Tezuch w SETL
Jackob Schwartz

196-48

6.12.72

Overall Impact of the Programming Problem

1. Cost (to users) of programming.
2. APAR cost.
3. Delay in implementing applications.
4. Unpredictability of large projects.

Difficulty of programming is the main obstacle to the application of computers.

An obvious technological imbalance:

We are rapidly approaching a situation in which 4th generation hardware will be available -- but programming techniques are only 2nd generation.

Conclusions

Substantial improvements in all these respects are attainable

4'. Improved project predictability
2'. Debugging methods reducing APAR fix cost
1',3'. Reduced programming costs
(Higher level language systems;
hardware support to be worked out).

Response to this problem:

'Modularization'

-- combat all pressures which lead to interrelatedness of elements.

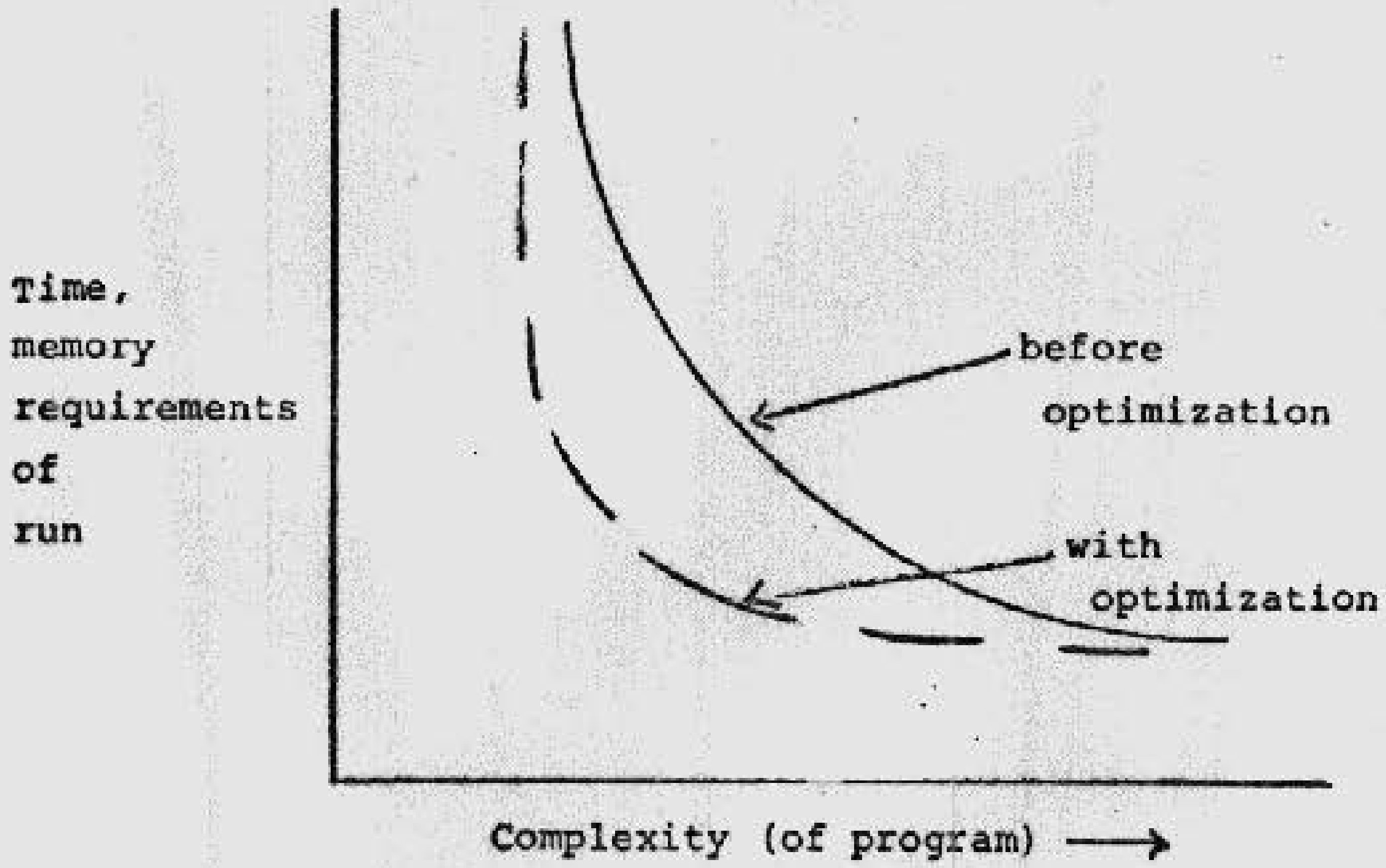
Use small number of powerful elements governed by uniform simple conventions.

'Language of maximum expressivity'.

However:

Systematic modularization leads to diminished efficiency.

Expressivity-efficiency tradeoff.



Sources of difficulty in programming

Programming is a construction process

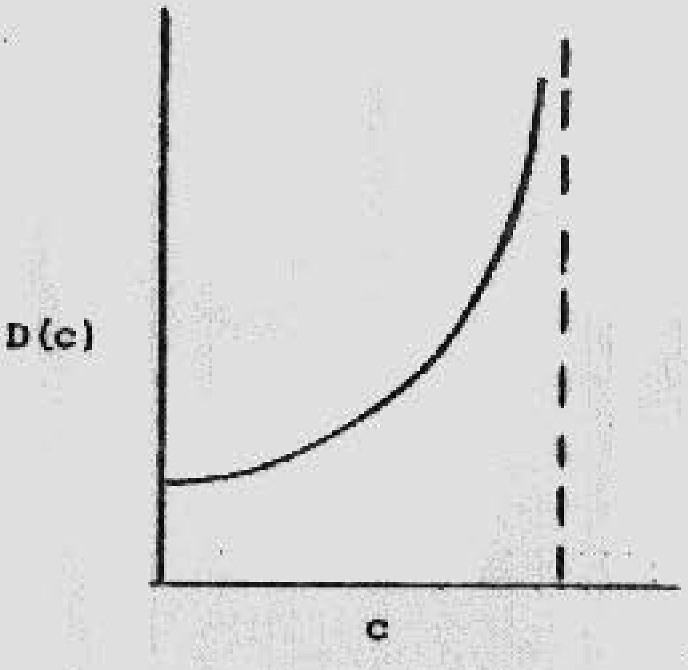
Elements $E_1, \ldots, E_n$ successively chosen

Local context of element E_j:

all aspects of other E's affecting choice of E_j.

'Complexity' of local context of E_j --- $c(E_j)$.

Difficulty rises steeply with $c(E_j)$.



Total difficulty of completing program

$$DTOT = D(c(E_1)) + \ldots + D(c(E_n))$$

Ineffficient but highly expressive language can be useful as prototyping tool

Two-stage programming technique.

Stage 1: Develop, debug algorithm using highly expressive algorithm-oriented language

Stage 2: Transcribe algorithm to production language, using higher level version as 'development matrix'.

Advantages

Function known in advance.

Able to test function adequacy.

Customer exposure to function.

Design known to be consistent.

Greatly improved implementation predictability.

B. <u>Uses in University or Laboratory Environment</u>

Non-production experimental programs

  Algorithm modeling and measurement

Programs used for bootstrapping

Documentation of algorithms for instruction.

<u>Possible Mode of Application</u>.

The 'programming test stand':

50 million inst / sec micromachine with
16 million memory bytes
would appear as
computer of 7094 class with 1 million bytes of storage
  on which
  <u>programming was speeded up by factor of</u> 10

Successful data structure elaborations would
give practical programming tool for commercial
  programming range.

Sources of Amodularity and Responses

Problem 1: 'All at once' design of function, logical structure, efficient encoding;

Response: 'decision postponement':

break development into orderly stages:

solve initial parts of problem without foreclosing possible approaches to remaining parts.

postpone choice of encoding until logical structure is worked out.

Problem 2: Common relationship of many processes to a smaller number of data structures.

Rigidity and specificity of code which reflects data structure details.

Response: Use logically powerful family of default data structures which enable many others to be modeled..

Develop declaratory approach to details of data structuring.

Allow functional treatment of storage sequences corresponding to presently available functional treatment of access sequences.

Allow declaratory specification of a variety of data objects to which operators apply in an object-dependent manner.

Problem 3: Present techniques require code to be written in order of eventual execution, rather than in logically most transparent arrangement.

Response: Break with linear coding style, and allow:
- 'footnoted style'
- 'remote code' dictions

Study 'whenever' dictions
- 'non-deterministic branch' dictions

Problem 4: Repetition of detail, with obligatory small variations, because of language-problem mismatch.

Response: Develop extension mechanisms, especially to allow:
- extensions of semantic object classes
- available declarations
- global, rather than merely local, transformations of source text

Develop mechanisms for reference resolution

Problem 5: Insufficiency of presently available debugging tools.

Response: Develop disciplined approach to statement of programmer assumptions.

Develop program-event oriented debugging language

Use high-level language to debug lower-level production programs.

Attainable tradeoffs:

FORTRAN - PL/1 Standard

Data expansion	1/1
Execution slowdown	1/1
Programming effort	1/1

High-Level Algorithm Oriented Language

A. Without 'data strategy' elaborations or hardware enhancement

Data expansion	8/1
Execution slowdown	30/1
Programming speedup	10/1

B. With 'data strategy' elaborations, but no hardware enhancement

Data expansion	1/1
Execution slowdown	5/1
Programming speedup	5/1

C. With elaborations and hardware enhancement

Data expansion	1/1
Execution slowdown	1.5/1
Programming speedup	5/1

Basic objects: *Sets* and *atoms*. Sets may have atoms or sets as members.

Atoms may be: Integer, real, bitstring, charstring, label, subroutine, function

or: Blank. *newat* is blank atom creator.
Special undefined Ω

All standard operations provided for atoms

Operations for sets. {x}, {x,y,z}, etc.

x ∈ a   *nℓ*   ∋a   #a
a *eq* b   a *ne* b   a *incs* b , etc.
a *u* b   a *u* {x} ≡ a *with* x
a - {x} ≡ a *less* x
pow(a)

Tuples: <x,y,z> $<x_1,\ldots,x_n>$
t(i)   *hd* t ≡ t(1)   *tℓ* t .
tuple x   *pair* x

Set former: {x ∈ a | C(x)}
{e(x), x ∈ a | C(x)}
{e(x,y), x ∈ a, y ∈ b(x) | C(x,y)}, etc.
{e(n), m ≤ n ≤ mm | C(n)}

Functional application:

f{x} = {*tℓ* x, x ∈ f | *pair* x}
f(x) = if #f{x} *eq* 1 then ∋f{x} else Ω

Compound operator:

```
[op: x ∈ a] e(x)
```

Example: [+: x ∈ a] e(x) ≡ $\sum_{x \in a} e(x)$

Quantifiers:

```
 ∃x ∈ a | C(x)              m ≤  ∃n ≤ mm | C(n)
∃[x] ∈ a | C(x)             m ≤ ∃[n] ≤ mm | C(n)
 ∀x ∈ a | C(x)
```

Algol 60 conditional expressions.

Statement forms: statements punctuated with semicolons.

```
  a = expn;                  <a,b> = expn;
f(a) = expn;
```

means: remove all tuples with first component a from set f; then re-insert <a,expn>

```
Algol 60   if-then-else
           go to <label>;
```

iteration headers:

```
      (while<cond>) <block>;
 or   (while<cond>) <block> end while;
      (∀x  ∈ a | C(x)) <block>;
      (m ≤ ∀n ≤ mm| C(n)) <block>;
      (mm≥ ∀n ≥ m | C(n)) <block>;
      quit ∀x;    continue ∀x;
```

counting sort:

```
 place = nℓ;  (∀x ∈ set) place(x) = #{y ∈ set | f(y) ℓe f(x)};
                                   end ∀x;
```

Huffman encode:

```
   huffcode = [+: 1 ≤ n ≤ #cstring] hufc(cstring(n));
```

Huffman decode:

```
   dehuf = nulc;  node = top;
   (1 ≤ ∀n ≤ #bstring)
   newnode = if bstring(n) eq 1 then ℓ(node) else r(node);
     if newnode eq Ω then
       dehuf = dehuf + node;  node = top;
     else node = newnode;  end if;  end ∀n;
```

Huffman tree:

```
   work=chars;  wfreq=freq;  ℓ=nℓ, r=nℓ;
   (while #work gt 1)
   c1 = getmin work;  c2 = getmin work;
   nd = newat;  ℓ(nd) = c1;  r(nd) = c2;
   wfreq(nd) = wfreq(c1) + wfreq(c2);
   work = work with nd;
   end while;
   top = ∋work;
   definef getmin set;  external wfreq;
   minfreq = [min: x ∈ set] wfreq(x);
   xmin = ∋{x ∈ set | wfreq(x) eq minfreq};
   set = set less xmin;  return xmin;  end getmin;
```

196-60

TOPOLOGICAL SORT

Problem

Suppose we are given a set S of arbitrary objects together with a partial ordering P on S. Suppose P is given as a set of pairs <a, b> with a, b ε S.

Arrange the members of S into a tuple T such that if a = T(i) and b = T(j), and <a, b> ε P (meaning a ≤ b), then i ≤ j.

Solution

1. We select an arbitrary member x of S which has no predecessor, and append that to T (T is initially null).
2. Having successfully placed x in T, we delete x from S and also delete all pairs beginning with x from P (if any exist).
3. We continue this process until S is null.

SETL Code

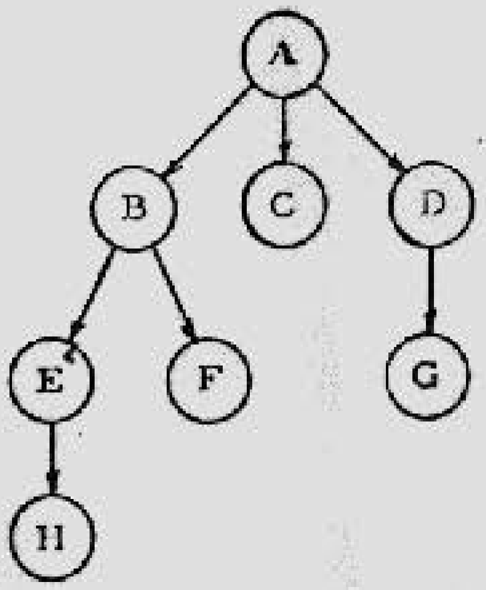
```
T = nult;
(while S ne nl)
   x = ∋{y ε S | not(∃pair ε P | pair(2) eq y)};
   T(#T+1) = x;
   S = S less x;
   P = P - {pair ε P | pair(1) = x};
   end while;
```

An ordered tree is a descendent function desc(node, j) defined for j in some finite (possibly null) range.

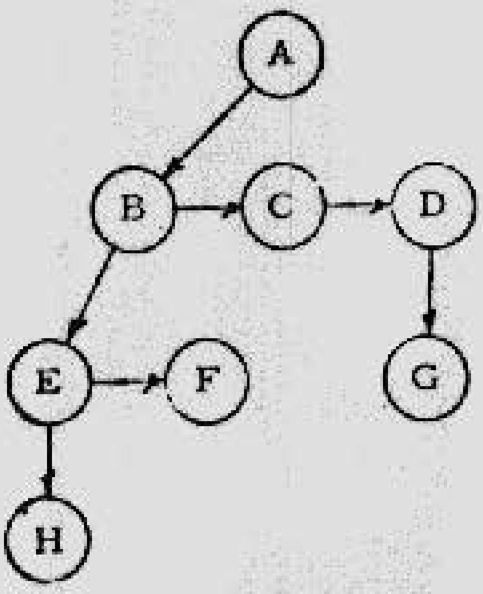
A binary tree is a pair of descendent functions L and R (left and right descendents).

The ordered and binary trees stand in an interesting 1-1 relationship that is illustrated below.

Ordered Tree



Binary Tree



Descendent Function

A	1	B
A	2	C
A	3	D
B	1	E
B	2	F
D	1	G
E	1	H

Descendent Functions

L		R	
A	B	B	C
B	E	C	D
D	G	E	F
E	H		

196-62

Ordered To Binary Tree Transformation

```
define OTB(desc, L, R);
L = {<x(1), x(3)>, x ε desc | x(2) eq 1};
R = {<x(3), y(3)>, x ε desc, y ε desc | x(1) eq y(1) and
                                        (x(2) + 1) eq y(2)};

return;
end OTB;
```

```
OTB:      PROCEDURE(DESC, L, #L, R, #R);
          DECLARE 1 DESC(*),
                      2 DESC1  CHAR(50) VARYING,
                      2 DESC2  FIXED  BINARY,
                      2 DESC3  CHAR(50) VARYING;
          DECLARE 1  L(*) CONTROLLED,
                      2 (L1, L2) CHAR(50)  VARYING;
          DECLARE 1  R(*) CONTROLLED,
                      2 (R1, R2) CHAR(50)  VARYING;
          DECLARE (#L, #R) FIXED BINARY;

BEGIN:    ALLOCATE  L(DIM(DESC1, 1));  L1, L2 = '';   #L = 0;
          ALLOCATE  R(DIM(DESC1, 1));  R1, R2 = '';   #R = 0;
          DO I = 1 TO DIM(DESC1, 1);
             IF DESC2(I) = 1  THEN DO;  #L = #L + 1;
                                        L1(#L) = DESC1(I);
                                        L2(#L) = DESC3(I);
                                        END;
             DO  J = 1  TO  DIM(DESC1, 1);
                IF DESC1(J) = DESC1(I)   &   DESC2(J) = DESC2(I) + 1
                             THEN DO;   #R = #R + 1;
                                        R1(#R) = DESC3(I);
                                        R2(#R) = DESC3(J);
                                        GO TO BUMP_I;
                                        END;

                END  /* DO J */;
BUMP_I;      END  /* DO I */;
          END OTB;
```