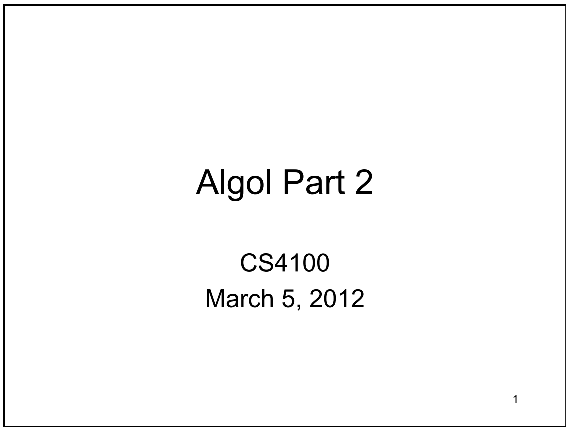
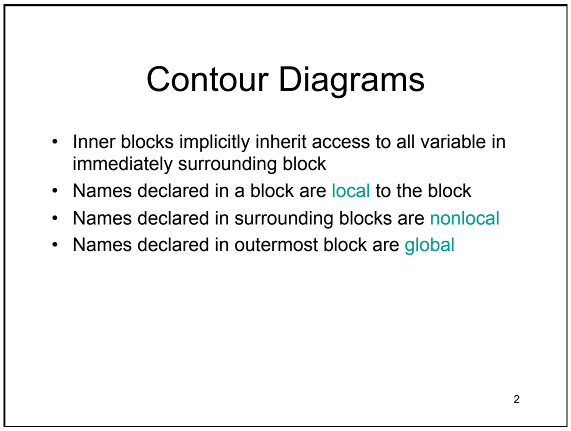


# Algol Part 2

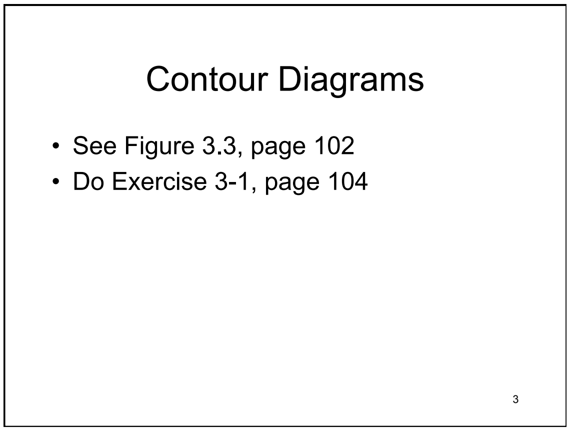
CS4100

March 5, 2012



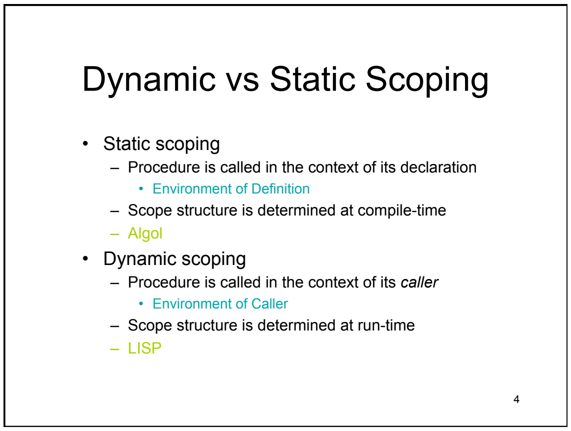
## Contour Diagrams

- Inner blocks implicitly inherit access to all variable in immediately surrounding block
- Names declared in a block are local to the block
- Names declared in surrounding blocks are nonlocal
- Names declared in outermost block are global



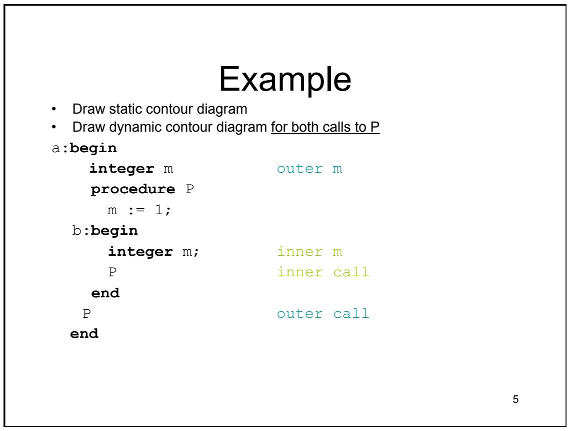
## Contour Diagrams

- See Figure 3.3, page 102
- Do Exercise 3-1, page 104



## Dynamic vs Static Scoping

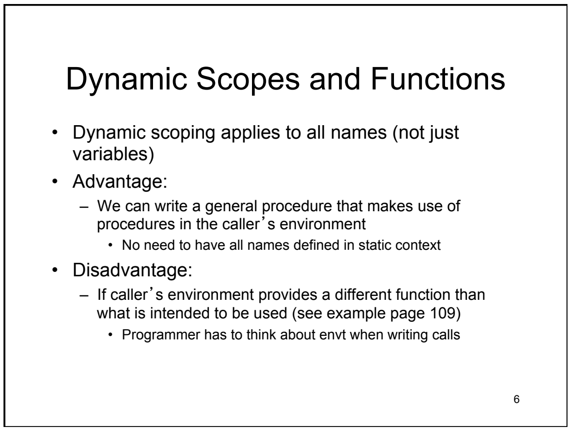
- Static scoping
  - Procedure is called in the context of its declaration
    - Environment of Definition
  - Scope structure is determined at compile-time
  - Algol
- Dynamic scoping
  - Procedure is called in the context of its *caller*
    - Environment of Caller
  - Scope structure is determined at run-time
  - LISP



## Example

- Draw static contour diagram
- Draw dynamic contour diagram for both calls to P

```
a:begin
    integer m          outer m
    procedure P
      m := 1;
  b:begin
      integer m;       inner m
      P                inner call
    end
   P                   outer call
  end
```



## Dynamic Scopes and Functions

- Dynamic scoping applies to all names (not just variables)
- Advantage:
  - We can write a general procedure that makes use of procedures in the caller's environment
    - No need to have all names defined in static context
- Disadvantage:
  - If caller's environment provides a different function than what is intended to be used (see example page 109)
    - Programmer has to think about envt when writing calls

# Which one is better?

- General rule:
  - What is natural to humans will cause less problems in the long run
  - If humans can understand static scoping better, than it will result in higher quality programs in the long run
- Dynamic scoping is confusing
  - Generally rejected (not used in new languages)
  - Static scoping agrees more with the program's dynamic behavior

# Blocks Permit Efficient Storage Management

- Fortran used EQUIVALENCE
  - Not safe, since there is no guarantee of exclusive use of memory
- Blocks permit reuse of memory

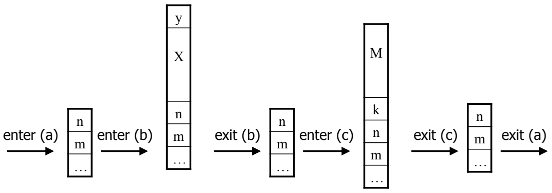
```
a:begin integer m, n;
  b:begin real array X[1:100], real y;
  ...
  end
...
  c:begin integer k; real array M[0:50];
  ...
  end
end
```

# Run-Time Stacks

- Variables in blocks *b* and *c* are never used at the same time
- When exiting *b*, its variables may be discarded
- Notice: Block entered last will be exited first
  - LIFO (last-in first-out) order
  - Can use a stack to organize activation records
  - When block is entered, its AR is pushed onto stack
  - When block is exited, its AR is popped off stack
  - Assumption: No local variables are initialized

# Example

- From previous program



# Responsible Design

- Algol designers did not include EQUIVALENCE
  - Programmers might have asked for it…
  - Instead, they looked at the root of the problem
  - "Don't ask what they want, ask how the problem arises"
  - Language designers are responsible for the features in the language, not programmers

# *Principles of Programming*

- The Responsible Design Principle
  - Do not ask programmers what they want, find out what they need.

# Data Structures

- Primitives
  - Mathematical scalars, like in Fortran
  - integer, real, Boolean
  - complex and double not included
- Double: platform dependent
  - Not portable
  - Why? Because we need to know the size of a word to know how big double is.
  - Alternate approaches:
    - specify precision
    - Let compiler pick precision

# Why no complex?

- Not primitive
  - Can be constructed using other types easily (2 reals)
- Is it easy to use *real*s for complex?
  - Yes, but inconvenient
  - Need supporting operations
    - ComplexAdd(x, y, z), etc.
- Designers' choice:
  - Is it worthwhile to add the complexity/overhead of another type? (conversions, coercion, operator overload, etc.)
  - Will they get enough use?

# Strings

- Yet another data structure that needs full support (operation, etc.)
- Algol designers included strings as second-class citizens
  - `string` type is only allowed for formal parameters
  - String literals can only be actual parameters
  - No operations
  - Strings can only be passed around in procedures
  - Cannot actually *do* anything with them
- What's the point???
  - String will end up getting passed to output procedure written in a lower (machine) language that can handle it

# Zero-One-Infinity

- Programmers should not be required to remember arbitrary constants
- Fortran examples
  - Identifiers have max. 6 characters
  - There are at most 19 continuation cards
  - Arrays can have at most 3 dimensions
- Regularity in Algol requires small number of exceptions
  - Gives rise to Zero-One-Infinity principle
  - E.g.: Identifier names should be either 0, 1 or unlimited length. (0 & 1 don't make much sense)

# *Principles of Programming*

- The Zero-One-Infinity Principle
  - The only reasonable numbers in programming language design are zero, one and infinity.

# Arrays are Generalized

- Arrays can have any number of dimensions
- Lower bound can be number other than 1
  - More intuitive, and less error prone than fixed lower bound
- Arrays are dynamic
  - Array bounds can be given as expressions, which allows recomputation every time the block is entered
  - Array size is set until block is exited
- (Fortran had fixed array sizes.)

## Strong Typing

- Strong typed language
  - Prevents programmer to perform meaningless operations on data
  - Not to be confused with legitimate type conversions (integer + real (coercion))
- Fortran
  - Weakly typed
  - Permits adding to a Hollerith constant, etc.
  - Equivalence allows setting up the same memory for different types
    - Security and maintenance problem
    - Intentional type violation is not portable
- Exception: System programming (C)
  - Have to treat memory cells as raw storage without regard to type

## Control Structures

- Primitive statements are similar to Fortran's
  - Assignment
  - Control flow
  - No input/output

## Controls are Generalized: *if*

- Fortran had many restrictions
  - `if (exp) simple statement`
    - Statement restricted to GOTO, CALL, or assignment
- Algol removes restrictions
  - All statements are allowed (even 'if' in body of 'if')
  - 'else' added to address *false* condition

## Controls are Generalized: *for*

- Algol's *for* is more general than Fortran's *do*

```
for i := 1 step 1 until N do
      sum := sum + Data[i]
```

  - Leading-decision loop:

```
for NewGuess := Improve(OldGuess)
  while abs(NewGuess – OldGuess) > 0.01
  do OldGuess := NewGuess
```

  - Same as while loop in newer languages:

```
NewGuess := Improve(OldGuess);
while abs(NewGuess – OldGuess) > 0.01 do
  begin
      OldGuess := NewGuess;
      NewGuess := Improve(OldGuess);
  end
```

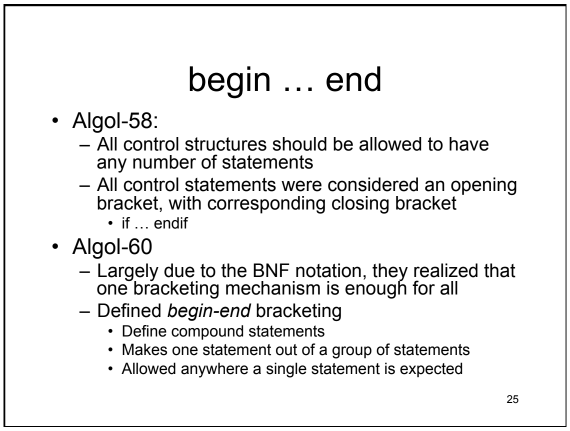
## Another for loop

```
for i := 3, 7,
     11 step 1 until 16,
     i ÷ 2 while i >= 1,
     2 step i until 32 do
   print( i );
```

3 7 11 12 13 14 15 16 8 4 2 1 2 4 8 16 32

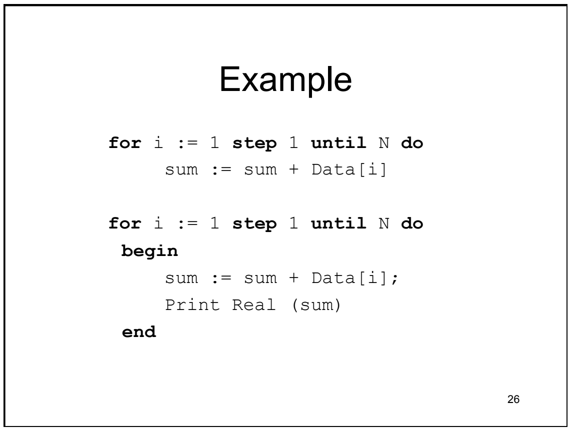
## Goal: Regularity

- Algol was designed around regularity
  - "Anything that you think you ought to be able to do, you will be able to do."
  - Elaboration on zero-one-infinity principle
    - Remove inexplicable exceptions from the language



## begin … end

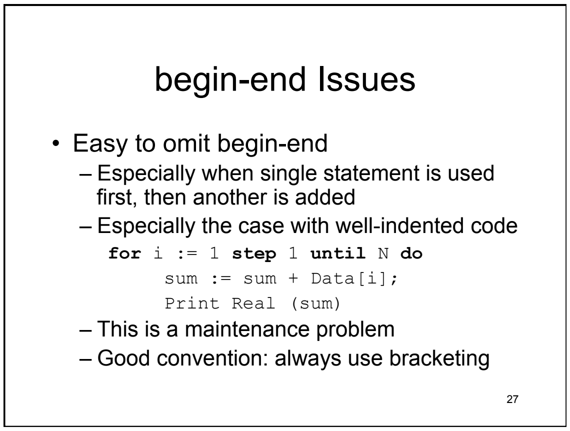
- Algol-58:
  - All control structures should be allowed to have any number of statements
  - All control statements were considered an opening bracket, with corresponding closing bracket
    - if … endif
- Algol-60
  - Largely due to the BNF notation, they realized that one bracketing mechanism is enough for all
  - Defined *begin-end* bracketing
    - Define compound statements
    - Makes one statement out of a group of statements
    - Allowed anywhere a single statement is expected



## Example

```
for i := 1 step 1 until N do
    sum := sum + Data[i]

for i := 1 step 1 until N do
 begin
    sum := sum + Data[i];
    Print Real (sum)
 end
```

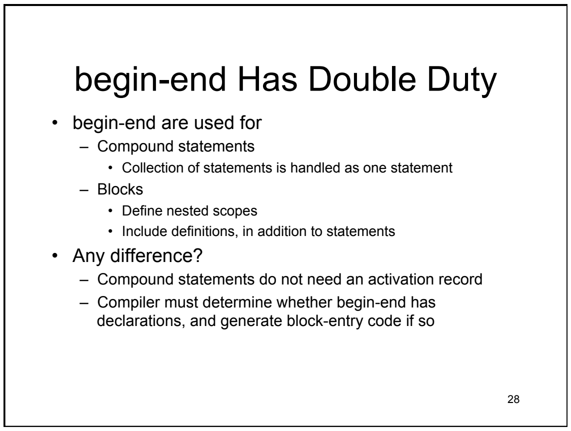


## begin-end Issues

- Easy to omit begin-end
  - Especially when single statement is used first, then another is added
  - Especially the case with well-indented code

```
for i := 1 step 1 until N do
    sum := sum + Data[i];
    Print Real (sum)
```

  - This is a maintenance problem
  - Good convention: always use bracketing



## begin-end Has Double Duty

- begin-end are used for
  - Compound statements
    - Collection of statements is handled as one statement
  - Blocks
    - Define nested scopes
    - Include definitions, in addition to statements
- Any difference?
  - Compound statements do not need an activation record
  - Compiler must determine whether begin-end has declarations, and generate block-entry code if so