Names

- o maximum length: readability vs. symbol table size
- o connector: (_), (-)
- o case-sensitive? MySalary, mySalary, mysalary
- o special word (keyword): reserve or not?

Ex 1. Fortran

```
INTEGER REAL
     :
REAL = 10
```

Ex 2. C

```
int main ()
{
    int auto, ford;
        ford:= 150;
        auto:= ford;
    printf ("%\n", auto);
}
```

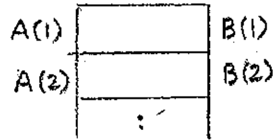
Variable

attributes (characters)

- . name
- . address
- . value
- . type
- . lifetime
- . scope

Aliasing

Fortran equivalence (A(1), B(1))



B DEFINES A (Cobol)
B REDEFINES A (PL/I)

Address vs. Value

$$\underline{A} := \underline{A} + 1;$$

↑ l-value (address) ↑ r-value (value)

Binding times:

o Type binding

Ex. real X,Y

:

Y := X + 10;

type of var X – compile time
value of var X – run time
set of possible types of X – language design time

Static Type Binding

- mostly compiled

{ Explicit declaration
{ Implicit declaration (Default)

Ex. Fortran/PL/I : I-N: integer, others real
Perl: $a: scalar, @a: array

Type Conversion

{ Explicit: PL/I,Pascal,Ada Ex. N:= ROUND(X);
{ Implicit: Fortran Ex. R = N * 3.14 (R: real; N: integer → real; 3.14: real)

Note. Coercion ≡ automatic type conversion

Dynamic Type Binding

- mostly interpreted
- flexibility
- no error-checking at compile time

APL, Snobol, Lisp

```
Ex.  A ← 5 3 7 2     // array //
     A ← 3.14        // real scalar //
```

o Type inference

- Type is inferred from the context whenever possible.
- If not possible, "type cast"

Ex. ML

```
fun leng(r)  = 2.0 * r * 3.14;        // real //
fun square(x)  =  (x: int) * x;       // int //
fun square(x) : real = x * x;         // real //
fun square (x: real) = x * x;         // real //
```

Type Checking

- check compatibility
- type-error or coercion

Strong Typing

- When type errors are always detected at compile time

① each name has a single type associated with it.
② The type is unchanging and known at compile time.

Fortran 77 - not strong: equivalence, lack of parameter type checking
Pascal – almost strong (except variant records)
Modula-2 – not strong: type-checking avoidance feature
Ada - almost strong: type checking suspension request by users
C/C++ - not strong (union and functions that don't require type checking of parameters)
Java – almost strong

Note. Assignment statement – side-effect

A: [17.4]
B: [8.0]

Static

A:= B; (Pascal,Ada)
A = B (Fortran,PL/1,C)
MOVE B TO A. (Cobol)

A: [8.0]
B: [8.0]

Dynamic

A ← B (APL)
(setq A B) (Lisp)

A [→] [17.4]
B [→] [8.0]

A [ ] [17.4]
B [ ] → [8.0]

- Type can be changed.

Type Equivalence

Name type equivalence - most strict, same type name.
Structure type equivalence - same internal structure

		Name	Structure
Ex	type celsius = real; farenheit = real;	no	yes

Ex Pascal:

Structure for most cases, but name for formal parameters

Declaration equivalence:

```
type  type1 = array[1..10] of real;
      type2 = array[1..10] of real;
      type3 = type2
```

type2 = type3 but type1 ≠ type2

Ex Ada – name

subtype SMALL_TYPE is INTEGER range 0..99;
SMALL_TYPE variables are compatible with INTEGER variable

A: array(1..10) of FLOAT; B: array(1..10) of FLOAT;	type LIST10 is array(1..10) of FLOAT A,B: LIST10;
No **(2 anonymous distinct types)**	**Yes**

Ex C structure except records
C++ name

- Which is better? Name : Safer, easy implementation

Scope

- visibility

Scope rule:

(1) Static – static structure of the program (environment of definition)
compile time, Algol 60

(2) Dynamic – caller's environment, run time

Note. (Static) inner block inherits access to variables of its immediate surrounding block

Static Scoping

```
proc Big
    X1: integer

    proc Sub1
      X2: integer
      begin
        :
      end

    proc Sub2
    begin
       X  ---------------------------- (*)          // X1 //
    end

begin (* Big *)
   :
end
```

Block: degenerated procedure

C	Algol	Ada

if (a < b) {
int temp
temp = a;
a = b;
b = temp;
}
	begin
	declarations
	statements
	end
		declare
		declarations
		begin
		statements
		end

Nested Block

Scope: local, nonlocal, global (outermost block)

Evaluation of Static Scoping

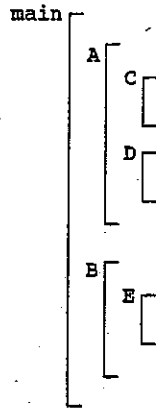


Figure 5.1
The structure of a program

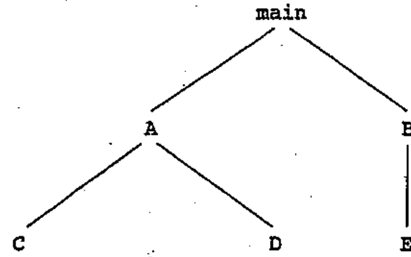
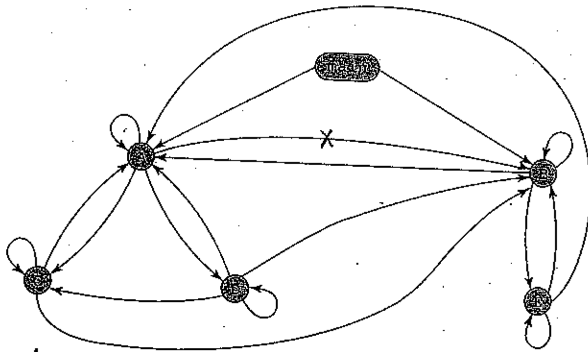
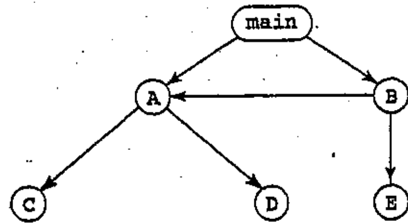


Figure 5.2
The tree structure of the program in Figure 5.1



Potential calls



Desirable calls

Shared Data

easily defined in block structured Languages

```
[array A[1:100]
  P [A
  Q [A
  R [A
```

Note Fortran Common

Dynamic Scope

- APL, Snobol, Lisp (Perl, Common Lisp)

```
proc  Big
   X1: integer

   proc  Sub1
      X2: integer
      begin
         :
      end

   proc  Sub2
      begin
          X    ----------------- (*)
      end

   begin  (* Big *)
      :
   end
```

(Case 1) Big → Sub1 → Sub2   X =

(Case 2) BIG → Sub2   X =

Lifetime

- how long a variable is attached to its memory location

Retention: history-sensitive   Fortran

Deletion: Algol family

Note. Scope vs. Life time

```
void  printheader() {
   ...
}
void compute() {
   int sum;
    :
  printheader();
}
```

Scope of sum - compute()

Lifetime of sum - printheader()

Referencing Environment

STATIC

```
program example
  var a, b : integer;
  ...
  procedure sub1;
    var x1, y1 : integer;
    begin  {of sub1}
    ...    <-----------------------1   x1, y1, a, b
    end;   {of sub1}

  procedure sub2;
    var x2 : integer;
    ...
    procedure sub3;
      var x3 : integer;
      begin   {of sub3}
      ...    <---------------------2   x3, a, b
      end;    {of sub3}
  begin   {of sub2}
  ...    <-------------------------3   x2, a, b
  end;    {of sub2}
begin   {of example}
...    <---------------------------4   a, b
end.
```

Dynamic

```
void sub1( ) {
    int a, b1;
        --------------------------- 1          a, b1, c2, d
}

void sub2( ) {
    int b2, c2;
        --------------------------- 2          b, c2, d
    sub1( );
}

void main( ) {
    int c3, d;
        --------------------------- 3          c3, d
    sub2( );
}
```

main → sub2 → sub1

Named Constants

- convenient

Ex.
```
const   max = 100;
  :
for i:= 1 to max do
    :
for k:= 1 to max do
    :
```

Variable Initialization

Fortran 77	INTEGER SUM DATA SUM /0/
Fortran 90	INTEGER SUM = 0
Pascal	const sum = 0;
Ada	sum: INTEGER := 0;
Java	final int sum = 0;