of allowing only one statement

A Minor Issue

```
for i := 1 step 1 until N do
        ReadReal(val);
        Data[i] := if val < 0 then –val else val;
for i:= 1 step ....
```

Should have been:

```
for i := 1 step 1 until N do
        begin
        ReadReal(val);
        Data[i] := if val < 0 then –val else val;
        end
for i:= 1 step ....
```

The brackets

- **begin-end** brackets
  - Group statements into compound statements
  - Delimit blocks, which define nested scopes

Procedures

- Inherently recursive
- Can pass parameters by value as well as by name

Parameter Passing by Value

```
Procedure switch(n);
        value n; integer n;
        n := 3;
   …
```

- Avoids FORTRAN constant changing problem
- Actual copied into variable corresponding to formal
- Secure; local variable will not overwrite actual parameter
- Does not allow output parameters (input only)
- Inefficient for arrays (or other non-primitive data structures)
- Copy must be made of entire array in activation record
- Copying takes time

Parameter Passing by Name

- Based on substitution

```
procedure Inc (n);
    value n; integer n;
    n := n + 1;
```

- What does Inc(k) do?
  - Nothing – call by value only
- Change to call by name (the default)

```
procedure Inc (n);
    integer n;
    n := n + 1;
```

Parameter Passing by Name

- NOT a call by reference.
- Substitutes “k” in the caller for “n” in the procedure.

Example

```
procedure S (el, k);
        integer el, k;
        begin
                k := 2;
                el := 0;
        end;

A[1] := A[2] := 1;
i := 1;
S (A[i], i);
```

Example (2)

- Executes as if it were written:

```
procedure S (A[i], i);
    integer A[i], i;
    begin
        i := 2;
        A[i] := 0;
    end;
```

- Note that this is not the expected A[i] := 0
- Implementation mechanism is called a *thunk*

Another example: Jensen's device

- $x = \sum_{i=1,n} V_i$ x := Sum(i,1,n,V[i])

```
real procedure Sum(k,l,u,ak);
  value l,u;  integer k,l,u; real ak;
  begin real S; S:=0;
    for k:=1 step 1 until u do
         S:= S+ak;
     Sum:=S;
  end;
```

- *This Sum procedure is very general*

$$x = \Sigma_{\mathrm{i=1,m}} \Sigma_{\mathrm{j=1,n}} A_{ij}$$

Implementation

1. Passing the text of the actual parameter to the procedure
   - Compile and execute this text every time the parameter was referenced.
2. Compile the actual parameter into machine code and then copy this code into the callee every where the parameter is referenced
   - This code would be copied many times
   - Each time with different size of code
3. Passing the address of the compiled code for the actual parameter, the ***thunk***

Thunks

- Simple parameterless subprogram
- Every time the parameter is referenced, the callee can execute the thunk by jumping to this address.
- The result of executing the thunk, an address of a variable, is returned to the callee.

Thunks

- x:= Sum (i, 1, m, Sum (j,1,n,A[i,j]))

Sum (k,l,u,ak)

k → i
l = 1
u = m
ak → thunk:
Sum(j,1,n,A[i,j])

Sum (k,l,u,ak)

k → j
l = 1
u = n
ak → thunk:
A[i,j]

Scope of variables in thunks

- The association for parameters are *back in the calling program*.

- call Sub(x) for invoking Sub(y),

  causes ambiguity if there is a x in Sub.

Pass by Name

- Is powerful
- Can be confusing
- Expensive to Implement
- Which would you rather be the default?

Pass by Name

- Write a swap procedure for swapping two variables.
- Does it work correctly for all actual parameters?
- Why?

Conceptual Models of a Programming Languages

- David Norman[1] (psychologist): "A good conceptual model allows us to predict the effects of our actions."
  - Designer's Model – reflects system construction
  - System Image – created by the designer; basis for User's Model – includes manuals, diagrams, etc.
  - User's Model – formed by the user based on the system image, personal competence and comfort.

[1]*Psychology of Everyday Things* (Basic Books, 1988)

Out-of-Block gotos

```
    begin
        begin
            goto exit;
        end
exit:
    end
```

Out-of-Block gotos

- No longer just a simple jump instruction
- Must terminate the block as if it left through the end
  - Release variables
  - Kill activation record
- Can you branch into a block?

For Loop

- Two basic forms:
  - for *var* := *exp* step *exp’* until *exp”* do *stat*
  - for *var* := *exp* while *exp’* do *stat*
- Also
  - for days := 31, 28, 31, 30, 31, …, 31 do *stat*
- *And it can get even worse*

Even Worse

```
for I := 3, 7,
        11 step 1 until 16,
        i/2 while i ≥ 1,
        2 step i until 32
  do print (i)
```

Violation of principle

- for i := m step n until k do …
- ALGOL specifies that m, n and k will be revaluated on every iteration of the loop
- Reevaluation must be done for each cycle, even if the values haven't changed or are constants
- Cost of doing this is distributed over all uses of for loops, even when m, n and k are constants.

The Localized Cost Principle

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

Switch Statement

```
          begin
                    switch marital status = single, married, divorced,
widowed;
                    …
                    goto marital status[I]
single:             … handle single case
                              goto done:
married:                      … handle married case
                              goto done:
divorced:                     … handled divorced case
                              goto done:
widowed:                      … handle widowed case
done:
          end;
```

Bizarre Switch

```
begin
        switch S = L, if i > 0 then M else N, Q;
        goto S[j];
end
```

Note that S can have 3 values:

- L
- If i > 0 the M else N
- Q

Summary

- Parameters can be passed by value.
- Pass by value is very inefficient for arrays.
- Pass by name is based on substitution.
- Pass by name is powerful.
- Pass by name is dangerous and expensive.

Summary

- Good conceptual models help users.
- Out-of-block gotos can be expensive.
- Feature interaction is a difficult design problem.
- The for-loop is very general.
- The for-loop is Baroque.
- The switch is for handling cases.
- The switch is Baroque.

Syntactic Issues: Algol-60

- Syntactic Structures
- Descriptive tools: BNF
- Elegance
- Evaluation and Epilog

Software Portability

Portability Principle

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

Free Format

- FORTRAN: fixed format

- Algol: free format

Free Format

- FORTRAN: fixed format

- Algol: free format

Programmers had to think about:

“How a program should be formatted?”

Free Format

- Example: FORTRAN style

```
for i:=1 step 1 until N do
begin
real val;
Read Real (val);
Data [i] := val;
end;
for i:=1 step 1 until N do
sum:= sum + Data [i];
avg := sum/N;
```

Free Format

- Example: English style

```
for i:=1 step 1 until N do begin  real val;
Read Real (val); Data [i] := val; end; for
i:=1 step 1 until N do sum:= sum + Data [i];
avg := sum/N;
```

Free Format

- Example: English style

for i:=1 **step** 1 **until** N **do begin** **real** val;
Read Real (val); Data [i] := val; **end**; **for**
i:=1 **step** 1 **until** N **do** sum:= sum + Data [i];
avg := sum/N;

Remember the Structure Principle!

Free Format

- Example:

```
for i:=1 step 1 until N do
        begin
                real val;
                Read Real (val);
                Data [i] := val;
        end;
for i:=1 step 1 until N do
        sum:= sum + Data [i];
avg := sum/N;
```

Levels of Representation

- No universal character set!
- How to design the lexical structure?
  - Using those symbols available in *all* char sets.
  - Design *independent* of particular char sets.

Levels of Representation

- Three levels of the language:

1. Reference Language

$$a\,[i+1] := (a\,[i] + pi \times r \uparrow 2\,)\,/\,6.02_{10}23$$

2. Publication language

$$a_{i+1} \leftarrow \{\, a_i + \pi \times r^2 \,\} \,/\, 6.02 \times 10^{23}$$

3. Hardware representations

```
a (/i+1/) := (a (/i/) + pi * r**2 ) / 6,02e23
```

Problems of FORTRAN Lexics

- Example:

```
IF  IF  THEN
                 THEN = 0;
ELSE;
                  ELSE  ELSE = 0;
```

Problems of FORTRAN Lexics

- Example:

```
IF  IF  THEN
              THEN = 0;
ELSE;
               ELSE  ELSE = 0;
```

Confusion!!

Problems of FORTRAN Lexics

- How does Algol solve the problem?

if procedure **then** until := until +1 **else** do := false;

Lexical Conventions

- *Reserved words:*

  reserved words cant be used by the programmer

- *Keywords:*

  used word by the language are marked in some unambiguous way

- *Keywords in Context:*

  used words by the language are keywords in those context they are expected.

Arbitrary Restrictions, Eliminated!

Remember the
Zero-One-Infinity Principle!

- Number of chars in an identifier
- Subscript expression

Dangling else

```
if B then if C then S else T;
```

- What does the above code mean?

```
if B then begin if C then S else T  end;
```

```
if B then begin  if C then S end else T;
```

Dangling else

if B **then** **if** C **then** S **else** T;

- What does the above code mean?

Algol solved the problem:

"Consequent of an **if**-statement must be an unconditional statement"

Algol’s lexical and syntactic structures became so popular that virtually all languages designed since have been “Algol-like”.

Various Descriptive tools

- Example: *numeric denotations*

1. Giving Examples:

Integers such as: '-273'
Fractions such as: '3.76564'
Scientific notations: '$6.02_{10}23$'

- Gives a clear idea,
- But not so precise.

Various Descriptive tools

- Example: *numeric denotations*

2. English Description:

1. A digit is either 0,1,2,…,9.
2. An unsigned integer is a sequence of one or more digits.
3. An integer is either a positive sign followed by an unsigned integer or … .
4. A decimal fraction is a decimal point immediately followed by an unsigned integer.
5. An exponent part is a subten symbol immediately followed by an integer
6. A decimal number has one of three forms: … .
7. A unsigned number has one of three forms: … .
8. Finally, a number … .

- ☐ Precise,
- ☐ But not so clear!

Various Descriptive tools

3. Backus-Naur Form (BNF)

Backus Formal Syntactic Notation

- How to combine *perceptual clarity* and *precision*?
- Example: FORTRAN subscript expressions

```
c
v
v + c  or  v - c
c * v
c * v + c′  or  c * v - c′
```

Backus Formal Syntactic Notation

- How to combine *perceptual clarity* and *precision*?
- Example: FORTRAN subscript expressions

c
v
v + c or v - c
c * v
c * v + c′ or c * v - c′

The formulas make use of *syntactic categories!*

Naur's Adaptation of the Backus Notation

- BNF represents:
  - Particular symbols by themselves,
  - And classes of strings (syntactic categories) by phrases in angle brackets.

<decimal fraction> ::== .<unsigned integer>

Describing Alternates in BNF

- Example: alternative forms

```
<integer> ::== +<unsigned integer>
             | -<unsigned integer>
             |  <unsigned integer>
```

Describing Repetition in BNF

- Example: recursive definition

```
<unsigned integer> ::== <digit>
                      | <unsigned integer><digit>
```

Extended BNF

- How to directly express the idea “sequence of …”?
  - Kleene cross
  - Kleene star

<unsigned integer> ::== <digit>$^+$

<identifier> ::== <letter>{<letter>|<digit>}*

<integer> ::== [+/-]<unsigned integer>

Extended BNF

- How to directly express the idea “sequence of …”?
  - Kleene cross
  - Kleene star

- Why preferable?

Remember the
Structure principle!

Mathematical Theory of PLs

- Chomsky type 0, or *recursively enumerable* languages
- Chomsky type 1, or *context-sensitive* languages
- Chomsky type 2, or *context-free* languages
- Chomsky type 3, or *regular* languages

Mathematical Theory of PLs

- Languages described by BNF are context-free.

- Mathematical analysis of the syntax and grammar of PLs.

Design Has Three Dimensions

Scientific
Efficiency

Social
Economy

Symbolic
Elegance

Efficiency

- Efficiency seeks to minimize resources used.

- Resources :
  - Memory usage
  - Processing time
  - Compile time
  - Programmer typing time

Economy

- Economy seeks to maximize social benefit compared to its cost.
- The public: The programming community.
- Trade offs are hard to make since values change unpredictably.
- Social factors are often more important than scientific ones. (E.g. major manufacturers, prestigious universities, influential organizations)

Elegance

- The feature interaction problem.
  - It is impossible to analyze all the possible feature interactions.
  - All we need to do is to restrict our attention to designs in which feature interactions look good.
- Designs that *look* good will also *be* good.
- Good designers choose to work in a region of the design in which good designs look good.
- A sense of elegance is acquired through design experience.
  - Design, criticize, revise, discard!

Evaluation

- Algol-60 never achieved widespread use.
- Algol had no input-output
  - Little uniformity among input-output conventions.
  - It was decided that input-output would be accomplished by using library procedures.
  - Eventually several sets of input-output procedures were designed for Algol: It was too late!

Algol directly competed with FORTRAN

- Same application area.
- FORTRAN had gained considerable ground.
- More focus on reductive aspects of Algol.
- IBM decided against supporting Algol.
- Algol became an almost exclusively academic language.

A Major Milestone

- Introduced programming language terminology.
- Influenced most succeeding languages.
- An over general language: Quest for simplicity that resulted in Pascal.
- The basis of several extension, like Simula.
- Computer architects began to support Algol implementation.

Second Generation Programming Languages

- Algol-60: The first second generation programming language.
- Data structures are close to first-generation structures.
- Hierarchically nested name structures.
- Structured control structures (e.g. recursive procedures, parameter passing modes).
- Shifted from free formats.

Principle of Programming Language