SSE2-PLDE_5

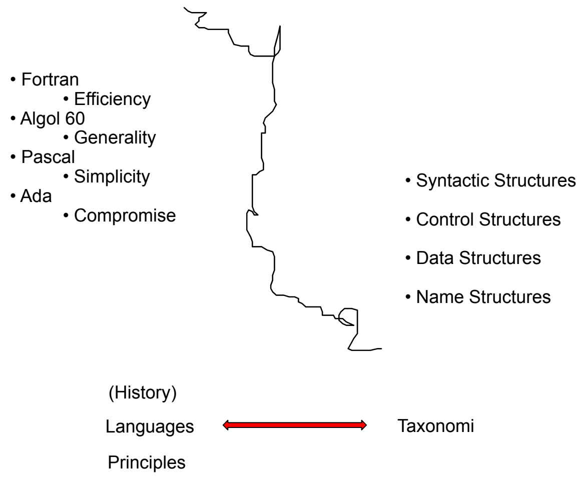
Contents:

- Language Design
- Programming Paradigms

Literature:

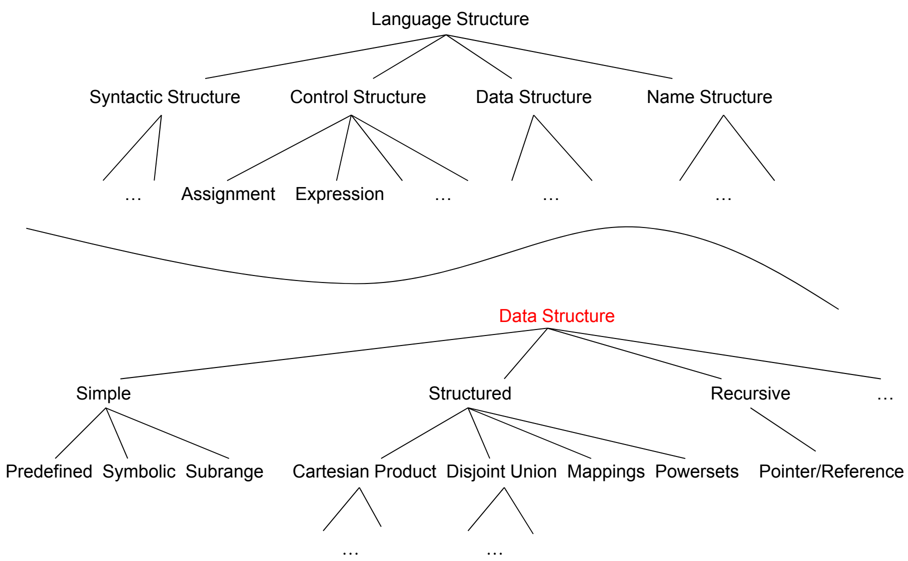
- MacLennan: Principles of Programming Languages (Design, Evaluation, and Implementation):
  - Pages 45-51, 86-89, 101-110, 121-126, 137-138, 180-190, 263-272

Principles of Programming Languages



… of (Imperative/Procedural) Programming …

Programming Language "Taxonomy"



Principles (of Programming Languages)

1.
2.
3. **Defense in Depth:** *Have a series of defenses so that if an error is not caught by one, it will probably be caught by another.*
4.
5.
6. **Information Hiding:** *The language should permit modules designed so that (1) the user has all the information needed to use the module correctly, and nothing more; and (2) the implementor has all the information needed to implement the module correctly, and nothing more.*
7. **Labeling:** *Avoid arbitrary sequences more than a few items long. Do not require the user to know the absolute position in the list. Instead, associate a meaningful label with each item and allow the items to occur in any order.*
8. **Localized Cost:** *Users should pay only for what they use; avoid distributed costs.*
9.
10.
11. **Portability:** *Avoid features or facilities that are dependent on a particular computer or a small class of computers.*
12.
13.
14.
15.
16. **Structure:** *The static structure of the program should correspond in a simple way to the dynamic structure of the corresponding computations.*
17. **Syntactic Consistency:** *Similar things should look similar, different things different.*
18. **Zero-One-Infinity:** *The only reasonable numbers are zero, one, and infinity.*

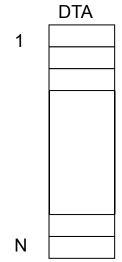
Fortran, Algol 60, Pascal, Ada

	Fortran	Algol 60	Pascal	Ada
Control Structures	*	***Structured**	Simplification	**Unification**
Data Structures			***Structured**	**...**
Name Structures		***Hierarchical**	Simplification	***Module**
Syntactic Structures	*Fixed form	**Free form**	**...**	**...**
				(Exceptions) **(Concurrency)**

FORTRAN

$$SUM = \left(\sum_{i=1\ldots N} DTA[i]\right) / N$$

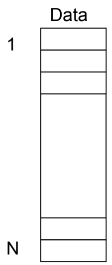
```
      DIMENSION DTA(900)
      SUM = 0.0
      READ 10, N
10    FORMAT(I3)
      DO 20 I= 1,N
      READ 30, DTA(I)
30    FORMAT(F10.6)
      IF (DTA(I)) 25,20,20
25    DTA(I)= -DTA(I)
20    CONTINUE
      DO 40 I= 1,N
      SUM = SUM + DTA(I)
40    CONTINUE
      AVG = SUM / FLOAT(N)
      PRINT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



Algol 60

$$SUM = \left(\sum_{i=1\ldots N} Data[i]\right) / N$$

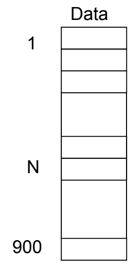
```
begin
   integer N;
   ReadInt(N);

   begin
      real array Data[1:N];
      real sum, avg;
      integer i;
      sum:= 0;
      for i:= 1 step 1 until N do
      begin real val:
         readReal(val);
         Data[i]:= if val<0 then -val else val
      end;
      for i:= 1 step 1 until N do
         sum:= sum+Data[i];
      avg:= sum/N;
      PrintReal(avg)
   end
end
```



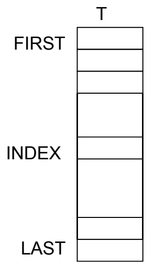
Pascal

$$SUM = \left(\sum_{i=1\ldots N} Data[i]\right) / N$$

```
program AbsMean (input, output);
const Max= 900;
type index= 1..Max;
var
   N: 0..Max;
   Data: array[index] of real;
   sum, avg, val: real;
   i: index;
begin
   sum:= 0; readln(N);
   for i:= 1 to N do
   begin readln(val);
      if val<0 then Data[i]:= -val
      else Data[i]:= val
   end;
   for i:= 1 to N do sum:= sum + Data[i];
   avg:= sum/N;
   writeln(avg)
end.
```



Ada




```
package TABLES is

   type TABLE is array (INTEGER range <>) of FLOAT;

   procedure BINSEARCH (T: TABLE; SOUGHT: FLOAT;
            out LOCATION: INTEGER; out FOUND: BOOLEAN) is
      subtype INDEX is INTEGER range T'FIRST .. T'LAST;
      LOWER: INDEX:= T'FIRST;
      UPPER: INDEX:= T'LAST;
      MIDDLE: INDEX:= (T'FIRST+T'LAST)/2;
   begin
      loop
         if T(MIDDLE) = SOUGHT then
            LOCATION:= MIDDLE:
            FOUND:= TRUE;
            return;
         elseif UPPER < LOWER then
            FOUND:= FALSE
            return
         elseif ...

         elseif T(MIDDLE > SOUGHT then
            UPPER:= MIDDLE -1;
         else LOWER:= MIDDLE +1:
         end if;
         MIDDLE:= (LOWER+UPPER)/2;
      end loop;
   end BINSEARCH;
end TABLES;
```

Syntactic Structures

Fixed format:	1-5, 6, 7-72, 73-80
Ignoring all blanks:	`DO 20 I = 1.1000` `DO 20 I = 1,1000`
No reserved words:	`IF (I-1) = 1 2 3` `IF (I-1) 1,2,3`
Reserved words – Keywords (marked):	<u>`then`</u> **`then`** `'then'` `THEN`

Dangling 'else' problem:
Fully bracketed syntax:
	`if C1 then if C2 then S1 else S2`
	`loop ... end loop`
	`if ... end if`
	…

Unique parentheses:	`procedure` <u>`N`</u> `... end` <u>`N`</u>

• Fortran:	Fixed format,	Linear
• Algol 60:	Free format,	Structure
• Pascal:	Free format,	Structure
• Ada:	Systematics,	Fully bracketed

Zero-One-Infinity Principle: *The only reasonable numbers are zero, one, and infinity*

FORTRAN:

Identifiers are limited to 6 characters
At most 19 continuation cards
Arrays have at most 3 dimensions

FORTRAN Control Structures

FORTRAN II Statements	704 Branch	
`GOTO n`	`TRA k`	`(transfer direct)`
`GOTO n, (n1, …, nm)`	`TRA i`	`(transfer indirect`
`GOTO (n1, …, nm), n`	`TRA i, k`	`(transfer indexed)`
`IF (a) n1, n2, n3`	`CAS k`	`(compare AC with storage)`
`IF ACCUMULATOR OVERFLOW n1, n2`	`TOV k`	`(transfer on AC overflow)`
`IF QUOTIENT OVERFLOW n1, n2`	`TQO k`	`(transfer on MQ overflow)`
`DO n i = m1, m2, m3`	`TIX d, i, k`	`(transfer on index)`
`CALL name (args)`	`TSX i, k`	`(transfer and set index)`
`RETURN`	`TRA i`	`(transfer indirect)`

Portability Principle: *Avoid features or facilities that are dependent on a particular computer or a small class of computers*

FORTRAN:

```
IF (e) n1, n2, n3
```

Assembly language for the IBM 704
Evaluate expression then branch depending on whether the result is negative, zero or positive
Exactly function of 704's CAS instruction

Defense in Depth Principle: *Have a series of defenses so that if an error is not caught by one, it will probably be caught by another*

FORTRAN:

Assigned GOTO:

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

Integer variables can hold a number of things besides integers, such as the address of a statement
Variables of type LABEL could hold addresses of statements

(If syntactic checking is ok, then type checking would not be ok)
(syntax analysis) (contextual analysis)

```
integer I

label I
```

Syntactic Consistency Principle: *Similar things should look similar, different things different*

FORTRAN:

Computed GOTO:

```
GOTO (L1, …, Ln), I
```

Assign number to I
Use this value as an index in jump table

Assigned GOTO:

```
GOTO N, (L1, …, Ln)
```

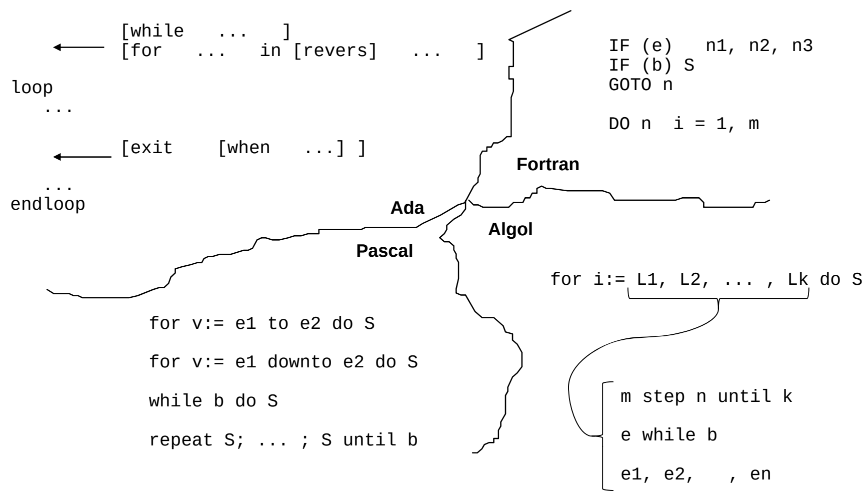
N address of a statement
Use the label in the list with this address

```
I = 3
  .
  .
  .
GOTO I, (20, 30, 40, 50)
```

ERROR

```
ASSIGN 20 TO N
  .
  .
  .
GOTO (20, 30, 40, 50), N
```

Control Structures (iteration)



Localized Cost Principle: *Users should pay only for what they use; avoid distributed costs*

Algol:

For-Loop:

Baroque
Binding time of loop parameters
No definite iteration

```
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
```

Structure Principle: *The static structure of the program should correspond in a simple way to the dynamic structure of the corresponding computations*

FORTRAN versus Algol:

```
        IF (.NOT. (condition)) GOTO 100
        statement;
           .
           .
           .
        statement
        GOTO 200
100
        statement;
           .
           .
           .
        statement
200     ...
```

```
if condition then
   begin
      statement;
         .
         .
         .
      statement
   end
else
   begin
      statement;
         .
         .
         .
      statement
   end
```

Labeling Principle: *Avoid arbitrary sequences more than a few items long. Do not require the user to know the absolute position in the list. Instead, associate a meaningful label with each item and allow the items to occur in any order*

Pascal:

case-Statement:
- Labels
- Multiple labels

```
case <> of
   <case clause>;
   <case clause>;
      .
      .
      .
   <case clause>
end
```

```
<constant>, <constant>, ... : <statement>
```

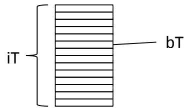
Expression & Assignment - Examples

```
V := V + 1

V1:= V2:= ... Vn:= E

V1, V2:= E1, E2

N += 1

( if B then i else j ):= E

i:= ...   ( if B then E1 else E2 )   ...
```

Pascal - Data Structures

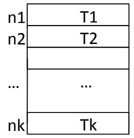
```
array [iT] of bT
```

index type and base type
multidimensional arrays
bT any type (also array type)



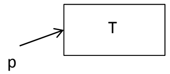

```
record n1: T1; ... ; nk: Tk end
```

heterogeneous data
Ti any type
named selector
variant records with tagfield




```
p = ^T
```

strong typing
T any other type



```
...   a.b[i][j]^.c.d^^   ...
```

Name Structures - Examples

FORTRAN:	Main program (global scope) Subprograms (local scope) COMMON blocks
Algol 60:	Block (global, nonlocal, local) Scope (static, (dynamic))
Pascal:	Record
Ada:	Data abstraction: Package Modularization (Module)

Regularity: *Regular rules, without exceptions, are easier to learn, use, describe, and implement*

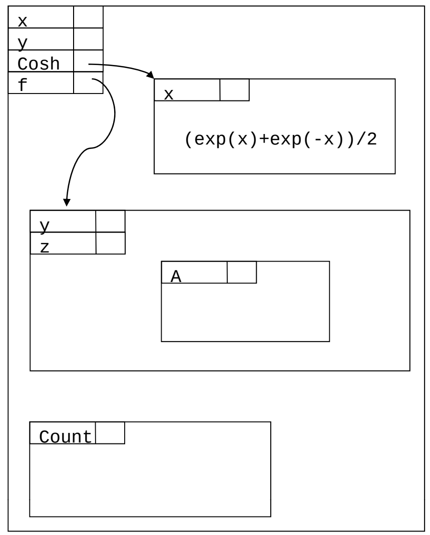
Algol: One statement & Compound statement (or block)

```
for i := 1 step 1 until N do
  begin
    if Data[i] > 1000000 then Data[i] := 1000000;
    sum := sum + Data[i]
    Print Real (sum)
  end


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

Algol - Block

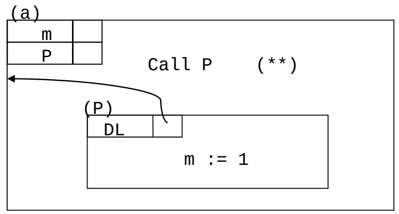
```
begin declarations; statements end
```

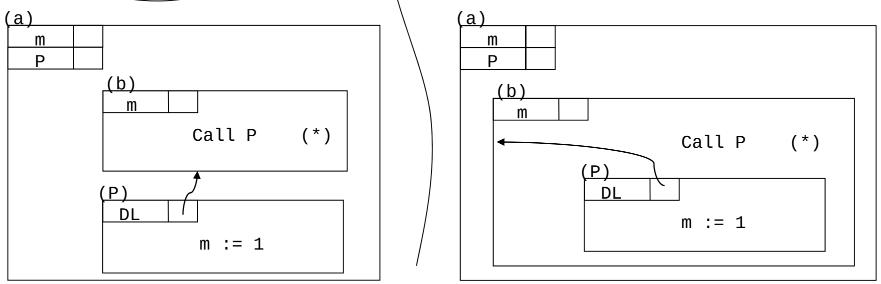
```
begin
  real x, y;

  real procedure cosh (x); real x;
    cosh := (exp(x) + exp(-x)) / 2;

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



Static and Dynamic Scoping

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





Ada - Package

generic ...
package ... is ...
[private ...]
end

package body ... is ...
[begin ... [exception ...]]
end

```
generic
   LENGTH: NATURAL:= 100;
   type ELEMENT is private
package STACK is
   procedure PUSH( X: in ELEMENT);
   ...
end Stack;


package body Stack is
   ST: array (1..Length) of Integer;
   Top: Integer range 00..Length := 0;

   procedure Push (X: in Integer) is
   begin
     ...
   end Push;
   ...
end Stack
```

```
package STACK2 is new STACK (200, INTEGER);
...
```

Information Hiding Principle: *The language should permit modules designed so that (1) the user has all the information needed to use the module correctly, and nothing more; and (2) the implementor has all the information needed to implement the module correctly, and nothing more*

ADA:

Parnas's Principles
ADA package

```
package … is
   … specification of public names …
end … ;


package body … is
   … implementation …
end … ;
```

Ada - Package

Libraries
(Shared Data Access)
Data Structures
Multiple Instantiation
Parameterization

```
package plot is
   ...
end Plot ;
```

```
package Stack is
   ...
 end Stack ;

 use Stack
 ...
 Push(I)
 ...
```

```
generic
package Stack is
   ...
end Stack;

package MyStack is new Stack;
package YourStack is new Stack;
```

```
generic
   length : Natural := 100;
 package Stack is
    ...
 end Stack;

 package MyStack is new Stack(100);
 package YourStack is new Stack(64);
```

Syntactic Consistency Principle: *Similar things should look similar, different things different*

Ada:

Block declaration:

```
declare
  <local declarations>
begin
  <statements>
exceptions
  <exception handlers>
end;
```

Procedure declaration:

```
procedure <name> (<formals>) is
  <local declarations>
begin
  <statements>
exceptions
  <exception handlers>
end;
```

Paradigms — Programming Languages

Procedural (imperative) programming

A program execution is regarded as a (partially ordered) sequence of operations manipulating data structures

Functional (applicative) programming

A program is regarded as a mathematical function describing a relation between input and output

Constraint-oriented (logic) programming

A program is regarded as a set of equations describing relations between input and output

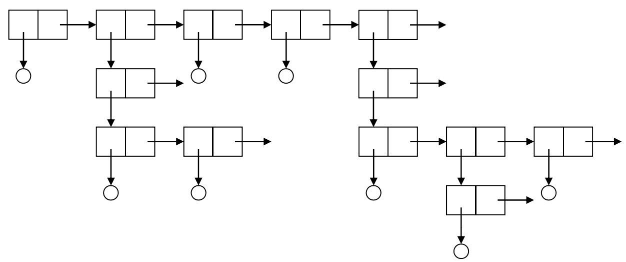
---

Object-oriented programming?

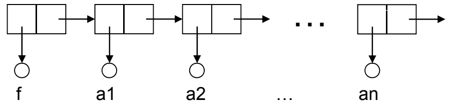
A program execution is regarded as a set of objects (and classes) responding to messages?

LISP (Functional)

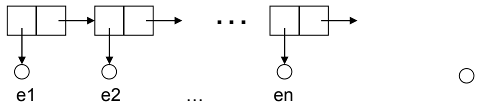
- Function application (Simplicity): (f a1 a2 … an)
- Datastructures—Data/Program (Simplicity): List (e1 e2 … en) ei is either a list or an atom
- Interactive interpreter (dynamic binding) (written in LISP: ~22 lines)



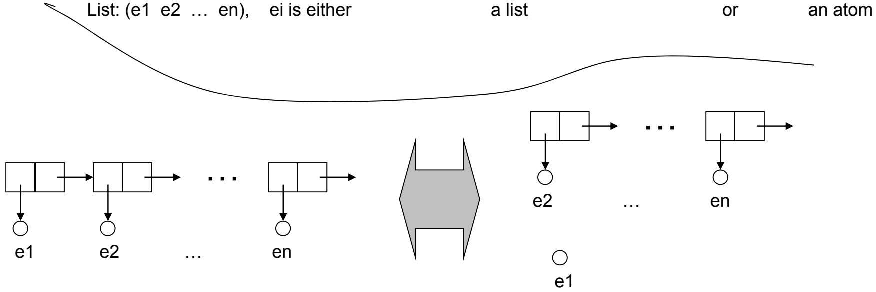
LIST



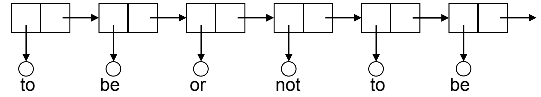
Function application: (f a1 a2 … an)

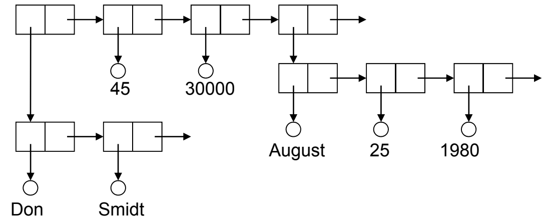


List: (e1 e2 … en), ei is either a list or an atom

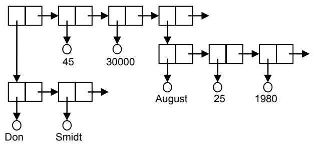


LIST





LISP



car and cdr

- (car L) first element of list L — e1
- (cdr L) list L, except first element — (e2 e3 .. en)

- (set 'DS '((Don Smidt) 45 30000 (August 25 1980))) — DS = (…)
- (car (cdr (car DS))) — Smidt
- (cadar DS) — Smidt

A quoted argument is not evaluated:

(set 'text '(to be or not to be))
(set 'text (to be or not to be)) — Try to evaluate function "to"

defun

- (defun f (n1, n2, … , ns) b)
- (defun hire-date (r) (cadddr r))
- (defun year (d) (caddr d))

- (hire-date DS) — (August 25 1980)
- (year (hire-date DS)) — 1980

LISP

cons — e0 and L = (e1 e2 … en) : (e0 e1 e2 … en)

- (cons (car L) (cdr L) = L
- (car (cons x L)) = x
- (cdr (cons x L)) = L

cond — if p1 then e1
elseif p2 then e2
…
elseif pn then en

- (cond (p1 e1) (p2 e2) … (pn en))

append — L = (e1 e2 … en) and M = (s1 s2 … sm):
(e1 e2 … en s1 s2 … sm)

- (defun append (L M)
  (cond
    ((null L) M)
    (t (cons (car L) (append (cdr L) M)))))

LISP

Functional

—either (or both) of

- One or more functions as arguments
- A function as its result

- (mapcar ‘add1 ‘(1 9 8 4)) (2 10 9 5)

- (defun mapcar (f x)
  (cond ((null x) nil)
  (t (cons (f (car x)) (mapcar f (cdr x))))))

Lamda expression ~ anonymous function

- (defun consval (x) (cons val x)) then (mapcar ‘consval L)
- (mapcar ‘(cons val x) L) ?

- (lampda (x) (cons val x)) anonymous consval function

- (mapcar ‘(lampda (x) (cons val x)) L)
- (mapcar ‘(lampda (n) (times n 2)) L)