A PROGRAMMING AND PROBLEM-SOLVING SEMINAR

by

Michael J. Clancy and Donald E. Knuth

STAN-CS-77-606
APRIL 1977

COMPUTER SC IENCE DEPARTMENT
School of Humanities and Sciences
STANFORD UNIVERSITY

A Programming and Problem-Solving Seminar

by Michael **J. Clancy and Donald E. Knuth**

Abstract:

This report contains edited transcripts of the discussions held in Stanford's course CS 204, Problem Seminar, during autumn quarter 19%. Since the topics span a large range of ideas in computer science, and since most of the important research paradigms and programming paradigms came up during the discussions, the notes may be of use to graduate students of computer science at other universities, as welt as to their professors and to professional people in the "real world".

The **preparation of this report was supported in part by IBM Corporation.**

Contents

Note: No notes were compiled for November 23, when generation of random trees was discussed, among other random topics related to problem 4.

Preface

During the autumn quarter of 1976, I had the privilege of teaching Stanford's seminar CS 204, which brings together most of our incoming graduate students during their first quarter here. It was a great pleasure to work with such a talented group of students, and it was an even greater pleasure to have Mike Clancy as my teaching assistant, since he prepared such beautiful summaries of what transpired during our classroom discussions. After reviewing his notes, it struck me that they would probably be instructive reading for a much wider audience; hence we have prepared this booklet.

The purpose of CS 204 is to teach skills needed for research in computer science, as well *as* programming skills. I assigned five programming problems drawn from different areas of computer science; none of these problems was easy, but they were all intended to be sufficiently enjoyable that the students would not mind working overtime. The programs were to be done in Sail, a local language in use on our PDP-10 computers; readers unfamiliar with Sail should still be able to understand most of what went on.

The five problems were based on:

1. map drawing (related to numerical analysis and computer graphics);
2. natural-language numbers (related to linguistics, interaction, and error recovery);
3. list representation (related to data structures and algorithmic analysis);
4. network design (related to program structures, list processing, and combinatorial optimization);
5. code generation (related to machine architecture and heuristic search techniques).

Incidentally, problem 3 was a research problem that I didn't know how to solve when I assigned it, although I had an idea that a solution would be possible. This added a touch of suspense as welt as an air of authenticity to our discussions of how to do research.

I handed out all five assignments at the first class meeting, so that everybody would know what they were in for, and so that the back of their minds could be working on problem 5 (say) while they were mostly thinking about problem 1. There also was a "warm-up" problem assignment about program control structures, so that we would have something to talk about during the second class meeting.

The reader can best put himself into the students' shoes by beginning the course as they did, reading the problem assignments first. Therefore all the problem statements appear immediately after this preface, followed by the notes Mike has prepared based on the class discussions. Mike has also interposed useful comments about the solutions which the students actually turned in. I hope these notes give pleasure to many people who study them.

D. E. Knuth
Stanford, California
March, 1977

Problem 0 (warmup problem, to be discussed October 6)

The original Algol 60 report concluded with the example procedure RK shown on the following page. After fifteen years have gone by, we think we know a little more about control structure and style.

Suppose Algol 60 has been extended to include the following new syntax

```
<basic statement> ::= <loop statement>
<loop statement>  ::= loop <statement list> while <Boolean expression>:
                          <statement list> repeat
<statement list>  ::= <statement> | <statement list>; <statement>
```

and the semantics that, if Γ_1 and Γ_2 are statement lists and B is a Boolean expression, the loop statement

loop Γ_1 while B: Γ_2 repeat

is equivalent to

```
Γ1;
if B then
    begin Γ2;
        loop Γ1 while B: Γ2 repeat
    end
```

Rewrite the RK procedure, using the new loop statement, in order to eliminate unnecessary go to statements which tend to obscure the control structure of the program in its original form.

Important note: Large segments of the program have been marked α, β, γ, etc. so that you need not recopy them; simply give your answer in terms of these abbreviations, as "procedure RK α" etc. You need not understand precisely what goes on inside those chunks of code in order to solve this problem. Don't make any changes to the actual sequence of computations, and don't try to do any tricky things like eliminating the Boolean variable *out* etc. Simply remove as many of the go to statements and meaningless labels as you can by taking advantage of the new loop construct.

```
procedure RK(x, y, n, FKT, eps, eta, xE, yE, fi); value x, y; integer n;
Boolean fi; real x, eps, eta, xE; array y, yE; procedure FKT;
comment: RK integrates the system y'k = fk(x, y1, y2, ..... yn) (k = 1, 4, ... n)
of differential equations with the method of Runge-Kutta with automatic search for
appropriate Length of integration step. Parameters are: The initial values x and y[k]
for x and the unknown functions yk (x). The order n of the system. The procedure
FKT(x, y, n, z) which represents the system to be integrated, i.e. the set of functions fk.
The tolerance vdues eps and eta which govern the accuracy of the numerical integra-
lion. The end of tk integration interval xE. The output parameter yE which re-
presents the solution at x = xE. The Boolean variable fi, which must always be
given the value true for an isolated or first entry into RK. If however the functions y
must be available at several meshpoints x0, x1, ...., xn, then the procedure must be
called repeatedly (with x = xk, xE = xk+1, for k = 0, 1, ..., n — 1) and then the later
calls may occur with fi = false which saves computing time. The input parameters
of FK T must be x, y, n, the output parameter z represents the set of derivatives z[k] =
fk(x, y[1], y[2], ..., y[n]) for x and the actual y's. A procedure comp enters us a
non-local identifier;
begin
    array z, yl, y2, y3[1:n]; real xl, x2, x3, H; Boolean out;
    integer k, j; own real s, Hs;
    procedure RK1ST(x, y, h, xe, ye); real x, h, xe; array y, ye;
        comment: RK1ST integrates one single Runge-Kutta step -with initial
        values x, y[k] which yields the output parameters xe = x + h and ye[k],
        the latter being the solution at xe.
        Important: the parameters n, FKT, z enter RK1ST as non-local entities;
        begin
            :
            : (omitted)
            :
        RK1ST;
Begin of program:
    if fi then begin H := xE — x; s := 0 end else H := Hs;
    out := false;
AA: if (x + 2.01 × H — xE > 0) ≡ (H > 0) then
    begin Hs := H; out := true; H := (xE — x)/2 end if;
    RK1ST(x, y, 2 × H, x1, y1);
BB: RK1ST(x, y, H, x2, y2); RK1ST(x2, y2, H, x3, ye?);
    for k := 1 step 1 untii n do
        if comp(y1[k], y3[k], eta) > eps then go to CC;
    comment: comp(a, b, c) is a function designator, the value of which is the
    absolute value of the difference of the mantissae of a and b, after the exponents
    of these quantities have been made equal to the largest of the exponents of the
    originally given parameters a, b, c;
    x := x3; if out then go to DD;
    for k := 1 step 1 until n do y[k] := y3[k];
    if s = 5 then begin s := 0; H := 2 × H end if;
    s := s + 1; go to AA;
CC: H := 0.5 × H; out := false; x1 := x2;
    for k := 1 step 1 until n do y1[k] := y2[k];
    go to BB;
DD: for k := 1 stop 1 until n do yE[k] := y3[k]
end RK
```

α

β

γ

δ

ε

ζ

η

Problem 1, due **October** 14, **1976**

The column "Mathematical Games" by Martin Gardner in the November, 1975 issue of *Scientific American* describes several classical projection techniques for making maps of the world.

The file WORLD.MAP[204,MJC]–see Appendix I-contains a sequence of ordered pairs $(x_1,y_1), \ldots, (x_n,y_n)$ of integer numbers, specifying latitude and longitude values of points on the globe, where $-90 \le x_i \le 90$ and $-180 \le y_i \le 180$. This list is such that you get a fair approximation to our earth's major land/sea boundaries if you draw polygons according to the following procedure:

```
3 ← 1;
loop while j < n:
    i ← j+1;
    loop
        Draw line from (x_{i-1},y_{i-1}) to (x_i,y_i);
    while (x_i,y_i) ≠ (x_j,y_j):
        i ← i+1;
    repeat;
    3 ← i;
repeat;
```

In other words, go from (x_1,y_1) to (x_2,y_2) to (x_3,y_3) etc. until getting back to the starting point, then lift the pen and start the same process at the next point, etc. Write a program or programs to draw the following maps:

(a) stereographic projections of the Northern hemisphere and Southern hemisphere;
(b) Mercator projection, but using two points along the equator at 0° and 180° longitude in place of the North and South Poles;
(c) two-point equidistant projection based on the two points New York (41,-74) and London (51,0);
(d) another world map of your choice.

The Sail manual tells how you can read the (x_i,y_i) pairs (e.g. with the INTIN function), To draw the maps, it Is suggested that you use system routines which display on a Data Disc screen (while debugging), converting to the similar routines that output on the XGP (for the final run). All these routines are explained in a separate handout.

Problem 2, due October 28, 1976

Write an interactive Sail program that says "Give me a number:", whereupon the user types the English name of a positive integer less than one billion followed by a carriage return, and the program says "Your number is n." where n is the decimal representation of the same number. Then comes "Give me a number:" etc. (If the given number is ungrammatical or out of the specified range, the program should of course make some other appropriate response.)

Grammatical examples:

twenty-four
one hundred [and] eighty-three
nineteen
six hundred million two hundred [and] one
eighty-two million seven hundred [and] thirty-three thousand
nineteen hundred [and] seventy-six

Ungrammatical examples:

twenty four
four and twenty
twenty-ten
six hundred hundred
one half
fourty
double-O seven

Your documentation should include a rigorous description of the syntax and semantics of what your program considers to be grammatical.

Problem 3, due November 11, 1B76

Design a data structure for storing a list of items, with operations of
(a) accessing the kth item,
(b) inserting a given item Just before the kth item,
(c) deleting the kth item,
given k. All three operations should take only O(log k) steps, regardless of the kngth of the list.

Give "clean" algorithms for these three operations, in an Algol-like language. . .

Problem 4, due November 23, 1976

(This problem is relevant to the design of networks in which remote sites are connected to a central node; most computer reservation systems, banking systems, etc. seem to be of this type. The same ideas have also been used in the design of offshore pipeline networks for natural gas, etc.) Consider a network in which n+1 points $v_1, v_2, \ldots, v_n$ are connected by n links $e_1, \ldots, e_n$. (Thus it is a free tree, and there is exactly one simple path from v_i to v_j for any given i and j.) Let e_i join $v_{a(i)}$ to $v_{b(i)}$.

There are finitely many ways to construct each link physically, depending on how much we are willing to pay to get a certain capacity for service; in other words, to get sufficiently reliable transmission, several devices are available but it costs more to transmit more bits per second. In practice we can use an estimate of the traffic across each link to convert transmission rates into delay times, so we can represent the possible ways to build each link as a delay *vs.* cost *tradeoff list* L; this is a list of the form

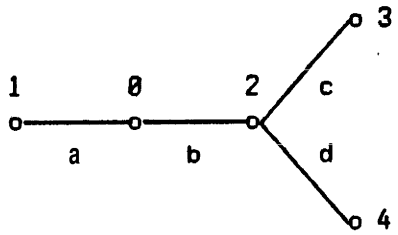
$$L = (d_1,c_1), (d_2,c_2), \ldots, (d_k,c_k)$$

where $d_1 > d_2 > \ldots > d_k$ and $c_1 < c_2 < \ldots < c_k$. Here (d_j,c_j) means that the Jth option for implementing the link has a cost of c_j but introduces a delay of d_j.

The tradeoff lists for different links will in general be different (because of differences in distance or traffic, or because telephone lines are more expensive in certain areas); all of these awkward nonlinear possibilities can be modelled by simply assuming that a tradeoff list L_i is given for each link e_i in the network.

The problem is to design a minimum cost implementation of the given network subject to the condition that the delay time from v_0 to v_j is at most a given value D, for all j. (The delay time from v_0 to v_j is the sum of the delays along all links in the path from v_0 to v_j; thus we assume zero "forwarding" delay at the intermediate stations.)

For example, consider the network



With

L_a = (150,6), (139,9), (118,14), (87,21), (75,30);
L_b = (67,6), (62,10), (52,15), (39,23);
L_c = (120,13), (111,17), (92,23), (66,29);
L_d = (94,8), (86,12), (80,18), (61,26).

It turns out that the best implementations corresponding to this data can be described by the tradeoff list

L = (187,33), (178,37), (173,41), (172,42), (161,43), (156,47), (154,51), (150,52), (146,55), (144,59), (139,63), (133,68), (131,72), (125,78), (118,84), (105,99).

For D ≥ 187 there is an implementation of cost 33; for 187 > D ≥ 178 the best is one of cost 37; and so on. In particular, the best implementation for D = 145 has cost 59 and can be obtained by choosing (139,9) for a, (52,15) for b, (92,23) for c, and (86,12) for d. This can be found by hand, using a method that is much better than simply trying all 5•4•4•4 = 320 possibilities!

Write a Sail program for this problem, testing it on the above sample data. Then run it on the larger data set in TREE.NET[204,MJC] (see Appendix 2), using the statement counter system described in Appendix F of the Sail manual. .

Problem 6, due December 9, 1976

The purpose of this problem is to obtain a general code-generation procedure for simple assignment statements on a wide variety of machine architectures. You should write a program that inputs an assignment statement and a machine description (in any format that is convenient for you to use), and which outputs an optimum or near-optimum program to perform the given statement on the given machine. (Here "optimum" means "fastest".) In order to keep this task within manageable size, you may assume that the assignment statement is--formed from distinct unsubscripted variables (so that there are no common subexpressions or constants to worry about); it has the form $v \leftarrow E$ where expression E uses only the binary operations +, -, •, and /. The associative law for addition and multiplication should not be used to rearrange the form of the expression; however, the commutative law and laws of signs should be exploited when possible. The machine description need only be general enough to handle the two historical examples below and others of their ilk. Actually it suffices for you to write two programs, one for each of the following two machines. (In class we will try to discuss the removal of these restrictions.)

Example 1 (based on IBM 704-709-7090-7094 series).
There are two registers A and Q; the instructions are as follows, where M denotes a memory location:

mnemonic(s)	operation	execution time
STO, STQ	M ← A, M ← Q	4.36 μs
CLA, CLS	A ← M, A ← -M	4.36 μs
LDQ	Q ← M	4.36 μs
FAD,FSB	A ← A+M, A ← A-M	13.95 μs
FMP	A ← Q•M and Q ← lower half	25.29 μs
FDP	Q ← A/M and A ← remainder	28.34 μs
XCA	A ↔ Q	2.18 μs
CHS	A ← -A	4.36 μs

Example 2 (suggested by CDC 6000).
There are three registers X1, X2, and X3; the instructions are as follows:

instructions	
X1 ← M, x2 ← M	0.5 μs
xi ← xj, xi ← -Xj (1 ≤ i,j ≤ 3)	0.3 μs
XI ← Xj±Xk (1 ≤ i,j,k ≤ 3)	0.4 μs
Xi ← Xj•Xk (1 ≤ i,j,k ≤ 3)	0.8 μs
Xi ← Xj/Xk (1 ≤ i,j,k ≤ 3)	0.9 μs
M ← X3	0.7 μs

Test your program on the following example statements:

```
a ← b-c•d
a ← b/c + d•e
a ← (b/c+d)/e
a ← (b-c•d-e•f)/(g-h/I-j/k)
a ← b - (c-d/(e•f+g•h))/((i-j)•(k+1)+m/n)
```

So as not to waste computer time, it would be best if your program has a running time linearly proportional (say) to the length of the given assignment statement, for any fixed machine description.

[For this problem we also referred students to the article "Optimal Code Generation for Expression Trees" by Aho and Johnson in *JACM* (July 1976), pp. 488-501, and distributed a preprint of the article "A Generalization of Dijkstra's Algorithm," by D.E. Knuth, *Information Processing Letters* (1977), to appear.]

Notes for October 6, 1976

Topics of today's discussion:

1. solution of problem A (removing gotos from a procedure for Runge-Kutta integration);
2. mechanical *goto* elimination;
3. good programming style (and why problem A is a poor problem);
4. features of programming languages which could make good programming easier.

Several solutions were proposed to remove gotos from the Algol procedure RK in problem *A* (listed in abbreviated form below).

```
procedure RK a; β
AA:     γ;
BB:     RK1ST(X,Y,H,X2,Y2);  RK1ST(X2,Y2,H,X3,Y3);
        for K:=1 step 1 until N do
                If COMP(Y1[K],Y3[K],ETA) > EPS then go to CC;
        δ;
        X:=X3;  If OUT then go to DD;
        ε; g o to AA;
CC:     ζ; go to B B ;
DD:     9
end RK;
```

Figure 1 – original program

One approach to solving the problem started with noticing that the code in section ζ is self-contained, since the statement preceding CC is a *goto*. Hence, without disturbing the sequence of operations in RK, ζ may be moved-for example, to replace the *go* to CC in the *for* statement following BB, since that *goto* is the only statement referencing CC. The code starting with BB and ending just before δ may then be turned into a *loop...while...repeat* loop. Subsequently the entire procedure may be turned into another *loop...while...repeat* loop to remove all remaining *gotos*. The result is listed below.

```
procedure RK a; β
    loop
            γ;
            loop
                    RK1ST(X,Y,H,X2,Y2);  RK1ST(X2,Y2,H,X3,Y3);  K:=0;
                    loop
                            K:=K+1;
                    while (K ≤ N) and (COMP(Y1[K],Y3[K],ETA) ≤ EPS): δ;
                    repeat;
            while K ≤ N: ζ;
            repeat;
            X:=X3;
    while not OUT: ε;
    repeat;
    9
    end RK;
```

Figure 2 – goto-less version

The top-down ("outside-in") approach reverses the steps of the previous solution. It involves first discovering the outer loop and where it exits, then noting the independence of the CC code, resulting in the following:

```
procedure RK a; ß
    loop
            ϒ;
            loop
                    RK1ST(X,Y,H,X2,Y2);  RK1ST(X2,Y2,H,X3,Y3);  K:=0;
            while {for some K, COMP(Y1[K],Y3[K],ETA)≤ EPS}: 8; ƒ
            repeat;
            X:=X3;
    while not OUT: ε;
    repeat;
    9
    end RK;
```

Figure 3 – partially abstracted

The inner code may be replaced by a loop, resulting in the procedure of Figure 2, or by a Boolean procedure. The latter is probably the cleanest solution, even though it requires the extra overhead of the procedure calls.

A note about the procedure's function: RK is a procedure for integration with automatic search for the appropriate integration step size. The transfer to CC results from finding approximations (to function values) which are too far apart-hence the step size for computing the approximations is decreased to get more accuracy. It would be nice to have an instruction implemented in the hardware which took the place of the COMP predicate and determined if two floating-point numbers were suitably close together. Apparently APL has such a feature.

Many people have studied the question of mechanical *goto* elimination. *Gotos* can always be replaced by composition, iteration, or conditional operations. In general, however, auxiliary variables and extra computation are introduced in the elimination process; there exist programs which grow exponentially in space required when gotos are removed (see [2,4]).

Automatic *goto*-eliminating procedures, applied to a badly-structured program, cannot be expected to improve the structure. For this reason problem A was a bad problem, in that one's goal should be to write a well-structured program in the first place, rather than to write a poorly structured program and then patch it up. The top-down solution to problem A essentially took the program apart and rewrote it, as if starting from scratch; for that reason the top-down approach was better. (Since we got a chance to discuss this, it wasn't such a bad problem after all. Anyway, one of the paradigms of research is to criticize or change a problem.)

We have noted that use of a Boolean procedure would be a good way to implement the BB-CC loop. Examples are listed below. In the first example, the lack of advanced control structures in pure Algol 60 forces the use of a *goto*. The second example is written in Sail. The third might have been written by a Lisp programmer.

```
Boolean procedure CLOSE(Y,YP,N,ETA,EPS); array Y,YN; Integer N; real ETA,EPS;
        begin integer K;
        CLOSE:=false;
        for K:=1 step 1 until N do
                If COMP(Y[K],YP[K],ETA)> EPS then go to DONE;
        CLOSE:=true;
DONE :  end CLOSE ;

Boolean procedure CLOSE(real array Y,YP; integer N; real ETA,EPS);
        begin   integer   K;
        for K:=1 step 1 until N do
                If COMP(Y[K],YP[K],ETA)> EPS then return(false);
        return(true)
        end;

Boolean procedure CLOSE(real array Y,YP; integer K,N; real ETA,EPS);
        if K > N then true
        else if COMP(Y[K],YP[K],ETA)> EPS then false
        else CLOSE(Y,YP,K+1,N,ETA,EPS);
```

Figure 4 – versions of the Boolean procedure CLOSE

The program of Figure 2 can be "optimized" by replacing "(K≤N)" in the first *while* by "(K(N)" and then changing the second *while* test to "COMP(Y1[K],Y3[K],ETA)>EPS". If the loop doesn't terminate with K≥N this avoids a redundant test. However, this optimization makes the program less robust, as it no longer works when N-O. (One rarely does Runge-Kutta integration in zero-dimensional space, but the point is that optimizations often do make programs behave differently in unexpected ways.)

Gotos, of course, aren't all bad, and Knuth discusses this at length in his paper C1]. For example, if arrival at some labelled statement has an exact meaning, then a transfer to that statement via a *goto* can often be justified. Usually this isn't true of *gotos* derived from flowcharts, and these are the *gotos* that should be replaced by other control structures. Overall clarity of the program's logic is the main concern; just as a variable should have a meaningful name, a label should have a clear purpose too, and if it does, then its use is quite reasonable.

If a nicer alternative to a *goto* exists, though, it should be used. What might alternative control structures look like? Well, many languages contain a *while* or *until* (= *while* not) statement. Others (Bliss, Sail) have facilities for exiting loops in a controlled way. The language Alphard [5] being developed at Carnegie-Mellon has a "first" construct for taking some action after the first occurrence of an event in a loop. APL has aggregate operations which remove the need for many loops. Knuth in [1] describes two other useful control structures: Dahl's *loop...while... repeat* construct, used in problem A; and Zahn's event construct (later renamed the "situation" construct-see [3]), for which a list of events which terminate the loop is specified, somewhere in the loop each event is signaled, and processing after the loop's termination depends on the event which caused exit).

Evaluating the "niceness" of all these is difficult, since not much is known about the psychology of computer programming. Do people really write worse programs, or find it harder to program, in Fortran than Algol? Is a construct which puts the termination criteria in the iteration condition (as in loop.. *.while...repeat*) better than one which has a trivial iteration condition–*while* true do-and exits via event signalling? One might decide that having the word repeat delimit a loop makes a program more readable, but how would he (or she) prove it? There are many such unanswered questions, even at the most elementary levels.

References:

[1] Knuth, D.E., "Structured Program with 'go to' Statements," *Computing Surveys 6,4* (Dec 74), pp. 260-301.

[2] Knuth, D.E. and Floyd, R.W., "Notes on avoiding 'go to' statements," *Information Processing Letters 1* (1971), pp. 23-31, 177.

[3] Knuth, D.E. and Zahn, C.T., letter to editor, *CACM 18,6* (June 1975), p. 360.

[4] Lipton, R.J., Eisenstat, S.C., and Demillo, R.A., "The Complexity of Control Structures and Data Structures," Yale Univ. C.S. Dept. Res. Rpt. 04 1, 1975.

[5] Shaw, M., Wulf, W .A., and London, R.L., "Abstraction and Verification in Alphard: Iteration and Generators," Carnegie-Mellon Univ. C.S. Dept. and USC Information Sciences Institute Res. Rpt. (1976).

Notes for October 7, 1976

Topics of today's discussion:
1. stereographic projections;
2. representation of data;
3. conformal projections;
4. Mercator projections;
5. modes of research;
6. equidistant projections.

Problem 1 involves the programming of various projection techniques for making maps of the world; formulas for producing the projections were discussed today.

The stereographic projection results from choosing a point A on the sphere and then projecting each other point on the sphere's surface to a plane tangent to the sphere at the point B opposite A. (A and B are called *antipodes.*) The projection points to use for problem 1 are the North and South Poles. To produce a stereographic projection, we need to choose a representation for the map. The data consist of coordinates of latitude and longitude-spherical coordinates, essentially-so a polar coordinate system seems appropriate to use for the map.

Let θ be the longitude east of Greenwich, and ϕ be the latitude north of the equator. Polar coordinates of the projection from the South Pole can be easily expressed in terms of θ and ϕ. The angle of rotation around the center is θ in both systems. The radius r on the map is determined as follows: the angle of projection ψ is half of $90-\phi$ and the radius ρ of the sphere is known, so $r = 2\rho \tan \psi = 2\rho \tan((90-\phi)/2) = 2\rho(\sec\phi - \tan\phi)$.

We might also use rectangular coordinate systems for both sphere and map, representing the earth as a sphere with radius 1 and origin (0,0,0); the North Pole is assumed to be (0,0,1), and the Greenwich meridian intersects the quator at (1,0,0). The equation of this sphere is $x^2+y^2+z^2 = 1$, and the rectangular coordinates x, y, and z may be determined from θ and ϕ using the formulas $x = \cos\phi\cos\theta$, $y = \cos\phi\sin\theta$, $z = \sin\phi$. The projection of a point (x,y,z) on tht sphere is the point at which the line going through (0,0,-1) and (x,y,z) intersects the plane $z = 1$. The quation of the line may be determined using a parameter t: a point on the line has coordinates of the form $(0,0,-1) + t(x-0,y-0,z-(-1)) = (tx,ty,tz+t-1)$. This line intersects the plane $z = 1$ at $t = 2/(z+1)$, so the projected point is $(2x/(z+1),2y/(z+1))$.

The main things we should consider when evaluating our solution method are its simplicity and its generality. For instance, projecting onto a polar coordinate system was easy in our example, but a projection point other than the North or South Pole would make things much more difficult (the θ on the map would no longer be the same as the longitude, and r would be a function of both longitude and latitude). The use of rectangular coordinates, although it requires data conversion effort, seems the nicer method of the two. The representation is symmetric in x, y, and z, which may make it easier to work with. Also, we can determine stereographic projections from any point in the same way as we did for the South Pole-finding the quations of the line of projection and the plane tangent to the sphere at the antipode, and then intersecting them-with only a little more effort.

There usually is no best way to solve a problem, so we optimize our solution for the particular applications. But wt should look for simple solutions-easy to understand, easy to extend—whenever possible.

A useful property of projections is *conformality*, that is, preservation of angles between lines on the globe. Stereographic projections are conformal. So are Mercator projections, which involve projecting the sphere to a cylinder and then stretching the cylinder until the map is conformal. To determine where a point (ϕ,θ) is projected, consider two small vectors dp and dq emanating from (ϕ,θ) on the sphere such that dp points toward the North Pole and dq points east. If these vectors are small enough, they are projected to vectors dX and dY on the map, and $dp/dq = dY/dX$ by the definition of conformality. Now dp is the same as $d\phi$, assuming a unit sphere, since dp and $d\phi$ are both measured along a great circle. On the other hand, dq is measured along the circumference of some parallel of latitude. The ratio of the circumference along this latitude to the circumference along the equator is $\cos\phi$, so $dq = d\theta \cos\phi$, and we have $dY/dX = 1/\cos\phi \cdot d\phi/d\theta$. On the map, we let X be θ (scaled by some appropriate amount) and Y be $f(\phi)$. Differentiating f with respect to ϕ, we have $dY/dX = f'(\phi)\, d\phi/d\theta$; integrating gives $f(\phi) = \ln(\sec\phi + \tan\phi)$. (Recall that the function $\sec\phi - \tan\phi$ came up in the stereographic projection. Is this just coincidence?) In Cartesian coordinates, $\sec\phi + \tan\phi$ is $(1+z)/\mathrm{sqrt}(1-z^2)$.

Suppose we choose to work with Cartesian coordinates, and we want to develop the idea of conformality from "first principles". Let (x,y,z) be a point on the sphere, and consider a straight line $(x,y,z)+t(a,b,c)$ which is tangent to the sphere. "Tangent" means that for small t the line is as close to the sphere as possible; thus we want $(x+ta)^2+(y+tb)^2+(z+tc)^2$ to be very near 1 when t is small. Well, this is $x^2+2tax+t^2a^2+\ldots = 1+2t(ax+by+cz)+O(t^2)$, so the line will be tangent if $ax+by+cz=0$. In other words, the dot product $(a,b,c)\cdot(x,y,z)$ is 0, i.e., the vectors are perpendicular. (We knew this, but it is perhaps interesting to derive it from the equation of the sphere.) We can normalize the line so that $a^2+b^2+c^2 = 1$. If we have another line tangent to the sphere at (x,y,z), say the line $(x,y,z)+t(\alpha,\beta,\gamma)$ where $\alpha^2+\beta^2+\gamma^2 = 1$, the definition of conformality says that the image of this line should have the same angle to the image of the other as the angle between (a,b,c) and (α,β,γ). Here it helps to know about the cross product of vectors: If we let $(\alpha,\beta,\gamma) = (a,b,c)\times(x,y,z) = (bz-cy,cx-az,ay-bx)$, it is easy to check that $\alpha^2+\beta^2+\gamma^2 = 1$ and that $(\alpha,\beta,\gamma)\cdot(a,b,c) = (\alpha,\beta,\gamma)\cdot(x,y,z) = 0$. Then the line defined by the vector $(a,b,c)\cos\psi + (\alpha,\beta,\gamma)\sin\psi$ makes an angle ψ with the line defined by (a,b,c), and so we can check conformality by seeing what happens to the lines (a,b,c) and (α,β,γ) and linear combinations of them.

For example, to check the conformality of stereographic projection by this method, consider the image of (x+at,y+at,z+at), namely

$$(a(x+at)/(z+1+ct), 2(y+bt)/(z+1\ tct)) = (2x/(z+1), 2y/(z+1))\ t\ 2t/(z+1)^2(cx+(z+1)a, cy+(z+1)b) + O(t^2).$$

Similarly the image of $(x+\alpha t,y+\beta t,z+\gamma t)$, where $(\alpha,\beta,\gamma) = (a,b,c)\times(x,y,z)$ as above, turns out to be

$$(2x/(z+1), 2y/(z+1)) + 2t/(z+1)^2\ (-cy-(zt\ 1)b, cx+(z+1)a)\ t\ O(t^2),$$

so the image lines are defined by orthogonal vectors of the form (u,v) and (-v,u), where $u=2(cx+(z+1)a)/(z+1)^2$ and $v=2(cy+(z+1)b)/(z+1)^2$ depend linearly on a, b, and c; the conformality condition follows.

Similarly, for the Mercator case, let us try to find the image of (x+at,y+bt,z+ct) for small t. First we define p and q by the conditions

$$x+at = \cos(\phi+pt)\cos(\theta+qt) + O(t^2)$$
$$y+bt = \cos(\phi+pt)\sin(\theta+qt) + O(t^2)$$
$$z+ct = \sin(\phi+pt) + O(t^2).$$

Thus we get

$$a = -p\sin\phi\cos\theta - q\cos\phi\sin\theta$$
$$b = -p\sin\phi\sin\theta - q\cos\phi\cos\theta$$
$$c = p\cos\theta$$

from which it follows that $ay-bx = -q\cos^2\phi$. If we map (x,y,z) into $(\theta,f(\phi))$, where f is chosen to make this conformal, we find that (x+at,y+bt,z+ct) for small t goes into $(\theta+qt, f(\phi)+ptf'(\phi)) + O(t^2) = (\theta,f(\phi))\ t\ t/\cos^2\phi\ (bx-ay, cf'(\phi)\cos\phi)$. The image of $(x+\alpha t,y+\beta t,z+\gamma t)$ similariy comes to $(\theta,f(\phi)) + t/\cos^2\phi\ (\beta x-\alpha y, \gamma f'(\phi)\cos\phi) = (\theta,f(\phi))\ t\ t/\cos^2\phi\ (c, f'(\phi)(ay-bx)\cos\phi)$. Thus conformality requires $f'(\phi) = 1/\cos\phi$, as before.

This derivation involved more computation but less insight, since it sticks more closely to "first principles". However, it still is not simple enough to explain how Mercator came up with the idea, since he lived from 1512 to 1594; that was not only before calculus, it was also before Descartes, so he didn't have Cartesian coordinates to work with either.

When doing research, it is instructive to not do an extensive literature search right away, but instead to think beforehand about the problem and different ways of solving it. This approach often gives us a better feel for the difficulty-we will encounter in solving the problem. Similarly, when confronted by the opportunity to plug some numbers into a formula out of the CRC book, it's a better idea to try to understand how the formula was derived in the first place. Even when reading somebody else's contribution to a problem, we should constantly ask not what was done but "how could I have thought of that?". This is perhaps the best way to develop research skills. Another interesting kind of research is historical inquiry into what the original discoverer of an idea had to say about it at the time; often the inventor had a better feel for the subject than many of the people who followed, and a study of original source documents is usually very rewarding. Thus it would be interesting to look up what Mercator said.

To produce the Mercator map with the usual north and south poles, we take (ϕ,θ) into $(\theta, \ln(\sec\phi + \tan\phi))$ in spherical coordinates; Cartesian-wise this becomes $(\arctan(y/x), \ln((1+z)/\mathrm{sqrt}(1-z^2)))$, where the arctan function has values between $-90°$ and $90°$ when $x>0$, between $90°$ and $180°$ or between $-90°$ and $-180°$ when $x<0$, or $90°\cdot\mathrm{sign}(y)$ when $x=0$. The details are a bit messy, but it is easy to use these formulas when the "north and south poles" change to the points $(\pm1,0,0)$, by permuting the coordinates. Alternatively we can stick to spherical coordinates if we rotate the sphere first; it is easy to see that any rigid motion of the sphere about its center can be achieved by a rotation about the earth's axis followed by a rotation along, say, the Prime Meridian followed by another rotation about the earth's axis. The first and last are trivial to do directly in spherical coordinates. Formulas for a rotation of the Prime Meridian can be obtained either by permuting the x,y,z coordinate axes, then converting to spherical coordinates, rotating, and permuting back again; or by using spherical trigonometry to find the other edges and angles of the triangle formed by the old North Pole, the new pole, any arbitrary point, and the great circles connecting them. In the latter case, it helps to know the law of cosines and the law of sines for spherical trigonometry:

$\cos a = \cos b \cos c + \sin b \sin c \cos A$ (a,b,c are sides; A ,B,C are angles)

$\sin A / \sin a = \sin B / \sin b = \sin C / \sin c.$

Hence a rotation of $90°$ gives $\cos\phi\cos\theta = \sin\phi'$ and $\sin\theta\cos\phi = \sin\theta'\cos\phi'$.

A two-point quidistant map has the property that the distances between an arbitrary point X and two selected points A and B are always in true scale. A rectangular coordinate system seems easiest for this projection. The distances between X and A and between X and B are found; they determine two circles centered at A and B, which intersect in two points, one of which is the projection of X. Which point it is can be determined as follows: Let (x_0,y_0,z_0) be New York and (x_1,y_1,z_1) be London, and let $(x_2,y_2,z_2) = (\text{New York})\times(\text{London})$ be a vector normal to the great circle of New York and London. The spherical distance of a point (x,y,z) to New York is θ_1 where $\cos\theta_1 = (x,y,z)\cdot(\text{New York})$, and the distance to London is θ_2 where $\cos\theta_2 = (x,y,z)\cdot(\text{London})$. We map (x,y,z) into the upper point having these given distances if $(x,y,z)\cdot(x_2,y_2,z_2)>0$, and into the lower point if $(x,y,z)\cdot(x_2,y_2,z_2)<0$. In other words, the idea is to take three dot products to determine the image of (x,y,z). The sphere is split along the great circle from (-New York) to (-London); that line maps into an ellipse, since all points on that line have the property that the sum of their distances to New York and to London is constant. Fortunately this great circle arc lies entirely in the ocean, so no country is split by this process.

Notes for October 12, 1976

Topics of today's discussion:
1. drawing smooth curves through points on a plane;
2. splines;
3. zeroing in on a good curve-approximating procedure.

For problem 1, we wrote programs which read data representing the boundaries of continents of the world, and produced maps on which the continents appeared as polygons. To make realistic-looking maps, this method needs a lot of data points; today we discussed ways of interpolating smaller sets of data to produce smooth curves which fit the data.

A simple solution to the problem is to connect successive points with straight lines, as we did for problem 1. If there are enough points, this gives the desired result-a "smooth" curve-since sharp points on the boundaries will be too small to show up on the map. In most cases, however, the set of points will not be dense enough for this method to succeed. Another, better, solution is to connect adjacent points with some more complicated curve, requiring in addition that consecutive curves meet in a nice way, e.g. with equal slopes at the point of intersection.

The problem is an old one. Shipbuilders built a framework of beams, then overlaid it with strips of wood to form the hull of the ship, effectively fitting a curve to the "data points" of the frame. A strip of wood so used was called a *spline.* The word is now used mathematically to mean a function which interpolates a list of points; between successive points the function is the same as some polynomial, and derivatives of adjacent polynomial "pieces" are equal at certain points where the pieces intersect. More information on splines may be found in [2].

For our first solution, the polynomial pieces were all linear, and probably none of the derivatives matched up. A natural step forward would be to choose quadratic polynomials. Given a sequence of data points (x_i,y_i), we will find a sequence of functions f_i, each of which is some quadratic polynomial connecting (x_i,y_i) and (x_{i+1},y_{i+1}). Actually f_i is a function of a parameter t, say, which has two components; it is convenient to consider $f_i(t)$ as a function from the interval [0,1] to the plane, where $f_i(0)=(x_i,y_i)$ and $f_i(1)=(x_{i+1},y_{i+1})$. In order to have a "smooth" curve, the derivatives should match up, namely

(1) $\qquad f_i'(1) = f_{i+1}'(0).$

For simplicity let's first consider only the y component; in a quadratic case we therefore can set $f_i(t) = (1-t)y_i + ty_{i+1} + t(1-t)a_i$. Note that $(1-t)y_i+ty_{i+1}$ is a function which has the correct values at the endpoints of [0,1]—we've defined $f_i(0)=y_i$ and $f_i(1)=y_{i+1}$—and $t(1-t)a_i$ is zero at the endpoints and sufficiently general to include all quadratic functions. To determine the coefficients a_i, we see that $f_i'(t) = -y_i + y_{i+1} + (1-2t)a_i$; condition (1) results in the recurrence $a_{i+1} = -(y_{i+2} - 2y_{i+1} + y_i) - a_i$.

Apparently quadratic splines give us one degree of freedom; we choose a_1, and all the f_i are determined. What would the best choice of a_1 be? Well, each a_i is the curvature of the corresponding parabola, so the "best" spline might be the curve which minimizes all these curvatures somehow, e.g., minimizes $\Sigma(a_i^2)$. Unfortunately such a curve fitted to the data points of Figure 1 is a very bad approximation to the best curve fitting only the middle points, no matter what the choice of a_0. We might think of choosing another metric for examining the a_i's, say $\Sigma(a_i^2w_i)$, in order to let the curve settle down on "flat" areas and ignore or minimize gigantic curvature at one part of the curve. No matter what our method for choosing a_0, however, we cannot avoid the "globality" inherent in quadratic splines. We choose one parameter, we get the whole curve, and sets of data which vary widely in one place but are regular in others won't be adequately representable by *a* quadratic spline.

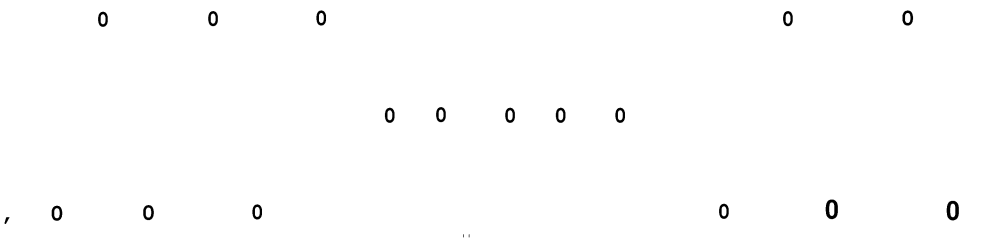


Figure 1 – points which a quadratic spline cannot adequately represent

We have several options. We can relax the condition that the curve hits all data points; much has been written (check any numerical analysis book) on, for example, least-squares curve fitting. Or we can abandon splines and try some other approximating curve. One method, called *quadratic* blending (see [I]), would have a quadratic for each three successive points in the data list; the curve constructed between 'any two successive points is the *average* of the two parabolas over the corresponding interval. This method avoids the main disadvantage of the quadratic spline, since it does not allow local curvature to affect the whole curve.

Our third alternative is to approximate the pieces of the curve with higher-order polynomials. We'll try cubics. As before, we'll examine functions of one parameter variable on unit intervals: $f_i(t) = (1-t)y_i + ty_{i+1} + t(1-t)(a_i(1-t)+b_it)$, such that $f_i'(1) = f_{i+1}'(0)$. Differentiating f_i and f_{i+1} and evaluating at 1 and 0 respectively yields the recurrence

(2) $$a_{i+1} = -y_{i+2} + 2y_{i+1} - y_i - b_i$$

This recurrence gives us n degrees of freedom for the spline, one for each point. Hence we can choose the "right" derivative locally for each point, instead of trying to pick some global value; so we have the smoothness of splines together with the important locality property of quadratic blending.

Before we choose the "best" slope at (x_i,y_i), we should decide what it is: perhaps the average of the slopes from (x_{i-1},y_{i-1}) and (x_{i+1},y_{i+1}) to (x_i,y_i), perhaps some combination of the corresponding vectors. We find another desirable characteristic of the spline curve: besides being local, so the curvature of one part won't affect another part, and having low degree, so it can be easily computed, the curve should have physical meaning, in that rotation or dilation or translation of the data set should not change the resultant curve. The average slope between two lines, for instance, doesn't give the average of the angles made by the lines; the latter would be invariant under rotation, while the former would not. Suppose we treat the lines from (x_{i-1},y_{i-1}) to (x_i,y_i) and from (x_i,y_i) to (x_{i+1},y_{i+1}) as vectors, and pick a derivative at (x_i,y_i) corresponding to a vector which bisects the angle formed by the other two. It is convenient to use complex coordinates: letting $z_j=x_j+iy_j$, and substituting z_j for y_i in formula (2), we have the situation pictured in Figure 2. The complex numbers z_j can also be described in polar coordinates (r,θ) such that $z_j-z_{j-1}=r_j\exp(i\theta_j)$. The vector represented in polar cwrdinates by $\mathrm{sqrt}(r_jr_{j+1})\cdot\exp(i(\theta_j+\theta_{j+1})/2)$, i.e. the square root of the product of the two vectors $z_{j-1}z_j$ and z_jz_{j+1}, is a vector which bisects the angles between the two vectors. Given this formula for the derivative, the numbers .a, and b, are determined (as are the corresponding numbers for the x component of f_i); it is perhaps most convenient to think of f_i as a complex-valued function on [0,1].

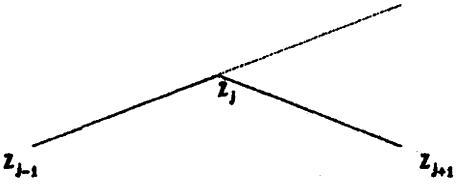


Figure 2 – vectors to which we wish to fit a reasonable curve

Some other mean between the two vectors might also be used to determine-the derivative at z_i—the sum of the vectors is computationally simpler, and someone suggested taking a sum using the inverse of the lengths in order to weight short segments higher. If one of the vectors is small, however, the resultant vector should probably be near zero, and neither vector sum has that property. One might possibly consider what should happen when two long vectors have a short vector between them, and how a contour that really does come to a point should be specified (e.g., by taking $(x_{i+1},y_{i+1})=(x_i,y_i)$). In n dimensions a similar approach can be used; to get the desired tangent line at (x_i,y_i) one simply works in the plane determined by (x_{i-1},y_{i-1}), (x_i,y_i), and (x_{i+1},y_{i+1}).

We can now step back and see how we came to this point. We proposed a very simple first solution (drawing straight lines between the data points), and evaluated its deficiencies. We chose to generalize an appropriate parameter of the solution (the degree of the approximating polynomial), and evaluated the deficiencies of the result. Each evaluation after the first yielded new features desirable in the "ideal" solution (smoothness, locality, invariance under rotation). In this way we homed in on our final answer. This method of research is a common one. In [3] are many examples of sorting algorithms which evolved in much the same way as our spline algorithm, e.g. Shellsort from straight insertion and Quicksort from bubble sort.

There are interesting unanswered questions in this area. Given a formula for producing a curve, which are the "best" choices of points for that formula? Can we efficiently determine the least number of points which define a given curve to within a specified tolerance?

References:

[1] Forrest, A., *Curves and Surfaces for Computer-aided Design*, Cambridge, 1988.

[2] Forsythe, G.E., Malcolm, M.A., and Moler, C.B., *Computer Methods for Mathematical Computation: Notes for CS 135*, Stanford C.S. Dept., 1975.

[3] Knuth, D.E., *The Art of Computer Programming: Volume 3, Sorting and Searching*, Addison-Wesley, 1973.

Notes for October 14, 1976

Topics of today's discussion:
1. curve-plotting algorithms for discrete graphics devices;
2. transformations to the algorithms to improve their performance.

Lines drawn on Data Disc screens, since they are raster-type devices, can only be approximated; "dots" on the screen are lit up which are as near as possible to the actual path of the line. Lines drawn using plotters are approximated in much the same way; normally a plotter pen can move in directions which are multipks of 45 degrees, and lines are approximated by combinations of the possible pen moves. We discussed algorithms for approximating lines and curves today.

We can take two approaches to studying such algorithms. One is to look up what's been done—systems for running plotters presumably include good ways of drawing lines-but some published algorithms are almost comment-free and poorly explained. (They appear more scholarly that way.) The other way is to solve the problem ourselves.

We can look for example at an 5x8 rectangle (see Figure 1), and try to draw a line from corner to corner. The accurate line is $y=5x/8$. For each x we can represent the corresponding y value by rounding to the nearest integer, finding y such that $5x/8-\frac{1}{2} \le y < 5x/8+\frac{1}{2}$; such a y is $\lceil 5x/8-\frac{1}{2} \rceil$.

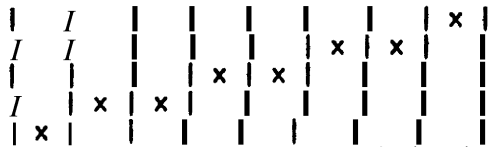


Figure I – trace of a line from (0,0) to (8,5)

For the general problem, an mxn rectangle, we need only look at those for which $m \le n$; a rectangle for which $m>n$ can be rotated, the trace constructed, and then the rectangle rotated back. The trace may be viewed as a series of pen moves, and we can easily see that we need only two: up or right. So the problem we'll examine is the following: given an mxn rectangle for which msn, generate a series of pen moves either to the right (0 degrees from horizontal) or. up and to the right (a 45 degree move), from the lower left corner to the upper right corner.

For each x, let the traced y value be f(x). An initial algorithm is then

```
for x←0 untll n do y[x]←f(x);
for x←1 until n do
    if y[x]=y[x-1] then move-0 else move-45;
```

This algorithm wastes a lot of space and time. The y array is not necessary; each f(x) can be computed incrementally, from f(x-l), instead of starting from scratch. So we need to tune it up.

Program tuneups aren't much in style these days. Computers are so fast that program inefficiency isn't an important concern, and anyway, given a halfway-decent algorithm, a good compiler can optimize out all the inefficiency-so say some experts. But optimizing compilers aren't all that smart, for one thing. Bob Sedgewick noted in [3] that the PL/I optimizing compiler produced the worst code of several processors (including Algol W and unoptimized Fortran H) run on versions of the Quicksort algorithm. The attitude against nitty-gritty program improvement is perhaps a reaction against the "hacker" image (see [4] for an entertaining description of hackers), and it's certainly harder to debug a program containing a lot of tricky optimizations before debugging the basic underlying algorithms. Programming, however, should be a two-phase activity: the planning and coding of the initial version of the program, and then the application of provably-correct transformations which improve the program's performance.

Back to the program at hand. Instead of using the array y, we can just use the function f and save the previous value of f. Taking advantage of the fact that successive y values differ by either 0 or 1, we have

```
Y ← f(0);                    comment This is 0;
for x←1 until n do           comment At this point, y=f(x-1);
    If f(x)=y then move-0
    else begin
        move-4 5;
        Y ← y+1
        end;
```

The inefficiency in this version occurs in the computation of f(x) = ⌈mx/n-½⌉ in each iteration of the loop. All we really need to check is if f(x-1) is within the bounds defined by rounding f(x), that is, if mx/n-½ ≤ y ≤ mx/n+½, where y=f(x-1). We can compute the check for this condition more efficiently by transforming it as follows:

2mx-n ≤ 2ny < 2mx+n	multiply through by 2n to eliminate fractions;
0 ≤ 2ny-2mx+n < n	subtract 2mx-n from all terms to allow comparison with 0.

The condition "*if f(x)=y*" is thus equivalent to "*if 0 ≤ 2ny-2mx+n < 2n*". We can prove that 2ny-2mx+n < 2n for allowable values of x and y, so we have

```
Y ← 0;
for x←1 until n do           comment y = f(x-1);
    if 0 ≤ 2ny-2mx+n then move-0
    else begin
        move-45;
        Y ← y+1
        end;
```

Another optimization is called *strength reduction,* which replaces an operation such as multiplication by an incremental operation which executes faster, such as addition. One example of its application is the following:

```
                                    t1←0;
for i←1 until n do                  for i←1 until n do
    begin                   =>          begin
    ...                                 ...
    t1←i*k;                             t1←t1+k;
    ...                                 ...
    end;                                end;
```

For our algorithm we could apply strength reduction for the operations involving either x or y; we can do even better by having a variable R for the quantity 2ny-2mx+n and then adjusting R during each loop iteration. As x goes up by 1, R decreases by 2m; as y increases by 1, R goes up by 2n. The addition of n can be absorbed in the initial value assignment to R. The result (with variables C and D containing the constants 2n and 2m):

```
C←2n;  D←2m;  R←n;
for x←1 until n do
    begin
    R ← R - D;              comment Reflects x changing by 1;
    if R ≥ 0 then move-0
    else begin
            move-4 5;
            R ← R + C;      comment Reflects y changing by 1;
            end;
    end;
```

Is the counter x really necessary? Not always, if a test on R suffices to show whether or not the upper-right corner of the rectangle has been reached. R is zero when the loop terminates, but in the 5x8 example R would have been zero when x was 4. Another refinement: if we halve the quantities R, C, and D, where R starts at $\lceil n/2 \rceil$, it is not hard to verify that the loop is the same. Now the counter isn't necessary when m and n are relatively prime. (The probability of this is $6/\pi^2$.)

Knuth estimates in [2] that 10-15% of all computing is spent on unconditional transfers to the beginning of loops. Hence we could turn the *for* loop in our example into a *while* loop to put the test at the start. Some computers have special loop-control instructions which reduce the need for such an optimization.

The algorithm we have for drawing lines generalizes nicely for drawing circles and ellipses. If we trace only 1/8 of a circle, we need once again only consider two moves: a move to the right (0 degrees) and a move down and to the right (-45 degrees). The function f(x) = rounded $\mathrm{sqrt}(r^2-x^2)$ specifies the position of the pen. During each iteration of the loop, we wish to test if y=f(x-1) is between $\mathrm{sqrt}(r^2-x^2)-\frac{1}{2}$ and $\mathrm{sqrt}(r^2-x^2)+\frac{1}{2}$. The following inequalities are equivalent:

$\mathrm{sqrt}(r^2-x^2)-\frac{1}{2} \le y < \mathrm{sqrt}(r^2-x^2)+\frac{1}{2}$	
$y-\frac{1}{2} < \mathrm{sqrt}(r^2-x^2) \le y+\frac{1}{2}$	adding $y-\mathrm{sqrt}(r^2-x^2)$ to both sides, then multiplying by -1;
$y^2-y+\frac{1}{4} < r^2-x^2 \le y^2-y+\frac{1}{4}$	squaring each term;
$0 < r^2-x^2-y^2+y-\frac{1}{4} \le 2y$	subtracting $y^2-y+\frac{1}{4}$ from each term;

Hence the test which determines the direction of the pen move is $0 < r^2-x^2-y^2+y-\frac{1}{4} \le 2y$. The middle term may be replaced by $\lfloor r^2-\frac{1}{4} \rfloor -x^2-y^2+y$ (since for any x, if $x \le n$, then $\lfloor x \rfloor \le n$). We can set R to this quantity and maintain variables $X = 2x+1$ and $Y = 2y-2$ to keep track of the changes to R during each iteration of the loop. The loop terminates when $X \ge Y$. What results is the following code, very much like the routine for tracing lines.

```
x ← 0; y ← ⌊r⌋;
R ← ⌊r²-¼⌋-x²-y²+y; X ← 2(x-1)+1; Y ← 2(y+1)-2;
while X < Y do
    begin
    X ← X + 2;     R ← R - X;
    if R > 0 then move-0
    else begin
        Y ← Y - 2;     R ← R + Y
        move-minus-45;
        end
    end;
```

A similar routine for cubic curves would make use of additional variables like $X = 3x^2+3x+1$ and $dX = 6x+3$, etc.

This problem was interesting in that it combined programming methodology with some mathematics. It was also fairly easy. Practice with small problems, however, improves our ability to organize solutions to bigger ones. Research is going on at Stanford and elsewhere on understanding and automating the programming process; Cordell Green and Dave Barstow, for example (see [1]), have attempted to codify the knowledge required to write a program to sort a list of keys. We understand more about programming now than ever before. Still, as we learn more about old techniques, people invent new ones, always staying a few jumps ahead.

What about tracing general curves, that is, approximating the curve f(x,y)=0? We can assume a "nice" function f, that is, one which is continuous and doesn't curve tw abruptly. One way to approximate f is to go from x to x+1 as before and find the corresponding approximate y value. For thls method, however, we need some special information about f; a general curve may wind around, and there may not be a unique y value for some x. Another way is to compute values of f at points around the current pen position. Some of these values will be positive, some negative, and maybe some zero. We can pick the place to move the pen either by inspection (taking the point whose f value is closest to zero) or by interpolation, depending on the fineness of the set of points we selected.

Yet another way, the most clever of the three, uses a triangular grid; each triangle on the grid represents a region on the display which can be lit up, and each "corner" on the grid is a point at which the function is evaluated. Any assignment of +'s and -'s to the points of the graph defines a set of contours on the display, which passes between points of different "parities". Each contour either winds around and intersects itself, or goes off the boundary of the display. This method also can be implemented so as to rquire very little memory space.

References:

[1] Green, C.C. and Barstow, D.R., "Some Rules for the Automatic Synthesis of Programs", *Proc. 4th IJCAI*, Sept. 1975

[2] Knuth, D.E., "Structured Program with 'go to' Statements," Computing *Surveys 6,4* (Dec 74), pp. 260-301. --

[3] Sedgewick, R., *Quicksort*, appendix C, Ph.D. thesis (CS-192), Stanford University, May 1975

[4] Weizenbaum, J., *Computer Power* and *Human Reason*, W.H. Freeman, San Francisco, 1975

Comments on solutions to problem 1

(Notes from the "show and tell" discussion of October 19, 1976 are also included here.)

I graded programs and writeups on several criteria:
(a) correctness (does the program work?);
(b) program organization (is the program well-written and efficient?);
(c) documentation and style (could I understand what was going on?); and
(d) generality/human engineering/originality (everything else).
Grades in the first three categories were A, B, C, D, or E, meaning excellent, good, mediocre, poor, or terrible. My grading standards are not far from those of a typical comprehensive exam committee; a listing of a good solution program is included in these notes. (The program is also a good example of the use of records and record pointers.) Remember, by the way, that these grades are for your own interest only; the course itself will be graded pass-fail, and most likely everyone will pass.

I was disappointed at the number of programs handed in without sufficient comments. Few programs are "self-documenting", especially when they are as complicated as these were; people programmed rotations, for instance, in several different ways. Comments in the following places are especially useful: at the beginning of the program, where they outline the program's flow of control; at the head of each procedure, explaining the function of the procedure, what other procedures it calls, and what external variables are referenced or changed; at any label, describing the conditions which are true upon reaching the iabeiied statement; and at declarations of global variables, explaining the purpose of the variable, what values it may take on, and where it is initialized and updated.

The writeups should contain derivations if appropriate, or they should at least refer explicitly to class notes which contain the derivations. Explanation of the program is helpful in a writeup, but even more helpful in the code itself. The other part of your writeup for CS 204 problems should describe your approach to solving the problem. (Few writeups did.) One of the goals of this course is to teach you better methods of problem solving, and your explanations in the writeup enable us to give you feedback on the ways you attacked the problem. Aiso, you may find that writing down the processes you used to soive a problem may in itself help you understand those processes better.

Your programs should be clear and well-organized. Locality of effects is important; we comprehend a program more easily if we don't have to remember a lot of global state information to trace the program's progress. Furthermore, our minds can oniy handle so much about a single procedure, so break down your program into reasonably-sized chunks. Top-down problem-solving should lead you to divide your program nicely into procedures. You should pick data structures which maximize locality of data references. Be especially wary of global variabies; comment references to them profusely.

Gerald Weinberg, in the book *The Psychology of Computer Programmfng* (a great book, by the way), notes the following: "Programming is, among other things, a kind of writing. One way to learn writing is to write, but in ail other forms of writing, one also reads. We read examples-both good and bad-to facilitate learning. But how many programmers learn to write programs by reading programs? A few, but not many." Weinberg goes on to say that a good way to set about studying programming (the topic of his book) is to study programs. But studying programs, as he implies in the above passage, is also a good way to make your own programs more clear and more organized. Try to make a habit of having other people read your code, to get an idea of how it can be made better (often they even spot your bugs). Also, please read (carefully) the accompanying solution to problem 1.

The maps you obtained for this problem were in many cases quite interesting. Some of you beautified your maps by adding axes; others implemented various windowing or parameter-selection features, or adjusted for the Data Disc's aspect ratio. Most interesting were your selections for the fourth map. These are listed below, but I'll describe a few of them in more detail. Frank Liang produced a gnomonic projection onto a dodecahedron. Bengt Aspvall and Kevin Karplus produced maps of arbitrary *azimuthal* (or *perspective*) projections; the stereographic and gnomonic projections are special cases of the azimuthal projection. Mike Plass produced a sinusoidal map; this is a special case of a *conic* projection, which projects the sphere onto one or more cones wrapped around it. The sinusoidal map results from using a different cone for each parallel of latitude, chosen so that areas are preserved. Andrew Robinson and Marsha Berger came up with a "North Pole Expedition Map", for which distances to the North Pole are all in true scale; this seems to be a refinement of the two-point equidistant map, where the two points are identical. Carolyn Taicott produced the Werner heart map mentioned in the Scientific American article; the equations she used to map (ϕ,θ) into (x,y) are

$$x = (r \sin |\theta'|) \cdot \text{sign}(y \text{ coord of } (\phi,\theta))$$
$$y = -r \cos \theta'$$

where $\theta' = (\theta \cos \phi)/r$ and $r = \pi/2 - \phi$.

For another nice generalization, Alex Strong computed various projections by taking a point z (in complex coordinates) mapped by the stereographic projection and mapping it to the point $w = (az+b)/(cz+d)$. This function is analytic and hence the resulting projections are conformal. He observed that Mercator projection corresponds to the analytic function Log z, thus explaining why that function $\sec \phi + \tan \phi$ occurred twice in our previous class discussion.

At the boundaries of a map, where distortion often occurs, the straight-line approximation used for the map turns out to be too crude. A common solution for this was simply to leave off all straight lines longer than a certain threshold. This was also a solution to plotting a line which crossed one boundary of the map and continued on the other side (a better way of course was to draw both segments of the line).

Feature list:

Cylindrical (Mercator without stretching): JED, MI, RXM, JRR
Gnomonic: AVG, BTH/REP, AZS, FML
Arbitrary azimuthal: BIA, K JK
Sinusoidal: MFP
North Pole Expedition: M JB/AMR
Rotated stereographic: IAZ
Two-world on orthographic projection: WP/CGN
Complex conformal mapping: A RS
Werner heart: CLT
Intersections for two-point equidistant: DAN
User selection of parameters: BTH/REP (especially nice), K JK, BIA, JED
Axes: CLT, JRR, M JB, K JK, RXM, DAN, BIA

Reference:

[1] Weinberg, G.M., *The Psychology of Computer Programming,* Van Nostrand Reinhold, New York, 1971.

```
BEGIN "MAPS"

DEFINE !="COMMENT", TIL="STEP 1 UNTIL", NONE= "NOT", NO" "NOT", STARTLOOP="WHILE TRUE DO",
       CRLF="('15&'12)", PTR="RECORD_POINTER", NULLR="NULL_RECORD", n="3.14159";

! Organization of this program ("MAPS")

  MAPS first calls GETWORLD, to sot up the data base of coordinate points. GETUORLO asks the user for
  a file which will contain lists of (latitude, longitude) coordinates which correspond to "land
  bodies", that is, continents or islands or inland seas. Each land body list starts and ends with
  the same coordinate pair. GETUORLO assembles the file into a linked data base consisting of a list
  of circular lists representing a list of land bodies. A pointrr to this data base is stored in the
  variable WORLD. GETUORLO uses thr subsidiary procedure GETLANDBODY to road in the coordinates for a
  single land body and construct the corresponding circular list.

  Once the data base is constructed, thr projections are produced in turn. There are three
  project ions: a stereographic projection of the Northern Hemisphere, a rotated Mercator projection
  using (0,0) as the North Pole, and a tuo-point rquidistant projection with respect to points London
  and Now Vork. Each projection routine first initializes screen parameters (scaling is inciudsd
  here), than loops through tho UORLO list to plot the points for each land body. Tho procedure
  DRAWBODY, internal to each projection routine, loops through the circular list of points for the
  body, transforming the points as required for the particular projection and "moving the pen" to draw
  the map. When the UORLO list has been completely plottrd, thr map is dispiayod using the procedure
  DISPLAYMAP.

  The mathematics for thr projection transformations may be found for the most part in tho class
  notes. There are special considerations for the MERCATOR and EQUIDISTANT routines. In the Mercator
  transformation, a line may cross the edge of thr map in connecting two points, one in each
  hemisphere, and that iinr must be piottod in two pieces. In thr two-point equidistant
  transformation, the projection coordinate values depsnd on each other and so, to avoid unnecessary
  calculation, they are computed together (and returned in REFERENCE parameters) instead of
  separately.

  Other procedures (NEW_HEADER, NEW_PO INT) exist to simplify thr creating of records. The procedure
  PRINTLANDBODY, a l though not referenced in the program, was useful during tho debugg ing stages;


! Definitions for coordinate point record and list header record;
RECORD_CLASS POINT(REAL XX, VV, ZZ; PTR(POINT) NNEXT);
DEFINE !X="POINT:XX", !Y="POINT:YY", !Z="POINT:ZZ", !NEXT="POINT:NNEXT";
RECORD_CLASS HEADER(PTR (POINT) LLANDBODY; PTR (HEADER) RREST);
DEFINE !LANDBODY="HEADER:LLANDBODY", !REST="HEADER:RREST";

! UORLO contains pointer to a list of HERDER records. Ths !LANDBODY field of each HERDER record
  points to "land bodies", i.e., aircuiar lists of POINT records. Each POINT record contains thr
  Cartssian coordinator of a point in the corresponding land body. UORLD is crea ted in GETUDRLD and
  passed to t ho map draw ing procedures;
PTR (HERDER) WORLD;

! Daciarations for • xtornal procedures;
REQUIRE "DDHDR.SAI[GRA,HPM]" SOURCE-FILE ;

! Definition of i/o record;
RECORD_CLASS IO(INTEGER CCHANNEL, BBREAKCHARACTER, EEOF);
DEFINE !CHAN="IO:CCHANNEL", !BRK="IO:BBREAKCHARACTER", !EOF="IO:EEOF";


! Position of pen on screen. These are given vaiurs in MOVEPEN and read in MERCATOR;
REAL CURNTX,CURNTY;
DEFINE WITHPENUP="FALSE";        !i.e. pen not down;
DEFINE WITHPENDOWN="TRUE";

PROCEDURE MOVEPEN(REAL X,Y; BOOLEAN PENDOWN);
! Move the pen to posit ion (X,Y). If PENOOUN, drau a linr while doing so. Update thr current pen
  position and return. MOVEPEN is called by all the map drawing routines. MERCATOR • xpiicitiy
  references (but door not change) the va lues of CURNTX and CURNTV;
    BEGIN "MOVE PEN"
    IF PENDOUN THEN LINE(CURNTX,CURNTY,X,Y);     CURNTX←X; CURNTY←Y
    END "MOVE PEN";
```

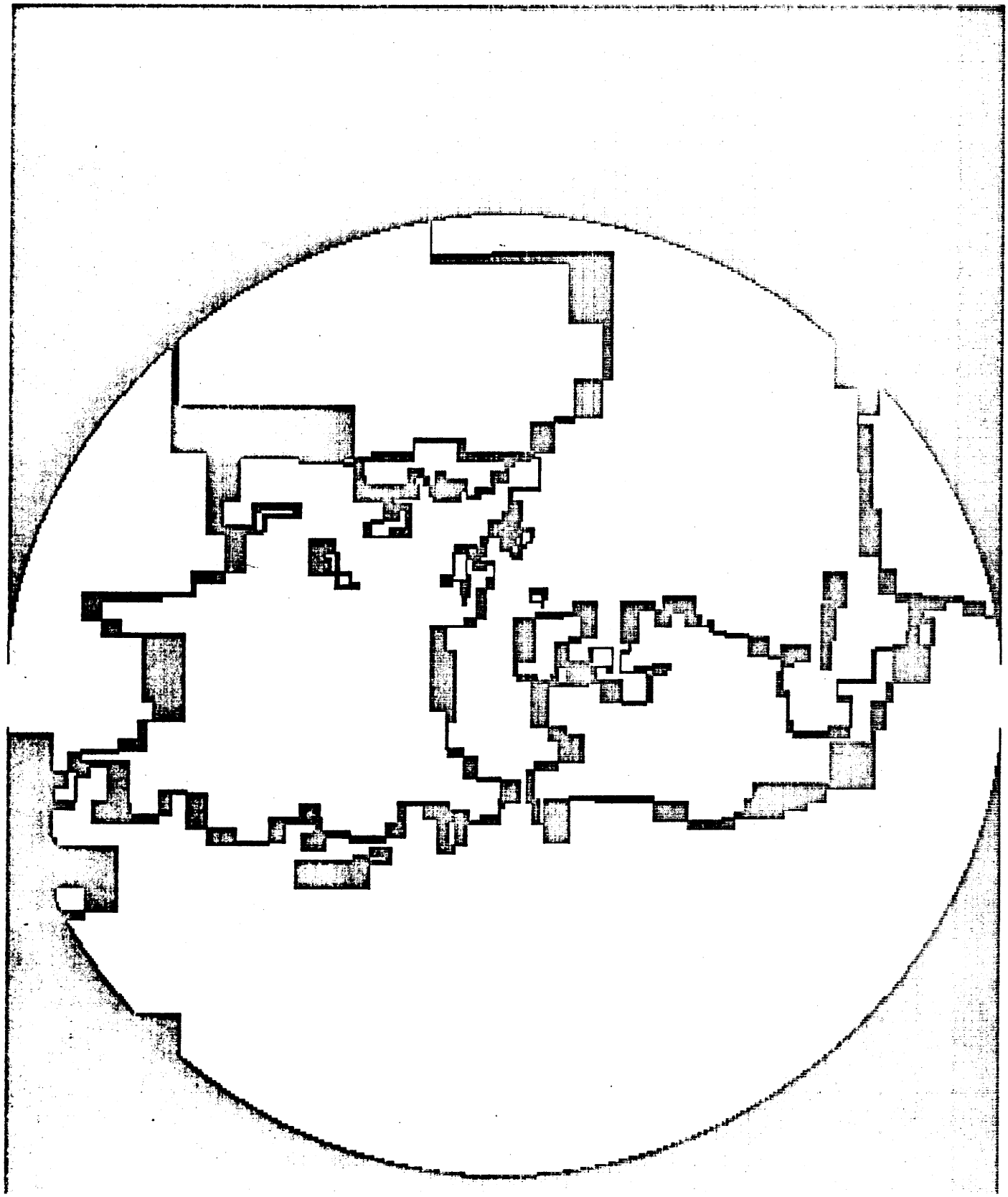
```
PTR (HEADER) PROCEDURE NEW-HERDER (PTR (POINT) L; PTR(HEADER) R (NULLR));
! L Is copied Into the !LANDBODY field of the new record, and R is copied Into the !REST field,
  Note that R is assumed null If not specified;
    BEGIN "NEW HERDER"
    PTR (HERDER) $NEW;     I Pointer to be returned1
    $NEW←NEW_RECORD (HERDER);
    !LANDBODY[$NEW]←L;   ! F i l l the f ields (too bad Sail doesn't let us do this al l at once);
    !REST[$NEW]←R;
    RETURN ($NEW)
    END "NEW HEADER";

PTR (POINT) PROCEDURE NEW_POINT(INTEGER LAT,LONG; PTR(POINT) N (NULLR));
! LRT and LONG are converted to Carteelan coordlnatss with respect to an origin at ths center of the
  • ◎□ th. The +x direct ion goes through Africa, the +y directton through the Indian Ocean, and the +z
  direction through the North Pole. The coordinates are coplsd into thr IX, IY, and IZ fields of the
  new record. N Is copied Into thr !NEXT field, and Is assumed null If not specified;
    BEGIN "NEW POINT"
    REAL CDSLRT, SINLRT, CDSLDNG, SINLONG;
    PTR (POINT) $NEW;     I Pointer to be returned;
    COSLAT←COSD (LAT);  SINLAT←SIND (LAT);
    COSLONG←COSD (LONG);  SINLONG←SIND (LONG);
    $NEW←NEW_RECORD (POINT);
    ! X [$NEW]←COSLAT*COSLONG;
    !Y [$NEW]←COSLAT*SINLONG;
    !Z[$NEW]  ●  SINLRTI
    !NEXT[$NEW]←N;
    RETURN ($NEW)
    END "NEU POINT";

PROCEDURE PRINTLANDBODY (PTR (HEADER) WORLD; INTEGER N);
! Prints the Nth land body In UDRLD.  Useful for dsbuggIng GETWORLD using Bail;
    BEGIN "PRINT LAND BODY"
    INTEGER I; PTR (POINT) P;
    FOR I←1 TIL N-I DO IF !REST[WORLD] THEN WORLD←!REST[WORLD];
    ! !LANDBODY [WORLD] now points at the des iced land body;
    IF P←!LANDBODY [WORLD] THEN
        DO PRINT(!X[P], !Y[P], !Z[P], CRLF) UNTIL (P←!NEXT[P])=!LANDBODY[WORLD]
    END "PRINT LRND BODY";

PROCEDURE DISPLAYMAP;
I Displays the contents of the display buffer on a free Data Disc channel. It waits for a line to
  be typed, and produces an Image of the display on the XCP If the line typed In starts with "X";
    BEGIN "DISPLAY MAP"
    INTEGER DCHRN, C;
    DCHAN←GDDCHN (-1);
    DPYUP (DCHAN);
    SHOU (DCHAN);
    IF (C←INCHWL)="X" OR C="x" THEN XGPUP(5);
    SHOW (-1);
    RDDCHN (DCHAN)
    END "DISPLAY MAP";
```

```
PTR(HEADER) PROCEDURE GETUORLD;
! This procedure inputs the date points in thr specified file and constructs a linked list, each
  member of which corresponds to a "land body" (a set of data points from the file whose first point
  is the same as Its last point). Each "land body" is a circular list of point records, each of which
  contains the coordinates of the associated data point. A pointer to the entire I lnked I ist Is
  re turned. Outside procedure calls: NEW_HEADER;
    BEGIN "GET WORLD"
    PTR(IO)$IN; BOOLERN BRDLDDKUP;
    PTR(HEADER) UORLD; PTR(POINT)BODY;

    PTR(POINT) PROCEDURE GETLRNOBODY;
    ! This procedure constructs the circular list corresponding to the points for one "land body",
      and returns a pointer to some point in the list. If no more input, the null pointer Is
      returned. Outside procedure calls: NEW_POINT;
        BEGIN "GET LRND BODY"
        INTEGER LATITUDE, LONGITUDE, FIRSTLRT, FIRSTLONG; PTR(POINT) BODY, FIRSTPT;
        ! BODY will be returned. As points come in, they will be added to the front of the land
          body, and when the last point is hit the end of the body will be linked to its beginning;
        FIRSTLAT←INTIN(!CHAN[$IN] ); FIRSTLONG←INTIN(!CHAN[$IN]);
        IF !EOF[$IN] THEN          !Assuming hero that ● of occurs only after a complete land body;
            RETURN (NULLR);
        BODY←FIRSTPT←NEW_POINT(FIRSTLAT, FIRSTLONG);
        LATITUDE←INTIN(!CHAN[$IN]); LONGITUDE←INTIN(!CHAN[$IN]);
        WHILE LATITUDE≠FIRSTLAT OR LONGITUDE≠FIRSTLONG DO
            BEG IN
            BODY←NEW_POINT(LATITUDE, LONGITUDE, BODY);
            LATITUDE←INTIN(!CHAN[$IN] ); LONGITUDE←INTIN(!CHAN[$IN])
            END;
        !NEXT(FIRSTPT)←BODY;
        RETURN (BODY)
        END "GET LAND BODY";


    ! Open a channel for Input, and connect It to the appropriate file;
    $IN←NEW_RECORD(IO);
    !CHAN[$IN]←GETCHAN;
    OPEN(!CHAN[$IN], "DSK", 1, 2  , 0, 150, !BRK[$IN], !EOF[$IN]);
    DO BEGIN
        PRINT("File to be read = ");
        LOOKUP (!CHAN[$IN], INCHUL, BADLOOKUP);
        END
    UNTIL ND BADLOOKUP;

    I WORLD will be returned as thr value of GETWORLD. As each land body comes along, rdd It to the
      front of the list;
    WORLD←NEW_HEADER(GETLANDBODY);
    WHILE BODY←GETLANDBODY DO WORLD←NEW_HEADER (BODY, WORLD);

    CLOSE(!CHAN[$IN]);
    RETURN (WORLD)
    END "GET WORLD";
```

```
PROCEDURE STEREOGRAPHIC(PTR(HERDER) WORLD);
! This procedure plots a stereographic projection of the Northern Hemisphere. UDRLD points to a
  linked list whose format is described e isewhors. The point (x,y,z) is mapped t o (2x/(z+1),2y/(z+1)),
  as derived in the class notes of 10/7/76. Outside procedure calls: DISPLAYMAP;
BEGIN "STEREOGRAPHIC"

    ! Coordinate transformations;
    REAL PROCEDURE XTRRNS (PTR (POINT) P);
        RETURN(2*!X[P]/(!Z[P]+1));
    RERL PROCEDURE YTRRNS (PTR(POINT) PI;
        RETURN(2*!Y[P]/(!Z[P]+1));

    PROCEDURE DRAWBODY(PTR(POINT)BODY);
    ! This procedure plots the land body pointed to by BODY. Firstthepenislif ted a n d moved t o
      the first point of the land body, thrn each point in the circular list is plotted. Outside
      procrdurr calls: MOVEPEN, XTRANS, YTRANS;
        BEGIN "DRAW BODY"
        PTR(POINT) P; REAL X,Y;
        P←BODY;
        X←XTRANS(P); Y←YTRANS(P);
        MOVEPEN(X,Y,WITHPENUP);
        DO BEGIN
            P←!NEXT[P];
            X←XTRANS(P); Y←YTRANS(P);
            MOVEPEN(X,Y,WITHPENDOWN);
            END
        UNTIL P-BODY
        END "DRAW BODY";

    !Ini t ialize the virtual display. The sap will be a circle with radius 2 (also derived in the
      class notes), We wil l construct the display with a border--everything between the circle of the
      map and the rectangle of thr screen will be "lightened";
    DDINIT;
    SCREEN(-2,-2,2,2);
    LITEN;
    RECTAN(-2,-2,2,2);          ! 1 ight up ths screen;
    DRKEN;
    ELLIPS(-2,-2,2,2);          ! Darken where t h s map will be;
    1 ITEN;

    ! Plot each land body;
    WHILE UORLD DO
        BEGIN
        DRAWBODY(!LANDBODY IUDRLDI);
        WORLD←! REST IUDRLDI
        END;

    D ISPLAYMAP

    END "STEREOGRAPHIC";
```

```
PROCEDURE MERCATOR(PTR(HEADER)WORLD);
! WORLD is a pointer to a linked list of land bodies. Each land body is a circular list of points
 (x,y,z) which specify a continent or island or inland sea. This procedure plots a Hercator
  project ion of the rotated world whose North Pole is at (1,0,0). The rotation takes (x,y,z) into
  (-z,y,x). The Hercator transformation maps (x,y,z) into (arctan(y/x),ln((1+z)/sqrt(1-z↑2))) as
  described in class notes for 10/7/76, so each point (x,y,z) from WORLD is mapped to (arctan(y/-z),
  ln((1+x)/sqrt(1-x↑2))). Outside procedure caller DISPLRYHRP;
    BEGIN "MERCATOR"
    RERL ZFOR75,UPPERBOUND;

    ! Coordinate transformations;
    RERL PROCEDURE XTRRNS (PTR(POINT) PI;
        ATAN2(!Y[P],-!Z[P]);
    RERL PROCEDURE YTRRNS (PTR(POINT)P);
        LOG((1+!X[P])/SQRT(1-!X[P]↑2));

    PROCEDURE DRRUBODY (PTR(POINT) BODY);
    ! This procedure plots the land body pointed to by BODY. First the pen is lifted and moved to
      the first point of the land body, then each point in the circular list is plotted. Outside
      procedure calls: MOVEPEN, XTRRNS, YTRRNS. Note also the reference to CURNTX and CURNTY;
        BEGIN "DRAW BODY"
        PTR (POINT)P; RERL X,Y,XDISPLACEMENT;
        P←BODY;
        X←XTRANS(P); Y←YTRANS(P);
        MOVEPEN(X,Y,WITHPENUP);
        DO BEGIN
            P←!NEXT[P];
            X←XTRANS(P); Y←YTRANS(P);
            !Check now for a move across thr edge of the map from one side to the other. Any pen
              move over a distance iongrr than n (i.e. half the map) is assumed to be such a move;
            IF ABS(CURNTX-X)>π THEN
                BEGIN
                ! Draw part of the line on the right side of the map and part on thr loft;
                XDISPLACEMENT←IF CURNTX>0 THEN 2*π ELSE -2tnl
                MOVEPEN(X+XDISPLACEMENT,Y,WITHPENDOWN);
                MOVEPEN(CURNTX-XDISPLACEMENT,CURNTY,WITHPENUP);
                MOVEPEN(X,Y,WITHPENDOWN)
                END
            ELSE MOVEPEN(X,Y,WITHPENDOWN)
            END
        UNTIL P=BODY
        END "DRRU BODY";

    ! Initial ize the virtual display. The x axis ranges from -π to +π, and longltudr intervals are
      constant, The upper bounds of the screen will correspond to latitudes of 75 degrees;
    DDINIT;
    ZFOR75←SIND(75);
    UPPERBOUND←LOG((1+ZFOR75)/SQRT(1-ZFOR75↑2));
    SCREEN(-π,-UPPERBOUND,π,UPPERBOUND);
    LITEN;

    !Plot each land body;
    WHILE UDRLD DO
        BEGIN
        DRAWBODY(!LANDBODY[WORLD]);
        WORLD←! REST [WORLD]
        END;

    DISPLAYMAP

    END "MERCATOR";
```

```
PROCEDURE EQUIDISTANT(PTR(HEADER)WORLD);
! This procedure plots a two-point equidistant projection of the world with respect to the tuo
  points New York (41 deg lat, -74 deg long) and London (51 deg la t, 0 dog long). The feature of such
  a map is that the distances betuern any point A and either New York or London are true to scale.
  The projection of a point A is found by taking dot products of A uith the vectors for Now York • nd
  London as described in the notes of 10/7/76. Outside procedure calls: DISPLAYMAP;
    BEGIN "EQUIDISTANT"
    PTR(POINT) NEUYDRK, LONDON, NORMAL;      RERL NEWYORK2LONDON, NEWYORK2LONDONSQ;

    RERL PROCEDURE DOT (PTR (POINT) P,Q);
        RETURN(!X[P]*!X[Q]+!Y[P]*!Y[Q]+!Z[P]*!Z[Q]);

    REAL PROCEDURE DISTANCE(PTR(POINT)P,Q); RETURN(ACOS(DOT(P,Q)));

    PROCEDURE TRRNS (PTR(POINT)P; REFERENCE RERL X, Y);
    ! Uo have a triangle for which we know the lengths of al I sides. This triangle Is formed by the
      points N=(0,0), L, and the point P we wish to plot. From tho Pythagorean relation we see thrt x↑2
      4 y↑2 = d(P,N)↑2, (d(N,L)-x)↑2 4 y↑2 = d(P,L)↑2. Hence x = (d(P,N)↑2+d(N,L)↑2-d(P,L)↑2)/2d(N,L).
      Also y = sqrt(d(P,L)↑2-(d(N,L)-x)↑2), with the sign posltivs iff dot(normal,p)>0 as specified In
      the notes of 10/7/76. Outside procedure calls: DISTANCE, DOT;
        BEG IN "TRRNS"
        RERL D;
        Xc(DISTANCE(P,NEWYORK)↑2+NEWYORK2LONDONSQ-(D←DISTANCE(P,LONDON)↑2))/2/NEWYORK2LONDON;
        Y←SQRT(D-(NEWYORK2LONDON-X)↑2);
        IF DOT(P,NORMAL)<0 THEN Yc-Y
        END "TRRNS";

    PROCEDURE DRRUBODY (PTR(POINT) BODY);
    ! This procedure plots the land body pointed to by BODY. First the pen Is I lfted and moved to
      the first point of the land body, then each point In the circular list Is plotted. Outside
      procedure cal Is: MOVEPEN, TRANS;
        BEGIN "DRAW BODY"
        PTR(POINT) P; RERL X, Y;
        P←BODY;  TRANS(P,X,Y);
        MOVEPEN(X,Y,WITHPENUP);
        DO BEGIN
            P←!NEXT[P] ; TRANS(P,X,Y);
            MOVEPEN(X,Y,WITHPENDOWN);
            END
        UNTIL P=BODY
        END "DRAW BODY";

    NEWYORK←NEW_PO 1 NT(4 1,-74); LONDON←NEW_POINT(51,0);
    NEWYORK2LONDON←DISTANCE(NEWYORK,LONDON); NEWYORK2LONDONSQ←NEWYORK2LONDON↑2;
    NORMAL←NEW_RECORD (POINT);
    !X[NORMAL]←!Y[NEWYORK]*!Z[LONDON]-!Y[LONDON]*!Z[NEWYORK];
    !Y[NORMAL]←!Z[NEWYORK]*!X[LONDON]-!Z[LONDON]*!X[NEWYORK];
    !Z[NORMAL]←!X[NEWYORK]*!Y[LONDON]-!X[LONDON]*!Y[NEWYORK];

    ! Initial Ize the virtual display. The map will be ♋■ • Illpss with major diameter 2π-d rnd minor
      diameter sqrt((2π-d)↑2-d↑2)/2, where d is the distance between Now York and London on the unlt
      sphere (it turns out to be .875). The ellipse represents the line connoctlng the • ntipodes of
      New York and London--for every point A on it, d(A,New York) 4 d(A,London) = 2π-d. New York will
      be mapped to (0,0). Ue will construct the display with a border--everything between the circle
      of the up and the rectangle of the screen will be "lightened";
    DDINIT;
    SCREEN(-2.5,-3,3.5,3);              ! Scale It a bit;
    LITEN;  RECTAN(-9,-9,9,9);          ! light up the screen;
    DRKEN;  ELLIPS(-2.4,-2.7,3.2,2.7);  ! Darken where the u p will be;
    LITEN;

    ! Plot each land body, then show the map;
    UHILE UDRLD DO BEGIN     DRAWBODY(I LRNDBDDY[WORLD]); WORLD←I REST[WORLD]     END;
    DISPLAYMAP

    END "EQUIDISTANT";

! Ha in program;
WORLD←GETWORLD; MERCATOR (WORLD) ; STEREOGRAPHIC (WORLD) ; EQUIDISTANT(WORLD)
END "MAPS";
```



"A different view of world affairs"

Frank Liang

stereographic projection

Notes for October 21, 1976

Topics of today's discussion:
1. languages, grammars, and representations of grammars;
2. a possible grammar for the language of problem 2.

The program for problem 2 may be viewed as a compiler which parses a sentence in a language (the set of English numerals) into the "code" for the sentence (the number represented by the numeral). As background for this problem, we discussed languages, the grammars defining them, and methods for parsing sentences in a language given a grammar. We then discussed a possible grammar for English numerals.

A *language* is a set of sentences, where each sentence is composed of tokens In some alphabet. One example is English (a *natural* language), for which the alphabet is the set of words and the sentences are English sentences. Another example is Algol; tokens are identifier names, reserved words, and various symbols, and sentences in the language are valid Algol statements.

For a language to be interesting, it must have structure. The *syntax* of a language, specified by rules of one kind or another, provides that structure, along with a first approximation to the meaning of sentences of the language. Syntax rules define how sentences are constructed, often by *productions* which specify the relations between syntactic categories (*nonterminals*) and the tokens of the language (*terminals*). A sample production is <expression> → <term> + <term> which specifies that a nonterminal <expression> may be written as some <term> followed by the terminal "+" followed by another <term>. A *grammar* is defined by its productions, terminals, and nonterminals. We generate sentences in a language by applying rules of the corresponding grammar to some initial nonterminal; we would *parse* a sentence by finding a sequence of applications of rules of the grammar which would produce it.

Grammars fall into several types depending on how general their productions are. A grammar for which every production has the form $\alpha \rightarrow \beta$, where α is a single nonterminal and β is any nonempty string, is called *context-free*. If every production of a grammar is of the form $A \rightarrow aB$ or $A \rightarrow a$, where A and B are nonterminals and a is a terminal, the grammar is called *regular*. (Every regular grammar is obviously also context-free.) Computer languages are often defined by either context-free or regular grammars.

There are several ways to represent the rules of a context-free grammar. One familiar notation is *Backus-Naur* Form *(BNF)* used to describe Algol, in which the rule <a> ::= <b> <c> | <d> means that an instance of <a> is either a <b> followed by a <c>, or a <d>. A simpler notation replaces "::=" by "→" and uses upper-case ktters for nonterminals; the above rule would be represented by the pair of productions A → B C and A → D, or by the single rule A → B C | D. The tatter seems nicer; one of the intended features of BNF was the grouping of productions with the same left-hand side in one rule. A notation might use abbreviations, e.g. A* to represent any number of A's. *Transition nets* are also sometimes used to represent context-free and regular grammars; these are directed graphs which resemble flow charts in that each node represents a point in a parsing program, and the edges are labelled with terminal symbols or nonterminal symbols (the latter representing a subroutine call); the edges are sometimes also labelled with actions to be taken upon parsing the symbol. Similar to transition nets are *state diagrams* (used mostly to describe regular grammars), in which the nodes represent states of the parse and the edges are labelled with the next input character and possibly with what to do when it's seen. A good representation for a problem will stress the psychology of the situation and allow one to see patterns more easily (this Is why if-and-only-if theorems are so popular in mathematics; scientific advances often come from viewing old problems in new lights), and there are applications for which each of the above notations is particularly valuable.

Our parse of a language may be *deterministic* or *nondeterministic*, depending on whether or not it is always clear what action to take next. A nondeterministic parse is usually implemented with a backtracking algorithm. One technique for converting a nondeterministic (ND) parse to a deterministic (D) one is to consider transitions between sets of states instead of single states; an ND parser with n states can thus be converted to *a* D parser with 2^n states. One of the outstanding unsolved problems of complexity theory is whether the set of languages parsable in linear time by a ND parser is the same as the set of languages parsable in linear time by a D parser.

We can try to represent the language of English numerals with a context-free grammar, one rule of which might be <English numeral> → <million numeral> | <teen hundred numeral> | zero. The word "zero" is used here as a terminal symbol, but we need to define our terms; is "zero" itself a symbol, or is it four letters, or does it include surrounding blanks? A good solution is to use a two-level system incorporating a *lexical grammar* or *scanner* to read tokens from the input. The lexical grammar for this problem parses the input text into words. and hyphens, treating blanks as delimiters; the next part of the program would then parse the resulting string of tokens into the corresponding number. (Example: the string "one hundred fifty-two" would be parsed into s, s_{100} s_{50} s_{-} s_{2} ⊣.) Figure I shows a state diagram for the grammar. Each character gets read onty once. With each transition an action is associated; typical actions are recording a character, analyzing a word when a delimiter is hit, and moving on to the next processing stage when end-of-input is encountered. (Alternatively the Sail SCAN function may be used to implement the lexical scanner very easily.)

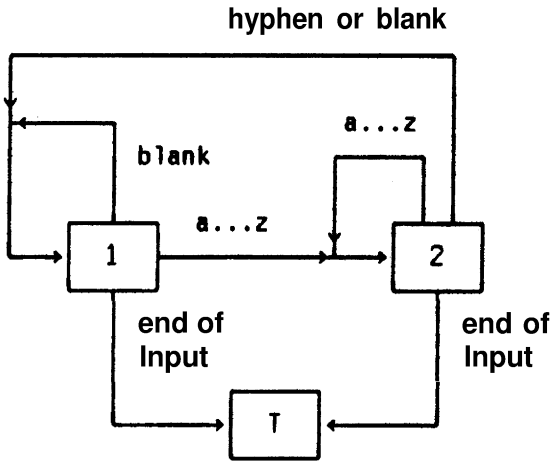


Figure I – state diagram of lexical scanner

The following grammar for English numerals was proposed.

<English #> → <million #> | <teen hundred #> | *zero*

This rule essentially splits up numerals into the nice ones (the millions, thousands, and units) and -the not-so-nice ones. We'll ignore the not-so-nice numerals for the time being.

<million #> → <hundred #> *million* <thousand #> | <thousand #> | <hundred #> *million*

"Million numerals" either specify some number of millions, or they specify some simpler number.

<thousand #> → <hundred #> *thousand* <hundred #> | <hundred #> | <hundred #> *thousand*

"Thousand numerals" are broken down in the same way as millions. Note that "two thousand and one" is not in this language, Stanley Kubrick notwithstanding.

<hundred #> → <digit> *hundred* <ten #> | <ten #> | <digit> *hundred* | <digit> *hundred and* <ten #>

<ten #> → <n-ty #> - <digit> | <n-ty #> | <teen #> | <digit>

<n-ty #> 3 *twenty*, ... *ninety*

<teen #> → *ten*, ... *nineteen*

<digit> → *one*, ... *nfne*

This grammar might be made more elegant by, for instance, allowing some of the above categories to include the null string and then combining right-hand-sides. The empty string is harder to deal with during parsing, but understanding how *a* parsing algorithm handles it is often very helpful for understanding the performance of the algorithm as *a* whole. Also, inelegant exceptions in the grammar are often more easily handled by accepting the corresponding strings and then disallowing them later during semantic processing.

This brings us to the "bad numerals" which we left out of the above grammar. What are examples of such numerals, and how do we handle them? Consider the following:

twenty-one hundred
twenty-one hundred thousand
two million twenty-one hundred
twenty-one hundred thousand twenty-one hundred
twenty-one hundred thirty thousand twenty-one hundred
twenty-one hundred thirty-one thousand twenty-one hundred

Are these proper English? How do we decide? The problem here is *a* common one for computer language designers. [To be continued.]

Topics of today's discussion:
1. defining a language (which constructs should be considered well-formed?);
2. top-down parsing;
3. semantics of a language.

Today we considered "bad" numerals, and discussed criteria for deciding which numerals should be accepted by programs for problem 2. We also discussed methods of parsing a language, and ways to assign meaning to statements of a language.

Which of the following expressions, each of which can be unambiguously parsed, are legal in English? More to the point, what output should the programs we write for problem 2 produce for them? The class voted, yielding the results in parentheses (twenty-two thought "twenty-one hundred" was well-formed, etc.); there were twenty-four students present, two of whom didn't like any numbers on the list.

twenty-one hundred (22)
twenty-one hundred thousand (16)
two million twenty-one hundred (15)
twenty-one hundred thousand twenty-one hundred (6)
twenty-one-hundred thirty thousand twenty-one hundred (6)
twenty-one hundred thirty-one thousand twenty-one hundred (4)

There were several opinions about which numerals should be allowed: (*i*) When there's a choice, the largest possible unit should be the only one allowed ("twenty-one hundred" should be "two thousand one hundred"). (*ii*) "Decomma-izing" idioms may be allowed ("*twenty-one hundred" is okay, "twenty-one hundred thousand" isn't because 2,100,000 is written with commas). (*iii*) Overlap (e.g. "ten thousand twenty-one hundred") should be illegal, but anything scannable left to right such that only zeros are filled in should be allowed. (*iv*) Anything that can be parsed should be legal, since a computer program is more useful if it doesn't impose too many restrictions. These all sound reasonable; how do we choose where to draw the line between meaningful and nonmeaningful strings?

One idea is to parse the input if possible (no matter how strange it sounds), then return both the expression's value and its degree of grammaticality. Also, the motivation behind our application is important. We're not trying to generate flawless English, just trying to recognize what comes in and translate it into some number.

Syntax rules in our program specify which forms have structure and which do not; *semantic* rules will assign the expression a meaning if possible. A well-structured program will probably separate the syntax functions from the semantic. Since we want to differentiate somehow between meaningful and nonmeaningful input, it seems desirable to put the code that does this into the semantic phase of the program, and define the syntax rules to allow as much as we can, easily. In the end, it comes down to programming considerations: the syntax should be as elegant and permissive as possible, but it should also be easily implemented. There is no one best answer to this dilemma; the differing points of view each have some validity. In most real-world applications we are faced with such tradeoffs and the impossibility of finding a truly clean or optimum solution to a given problem; the best we can do is understand the relevant criteria for such decision making.

We can now consider ways to implement syntactic analysis in a computer program. We'll use the grammar discussed last class, listed (in abbreviated form, ignoring the "bad" numerals) in Figure I.

$E \rightarrow M \mid s_0$ (E = English *, M = million *)
$M \rightarrow H\ s_{1000000}\ K \mid K \mid H\ s_{1000000}$ (H = hundred *, K = thousand *)
$K \rightarrow H\ s_{1000}\ H \mid H \mid H\ s_{1000}$
$H \rightarrow d\ s_{100}\ T \mid T \mid d\ s_{100} \mid d\ s_{100}\ s_{and}\ T$ (T = tens *, d = digit)
$T \rightarrow n\ s\_\ d \mid n \mid d \mid a$ (n = n-ty *, a = teen *)

Figure 1 – a grammar for English numerals

Parsing proceeds either in a "bottom-up" or a "top-down" fashion. Bottom-up parsing starts with a sequence of terminal symbots and attempts to "reduce" it to the initial nonterminal symbol (<English *> in problem 2); we defer that until next time. The top-down method works in just the opposite way, starting with the initial nonterminal and applying productions to it which finally yield the input string. Implementation of a top-down parser conceptually involves recursive subroutines, which match the input to various patterns. Such a routine for our sample grammar is listed in Figure 2.

```
Boolean procedure find_thousand#;
        return (find_hundred#_followed_by_s1000_followed_by_hundred#
                or find_hundred#
                or find_hundred#_followed_by_s1000);
```

Figure 2 – procedure for matching a "thousand numeral"

Straightforward top-down parsing has several drawbacks. For one thing, it can't be used with a grammar which contains left-recursive rules, i.e., rules of the form $A \rightarrow A\,B$, since the corresponding procedure *find_A* would contain as its first instruction a call to itself, and an infinite loop would result. A more subtle drawback is illustrated by the *find_thousand#* procedure. If the procedure *find_hundred#* always "finds" a nonnull string, there will be no way to recognize the string "two hundred thousand" since the second alternative would find "two hundred" and never get a chance to include "thousand" in the string being sought. In general, top-down algorithms don't work on grammars which require looking ahead in the source string.

Often, however, the grammar may be transformed so that a top-down algorithm may be used. If the grammar is not *self-imbedding* (i.e. there is no X such that $X \rightarrow^* \alpha\,X\,\beta$ where α and β are nonempty), then it may be transformed (by expanding the right-hand sides) to a regular grammar. The result may be quite large, but, knowing that a regular grammar exists, we can usually start from scratch to construct one with fewer rules. Another useful transformation on grammars is reordering the right-hand sides of rules; for example, changing the "thousands" rule to $K \rightarrow H\,s_{1000}\,H \mid H\,s_{1000} \mid H$ would have fixed the bug mentioned above. A third transformation on grammars is factoring, which allows us to avoid backup during the parse. Factoring the rule $X \rightarrow a\,b \mid a$, for instance, results in the two rules $X \rightarrow a\,B$ and $B \rightarrow b \mid \epsilon$, where ϵ is the empty string. Factoring our English numeral- grammar results in the grammar of Figure 3.

$E \rightarrow M \mid s_0$
$M \rightarrow H\ M'$
$M' \rightarrow K' \mid s_{1000000}\ M''$
$M'' \rightarrow K \mid \epsilon$
$K \rightarrow H\ K'$
$K' \rightarrow K'' \mid \epsilon$
$K' \rightarrow s_{1000}\ K'''$
$K''' \rightarrow H \mid \epsilon$
$H \rightarrow T' \mid d\ H'$
$H' \rightarrow H'' \mid \epsilon$
$H'' \rightarrow s_{100}\ H'''$
$H''' \rightarrow T \mid H'''' \mid \epsilon$
$H'''' \rightarrow s_{and}\ T$
$T \rightarrow d \mid T'$
$T' \rightarrow a \mid n\ T''$
$T'' \rightarrow T''' \mid \epsilon$
$T''' \rightarrow s\_\ d$

Figure 3 – factored version of the grammar of Figure 1

The factoring can be more compactly represented by using parentheses, as in Figure 4.

$E \rightarrow M \mid s_0$
$M \rightarrow H\,(K' \mid s_{1000000}\,(K \mid \epsilon))$
$K \rightarrow H\,K'$
$K' \rightarrow s_{1000}\,(H \mid \epsilon) \mid \epsilon$
$H \rightarrow T' \mid d\,(s_{100}\,(T \mid s_{and}\,T \mid \epsilon) \mid \epsilon)$
$T \rightarrow d \mid T'$
$T' \rightarrow a \mid n\,(s\_\,d \mid \epsilon)$

Figure 4 – compact representation of the grammar of Figure 3

For efficiency, we would like to avoid backup if possible. It is also the case that, although the problem of deciding if top-down parsing will work on an arbitrary context-free grammar is unsolvable, we can determine whether or not top-down parsing with no backing up will work. Hence parsing without backup has theoretical as well as practical interest. Knuth shows in [2] (a tutorial explanation of top-down syntactic analysis) that, for a grammar whose rules are all of the form $X \rightarrow Y_1 \mid \ldots \mid Y_m \mid Z_1 \ldots Z_n$ for $m,n \geq 0$, straightforward top-down parsing with **no backup may be used** if and only if the following conditions hold:

(a) the grammar contains no left-recursive nonterminals;
(b) the sets $first(Y_1), \ldots, first(Y_m), first(Z_1 \ldots Z_n)$ have no letters in common;
(c) if $Z_1, \ldots, Z_n$ can ultimately produce the empty string (or if $n=0$ so that it already is the empty string), then $first(Y_1), \ldots, first(Y_m)$ contain no letters in common with follow(X);
(d) $Y_1, \ldots, Y_n$ correspond to subroutines that are not "nonfalse".

Here *first*(X) denotes the set of terminals that can be first in a string parsable as X, and *follow*(X) denotes the set of terminals that can follow a string parsable as X in a string parsable as E-l. A nonterminal is called *nonfalse* if the corresponding subroutine will never return the value false; for a technical explanation of this condition, see [2].

The above factored grammar without parentheses has all productions in the desired form, and the conditions hold (here ⊣ denotes the end of input) as shown in Figure 5. Therefore the grammar can be used for top-down parsing.

X	first(X)	follow(X)	nonfalse?
E	and s_0	⊣	no
M	and	⊣	no
M'	s_{1000} $s_{1000000}$	⊣	no
M''	and	⊣	yes
K	and	⊣	no
K'	s_{1000}	⊣	yes
K''	s_{1000}	⊣	no
K'''	and	⊣	yes
H	and	s_{1000} $s_{1000000}$ ⊣	no
H'	s_{100}	s_{1000} $s_{1000000}$ ⊣	yes
H''	s_{1000}	s_{1000} $s_{1000000}$ ⊣	no
H'''	dan s_{and}	s_{1000} $s_{1000000}$ ⊣	yes
H''''	s_{and}	s_{1000} $s_{1000000}$ ⊣	no
T	dan	s_{100} $s_{1000000}$ ⊣	no
T'	an	s_{1000} $s_{1000000}$ ⊣	no
T''	$s\_$	s_{1000} $s_{1000000}$ ⊣	yes
T'''	$s\_$	s_{1000} $s_{1000000}$ ⊣	no

Figure 5 – characteristics of the factored numeral grammar

Parsing is, of course, only part of the problem. To assign a meaning to the numeral (its value), we will have a "meaning function"* m, and we will associate a semantic rule with each syntactic production which will define m in the desired way. For example, for the production $H \rightarrow T_1 \, s_{100} \, T_2$, we might have the semantic rule $m(H) = m(T_1) \cdot 100 + m(T_2)$. A nonterminal symbol may be given several *attribute* functions like m; for example, if we want to rate the grammaticalness of a number, we can check for things like overlap or decommaizing by defining a "g" attribute. When a grammar is transformed by factoring, the associated semantic rules may be likewise transformed; additional attributes may be necessary, however, since the transformation "factors" the meaning of the rules. For example, for the production $M \rightarrow H \, M'$, some information must be passed along telling what kind of M' was found, say, by a function b(M') which says if $M' > 1{,}000{,}000$. It is perhaps best to reconstruct the parse tree corresponding to the unfactored grammar, then apply the original unfactored semantic rules.

The function m is an example of a *synthesized* attribute, whose values are based on the attributes of descendants in the derivation tree. *Inherited* attributes are based on attributes of ancestors in the tree. Inherited attributes are useful wherever part of the meaning of some construct depends on the context in which that construct appears ("block structure" in programming languages is such a construct). Knuth explores the simultaneous use of synthesized and inherited semantic attributes in [1].

References:

[1] Knuth, D.E., "Semantics of Context-Free Languages", *Mathematical Systems Theory* 2,2, June 1968.

[2] Knuth, D.E., "Top-Down Syntax Analysis", *Acta Informatica* I, pp. 79-110 (1971).

Notes for October 28, 1976

Topics of today's discussion:

1. parsing with an arbitrary context-free grammar;
2. finding all parses of a given string;
3. bottom-up parsing.

Although transformations (factoring, etc.) can sometimes be applied to allow straightforward no-backup top-down parsing with a context-free grammar, this is not normally the case, so we should look for some other way to remove the inefficiency in a backtracking top-down parsing algorithm. Consider the main source of inefficiency. When the parser backtracks through the parse tree, it throws away everything it's done between the point it failed and the point it makes another choice; processing the second choice, however, may require duplicating much of the work it did the first trip down the tree. A good algorithm should somehow save the results of its computations in case they're needed later.

A highly useful technique based on this idea is *dynamic programming.* It is applicable to any problem whose solution depends on solving subproblems, many of which are identical. In a backtracking algorithm, results of subproblems are recomputed; in a dynamic programming algorithm, they are saved in a table when computed the first time, and looked up thereafter. The resulting speedup depends on how much duplication there is in the subproblems, but is often quite large.

One problem to which dynamic programming may be advantageously applied is that of finding all parses of a string with respect to a given context-free grammar. One might want to do this in a natural language processor, in order to find the parse which was most appropriate for the given context. One might also want to determine if all parses of a given construct were semantically equivalent.

Take, for example, the grammar $S \rightarrow S \cdot S \mid x$; the strings produced by this grammar are x, x•x, x • x•x, etc. The grammar is highly ambiguous; the number of parses of a string with n dots and n+ l x's is binomial(2n,n)-binomial(2n,n-1), the nth Catalan number, which is approximately equal to $4^n/\sqrt{\pi n}$. We will use a dynamic programming approach to find all parses of a string-there are exponentially many-in polynomial time. Supposing the input stream is $a_1 a_2 \ldots a_n$, we will let table entry S_{ij} tell us if $a_i \ldots a_j$ can be parsed as S, and if so how. Since a string in the language is produced by concatenating two other strings, with a dot in between, each subproblem is derived by choosing a dot in the string, parsing the left part, then parsing the right part; hence each possible subproblem is defined by i and j, the indices of the left and right x's for the corresponding string. The table for S_{ij} can be defined as follows:

S_{ii} = true if a_i = "x", *false* otherwise;

S_{ij} = true if there is a k such that $S_{i,k-1}$ is true and a_k = "•" and $S_{k+1,j}$, *false* otherwise.

(We're not worrying about empty strings here.) The above table merely lets us recognize legal strings. If we wanted to parse a string-that is, give its derivation tree-we would let S_{ij} be a list of all essentially different ways that $a_i \ldots a_j$ could be parsed (a list of the k's would work).

For another example, consider the grammar whose rules are $S \rightarrow S \cdot T \mid x$ and $T \rightarrow T\ S\ T \mid y\ x\ S$. We keep two tables this time: S_{ij} tells if (and how) $a_i \ldots a_j$ is parsable as S, and T_{ij} tells if $a_i \ldots a_j$ is parsable as T. The conditions on S_{ij} and T_{ij} are the following:

S_{ij} = true if either i = j and a_i = "x"
or there is a k such that $S_{i,k-1}$ and $T_{k+1,j}$ are true and a_k = ".",
false otherwise;
T_{ij} = *true* if either a_i = "y" and a_{i+1} = "x" and $S_{i+2,j}$ is true
or there are k and l for which $T_{i,l-1}$ and $S_{l,k-1}$ and $T_{k,j}$ are all true,
False otherwise.

We can compute all S_{ij} and T_{ij} in increasing order of j-i: first we fill in S_{ii} and T_{ii}, then S_i and T_i, etc. Finally we have evaluated S_{1n} and T_{1n}, after filling about $2(n^2/2)$ table entries. Filling in S_{ij} requires checking all k's between i and j, in linear time; for T_{ij} all pairs (k,l) between i and j, k<l, must be checked, in $O(n^2)$ time. Hence the amount of work needed to recognize a string of length n is $O(n^4)$, while a backtracking algorithm would have taken exponential time. The algorithm can still be asymptotically improved by reducing the $O(n^2)$ in the computation of T_{ij} to O(n); we add a new nonterminal U, and replace the rule $T \rightarrow T\ S\ T \mid y\ x\ S$ by the two rules $T \rightarrow T\ U \mid y\ x\ S$ and $U \rightarrow S\ T$. We then have an extra table to fill, but both the T and U tables can now be filled in time O(n). The "improved" algorithm takes time $O(n^3)$ overall, although for reasonably sized n it will be slower than the $O(n^4)$ version.

We could use such an algorithm for the numeral grammar of problem 2, by keeping a meaning table E_{ij} instead of a parse table. For instance, E_{1n} could be the value of $a_1 \ldots a_n$ if it were parsable as E and -1 if not.

The above procedure is, of course, general; *any* context-free grammar can be parsed in $O(n^3)$ time. This bound has been lowered still farther by Leslie Valiant; by showing (in [6]) that finding all parses was equivalent to matrix multiplication, he lowered the bound to $O(n^{2.8})$, but his method is not of practical importance. Unfortunately no one knows of any context-free language which requires nonlinear time to parse-so there's quite a gap, between O(n) and $O(n^{2.8})$. Sheila Greibach [3] has found a particular context-free language L which is as hard as possible to parse: if her language L can be parsed efficiently, so can all context-free languages. Incidentally, L is an easy language to comprehend, but apparently hard to parse. The best general algorithm for parsing context-free languages was devised by Jay Earley [2]. Its worst-case is $O(n^3)$; it runs in time $O(n^2)$ on unambiguous grammars and in linear time for a large class of grammars.

An alternative to top-down parsing is *bottom-up* parsing, which is excellently explained by Terry Winograd in [7]. Winograd also offers a clear description of the various types of grammars and representations for grammars. There have also been many good papers describing the application of parsing techniques to compiler writing: for instance, Melvin Conway's use of transition nets in a Cobol compiler [1], and any of the papers reprinted in part four of Saul Rosen's book [5].

The top-down and bottom-up approaches may also be applied to problem solving. For example, in proving a theorem, we might decompose it into lemmas which when proved would yield the final result. -Or we might first start with all the given information and begin to prove everything that follows as "consequences", dealing with the problem at "bit-level" in order to familiarize ourselves with the domain. In most cases, we actually use a combination of the two approaches. When we start solving a problem, we have some idea of easy subproblems, from which we advance in both top-down and bottom-up ways. When we write a program, we often do the same thing.

A large portion of research in artificial intelligence and combinatorial algorithms concerns *heuristic search,* ways to have a computer program move efficiently from problem statement to solution. Search techniques proceed top-down, finding all possible subgoals that could lead to the desired goal, or bottom-up from the problem statement, first finding lots of trivial things, then less trivial ones, and so on. Ira Pohl [4], among others, has proposed methods of bidirectional search, in order to reduce fan-out of the search trees by building them toward each other.

References:

[1] Conway, ME., "Design of a Separable Transition-Diagram Compiler", *CACM 6*,7 (July 1963), pp. 396-408.

[2] Earley, J., "An Efficient Context-Free Parsing Algorithm", *CACM 13*,2 (February 1970), pp. 94- 102.

[3] Greibach, S., "The Hardest Context-Free Language", SIAM *J. Computing 2* (1973), pp. 304-310.

[4] Pohl, I., "Bidirectional Search", *Machine Intelligence* 6, B. Meltzer and D. Michie (eds.), Edinburgh Univ. Press, Edinburgh (1971), pp. 127-140.

[5] Rosen, S. (ed.), *Programming Systems and Languages,* McGraw-Hill, New York, 1967.

[6] Valiant, L.G., "General Context-Free Recognition in Less Than Cubic Time", *J. Computer and System Sci. 10* (1975), 308-315.

[7] Winograd, T., *Language as a* Cognitive *Process* (draft), notes for CS 265-266, 1975.

Comments on solutions to problem 2

Programs and writeups were graded on the following criteria:
(a) correctness;
(b) niceness of the grammar;
(c) program organization; and
(d) documentation and style.
I was gratified to see more comments in your code.

Most solutions used the top-down technique of *recursive descent;* for each nonterminal U, the program contains a procedure which parses phrases for U. (For an arbitrary context-free grammar, these routines would contain recursive calls, but the simplicity of the numeral grammar made recursive calls unnecessary.) With each such procedure is associated one or more semantic rules to be applied when the parse succeeds (see notes of October 26); the rules define the "value" of the numeral or numeral segment associated with the nonterminal being parsed, and in some cases kept track of other information to aid error diagnosis.

Several of you factored the grammar (cf. October 26 notes), either explicitly or implicitly, in order to eliminate backup. If the parser doesn't have to backtrack, it also doesn't have to undo any of its semantic processing (semantic backtracking can be expensive). Bengt Aspvall implemented the parser described in the notes of October 28, which incorporated *a* dynamic programming algorithm to build the parse tree. Others ignored the backup problem and either maintained a stack of semantic information, or built a semantic tree to process later, or let Sail's recursion do the book keeping for them.

The vocabulary and grammar for numerals were simple enough that your routines could make tests for most of the possible errors; some of you did this more cleanly than others. Since almost all of you treated the hyphen as an operator, hyphen errors were easy to detect and diagnose as such. Other diagnosable errors were too large a number and misplaced "and" or "zero". Error correction was harder to implement; Greg Nelson tried, producing error messages and computing the correct results for the following:

thirty-zero	THE "-ZERO" SHOULD BE OMITTED
one thousand hundred	IT IS VERY UNUSUAL TO MODIFY "HUNDRED" WITH A NUMBER EXCEEDING NINETY-NINE
fifty thousand duodecillion	IT IS NOT CUSTOMARY TO MODIFY "DUODECILLION" BY A NUMBER EXCEEDING ONE THOUSAND
ninety hundred	"NINETY HUNDRED" SOUNDS TERRIBLE. SAY "NINE THOUSAND" INSTEAD.

The representation Greg used may have made his error correction easier. He treated "million", "thousand", "hundred", "hyphen", and "and" as operators with precedence, and parsed the resulting operator precedence grammar. (See [1] for a description of operator precedence grammars.)

An alternative to top-down parsing is bottom-up parsing, and several solutions used this method. The numeral grammar may be transformed to a finite-state grammar, and a bottom-up recognizer written to parse the numeral strings merely by stepping from transition to transition based on the input. The input may be parsed in either direction; the left-to-right method requires the maintenance of extra semantic information, either explicitly keeping track of what's been seen or determining it from the values computed so far. The solution program uses the latter method. Kevin Karplus generalized the transition net approach, writing a program to read in an arbitrary set of transitions and parse according to that set. Jim Davidson generalized the same approach a bit more, programming a parser to handle arbitrary *recursive transition network* grammars [2].

The compiling techniques mentioned above are described in better detail in several books on compiling, e.g. [1].

Most programs had scanners to transform the input from strings to "tokens". Scanning procedures are usually useful because it's easier to work with integers than with strings. Also those programs which did not include a scanner had to be structured carefully to avoid treating, e.g., "eighteen" as "eight". Everyone but Bengt Aspvall used gigantic if statements or table lookup to scan symbols; Bengt discovered the feature of Sail (the CVSI function) which uses an internal hash function to associate item names with strings. Bugs in your programs included failure to catch misplaced "and", failure to reject "eleven hundred million" as too large, and scanner errors.

The sample solution contains a few interesting features. The approach is bottom-up; two consecutive tokens, along with semantic information, are examined at each step. The scanner treats a hyphenated numeral as a single token; Kevin Karplus was the only other person to do this. I think I was able to code my parser more cleanly by making the scanner do more work; I also think that "twenty-one" is just as much a "primitive" quantity as "twenty", and they should be handled equivalently.

Feature list:

recursive descent: MJB, AVG, RXM, MFP, WP, JRR, AMR, DAN, AZS, CLT, IAZ/MI
factored grammar (implicit or explicit): MJB, AVG, MFP, WP, JRR, AMR, CLT, IAZ/MI
bottom-up: MP, BTH/REP, KJK, ARS
operator precedence: CGN
right-to-left: IAZ/MI, MP, K JK
table construction: BIA
arbitrary recursive transition network grammars: JED

References:

[1] Cries, D., *Compiler Construction for Digital Campers,* Wiley, New York, 1971, chapters 4-6.
[2] Woods, W.A., "Transition Network Grammars for Natural Language Processing," *Comm. ACM 13,10* (October 1970), pp. 591-606.

```
BEGIN "NUMBER"

DEFINE  !="COMMENT",  TIL="STEP 1 UNTIL",  NONE="NOT", NO="NOT", STARTLOOP="WHILE TRUE DO",
        TAB='11, CRLF="('15&'12)";

REQUIRE "<><>" DELIMITERS;

INTEGER PAST_BLANKS, UP_TO_HYPHEN, UP_TO_BLANK;

PROCEDURE INITBREAKTABLES;
!Initialize some useful break tables. All tho variabirs are global;
    BEGIN "INIT BREAK TABLES"
    SETBREAK(PAST_BLANKS←GETBREAK," "&TAB, NULL, "XKR");
    SETBRERK (UP_TO_HYPHEN←GETBREAK,"-", NULL, "IKS");
    SETBRERK (UP_TO_BLANK←GETBREAK,""&TAB, NULL, "IKS");
    END"INIT BREAK TABLES";

STRING PROCEDURE INPUTFROMUSER;
    BEGIN "INPUTFROMUSER"
    PRINT("Type a numeral foiiouod by <return>, or just <return> if finished.",CRLF);
    RETURN(INCHWL)
    END "INPUTFROMUSER";

STRING NUMERAL;          ! fhe string which the user submits;

! Explanation of thr program:

  This parser recognizes and evaluates numerals formed by the following syntax.

  <0-999999999> → zero | <1-999> million [<1-999999>] | c1-999999,
  <1-999999> →     <1-999> thousand [<1-999>] | <1-9999>
  <1-9999> →       <teen> hundrrd [[and] <1-99>] | <hyph> hundrrd [[and] <1-99>] | <1-999>
  <1-999> →        <unit> hundred [[and] <1-99>] | <1-99>
  <1-99> →         <unit> | <teen> | <nty> | <hyph>

  We may expand tho above grammar to get thr following.

  <0-999999999> → zero
                  | <1-999> million
                  |<1-999> mi l l ion <1-999> thousand
                  |<1-999> mil l ton <1-999> thousand <1-999>
                  |<1-999> mi l l ion d-99997
                  |<1-999> thousand
                  |<1-999> thousand <1-999>
                  | d-99997
  <1-9999>, <1-999>, <1-99> a s above.

  While parsing, the program keeps track of srmant ic informat ion In the variables NUMBER and TEMP.
  NUiIBER contains the vaiur seen up through the iast "niiiion" or "thousand", while TEMP contains the
  yaiue tarn since the last "ml ii ion" or "thousand", TEMP is formed uhiir parsing <1-99>, <1-999>, or
  <1-9999>, and the value of TEMP indicator which of these has just been parsed (TEMP=0 => neither,
  TEMP>999 => <1-9999>, 100≤TEMP≤999 => x1-9997, otherwise <1-99>). If TEMP≥100, "hundred" has been
  seen. 1 f(TEMP mod 100)≠0, we've just scanned <1-99>. NUMBER=0 If neither "million" nor "thousand"
  has been seen, NUMBER≥1000000 if "million" has been seen, and (NUMBER mod 1000000)≠0 if "thousand"
  h a s been seen. This semantic information ● nabios us to drtrct illegally specified numerals a s
  foi iows.
  1. "million" must always be preceded by <1-999>, and no other "million" or "thousand" may precede
     it in the string. Hence 1≤TEMP≤999 and NUMBER=0 for "million" to be legal.
  2. "thousand" must also be prrcrdrd by <1-999>, and no othrr "thousand" may precede It. Hence
     1≤TEMP≤999 and (NUMBER mod 1000000)=0.
  3. Only <teen> or <hyph> or <unit> may precede "hundred". Hence 1≤TEMP≤99 and (TEMP mod 10)≠0. A lso,
     if "thousand" has been seen, only <unit> may precede "hundrrd", so if (NUMBER mod 1000000)≠0 than
     1≤TEMP≤9.
  4. Numbers loss than 188 cannot be adjacent. When <1-99> It scanned, (TEMP m o d 100)=0.

  Ftdditionai information is krpt in LASTTYPE to ● nabio us to detect syntactic errors. LASTTYPE could
  be used to detect some of the errors mentioned above (here we venture into the murky land dividing
  syntax and semantics);
```

```
! GETNEXTTOKEN and token table definitions;

! Total number of tokens;
    DEFINE !N=32;

! Possible token types-nicar to refer to them symbolically;
    DEFINE !BEGINTYPE=0,!ENDTYPE=8,
           !ZEROTYPE=1, !UNITTYPE=2, !TEENTYPE=3, !NTYTYPE=4, !HYPHTYPE=5,
           !HUNDTYPE=6, ITHOUTYPE.7, !MILLTYPE=8,!ANDTYPE=9,!ERRORTYPE=10;

! Useful macro-somewhat Inefficient but clear;
    DEFINE RETURNTOKEN (TYPE,VAL)=cBEG IN TOKENTYPE←TYPE; TOKENVAL←VAL; RETURN END>;

PROCEDURE GETNEXTTOKEN(REFERENCE STRING INPUTSTRING; REFERENCE INTEGER TOKENTYPE, TOKENVAL);
! This procedure roads the next symbol from the front of INPUTSTRING and looks It up In the symbol
  table. The type and value of the symbol are returned In the TOKENTYPE and TOKENVAL variables. All
  types (including the error type, for when an unrecognized symbol Is scanned) are positlve integers.
  The value of "zero" Is 0, and other values are positive. Thr token may be a hyphenated numeral, in
  uhlch case its value Is bull t from Its component numerals. If there are no symbols leftin 9, zero
  Is returned In both TOKENTYPE and TOKENVAL;

    BEGIN "GET NEXT TOKEN"

    INTEGER PROCEDURE INDEXOF (STRING S; STRING ARRRY TABLE );
    ! Returns the Index of tha glven string In the given array, or 8 If it's not therm, The array Is
      assumed to be single-dimensioned, Brn for n>0 (note the crafty optimization);
        BEGIN "INDEX OF"
        INTEGER I;
        I←ARRINFO(TABLE,2)+1;     ! For almost-complete generality;
        TABLE[0]←S;               ! Search will always succeed)
        DO I←I-1 UNTIL EQU(S,TABLE[I]);
        RETURN(I)
        END "INDEX OF";

    PRELOAD_WITH
        NULL, "ZERO", "ONE", "TWO", "THREE", "FOUR", "FIVE", "SIX", "SEVEN", "EIGHT", "NINE", "TEN",
        "ELEVEN", "TWELVE", "THIRTEEN", "FOURTEEN", "FIFTEEN", "SIXTEEN", "SEVENTEEN", "EIGHTEEN", "NINETEEN",
        "TWENTY", "THIRTY", "FORTY", "FIFTY", "SIXTY", "SEVENTY", "EIGHTY", "NINETY",
        "HUNDRED", *THOUSAND", "MILLION","AND";
    OUN STRING RRRRY TOKENS[0:!N];

    PRELOAD_WITH
        !ZEROTYPE,[9] IUNITTYPE, [10]!TEENTYPE, 181 !NTYTYPE,
        !HUNDTYPE, !THOUTYPE, !MILLTYPE, !ANDTYPE;
    DUN INTEGER ARRRY TOKTYPES[1:!N];

    PRELOAD_WITH
        8, 1, 2, 3, 4, 5, 6, 7, 8, 9, 18, 11, 12, 13, 14, 15, 16, 17, 18, 19,
        28, 30, 48, 58, 68, 70, 80, 98, 188, 1000, 1000000, 0;
    OUN INTEGER ARRAY TOKVALS[ 1 :!N];

    STRING S, S1; INTEGER BRERKCHR, I, N1,N2;

    SCAN(INPUTSTRING,PAST_BLANKS,BREAKCHR);          ! Skip blanks, then read symbol;
    S←SCAN(INPUTSTRING,UP_TO_BLANK,BREAKCHR);
    IF NO 9 THEN RETURNTOKEN(0,0);                   ! There was no symbol;

    I←INDEXOF(S,TOKENS);
    IF I THEN RETURNTOKEN(TOKTYPES[I],TOKVALS[I]);   ! Found it;

    S1←SCAN(S,UP_TO_HYPHEN,BREAKCHR);
    IF NO BRERKCHR THEN RETURNTOKEN(!ERRORTYPE,0);   ! Not In table and not hyphenated1

    N1←INDEXOF(S1,TOKENS); N2←INDEXOF(S,TOKENS);
    IF NO N1 OR NO N2 OR TOKTYPES[N1]≠!NTYTYPE OR TOKTYPES[N2]≠!UNITTYPE THEN
        RETURNTOKEN(! ERRORTYPE, 0);                 ! One of hyphenated parts Is bad;

    RETURNTOKEN (!HYPHTYPE,TOKVALS[N1]←TOKVALS IN21 )   ! legal hyphenated numeral;
    END "GET NEXT TOKEN";
```

```
!Ma in program;

DEFINE ERROR(S) = cBEGIN PRINT(S,CRLF); CONTINUE "NUMERAL LOOP" END>;

INITBREAKTABLES;
WHILE NUMERAL←!INPUTFROMUSER DO
    BEGIN "NUMERAL LOOP"
    INTEGER TOKENTYPE, TOKENVRL, LASTTYPE,TEMP,NUMBER;
    TEMP←NUMBER←0;  TOKENTYPE←!BEGINTYPE;
    STARTLOOP
        BEGIN "TOKEN LOOP"
        LASTTYPE←TOKENTYPE;
        GETNEXTTOKEN(NUMERAL,TOKENTYPE,TOKENVAL);
        CASE TOKENTYPE OF
            BEGIN "TOKEN CRSES"

[!ERRORTYPE]    ERROR("Unrecognized symbol.");

[!ZEROTYPE]     IF LASTTYPE≠!BEGINTYPE THEN ERROR("Misplaced ZERO.")
                ELSE    BEG IN
                        GETNEXTTOKEN (NUMERAL , TOKENTYPE, TOKENVAL);
                        IF TOKENTYPE≠!ENDTYPE THEN ERROR("Misplaced ZERO.") ELSE DONE
                        END;

[!ENDTYPE]      IF LASTTYPE=!ANDTYPE THEN ERROR("Numeral can't end with AND.") ELSE DONE;

[!ANDTYPE]      IF LASTTYPE≠!HUNDTYPE THEN ERROR("AND must follow HUNDRED.");

[!HUNDTYPE]                 --
[!THOUTYPE]
[!MILLTYPE]     IF LASTTYPE=!ANDTYPE THEN ERROR("Illegally placed AND.")
                ELSE CASE TOKENTYPE OF
                    BEGIN "HTM CRSES"

    [!HUNDTYPE]         IF TEMP=0 OR TEMP>99 THEN ERROR("Illegally placed HUNDRED.")
                        ELSE IF (TEMP MOD 10)=0 THEN ERROR("Numeral is like twenty hundred not allowed.")
                        ELSE IF (NUMBER MOD 1000000)≠0 AND TEMP>9 THEN ERROR ("Hundreds over lap thousands.")
                        ELSE TEMP←TEMP*100;

    [!THOUTYPE]         IF TEMP=0 THEN ERROR("Something must precede THOUSAND.")
                        ELSE IF TEMP>999 THEN ERROR("Numeral is like eleven hundred thousand not allowed.")
                        ELSE IF (NUMBER MOD 1000000)≠0 THEN ERROR("Thousands of thousands not allowed.")
                        ELSE    BEGIN
                                NUMBER←NUMBER+TEMP*1000;
                                TEMP←0
                                END;

    [!MILLTYPE]         IF TEMP=0 THEN ERROR("Something must precede MILLION.")
                        ELSE IF NUMBER THEN ERROR("Too large.")
                        ELSE IF TEMP>999 THEN ERROR("Too large.")
                        ELSE    BEG IN
                                NUMBER←TEMP*1000000;
                                TEMP←0
                                END

                    END "HTM CRSES"!

[!UNITTYPE]
[!TEENTYPE]
[!NTYTYPE]
[!HYPHTYPE]     IF TEMP MOD 188 THEN ERROR("Illegal numeral.")
                ELSE TEMP←TEMP+TOKENVAL

            END "TOKEN CRSES"
        END "TOKEN LOOP";

    NUMBER←NUMBER+TEMP;                 !Finish off the number;
    PRINT("Value is ",NUMBER,".",CRLF)
    END "NUMERAL LOOP"
END "NUMBER";
```

Notes for November 2, 1976

Topics of today's discussion:
1. preliminary approaches to problem 3;
2. height-balanced trees,

For problem 3, a data structure for a list of items must be designed so that each of the following three operations on the kth element will take only O(log k) steps: accessing the kth item, inserting a given item just before the kth item; and deleting the kth item. We discussed possible data structures today, concentrating on height-balanced trees, which seem particularly appropriate to the problem.

Problem 3 may require some adjustment in thinking for the typical computer science student. It's a purely theoretical problem, calling for a data structure which can be updated in time proportional to log k; note that the data structure need not be simple or practical, and the constant multiplier for the log k need not be very small. (Of course everything can be tuned up later if possible.) Such considerations are typical of research in complexity theory. Studies of an problem's complexity find asymptotic bounds on its running time or space; only when the lower bound is found to grow as fast as the upper bound are the constant coefficients studied. The idea is to find the best way to solve a class of problems as the size of those problems gets large, which involves first finding the order of magnitude of an optimal solution. The practical value of complexity theory is that it helps us discover bottlenecks in our algorithm; and it also makes us feel secure when we use *a* "best" algorithm. Lower-bounds are determined in various ways, but upper bounds are usually found by constructing a particular algorithm to solve the problem. That's what we're doing here; so the same kind of thought processes that we ordinarily use for algorithm design are involved except that we are "thinking big" (i.e. asymptotically) and at a somewhat more abstract level.

One way to start is to consider data structures which can be updated in O(log n) time, where n is the size of the data structure. A tree would work, if it's built right. We will speak of each of the nodes (or records) of a tree as having several fields: *x.info* is the contents of the *info* or *key* field of node x; *x.L* is the root of x's left subtree; and *x.R* is the root of x's right subtree. We wish the ordering of the tree to give some indication of the relation between the various keys. One way is to make sure that the key at the head of x's left subtree is less than the key of x (vice versa for the right subtree), but this is not enough; we must require ail the nodes in x's left subtree to have keys less than x.info, and all the keys in x's right subtree exceed x.info. It's easy to search for a key in such a tree, and the algorithm in Figure I does so.

```
if x is empty then return(false)
else if key = x.info then return(true)
else if key < x.info then set x to x.L and repeat
else if key > x.info then set x to x.R and repeat;
```

Figure I – searching in a binary search tree

In order to be able to search the tree by rank, we must store the rank of each node somewhere in the node. We could explicitly store the rank, but that would require too much work when inserting (a node inserted at the front of the list changes the rank of *all* the other nodes). Better is to store the rank so that only the nodes on the path from the root to the point of insertion would be affected. One way of doing this is to kt the *rank* field of x contain the number of nodes in the tree of which x is the root; the position of x in the list is then the rank of the left subtree, plus 1. Even more efficient is to let x.rank contain x's rank restricted to its own subtree, nameiy 1 plus the size of its left subtree.

```
procedure INSERT( integer POSITION; ptr(NODE)NEWNODE; reference ptr( node) TREE) ;
        if TREE then
                if POSITION ≤ RANK[ TREE ] then
                comment Update rank of current node to account for insertion in front of it, and
                        insert new node in left subtree;
                        begin
                        RANK[ TREE ] ← RANK[ TREE ] + 1 ;
                        INSERT( POSITION, NEWNODE, L[ TREE ])
                        end
                else
                comment Insert new node in right subtree, in position defined by position
                        relative to root of TREE;
                        INSERT( POSITION-RANK[ TREE ], NEWNODE, R[ TREE 'J)
        comment Otherwise we've reached a leaf and POSITION must be 1;
        else TREE ← NEWNODE;
```

Figure 2 – insertion into a tree by rank instead of by key

A Sail procedure for insertion of an element just before a given position in the list, using the data structure we have just defined, is given in Figure 2. Note that insertions only take place at leaves of the tree. Note also that this algorithm is recursive, as are many algorithms for processing trees, but that it may be converted to an iterative algorithm by replacing the recursive calls by gotos to the beginning of the procedure (at a cost of making the code less clean). It seems that the updating of RANK[TREE] should be interchangeable with the call to INSERT, and a good reason for having it follow the INSERT call would be to make sure the insertion has been done before updating the data base. However, were the two statements switched, the recursion-to-iteration optimization could not be made. RANK need not be updated in INSERT at all; instead, a second phase could go through the tree and update ail the affected nodes at once.

For random data, the INSERT algorithm will produce good search trees; a sequence of insertions at the front of the list, however, would essentially reduce the case to that of a linear list. We might, then, consider ways to keep the tree balanced, in order to keep the work we do for any one node down to log n. A data structure which serves this purpose *is* a ***height-balanced tree*** (also called an ***AVL*** tree for its originators, the two Russians Adel'son-Vel'skii and Landis). By the height of a tree we mean the length of the longest path from the root to a leaf; in a height-balanced binary tree, the heights of the left and right subtrees of any node differ by no more than one. (Knuth discusses height-balanced trees in [3], calling them simply "balanced trees".)

We would therefore like to show that the height of a height-balanced binary tree t with n nodes is no more than O(iog n), that is, that there is a c such that h(t) ≤ c log n. It's sometimes better to turn such a problem around: instead of examining all balanced trees with n nodes and showing that their height cannot be large, we might assume we have a tree of a certain height and show that the number of nodes cannot be small. The restated problem is to see, given h which represents the height of some balanced tree, how small n can get.

A good idea when starting to think about a mathematical problem is to think about some simple cases. Simple cases for this problem are small trees. The smallest tree with height 0 has no nodes, the smallest with height 1 has one node, and the smallest with height 2 has two nodes. In the smallest tree of height 3, one subtree would have height 2 and the other height at most 2; since the tree is balanced, the height of the other subtree is at least 1. Constructing the smallest S-height-tree from the smallest 2- and I-height trees, we see it has four nodes. For h-4, n≥7.

We now have the sequence of n values 0,1,2,4,7, corresponding to h=0,1,2,3,4. The sequence isn't immediately recognizable, although we might notice that consecutive differences form the Fibonacci sequence. We also notice a recurrence for n values: n_h, the minimal number of nodes in a balanced tree with height h, is $n_{h-1} + n_{h-2} + 1$ (the last term represents the node for the root). This recurrence is easier to work with if we add 1 to each side; we then have $(1+n_h) = (1+n_{h-1}) + (1+n_{h-2})$. Sure enough, $n_h = F_{h+2}-1$, where F_m is the mth Fibonacci number. Some facts about Fibonacci numbers let us finish the proof: For positive m, $F_m \geq \phi^{m-1}$, where ϕ is the golden ratio $(1+\sqrt{5})/2$, a number which satisfies $\phi^h = \phi^{h-1} + \phi^{h-2}$. Hence $n \geq F_{h+2}-1 \geq \phi^{h+1}-1$, and $h \leq \log(n+1)/\log \phi - 1$. The constant c can be easily computed from the latter relation.

Preserving the balance of the tree requires some extra work when inserting new nodes; the tree needs to be shifted at points of imbalance, starting from the bottom up. There are two cases to consider, both displayed in Figure 3: the insertion is made in the right subtree, rooted at B, and either the right subtree or the left subtree of B is higher. (Other cases are reflections of these two.) The tree is rotated as shown in Figure 3. To keep track of how balanced a tree is, we need an extra field In each record containing the height of the corresponding subtree. Better yet, if space is a crucial consideration, we might make the field a "balance indicator" with value +1 if the right subtree is higher (by 1), -1 if the left subtree is higher, and 0 if the two subtrees are equally high. Such refinements are unimportant when we are only looking for order-of-magnitude upper bounds, however; we are instead looking for clean algorithms, not optimized when the optimization provides only at most bounded speedup. For similar reasons we need not change recursion to iteration if we don't mind a bounded slowdown.

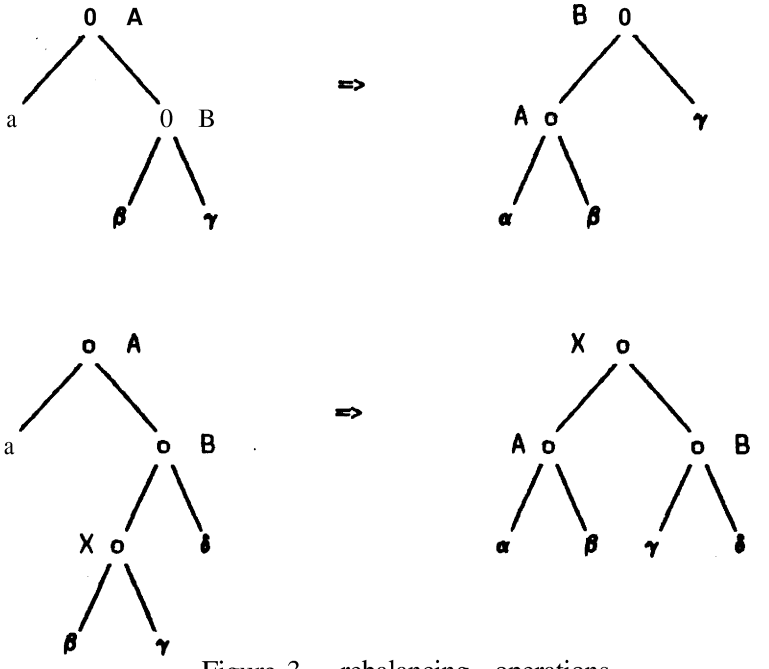


Figure 3 – rebalancing operations

The rebalancing operation preserves the symmetric ordering, and affects no nodes of the tree above node A since the height of the new tree is the same as the height of A was. This property assures that rebalancing will not propagate through a non-zero-balanced node, a fact which will be used in the insertion algorithm in Figure 4.

```
procedure SINGLELEFTROTATE(reference ptr(NODE) A);
! Counterclockwise single rotation.  Former root is A, new root will be B;
    begin "SINGLELEFTROTRTE"
    p tr (NODE) B;
    B←!R[A];      ! Point B at now root, thrn adjust baianco, ranks, and links;
    !BALANCE[A]←! BALANCE [B]←0;    ! RANK [B] . !RANK[A]+! RRNK [B];
    !R[A]←!L[B];   !L[B]←A;
    A←B           ! Ropiace root;
    end "SINGLELEFTROTRTE" ;

procedure DOUBLELEFTROTATE(reference ptr(NODE) A);
! Counterclockwise double rotation,  Former root is A, new root will be X. B In Figure 3 It !R[A];
    beg in "OOUBLELEFTROTRTE"
    p tr (NODE) Xi
    X←!L[!R[A]];          ! Point X at new root, thrn adjust balances, ranks, and links;
    if !BALANCE[X]=0 then !BALANCE[A] . !BALANCE[!R[A]]←0
     else if !BALANCE[X]=-1 then begin !BALANCE[A]←0; !BALANCE[!R[A]]←1 end
       else begin !BALANCE[A]←-1; !BALANCE[!R[A]]←0 end;
    !BALANCE[X] ● @i
    !RANK[!R[A]]←!RANK[!R[A]]-!RANK[X];   !RANK[X]←!RANK[X]+!RANK[A];
    !L[!R[A]]←!R[X];   !R[X]←!R[A];   !R[A]←!L[X];   !L[X]←A;
    A←X                    ! Replaco root;
    end "DOUBLELEFTROTATE";

procedure REBALANCE(reference ptr (NODE) A; Boo lran DIRECTION) ;
! Decides how to rebalance thr tree.  DIRECTION is the direction of imbalance;
    begin "REBALANCE"
    If DIRECTION=LEFT thrn       ! Tree is left-heavy so rotate clockwise;
        if !BALANCE[!L[A]]=-1 thrn SINGLERIGHTROTATE(A) else DOUBLERIGHTROTATE(A)
     else                        ! Tree is right-heavy so rotate counterclockwise;
        if !BALANCE[!R[A]]=1 then SINGLELEFTROTATE(A) else DOUBLELEFTROTATE(A)
    end "REBALANCE";

recursive Boo loan procrdurr INSERT(integer POSITION; ptr (NODE) NEWNODE;
                                    reference ptr (NODE) TREE) ;
!· This routine inserts the node NEWNODE into the balanced tree TREE Just before positlon POSITION.'
  The neu node is always inserted as a loaf.  INSERT returns true or false based on whether or not the
  insertion increases thr height of TREE;
   begin "INSERT"
    I f TREE then          ! Not yet down to leaf level—decide which subtree to insert Into;
        if POSITION≤!RANK[TREE] then     I Insert into left subtree;
            begin
            !RANK[TREE]←!RANK[TREE]+1;   ! One more element in loft subtree;
            I f INSERT(POSITION,NEWNODE,!L[TREE]) then
            ! Height of left subtree has increased;
                I f !BALANCE[TREE]=-1 then        ! Tree is too left-heavy;
                    begin REBALANCE(TREE,LEFT); RETURN(false) end
                 else I f !BALANCE[TREE]=0 then ! Tree is left-heavy enough;
                    bog in !BALANCE[TREE]←-1; RETURN(true) end
                  else                          ! Tree was right-heavy;
                    begin !BALANCE[TREE]←0; RETURN(false) end
            I Otherwise subtree height didn't increase—just pass the word on;
             else RETURN(false)
            end
       else     ! Do the same things for tho right subtree;
            i f INSERT(POSITION-!RANK[TREE],NEWNODE, !R[TREE]) then
                if !BALANCE[TREE]=1 then begin REBALANCE(TREE,RIGHT); RETURN(false)  ● nd
                 else if !BALANCE[TREE]=0 then brgin !BALANCE[TREE]←1; RETURN(true) end
                  ● ise begin !BALANCE[TREE]←0; RETURN(false) rnd
             else RETURN(false)
     else       ! We've reached the bottom of the tree [with POSITION=1), so Insert the new node;
        beg in TREE←NEWNODE; RETURN( true) rnd
    end "INSERT";
```

Figure 4 – Sail code for insertion into a height-balanced tree

One might wonder whether relaxing the height-balance requirement would speed up operations on the tree. Karlton et al. [2] described the performance of trees in which the subtree heights may differ by $k \geq 1$, and came to the conclusion that I is best. Other kinds of balancing have also been investigated. Aho, Hopcroft, and Ullman [1] describe algorithms for 2-3 *trees*, for which each non-leaf has two or three descendants and every path from the root to a leaf is of the same length. Insertion, deletion, etc. aye analogous to the operations for height-balanced trees. Knuth in [3] also discusses weight-balanced trees, for which the number of nodes in the right subtree approximately equals the number of nodes on the left. Such trees may require slightly less space than height-balanced trees, but usually the weight balance is harder to maintain.

References:

[1] A ho, A.V., Hopcroft, J.E., and Ullman, J.D., *The Design* and *Analysis of Computer* Algorithms, Addison-Wesley, Reading, Mass., 1974, sec. 4.9-4.12.

[2] Karlton, P.L., et al., "Performance of Height-Balanced Trees", *Comm.* ACM *19*, I (Jan. 1976), pp. 23-28.

[3] Knuth, D.E., *The* Art *of Computer Programming*, Addison-Wesley, Reading, Mass., 1973, sec. 6.2.3.

Notes for November 4, 1976

Topics of today's discussion:
1. more ways to approach problem 3;
2. a related problem: counting in base 2 in bounded time per step;
3. a case study of problem solving.

For problem 3 we wish to find a data structure for which the operations of insertion, deletion, and search for the kth highest element all take time O(log k). Last time we discussed balanced trees, for which these operations take time O(log n), where n is the size of the tree; today we discussed modifications to balanced trees which might improve the running time. We also considered today the problem of counting in base 2, doing a constant amount of work at each step; the problems are related because the carry operation in addition and the borrow operation in subtraction are somewhat analogous to the tree-balancing operations for balanced trees. In general we are exploring a previously unsolved research problem, and we are going to illustrate our thought processes by recording the false starts we made and the ways we recovered.

A data structure for problem 3 must work in particular for the first element in the ordering; accessing the first element must take time O(I), that is, bounded by a constant. Hence if we use a balanced tree, we should store small numbers (assuming we order elements by the size of their keys) where they are easy to access. (Note that problem 3 may be interpreted in two ways: either as search, insertion, and deletion of the kth smallest key in the data structure, or as search etc. of element k where the data structure contains elements I, ... n.) One way would be to store the small numbers near the top of the tree, as in Figure la. To reach the kth element in this tree, we would examine the binary representation of the number and descend through the left or right subtree of a node depending on the parity of the appropriate bit. Clearly the time to access k is bounded by c·log k, c representing the overhead of deciding which branch to take; but it is not at all clear how this data structure could be kept in such nice order if arbitrary additions and deletions were allowed,

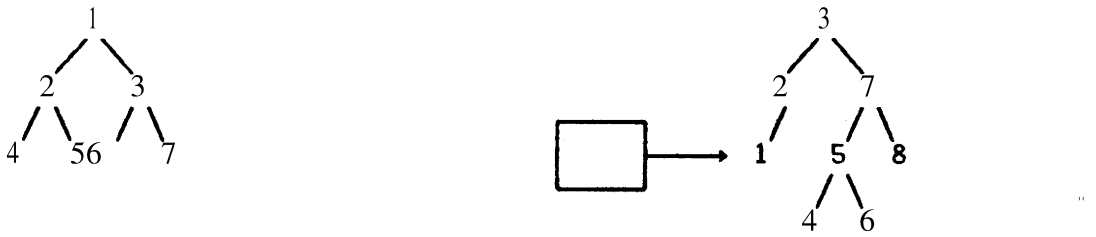


Figure I – Balanced trees in which a search for k takes O(log k) time

Suppose that information in the tree is stored in symmetric order, like in the examples we considered last time. In a balanced tree the "Jeftmost" leaf can be an arbitrary distance from the root–O(log n), where n is the size of the tree-but we might succeed by saving not only a pointer to the root of the tree as usual but also a pointer to the leftmost node, and having two-way links throughout the tree. (A good principle, if time is more important than space: when something is hard to access, save a link to it.) Figure lb contains an example of such a tree. Will it work? If we could prove that the root is always near the middle of the tree, then all keys on the right side of the tree would be at least n/2; it would take log n operations to reach the top and log n to reach the bottom-at most 2·(log(n/2) + log 2) operations to reach any node on the right side of the tree. But we can't prove that; if the tree has the least possible nodes in the left subtree (something like F_h) and the most possible in the right (something like 2^h), then the rank of the root tends toward 0 as the size of the tree increases. Fortunately we don't need to prove that the root is near n/2. We merely notice that if we go up m levels, we need go down at most 2m, so somehow the amount of work we do is proportional to the depth of the smaller of the two portions of the tree.

We see that it's probably a good idea to study "bad" trees, those which are worst cases in some respect. In general, it is a good idea to start thinking about a problem at a low level in order to build up intuition about how the problem might be solved; by familiarizing ourselves with the concepts of the problem, we become able to think of them in higher-level terms. (We may encounter a similar situation when programming. We determine "primitive" operations which we know we can implement, and build up the rest of the program in terms of these primitives. Compare the top-down bottom-up discussion in the notes of October 28.) Our low-level thinking will involve **"miniproblems"** which when solved give us more information about just what concepts are needed to solve the larger problem. In this case, the miniproblems involve bad trees.

We can state an algorithm for accessing the kth element in the tree structure described above. We go up the tree, stopping just before the first node whose rank n_i exceeds k, doing this in m steps. We then go down the right side of the tree until we find k, taking no more than. **2m** steps. We know, since the tree is balanced, that $m = O(\log n_i) \geq O(\log k)$. This relation in itself doesn't help us much, since the inequality goes the wrong way; how can we get an inequality in the right direction? We know that if k is in the right part of the tree, n, isn't much bigger than k. We can **also** note that if we stop climbing the tree just one step earlier, we've used m-l steps, where m- **1** is bounded by the log of some number n_0 (the rank of the prior node), and n_0 is less than or equal to k. We have m-l $= O(\log n_0) \leq O(\log$ k), the bound we want; the total time we spend is **3m** and therefore is also $O(\log k)$.

We need now to consider insertion into our data structure. What we'd like to do is find the place to do the insertion, and then just make some kind of local fix. Considering some examples will help turn up possible difficulty. Suppose we insert into position **1,** many times. Eventually the left side of the tree will fill up and the tree will need to be rebalanced at its **root;** to do this, though, we will need to go all the way up the tree, and we can't do this in constant time.

Another miniproblem we might consider is that of counting in base two, with the constraint that we do a constant amount of work at each step. If we just use the digits 0 and 1, we see that adding one to a number n occasionally results in a lot of carries, taking time proportional to the length of n. One idea is to code the numbers so that only a few digits change at any step; the carrying operation is essentially being deferred until later. A code in which successive numbers differ by only one digit is called a ***Gray code,*** and we could use it if we could figure out in constant time which digit was the next to change. (There is a tricky way to do this **[1],** but Knuth suggested not pursuing this since it is so tricky it probably won't generalize to our maxiproblem.) Another idea is to use a redundant representation for numbers, say, using the digits 0, I, and 2. Since most numbers (although not all-e.g., not **11...** I) would have multiple representations in this system, maybe we could pick for each number the representation that minimizes the work we must do.

Studying this counting problem is useful because the difficulty presented by carries is similar to that presented by tree-balancing in our other problem. Carries that don't affect much of the number 'correspond to insertions that don't affect much of the tree. Eventually, the number "fills up" just like the tree did, and fixes must be made farther left in the number and farther up the tree. Also, it is always worthwhile to **look** for representations of a problem which can be handled more conveniently. In this case, it's probably easier to deal with numbers and sequence lengths than with limbs of trees, and for many problems it's possible to change the notation used, or the representation of the data, or some other feature in order to view the problem in a different (brighter) light. The art of devising analogous but simpler miniproblems is an important part of research work.

Using binary numbers $a_k \ldots a_0$, we get a carry out of a, half the time, a carry out of a, ¼ of the time, and so on; this averages to one carry for each addition we perform, but the average includes the bad cases where many carries are done. What we want is to spread out the work we do so as to do only a little, maybe just one operation, at each step, and the redundant number representation may make this easier. We note that additions to 0 or I are easy; only adding to 2 now causes a carry. However, a **2** can be changed to a 10-a "difficult" case becomes two potentially easier cases. One possibility is to fix up a 2 after each addition. The resulting sequence is 0, 1, 2 fixed to 10, 11, **12→20, 21→101, 102→1** IO, **111,** 112420, **121→201,** 202. Now what? We need to do two things. First, we must decide what **2** to fix up when we have a choice. Second, we must prove that the **fixup** never leads to more than a constant amount of work; fixing up the rightmost 2 in the sequence **22...2** is just as bad an operation as the worst case of the binary number system.

Assuming we fix the leftmost **2** in the number (apparently as good a choice as any), we want to determine the maximal amount of work to be performed at each step. We'll use the following procedure for addition: first fix up the leftmost 2 by changing x2 to **(x+1)0,** then add 1 to the number by changing 0 to 1 and x 1 to **(x+1)0.** This should result in one carry at each step (fixing the 2) and one carry every other step (from the last digit). We note that if x is 2 in the latter case, the carry will propagate, and we hope that won't happen; unfortunately it may, in the sequence 11 **11** 11 I, I **11** 1120, **1111201,** 1112010, **i** I2001 1, 1200020, 2000021, 10000030 (illegal). A patch to the algorithm: fix the rightmost instead of the leftmost 2. The following sequence results: 0, 1, 10, 11, **20,** 101, 110, **111,** 120, **201, 1010, 1011,** 1020, 1101, 1110, **1111,** 1120, 1201, 2010, **10111,** . . .

The algorithm we have so far is

Rule One: (Fix a 2.) Find the rightmost 2 if there is one, and change x2 to **(x+1)0.**

Rule Two: (Increment the number.) If the rightmost digit is 0, change it to 1; otherwise in the last two digits, change x I to **(x+1)0.**

The last digit alternates between 0 and **1.** Rule Two never causes a carry into a 2 because a 2 in the next-to-last position is fixed by Rule One. We need only worry about carries from applying Rule One, i.e. consecutive 2's in a number. How can consecutive 2's arise? From an application of Rule Two: **212xx→220xx.** And how did 212 happen? From an application of Rule Two: **2112x→2120x.** (So far we ignore the effects of Rule One.) Showing that consecutive 2's cannot occur reduces to the problem of showing that 2112 cannot occur, and this problem doesn't seem any easier.

Looking at the sequence of numbers (after extending it a bit: iooii, 10020, 10101, 10110, 10111, 10120, 10201, 11010, 11011, 11020, i 1101, **11110,** iii ii, ii **120,** i **1201,** 12010, 20011, 100020, . . . **),** we might be able to figure out a formula for the digits and prove using that formula that 2's can't occur consecutively (one student did this-see below). We saw by looking further ahead in the sequence that it was possible to generate two 2's **(1** 11 **111, 111120,** 11 1201, **112010, 120011,** 200020, etc.) so we do have to deal with the 2's somehow. If we can't figure out a formula, we might try to show a minimum distance between 2's. Or we might note that 2's seem only to precede O's and try to prove that. This last seems the easiest to check. It is also a stronger statement than what we were trying to prove before, and illustrates a good problem-solving procedure: ***if*** an ***induction isn't*** *working, strengthen the induction hypothesis.*

It indeed turns out that any 2 must be followed by a 0. Application of Rule One changes x02 to x 10 and x 12 to x20, it will fail when x is a 2, and now we have to rule out 202. Once again we strengthen the induction hypothesis: between any two 2's are at least two O's, and every 2 is followed by a 0. To prove this, we look at the first number n for which it is not true, along with the number n-l immediately preceding it. First, if there are two O's between any two 2's and each 2 is followed by a 0 in n- 1, each 2 will also be followed by a 0 in n. If not, the exception-21 or 22-arose either from Rule One fixing a 2 in 202 or 212, or from Rule Two adding to 21 I or 201 to get 220 or 2 10. The first case and half of the second is ruled 'but by the induction hypothesis. If n-l ended with 201, the 2 would have been fixed up by Rule One since there's no rwm for any other 2 to interfere. Next we show that between any two 2's in n are at least two 0's. If not, we have the pattern 2011 1... 12 in n. Again there are two cases depending on how the latter 2 was created. If from Rule Two, then before the addition we had 20111...11-1; either Rule One should have fixed that 2 or there was some other 2 which violated the induction hypothesis. If the latter 2 came from Rule One, then n-l contained the pattern 2011 1...112 which was changed to 20111...120. Such a pattern violates the induction hypothesis.

The strengthened inductive solution above was not discovered during class. One of the students began working on his own and essentially found a pattern for the behavior of the kth digit from the right. The pattern turns out to be even simpler for a related sequence we were simultaneously considering:

Rule One': Same as Rule One.

Rule Two': If the rightmost digit is 0, change it to I; otherwise change it to 2. (It can't be a 2 because Rule One' has already acted.)

This sequence begins 1, 2, 11, 12, 21, 102, 111, 112, . . . ; the rules are somewhat more elegant, and only one carry is done per step instead of 1.5 carries per step in our previous method. Since the rules are more elegant, the analysis of this sequence can be expected to be a little simpler, and since the carries are done at the average rate we can expect any difficulties with 2's to show up faster. It turns out that the rightmost digits alternate 1, 2, I, 2, ..., the next-to-rightmost digits go 0, 0, 1, 1, 2, 0, 1, I, 2, ... the next go at just half speed of these, and so on. (Thus the positions where 2's are "fixed up" in this latter sequence is just the sequence of positions needed to generate Gray code with constant time per step, so we have stumbled into another approach to the problem solved in [1] !)

Both procedures give us a way to count, doing at most two operations at each step (an implementation of either method would use a stack to keep track of the rightmost 2). Counting with numbers corresponds to insertion at the left in trees; although a general solution to problem 3 will require us to consider deletions and arbitrary insertions, this miniproblem perhaps sheds some light on how to go about it.

Referenci:

[1] Bitner, J.R., Ehrlich, G. and Reingold, E.M., "Efficient Generation of the Binary Reflected Gray Code and Its Applications", *Comm.* ACM *19*,9 (Sept. 1976), pp. 517-521.

Notes for November 9, 1976

Topics of today's discussion:
1. counting either forward or backward in constant time' per step;
2. use of cellular automata to model a process which moves from state to state.

Last class we discussed the problem of counting in constant time; in today's class, we studied the more general problem of either incrementing or decrementing (the choice being arbitrary) in constant time. The emphasis of last class was on finding a good counting algorithm; we tried several approaches, failing and recovering a few times. This time we worked on a generalization of an algorithm we'd previously found, concentrating on proving it correct. The proof used the concept of a cellular automaton, with which the state changes resulting from incrementing and decrementing could be represented conveniently.

Just as a redundant number representation gave us flexibility in the addition problem of fast class, it should help us when we need to do both incrementing and decrementing in constant time. We will use the digits {1̄,0,1,2} to represent numbers (where i is "minus one"). It is possible to add and subtract in bounded time using just the digits i, 0, and l-computers have been built which use this ternary number system (see [1])—but we add the extra digit to simplify the problem.

We will also attempt to generalize one of the incrementing algorithms we discovered last time; the algorithm we will consider is the following.

Rule One: Fix the rightmost carry or borrow if one exists.
Rule Two: Add or subtract one as desired.

Fixing a carry consists of converting x2 to (x+1)0 as before; to fix a borrow, we change xi to (x-1)1. We will show that x will not be 2 when a carry is fixed, nor i when a borrow is fixed.

Some initial observations are in order. First, we can't just count backward by reversing the way we counted forward. Our incrementing algorithm of last time maintained a stack to keep track of 2's to fix up; to subtract by undoing an add would involve "unpopping" this stack and would not be easy to implement. Second, it really is in our interest to fix a carry or borrow at each step. Although, for instance, it's less work to decrement a 02 than a 10, it is more work to increment the 02; we don't know what operation will come next, so biasing the algorithm in favor of adds or subtracts is probably a bad idea. Thus, it is reasonable to consider the algorithm stated above.

Our first concern is whether or not the algorithm works. For the problem of last time, which irivolved only addition, we could find a pattern in the counting sequence, but that will not be possible here. (One of the nice features of this problem is that it requires a more powerful proof technique, which we discuss below.) Another way, which we also used last time, is to find an invariant which is true for any numeral our algorithm would produce.

To fix a borrow, we change xi to (x-1)1; if x is i, we have an error, so it should be the case that ii can never occur. Our first stab at an invariant: no numeral contains consecutive i's (or consecutive 2's). Consecutive i's could have resulted from fixing 1̄01̄, which could have resulted from fixing 1̄001̄, and so on; soon the rightmost i has run off the end of the string, an impossibility. Last time we used the "successive approximation" technique to home in on an invariant; the result is some statement about which patterns of digits may occur between two i's, and which patterns may occur between two 2's. Determining the correct invariant is probably a tedious process, however, because of the constant annoyance of dealing with what's going on in the right-hand part of the digit string: where is the next i? what's between the two i's? when does the second i get fixed up? what happens when it does, and can anything get in the way? and so on. We want to *localize* our examination of the problem, to represent the problem in such a way that changes in the string five or ten or twenty digits to the right can be ignored or treated in a convenient way.

Suppose we think of the digit positions as people (or machines, depending on how we feel about anthropomorphizing computers) holding up cards; these people talk only to their neighbors, receiving messages from their right and delivering messages to their left. Each message is either ***fixup, add*** *one,* or *subtract one.* The action taken by a person holding a given card and receiving a given message is specified by the table of Figure 1.

Person holds card		2	1	0	i
Message is ADD 1	New card:	?	2	1	0
	Action:	ERROR	PUSH		POP
Message is SUB 1	New card:	1	0	i	?
	Action:	POP		PUSH	ERROR
Message is FIXUP	New card:	0	1	0	1
	Action:	SEND + POP	SEND F	SEND F	SEND - POP

Figure I – transition table for counter

The error action does the appropriate thing. A ***push*** saves the location of a 2 or i on a stack; *pop* takes the location off the stack. The ***send*** action transmits a message to the person on the left (e.g. "SEND **+**" means to send the message "ADD 1"). Initially, **all** people start out holding a zero card. Messages, alternately ***fixup*** and either ***add*** or sub, are input at the right. The transition table is not quite an accurate description of the counting process, since the propagation implied for the **fixup** operation doesn't occur (the location of the rightmost i or 2 is kept on the stack), but it is a good enough model for the purpose of proving the algorithm's correctness.

The table in Figure 1 describes a *cellular automaton,* a model of **computation** introduced by John von Neumann. A general cellular automaton is two-dimensional; like Turing machines, cellular automata can compute all computable functions. Cellular automata seem especially useful as models for parallel processes found in large computing systems, for computing systems which can simulate their own behavior, and for biological reproduction; they are discussed in **[2]** and **[3].**

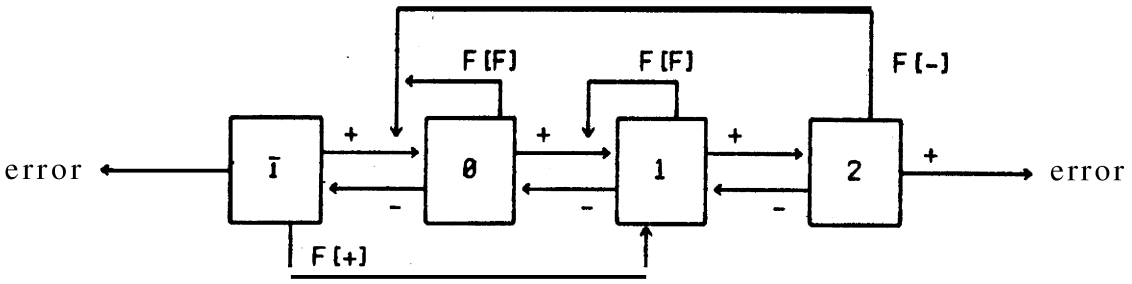


Figure 2 – transition diagram for counter (output from a cell is specified in brackets)

Figure 2 contains a transition diagram for each cell of our automaton; digits correspond to states in the diagram. We need to show that the automaton never reaches the error state, that is, that it never tries to add to a 2 or subtract from a 1. For the rightmost cell it suffices to show this for the input sequence (F{+|-})* described by the algorithm-a **fixup**, followed by a + or -, followed by a **fixup**, and so on-and we can do that, but this argument would not be sufficient to show that the other cells never get into the error state, because the messages they receive need not strictly alternate between **fixups** and +/-'s. Therefore we need to find a more flexible criterion which will guarantee that the error state cannot occur. Several adds or subtracts in a row certainly lead to error, since no **fixups** are done to avoid overflow. If we forbid, say, two consecutive adds or subtracts, can we ever wind up in the error state? Well, an error caused by incrementing can only come from adding to 2, and the only way to reach 2 is by adding to 1; hence at least two adds in a row must be input to cause such an error. Similarly, only from two consecutive subtracts can a decrementing error arise. Can errors arise indirectly, as a result of two adds or subtracts being transmitted? We easily see by examining Figure 2 that a **fixup** or subtract command must separate any two adds which are transmitted, and a **fixup** or add is sent between any two subtracts. We have shown that for any sequence of +'s, -'s, and F's which contains neither two consecutive +'s nor two consecutive -'s, a cell of the automaton will function without errors, and it will transmit a sequence of +'s, -'s, and F's without consecutive +'s or -'s. This argument does not quite characterize the set of inputs which don't cause errors, but it does show that our algorithm works.

So what have we done? We have proved that certain sequences of commands (each command being either add 1, subtract 1, or **fixup**), when applied to the digit string consisting of all zeros, yield a string of legal digits. We were able to determine "good" command sequences, using a cellular automaton model of a digit string, by examining each digit position locally, ignoring for the most part what was happening to other digits farther down the line. Among the "good" sequences are those generated by our counting algorithm. The semantics of our algorithm guarantee that the resulting digit string will represent the correct value; the algorithm, therefore, is a way to count, either incrementing or decrementing, in constant time per step. We noted last time that incrementing a binary number could be interpreted as insertion at the left of a balanced tree. Today's algorithm, as a generalization of the counting algorithm of last time, suggests a method of handling deletions from a tree as well as insertions.

We observed earlier that the cellular automaton did not accurately model the processing of **fixups**. The implementation of our algorithm includes an optimization: keeping the **fixup** location on a stack to avoid **fixup** propagation. More generally, we might associate with each position x a pointer to the next 2 or i to the left, in order to allow additions or subtractions of more than 1. Apparently no work has been done on formalizing optimizations of this kind within the framework of cellular automata. Such a formalization, if it existed, could be applied to other computer algorithms; for example, coroutines in a program can be modelled by a cellular automaton, and it would be nice to have a way to represent optimizations of the coroutine structure.

When we apply the counting algorithm to balanced trees, we encounter another difficulty: our assumption that integer operations are performed in constant time. For large enough integers, this assumption is clearly in error, and a factor of $\log_k n$ (k some large constant) representing the length of the number creeps into our "constant time" operations. Paradoxically-since we're interested in behavior of the algorithm for large n, which is exactly where we get into trouble-we ignore the problem, since the unreal theoretical model we are using is actually most relevant to practical programming.

The similarity between counting and tree maintenance is perhaps more obvious if we use {0,1,2,3} to represent numbers, instead of {1,0,1,2}. Each digit might then have a meaning of, say, the number of items in a tree, or the number of balanced trees in a group of trees. Ail nonnegative numbers are still representable (interesting digression: we can represent ail positive base ten integers using the digits {1,2,...,A = 10)) but we no longer have a way of representing negative numbers. Figure 3 contains the corresponding transition table; examining it as before, we see that any input sequence which starts with an add, which contains neither consecutive adds nor consecutive subtracts, and for which the number of adds exceeds the number of subtracts, yields a legal digit string.

	0	1	2	3
ADD	1	2	3	error
SUB	error	0	I	2
FIX	2[-]	[F]	[F]	1[+]

Figure 3 – transition table for base two counter using digits {0,1,2,3}

References:

[1] Avizienis, A., "Signed-digit number representations for fast parallel arithmetic", *IRE* Trans. *Electronic Computers 3 (196* 1), pp. 389-399.

[2] Burks, A.W. (ed.), *Essays on* Cellular *Automata,* University of Illinois Press, Urbana, lilinois, 1968.

[3] Codd, E.F., *Cellular Automata,* Academic Press, New York, 1968.

Comments on solutions to problem 3
incorporating notes from November 11 and 18, 1976

Several different solutions to problem 3 were submitted; two representative solutions are described and compared.

Solution A, using a balanced tree (Don Knuth)

Nodes in the tree are stored in symmetric order; each node contains a key or a rank field indicating the size of its left **subtree** as in [3]. There is also a header node which points to the root of the tree. The left path of the tree (including the header node) is doubly linked, and a pointer to the leftmost node is maintained. Nodes not in the left path are **labelled +, -,** or •, as in [3]; a node's label indicates its *balance* factor, respectively either + **l,** -1, or 0. The header node is iabelied R. Other nodes in the left path may be iabeiied **+, -,** •, B, or C. Here B stands for "borrow" and indicates a height deficiency; **it** labels a node whose balance factor is 0 and whose parent node's balance factor is (erroneously) based on the assumption that this node's height is one more than it really is. The code C stands for "carry" and indicates a height surplus; it labels a node whose balance factor is -1 and whose parent node's balance factor **is** (erroneously) based on the assumption that the node's height is one less than it really is.

The insertion algorithm introduces C's into the left path, and the deletion algorithm introduces **B's**. The **fixup** algorithm in turn eliminates them, either by moving them up the left path or by rebalancing the tree. A **fixup** uses one of the following transformations:

• => •	(single right rotation, node removed from left path)
C • => - c	(no rotation; merely moves C up)
c + => - •	(no rotation; merely corrects balance factors)
C B => - -	(no rotation)
C R => - R	(corrects balance factor)
B- => • B	(no rotation)
B • => • +	(no rotation)
B+ => • • B or • - B or • + - or • • + or • - +	(single or double left rotation, depending on the state of subtrees, with new node inserted into left path)
B C => • •	(no rotation)
BR => • R	(corrects balance factor)

Examples of these transformations are given in Figure 1.

Several operations are used to maintain the balanced tree. The operation **"fixup"** finds the leftmost B or C and applies one of the above transformations there; or, if there are no B's or C's, **"fixup"** does nothing. This takes bounded time since positions of B's and C's can be kept on a stack (leftmost on top). The operation "fix **B"** does a **fixup** only if there is a B on top of the stack; "fix **C"** does a **fixup** only if there is a C on top of the stack. The operation **"moveup"** consists of alternating **"fixup"** and "compare" where the comparison stops at the first node x from left to right such that the desired node is in the left **subtree** of x (the header node insures that x exists). The **fixup** insures that the node being compared is never C or B. We could also do the **fixup** only if we encounter a C or a B. The operation "search" consists of **"moveup"** plus a search in the left **subtree** of x. The operation "insert" consists of **"moveup"** plus an insertion into the tree x; then if **x≠R** and **subtree** x has grown in height (it must now be - since the insertion goes into the left subtree), we do the "fix C" operation *twice* and then label x with a C. The operation "delete" consists of **"moveup"** plus a deletion in the tree x; then if **x≠R** and **subtree** x has decreased in height (it must now be ●), we do the "fix B" operation and then label x with a B.

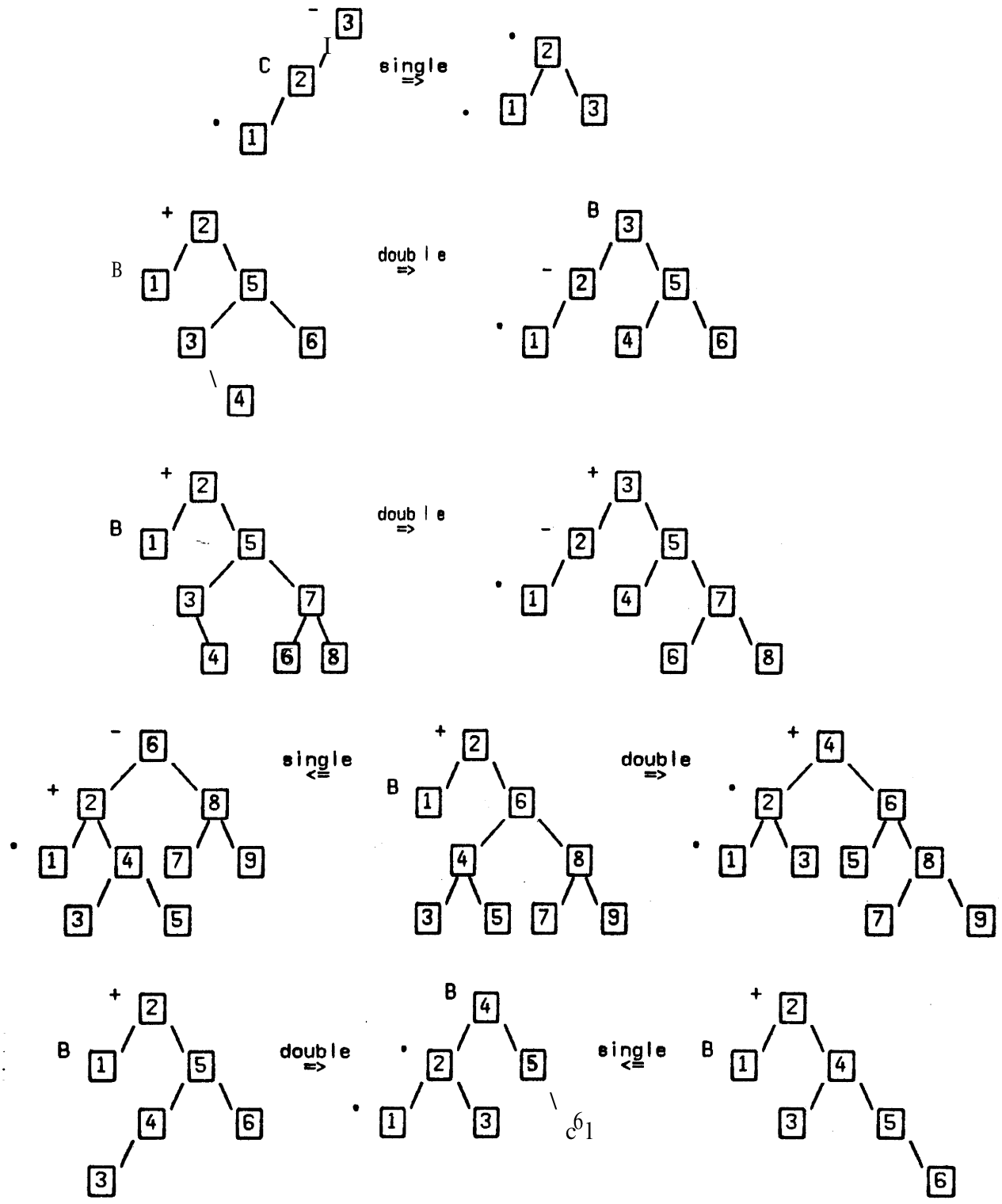


Figure **l – fixup** examples

Note that no transformation covers the cases of two adjacent B's or two adjacent C's. We now show that these cases will never arise. In fact, the following two conditions are invariant between search/ insert/delete operations:

(i) After a B there is a C or a . before the next B.

(ii) After a C there is a B or at least two -'s before the next C.

The **fixup** transformations clearly have no effect on (i) and (ii). Furthermore after we do a "fix B" operation it is clear that if the top of the stack is now a B, then that B is preceded by a • . After we do one "fix C" operation, the top of the stack is either B or there is a - preceding the first C; after we do two "fix C" operations, the top of the stack is either B or there are two -'s preceding the first C. (If the "fix C" operation was C- => . then by (ii) another - is "uncovered".) Therefore when insertion and deletion change x to C or B the invariants are preserved.

Using these two invariants, we can prove that the kth node can be accessed as required in O(log k) time. Let m be the number of fixup-compare cycles during the **moveup** operation **until** node **X is** found; thus m is the length of the left path of X. We showed in the notes of November 4 that the height of X is O(iog k). Thus m = O(log k) and the "search" operation takes O(log k) steps. Since "insert" and "delete" add only bounded time to "search", the total running time is bounded by O(log k).

For an example of how the tree is built, we'll examine the case of multiple insertions into the beginning of the tree. The following sequence results, assuming that the **moveup** operation does nothing.

R				
•R	*C--R	*C--CR	*C--C-R	• C--C•R
•CR	• C*-R	•C•--R	•	•C•--CR
•C-R	• -•R	•	•C-••R	*C-*-CR
• +R	• C**R	•C••-R	•C•••R	•C••--R

After every fourth step we get • C**a where • Ca occurred earlier, so the right part (corresponding to the top of the tree) of the sequence repeats ¼ as quickly, then 1/16, and so on. The tree after m insertions, where m is a power of two, is a "complete" binary tree, perfectly balanced except for an extra node stuck on the left.

Solution B, using a list of 2-3 trees (based on that of Janet Roberts)

This solution uses 2-3 trees, described in [1]. In a 2-3 tree, the lengths of paths from the root to the leaves are equal. This property is maintained by having two different kinds of nodes, 2-nodes and S-nodes; 2-nodes have two **subtrees** and contain one key, and 3-nodes have three **subtrees** and contain two keys.

Insertion into a 2-3 tree is done only at its leaves. If the appropriate leaf L is a 'L-node, it is made into a S-node. Otherwise it becomes two 2-nodes and the insertion must be propagated up the tree. If all nodes along the path between L and the root are S-nodes, the root itself splits into two **2-nodes** and the tree increases in height.

Deletion in a 2-3 tree is somewhat more complicated, since it may take place anywhere within the tree. A **nonleaf** may replaced by its successor or predecessor (both are leaves) and the leaf deleted. A leaf is deleted by removing it and moving in a key from its brother to replace it. If the brother was a 2-node, it has no keys to spare; the two nodes, together with a key from their parent, may be combined into a single node. If the parent was a 'L-node, the process is repeated. If this process results in deleting a key from the root, and the root is a 2-node, the tree decreases in height.

The data structure for this solution is a list of groups of 2-3 trees. The group of trees at position h in the list contains-between one and six 2-3 trees of height h. Trees in each group are ordered-e.g. all keys in tree #1 are less than the smallest key in tree #2. The trees are separated by ***predecessor keys*** to simplify maintenance operations. Keys are stored in all nodes of each tree (i.e. there are no "dummy nodes"), in symmetric order. A sample structure is shown in Figure 2.

Also maintained are two lists of pointers as shown at the top of Figure 2: one list containing pointers to groups which contain six trees, the other containing pointers to groups which contain only one tree.

To insert a key into the structure, we begin by finding the appropriate tree. Insertion into the 2-3 tree proceeds normally; the only problem arises when the tree fills up (the root then contains three keys) and the height must increase. There are two cases. If there is room in the current group (i.e. it contains fewer than six trees), the tree is split in two; the left half replaces the old tree, the right half becomes a new tree, and the key which would have become the new root in straightforward 2-3 tree insertion is made the predecessor key of the newest tree. If the new tree is the sixth tree of the group, a pointer to the group is added to the list of pointers mentioned above. If, on the other hand, the group of trees overflows into seven trees, trees #6 and #7 are combined, along with the predecessor key for tree #7, to form a tree of height h+1. This tree is added to the beginning of the h+ 1st group; the predecessor key for tree #6 is moved to become the predecessor for the new tree. The pointer list for groups which contain six trees is then modified according to rules presented below. An example of overflow processing is shown in Figure 3a, with four trees displayed instead of seven in order to keep the example small.

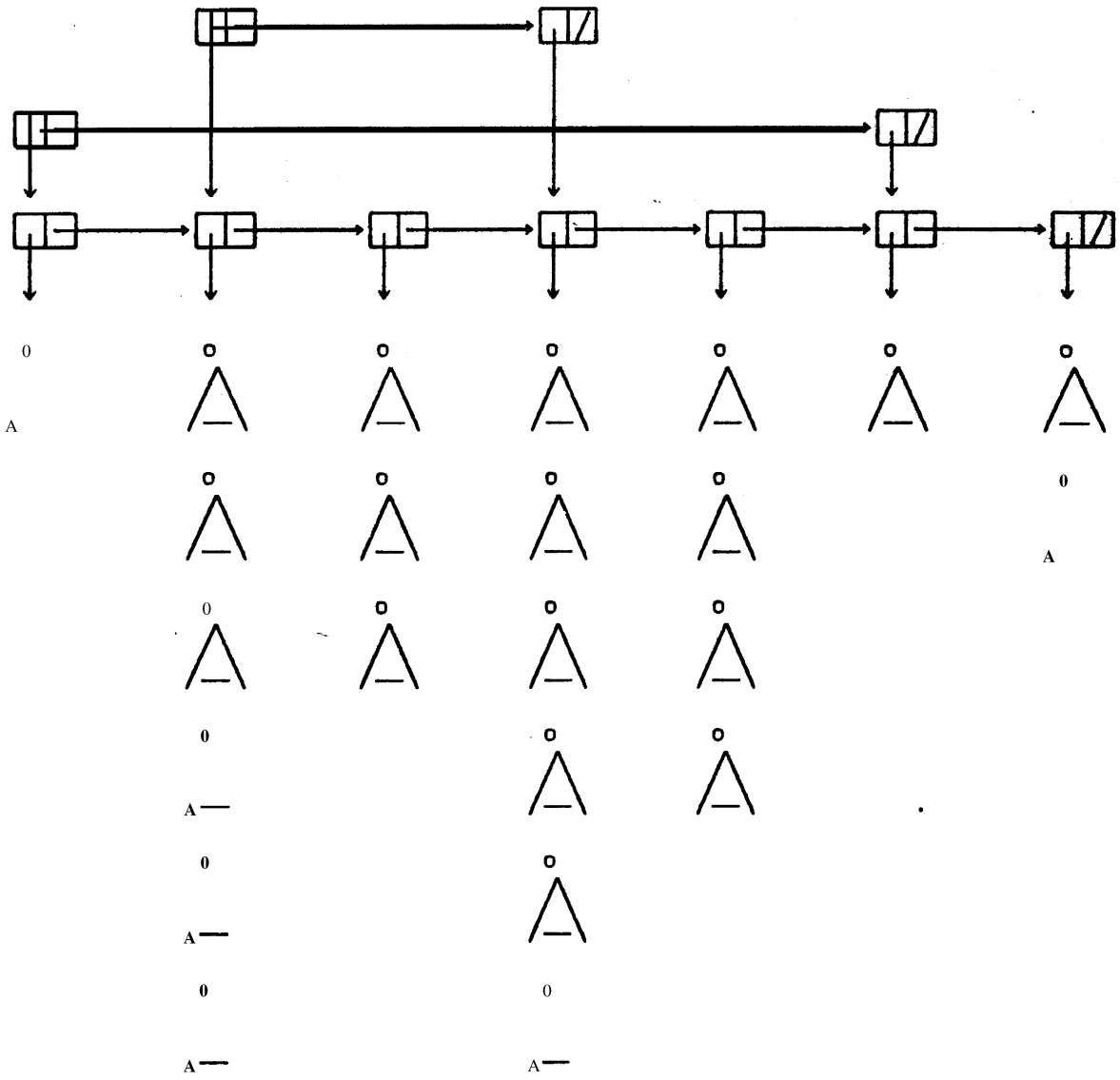
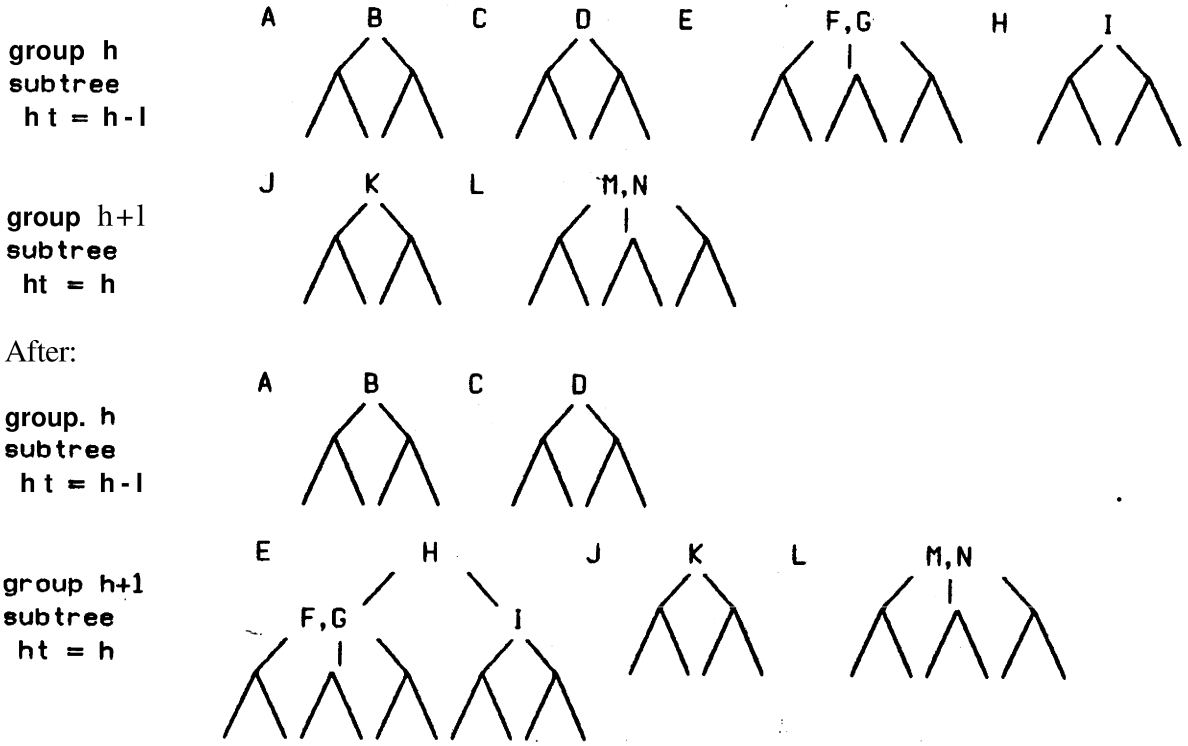


Figure 2 – structure for 1636412

Carry out of position h into **h+1**. Before:



Borrow into position h from h+ 1. Before:

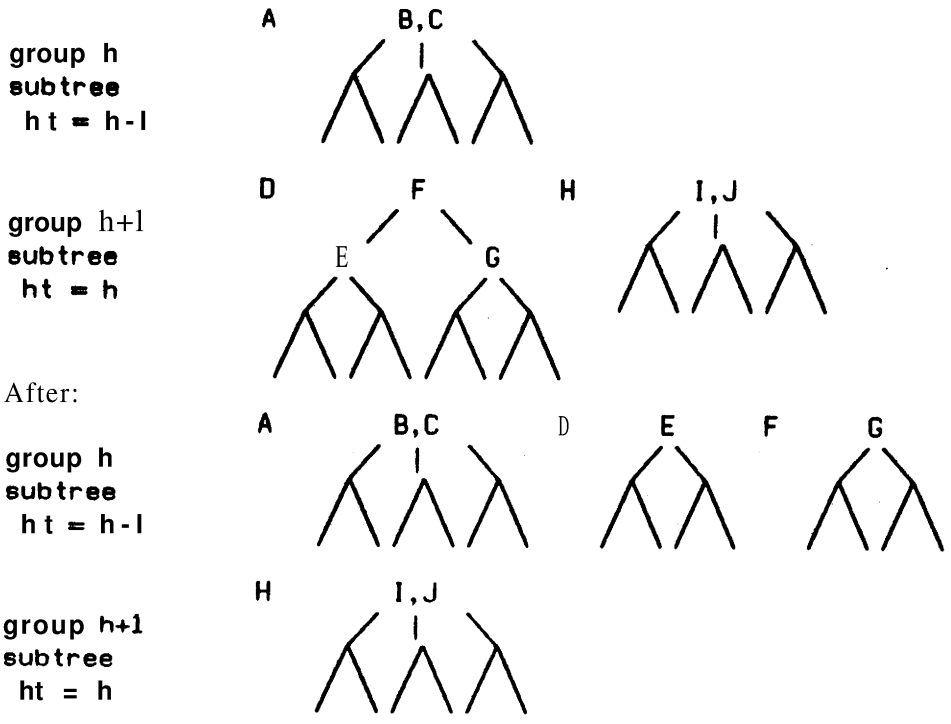


Figure 3 – processing of carries **and** borrows in Solution B (only four trees are used instead of seven)

The problem for deletion occurs when the key's removal causes its tree to decrease in height; when this happens, the one key in the root is forced to move down a level. This tree is combined with one of the adjacent trees in the group if possible. If the adjacent tree had two keys in its root, one of its **subtrees** is concatenated with the deficient tree, and the size of the group stays the same; otherwise the two trees coalesce into one, and the size of the group drops by one. If the deficient tree is the only tree in the group, it is combined with the smallest tree from the next higher group. If there are no trees in the next higher group, the single tree either splits into trees which are added to the next lower group, or else receives a carry from the next lower group. The pointer list for groups which contain one tree is then modified according to rules presented below. Figure **3b** is an example of borrow processing.

The rules for using and updating the two pointer lists are similar to rules we've already discussed for adding and subtracting powers of two in a redundant binary number system. We may view the structure as a digit string, where a digit represents the number of trees in the corresponding tree group. The operations we use to add and delete trees from the structure are done in such a way that only one "carry" or "borrow" is done at each step. There are two invariants which insure that no two carries or borrows appear "too close" to each other, and hence that carries and borrows propagate no more than one place:

Invariant I: Between any two 6's in the digit string is at least one **1,** 2, 3, or 4 which we will call a *buffer digit* for the 6's.

Invariant 2: Between any two I's in the digit string is at least one 3, 4, 5, or 6, and there is at least one **3, 4,** 5, or 6 to the right of every 1 unless the structure contains only one element. The 3's, etc. are buffer digits in this context.

A carry out of some position is done to "fix up" a 6; it transforms x6 to **(x+1)4** by combining the last two trees in the group, with their predecessor keys, into a 2-3 tree of the next greater height. A new 6 may be created in the process if x was 5, but the buffer digit for the old 6 still left-buffers the new one, and the 4 just created buffers the new 6 on the right.

A borrow into a position is done to fix up a 1. There are several cases. If there is a next higher group to borrow from, its first tree is split into two or three trees, depending on whether the root is a **2-node** or a S-node; this converts x **1** to **(x-1)3** or **(x-1)4.** If the I is leftmost in the iist and a 2 Immediately follows it, a borrow is done for the next lower position; the larger tree is split and added to the lower group, converting l-12 to l-4 or l-5. If any other digit follows the I (note that the next digit cannot also be a I), a carry is done from that position, converting l-lx to **l-2(x-2).**

To add a tree into position s of the list, we must consider also the positions of nearby 6's and buffer digits. (A group will be identified by the digit representing the number of trees it contains.) Let d_s be the digit at position s before the addition, and d', be the digit at position **s** after the addition. (Positions in the digit string are numbered right to left, starting at l-the zero position will indicate an imaginary digit at the right of the string.) Let t be the position of the first 6 to the right of d,, r the position of the first 6 to the left of d,, and q the position of the second 6 to to the left of d_s (t may be zero and q and r ∞ if no such 6's exist). Also let x be the position of the leftmost buffer digit (1, 2, 3, or 4) separating d_r and d_t (x is only used when d_s=5). We have one of the situations shown in Figure 4; rules for addition are expressed in Table 1.

Figure 4 – cases for addition

	d_s		d'_s		carry, out of position ...
case *i*	1		2		r
case *ii*	2		3		r
case *iii*	3		4		r
case *iv*	4		5		r
case v	5,	x>s	4		s (Figure 4a)
case *vi*	5,	x<s	6		r (Figure 4b)
case *vii*	6		5		s

Table 1 – rules for addition of one into position s

To see that these rules preserve Invariant 2, we need only note that no new i's are created by addition, and anywhere a 3, 4, 5, or 6 was acting as a buffer between two i's, there is still a 3, 4, 5, or 6. To verify Invariant 1, we will analyze the cases. Cases f-iv present no problem, since the **fixup** at position r provides a buffer digit if one is necessary. For case v, a new 6 which may appear as d'_{s+1} is buffered by d on the left and d', (=4) on the right. Also, a buffer digit at d_{s+1} may have disappeared, but the 4 in d_s will assume its role. Case *vi* merely moves a 6 right past a sequence of 5's, not affecting buffer relationships. In case *vii*, a buffer digit at d_{s+1} is removed, hut so was the 6 it was buffering.

The rules for removal of a tree from position s are much like those listed above for addition.' The rules preserve both invariants, and the proof proceeds in the same way as the proof for addition.

We now show that insertion or deletion of the kth key can be done in time O(iog k). Either process begins with moving down the list of groups to find the group containing the kth key. Group sizes increase exponentially, so this operation is done in time O(log k). Simultaneously, the two lists of pointers to groups which are too heavy or too light are scanned; for whatever group contains the kth key, this scanning yields pointers to the heavy and light groups on its left, again in O(iog k) operations. Properties of 2-3 trees allow us to find the kth key within its group in O(iog k) time. Once we've found the key, insertion or deletion may just involve changing a few links. The worst case, however, arises when the group structure needs to be changed; this involves updating the pointer lists, and constructing or dissecting 2-3 trees. Elements of the pointer lists are either removed (when a 6 or I is completely fixed up) or incremented (when a 6 or I is merely moved over), and this operation is done in constant time. Operations on the 2-3 trees are also done in constant time as described before.

Other solutions and ways of arriving at them

Discussions in previous classes (see notes of 11/2, 11/4, and 11/9) yielded several ideas for attacking problem 3, the most important being the use of balanced trees to achieve fast access,' and the use of redundant representation to allow fast worst-case insertion and deletion. However, there were still many details to be resolved: whether there should be one balanced tree or many; what kind of balanced tree to use (AVL or 2-3, for example); how to implement redundancy. The two solutions just described are extremes in some sense, in that they resolved all these details differently; other solutions, which combined features of solutions A and B, were equally interesting. Possible paths to a problem solution are described below.

The method used to increment or decrement a binary number, derived in the discussion of 11/9, we can generalize to a method for adding 1 into or subtracting 1 from any digit position of the number, i.e. adding or subtracting a power of two. Our resulting algorithm might alternate fixups with adds and subtracts, as in the previous method. But where do we do the fixups? If, say, only the *last* 2 in the digit string is fixed up, we could get in trouble from repeated insertions at the beginning of the string. One answer is to fix up all 2's and i's in the digit string to the *right* of the insertion/deletion position k; there are no more than log k such trouble spots, and fixups are done in constant time, so that can be done quickly enough. Another answer is to fix up trouble spots around the point of insertion/deletion, so that it may be made in conditions of calm rather than turmoil. We have a choice between the following two algorithms for insertion into position k:

Algorithm I: Move left from the end of the digit string, fixing up 2's and i's until reaching position k. (There are at most log k fixups.) Increment or decrement the kth digit as desired; no carry or borrow happens because a 2 or a i in position k has been fixed up. Then do (at least) one more fixup, just as we did a fixup after performing an addition or subtraction in the algorithm of 11/9.

Algorithm II: Increment or decrement the kth digit as desired. (This may tead to underflow or overflow.) Then fix up either the new kth digit, or some other nearby digit, in order to insure that no two consecutive digits can cause each other trouble.

Solution A used a variant of Algorithm I; Solution B, Algorithm II. Algorithm II involves more special cases but appears to do less work, and it was consequently more difficult to prove correct.

Once we've figured out an appropriate generalization to the increment/decrement algorithm, we need to decide on a data structure to which it may be applied. A doubly-linked height-balanced tree with an additional pointer to the leftmost node was suggested in the notes of 11/4 as a possibility; search time, at least, is good for this kind of tree, and it's as good a start as any. Each node on the left path of such a tree is associated with a quantity which exponentially increases as the tree is climbed: the number of nodes in its left subtree. The various states of the trees on the left path (i.e., their balance factors) are somewhat analogous to strings of digits in a redundant number system, since the local balancing transformations are like the local carrying transformations (except that balancing may shorten or lengthen the string). Thus the slightly unbalanced nodes C and B are analogous to the 2's and the i's in redundant binary numbers.

Our last major design decision involves our method for updating the structure; to what extent do we translate the binary number algorithms into our solution? Fixup operations in most solutions exactly paralleled fixups for addition and subtraction in a redundant binary number system. Our choice for representation of redundancy, however, may lead us to reject the direct translation of the binary number operations. Suppose, for example, we use a balanced tree whose left path contains nodes which may be slightly unbalanced; it is unlikely that the rotation transformations we'd use to fix up such a tree correspond exactly to arithmetic fixups.

Our solution has now taken shape. We have selected both a data structure and a representation of redundancy within the structure, and we have a fair-to-middling idea, derived from our discussions of binary counting, of how to update the structure. We now examine the consequences of our initial selections.

Suppose the kth node on the left path of our data structure points to a list of one, two, three, or four balanced trees of height k. For easy access of nodes within a given list, we order the trees of the list: the nodes of tree #1 are all less than those of tree #2, etc. The left path then represents a string consisting of digits {1,2,3,4}.

We must devise "incrementing" and "decrementing" algorithms for creating and deleting trees within a list, along with "carrying" and "borrowing" algorithms for transferring trees between lists. When tree #1 fills up, we want to overflow into tree #2, either a new tree or the tree #2 which already exists. From a "full list"-one containing four trees-we want to remove some trees, combine them into .a bigger tree, and add the new tree to the next higher list. For deletion we will want to coalesce trees within a list, or else split up a tree into smaller trees to be added to the next smaller list. Furthermore, we want to perform these operations in constant time if possible, since the number of fixups done by Algorithm I before an insertion or deletion is proportional to log k. Straightforward algorithms for splitting or concatenating arbitrary AVL trees take time O(log n), where n is the size of the tree ([2], summarized in [3]), but there are a number of ways to speed them up at the expense of some pointer space. The use of dummy nodes at nonleaves in the tree, for instance, makes splitting and concatenation trivial (although it makes insertion and deletion much more difficult). Keeping pointers to the first and last nodes in an AVL tree also simplifies concatenation, since the pointers allow easy access to some node which can be made the root of the new larger tree. Separator keys between trees of the same height would serve the same function. Neither technique, however, is useful for dividing an AVL tree into two other trees one less in height; assuming we stick to a direct implementation of a binary counting algorithm, we therefore still have no way to "*borrow" in constant time.

This observation suggests that we might adopt a more flexible algorithm for borrows and carries, which for instance might allow splitting a tree of height h into trees of height h-l and h-2 and adding one to each of the next two digit positions. The resulting invariant, if one exists, would be more complicated than the invariants for Algorithms I and II. The observation also suggests that the AVL tree is an inappropriate data structure for situations in which the height of the tree plays an integral part. Another data structure, 2-3 trees, was described in the notes of I 1/2 and may be more appropriate for this solution attempt. Path length from root to leaf in a given 2-3 tree is constant, so deleting the root results in trees which are ail the same height. The keys stored in each node serve the function of separator keys for the subtrees. Concatenation and division can be done quickly when they occur as a result of insertion and deletion.

Several solutions, including Solution B, used 2-3 trees. Typically, the data structure was a list of lists of trees, the top-level list corresponding to the left path of the AVL tree described earlier. (Mike Piass, however, noticed that the top-level list could be incorporated into a slightly-modified 2-3 tree whose left path contained nodes with 0, 1, 2, or 3 keys. Algorithms for maintaining his tree thus had to deal with only one type of data structure.)

Solutions which used lists of trees along with direct implementation of one of Algorithms I and II failed to handle one possibility correctly: deletion from the leftmost (largest) tree. Continued deletion from this tree would eventually make it too small for its position, resulting in a borrow-but from where? The borrow must be from the right rather than the left; it's equivalent to a carry out of the next position. This bug, if noticed, can be fixed in one of two ways: patching the proofs for invariants to account for the special action, or expanding the number system to allow even more redundancy than before. Solution B is the result of the latter fix; attempts at patching the proofs seemed only to grossly complicate them.

Another way to represent redundancy is to vary the height of each node along the left path between certain bounds; Alex Strong did this, using AVL trees. Concatenation of trees involves joining them with a root, then rotating if necessary; the proof that this preserves balance (in constant time) is long, containing many cases, but easy to understand.

A third way we might implement redundancy is by expanding the possible balance factors. One obvious way to do this is to add extra balance factors tt and --, meaning unbalanced by two on the right or the left. We wish, however, to make the balance factors ***local*** to the nodes of the tree; we would not, for instance, want an insertion at the beginning of the tree to affect nodes far up the tree, since this insertion must be done in constant time. Solution A bases the balance factor of a node on the ***assumed*** heights of its subtrees. Carolyn Talcott's solution used balance factors {+, -, ●), but also associated with each node on the left path a possible "pending carry" which'augmented the node's balance factor. Both these solutions used tree transformations to process carries and borrows. Both also maintained invariants which insured that left- and right-heavy nodes would not bunch up.

Students whose solutions used lists of trees were able to view the problem at a higher level of abstraction than those who tried other structures. Updating operations for tree lists need not be concerned with properties of individual nodes in the structure; they need merely know that the trees split and combine in various specified ways. Other solutions necessitated more attention to the details of defining basic terms, enumerating cases, deducing invariants, and testing. Common mistakes in approaches which used a single tree for the data structure involved insufficient attention to detail: an inadequately specified invariant, so that one was never quite sure what was properties the algorithm was maintaining; an incompletely specified **fixup** routine, or incorrectly ordered **fixups** *vis-a-vis* insertion and deletion; omission of **fixup** cases in proof of the algorithm.

Testing one's algorithm was probably difficult no matter what approach was used. The answer, of course, is to prove it correct, but some people have an emotional need for experimental evidence; also, correct **fixup** sequences and invariants are more easily inferred from a batch of data than from introspection. Most algorithms were too complicated to implement on the computer, at least given the time constraints of students in the course. The single-tree data structure had an advantage here of being simple to manipulate, and therefore possibly programmable; trying to keep track of rotations by hand, however, could have easily proved frustrating. Repeated insertion at the beginning of the data structure was typically the worst case, since it had to be performed in constant time (insufficient for massive structural renovation).

Summary

This problem had never been solved before, not even by Knuth when he assigned it; consequently it illustrated a number of interesting and typical facets of research. For its solution, we first considered a related problem into which the problem we had could be mapped. It's not unusual for an alternate representation of a problem to exist-the hard part is to find it-and by examining, the problem in a different light we often develop some insight into how to solve it, especially with regard to the types of proof techniques that will work.

Our next step was to determine the relevant properties of the related solution. We discovered that a redundant representation would be useful, and we saw that algorithms we used for carries and borrows could be applied to the resulting data structure.

Data structures for this problem all involved some form of balanced tree, but the AVL trees described in class were found in some cases to be inappropriate. A literature search (at least through [3]) was then required to see what other kinds of balanced trees there were. Searches through the literature are of course an integral part of most research.

Defining the basic terms of the solution came next: what a fixup did; when fixups occurred; what invariants were maintained. This task was difficult; testing these components of the solution was the main way of seeing that they worked, and refinement of one condition or another was often necessary. (When Knuth was asked if he had any insights into how he came up with Solution A, he replied as follows: "I kept fiddling until there were no more loopholes, guided only by loose analogies based on the proof techniques I was getting familiar with. Also I was watching a movie at Festival Cinema at the time; this may have helped.") It was important to state the basic properties of the algorithm and data structure exactly, and several solutions contained bugs resulting from sloppiness of basic definitions.

Thus, one needs careful attention to detail at the low level of prwf, while being guided, by high level ideas about the overall goals and interrelationships of problem components. In this way the low level details can be "parsed" into conceptual structures that allow us to keep afloat.

Finally, research does not stop with the solution to the problem; the next question is, what more can be done with the ideas we have just gained. In this case there was an excellent spinoff to problem 3, namely a paper to be published by Leo J. Cuibas, Edward M. McCreight, Michael F. Piass, and Janet R. Roberts entitled "A New Representation for Linear Lists". In this paper the authors discuss lists with a finite number of fingers pointing to positions of interest; it takes O(ftiog k) steps to access, insert, or delete at distance k from one of f fingers. We considered the special case of one finger pointing to the beginning of the list. The solution in this paper is based on B-trees (a generalization of 2-3 trees), of order 24 or more, and it would be interesting to see if a smaller-overhead solution based on AVL trees could be found.

References:

[1] A ho, A.V., Hopcroft, J.E., and Ullman, J.D., *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.

[2] Crane, CA., "Linear Lists and Priority Queues as Balanced Binary Trees," Stanford Univ. C.S. Dept. Res. Rpt. CS-259, February 1972.

[3] Knuth, D.E., *The Art of Computer Programming, Vol. 3: Sorting and Searching*, Addison-Wesley, 1973.

Notes for November 16, 1076

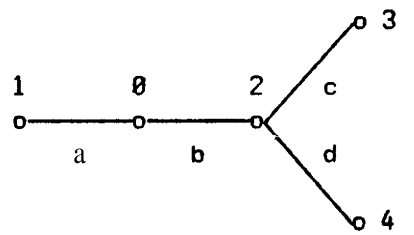
Topics of today's discussion:
1. algorithms for manipulating and combining tradeoff lists (to be used for problem 4);
2. coroutines.

Many networks in the real world (e.g. computer systems, gas pipeline systems, irrigation systems) are centralized, in that they consist of a central node connected to remote sites. There are normally many ways to construct the links between sites in the network; each possible way costs a certain amount, and yields a performance level commensurate with the cost. We can represent the ways to build each link as a list L of delay vs. cost *tradeoff* pairs:

$$L = (d_1,c_1), (d_2,c_2), \ldots, (d_k,c_k)$$

where $d_1 > d_2 > \ldots > d_k$ and c, $< c_2 < \ldots < c_k$. L will be called a *tradeoff list*; the pair (d_j,c_j) indicates that the jth option for constructing the link associated with L costs c_j but results in delay d,. Problem 4 involves finding an overall tradeoff list for the network, given the tradeoff lists for each link. A tradeoff pair in this overall list will represent, for the corresponding implementation, its total cost along with the maximal delay encountered for a transmission from the central node to any other site. (This problem is discussed in [2]; algorithms for join and sum are also described.)

We make two assumptions before proceeding. One is that we can represent the network by a tree rooted at the central node;-. The other is that nodes introduce zero forwarding delay-for a message from node 1 to node 3 through node 2, for instance, the total delay will be the sum of the delays along the links from I to 2 and from 2 to 3.



L, = (150,6), (139,9), (118,14), (87,2 I), (75,30)
L_b = (6 7,6), (62,10), (52,15), (39,23)
L_c = (120,13), (111,17), (92,23), (66,29)
L_d = (94,8), (86, 12), (80,18), (6 1,26)

Figure 1 – sample network

Figure I shows a small network, along with the tradeoff lists associated with each link. Each tradeoff list contains delay/cost pairs, in order of decreasing delay-for instance, it costs only 6 units to get a large (150) delay through link a, but at higher cost the delay can be reduced. Delays are always in decreasing order and costs in increasing order in tradeoff lists. If in some tradeoff list there were two tradeoff pairs (d_i,c_i) and (d_j,c_j) such that d, ≥ d, and $c_j \geq c_i$, option j would be either the same as or clearly inferior to option i since it costs at least as much yet introduces as much or more delay; such uneconomical options are therefore left out of the tradeoff lists. We will speak of one option *absorbing* another; in the above example, option i absorbs option j. Given two pairs (d,c) and (d′,c′), there are nine possibilities:

	c′	c=c′	c>c′
d<d′	(d′,c′) absorbed	(d′,c′) absorbed	
d=d′	(d′,c′) absorbed	either absorbed	(d,c) absorbed
d>d′		(d,c) absorbed	(d,c) absorbed

So the only cases where one does not absorb the other are when $d<d'$ and $c>c'$ or $d>d'$ and $c<c'$. If (d,c) absorbs (d',c') and (d',c') absorbs (d'',c''), then (d,c) absorbs (d'',c''). Given a large set of pairs, the ones which are not absorbed by any others are mutually incomparable, so they can be arranged into a tradeoff list $(d_1,c_1), \ldots, (d_k,c_k)$ as defined above. We can find this tradeoff list by removing any pair (d,c) that is absorbed by any other, until this operation is no longer possible; it doesn't matter which order we do the removal, since we simply want to find those pairs not absorbed by *any* other pairs. (Think about it.)

So we wish to construct an overall tradeoff list for the network, in order to find a minimum cost implementation for its links such that the delay time from the root to any node is at most some given value D. One way to do this would be to just try all possible combinations of options for the individual links, and see what results. This naive method would take far too long to be practical.

Another way is to-break the tree into subtrees, find the overall tradeoff lists for the subtrees, and then combine them somehow into the overall list. Once we find the overall tradeoff list L for a subtree, we can treat that subtree as an edge represented by L; if we find ways to combine tradeoff lists for edges, we can find the overall tradeoff list merely by reducing the subtrees of the root to edges and then combining them. An edge in the tree is oriented with respect to the root of the tree: its "head vertex" is closer to the root, and the "tail vertex" is farther away. Two edges are connected either head-to-head or head-to-tail. We will derive algorithms for combining tradeoff lists in each of the two *cases.*

An example of head-to-head adjacency is the subtree of the tree of Figure I with vertices 2, 3, and 4, and edges c and d. We will call the tradeoff list of this subtree the *join* (abbreviated v) of the lists for its constituent edges. To see how to compute it, we can pick an option for each edge and see what the resulting delay and cost is for the tree. The delay is measured from the root, so it's the maximum delay of the two options; the cost is obviously the sum of the two costs. So we compute $L_c \vee L_d$ as follows: The first tradeoff pair is (max(120,94),13+8)=(120,21). The pair (120,13) from L_c may now be discarded; (120,2 I) will absorb (120,13)v(any other pair from L_d), since the delays for these new pairs will remain 120 but the costs will increase. Moving along, we compute the next tradeoff pair-(max(111,94),17+8)=(I I 1,25)—and remove (I 11,17) from L,. The third pair will be (max(92,94),23+8)=(94,31), and (94,8) is removed from L_d. And so on. We finally reach the end of L_c by computing the pair (max(66,61),29+26) and removing (66,29) from L_c. At this point we know we're done, since any option for $L_c \vee L_d$ must have delay at least 66; this means, for instance, that if L_d contained an extra pair, say (40,40), it would be ignored. The resulting tradeoff list is ((120,21), (1 I 1,25), (94,31), (92,35), (86,4 I), (80,47), (66,55)). We can prove by an easy induction that this sort of algorithm works, i.e. it produces a legal tradeoff list. Furthermore, it uses O(sum of the lengths of L_c and L_d) operations, an improvement on the naive algorithm which joins each option of L_c with each option of L,, and then discards each pair absorbed by another pair in the list. To code the algorithm, we need to worry about the boundary cases-when one or the other list is empty-and also the case when the delays of the "top" pairs are equal: we discard both pairs in that case. Figure 2 contains the coded algorithm, written In a Lispish dialect of Sail, where list is a record structure containing three fields (integer delay, cost; list rest).

```
recursive list procedure join(list L,M);
    begin
    if L=null or M=null then return(null)
    else if delay[ L]>delay[M] then
        return(newlist(delay[L],cost[L]+cost[M],  join(rest[L],M)))
    else if delay[ L ]<delay[M] then
        return(newlist(delay[M],cost[L]+cost[M], join(L,rest[M])))
    else if delay[L]=delay[M] then
        return(newlist(delay[L],cost[L]+cost[M], join(rest[L],rest[M])))
    end ;
```

Figure 2 – code to construct LvM

Head-to-tail adjacency is somewhat more complicated. An example from Figure 1 is the subtree with vertices 0, 2, 3, and 4, and edges b, c, and d; if we assume that we've replaced $L_c \vee L_d$ by L_e, we now wish to compute the *sum* L_b+L_e. For any choice of options for the two edges, the total cost is the sum of the individual costs as before; the delay is also the sum of the two delays, since a transmission from the root to a leaf is delayed in its travel both through edge b and through one of edges c or d. Figure 3 contains a table of sum pairs for L_b and L_e.

	(120,21)	(111,25)	(94,31)	(92,35)	(86,41)	(80,47)	(66,55)
(67,6)	(187,27)	(178,31)	(161,37)	(159,41)	(153,47)	(147,53)	(133,61)
(62,10)	(182,31)	(173,35)	(156,41)	(154,45)	(148,51)	(142,57)	(128.65)
(52,15)	(172,36)	(163,40)	(146,46)	(144,50)	(138,56)	(132,62)	(118,70)
(39,23)	(159,44)	(150,48)	(133,54)	(131,58)	(125,64)	(119,70)	(105,78)

Figure 3 – sum pairs for L_b+L_e, where $L_e=L_c \vee L_d$

We noted previously that certain areas of the L-by-M table for LvM could be ignored. When the top tradeoff pair P of L had a larger delay value than the top tradeoff pair Q of M, PvQ would absorb the rest of P's row; when Q's delay value was higher, PvQ absorbed the rest of Q's column. There appears to be no analogous optimization for computing L+M (see Figure 4), so we will just straightforwardly generate the table and systematically remove absorbed pairs.

	(120,21)	(111,25)	(94,31)	(92,35)	(86,41)	(80,47)	(66,55)
(67,6)	(187,27)	(178,31)	(161,37)	***	***	***	***
(62,10)	***	(173,35)	(156,41)	(154,45)	***	***	***
(52,15)	(172,36)	***	(146,46)	(144,50)	***	***	(118,70)
(39,23)	***	***	(133,54)	(131,58)	(125,64)	***	(105,78)

Figure 4 – sum pairs for L_b+L_e with absorbed pairs removed

In order to construct a single tradeoff list from the table, we will use a subsidiary function *union*. The union of two tradeoff lists has its own interpretation: it is the list of "best case" options (say, for choosing the data link to procure from between competitors A.T.&T. and G.T.E.). To compute the union of L and M, we essentially put them together and remove all absorbed options; and to do this efficiently, we take advantage of the ordering property of tradeoff lists as follows. We remove pairs from the tops of L and M until neither top pair absorbs the other; we then remove the least costly top pair from L or M, add it to the end of the list to be returned by *union*, and repeat. The algorithm is coded in Figure 5.

```
recursive list procedure union(list L,M);
    begin
    if L=null then return(M)              comment Don't return null thistime;
    else If M=null then return(L)         comment Use "other supplier" instead;
    else If delay[L]>delay[M] then
        if cost[L]≥cost[M] then           comment Top M pair absorbs top L pair;
            return(union(rest[L],M))
        else return(newlist(delay[L],cost[L], union(rest[L],M)))
    else if delay[L]<delay[M] then
        if cost[L]≤cost[M] then           comment Top L pair absorbs top M pair;
            return(union(L,rest[M]))
        else return(newlist(delay[M],cost[M], union(L,rest[M])))
    else If delay[L]=delay[M] then        comment More costly pair gets absorbed;
        return! if cost[L]≥cost[M] then union(rest[L],M)
                                  else union(L,rest[M]))
    end;
```

Figure 5 – code to construct LuM

The union of two tradeoff lists is clearly also a tradeoff list. Now that we have a procedure to construct the union, we can compute the sum of two lists:

$$L+M = \text{union}((d+d',c+c') \mid (d,c)\in L, (d',c')\in M\}.$$

This is just the union of the rows of the table of sums of individual options.

So, using the *join* and sum procedures, we can combine edges of the tree, starting at the leaves and working in toward the root. For the tree of Figure 1, this process computes the tradeoff list represented by the expression $(L_a \vee (L_b + (L_c \vee L_d)))$. Both the join and sum operations are associative–$(L_1 \vee (L_2 \vee \ldots \vee (L_{k-1} \vee L_k))) = (L_1 \vee \ldots \vee L_k)$, the union of all the joined pairs in the L_1-by-...-by-L_k table, and $(L_1 + (L_2 + \ldots + (L_{k-1} + L_k))) = (L_1 + \ldots + L_k)$–so we may perform these operations in any order when, for instance, a vertex in the tree has three or more sons. We eventually reduce the tree to a single edge and a single tradeoff list, which list represents the tradeoffs for the entire tree.

A solution to problem 4 will not only compute this overall tradeoff list, but it will also keep track of how each option in the list was derived in order to be able to reconstruct the various implementations for the network as desired. Ways to do this bookkeeptng will be discussed later.

Analysis of our procedures for manipulating tradeoff lists leads us to several other questions; Are our procedures optimal (in terms of. time or space used)? How much can we optimize processing by varying the order in which tradeoff lists are combined? How long, on the average, will the intermediate tradeoff lists be? We leave the latter two questions to the curious reader; the first, however, leads us to the study of coroutines.

Procedures we are used to in Algol are always dynamically nested, so that a subroutine completes execution (in the sense that it can't be reactivated except by starting it over) before returning to its caller. This requirement that procedures are always resumed in a last-in, first-out order imposes an asymmetry between the calling procedure and the called procedure. Suppose we relax this requirement, so that procedures A and B can call each other, with A proceeding from where it left off when B returns and vice-versa. We have thus removed the asymmetry between caller and callee; such "symmetric subroutines" are *called coroutines*. Whenever a coroutine is activated, it resumes execution of its program at the point where the action was last suspended.

We now apply the concept of coroutines to problem 4. Recall that to join two lists, we removed the top pair of one list or the other until one of the lists became empty; the remaining pairs in the other list were ignored. Hence it would have been nice not to have computed them at all. We can set up a coroutine for each tradeoff list which generates options in the tradeoff list as they are needed; there is then one coroutine for each leaf in the tree, along with at least one coroutine for each nonterminal vertex. Whenever we need to know another option at the root, we **query** its descendants, who query their descendants and so on as required.

This application is best illustrated by an example. Consider the tree of Figure I, and associate coroutines with its vertices as follows:

Coroutine 0 computes the overall tradeoff list-(l's list) v (2's list), briefly 1 v 2.
Coroutine I computes L_a.
Coroutine 2 computes 6 + 5.
Coroutine 5 computes 3 v 4.
Coroutine 3 computes L_c.
Coroutine 4 computes L_d.
Coroutine 6 computes L_b.

To answer the question "Give me your next," coroutine 0 will request pairs from 1 and 2, and join them. Coroutine 1 returns its next: **(150,6).** Coroutine 2 asks 5 and 6 for their next pairs in order to return their sum. Coroutine 6 returns **(67,6);** coroutine 5 gets the next from **3–(120,13)–and** the next from **4–(94,8)–saves** the latter, and returns **(120,21).** Coroutine 2 then returns **(187,27),** and 0 returns **(187,33).**

We encounter a complication when computing the sum of two lists L and M: at some point we will want not the next pair from L, but some previous pair. To avoid this difficulty, we rewrite our list of coroutines as follows:

o - 1 v 2
1 – L_a
2 – 7 ∪ 8 We've expanded the + routine here.
5 – 3 v 4
3 – L_c
4 – L_d
6 – L_b
7 – scalarsum of **(120,21)** and 6; **(120,21)** is the first column in the original sum table.
8 – 9 + 5
9 – L_b A new copy of L_b.

We now continue processing. Next(7) is **(182,31),** produced from **(62,10). Next(8)** is **(178,31),** derived from **next(9)** and next(5) (soon to come). **Next(9)** is **(67,6),** starting over in L,. Next(5) is **(111,25),** derived from next(3) and the saved value from the previous call to 4. Next(S) is **(111,17)–3** has now been called twice. Back in 2, the union of **(182,31)** and **(178,31)** is computed; **(182,31)** is discarded since **(178,31)** absorbs it, and **next(7)** is requested again. Coroutine 4 returns **(172,36),** calling 6 again in the -process, and 2 outputs **(178,3** 1). Next(O) is then **(178,37).** There are still details **in** this example to be cleaned up; these will be discussed later.

Coroutines are described in more detail in **[3]** and **[4],** and especially in **[1].** They are quite useful in simulations of parallel processes, and are implemented in, e.g., the **Simula** language. In **this** case we are actually using ***semicoroutines,*** where there is an asymmetric tree-structured relationship between caller and **callee:** the caller resumes the **callee** at the point where the caliee last left off, and the **callee** always returns to its caller. This important special case of coroutines **is** worth noting since it seems to arise more frequently than the general case.

References:

[1] Dahl, O.J., Dijkstra, E.W., and Hoare, C.A.R., *Structured Programming,* Academic Press, 1972, chapter 3.

[2] Frank, H., et al, "Optimal Design of Centralized Computer Networks," *Networks 1* (1971), pp. 43-57.

[3] Knuth, D.E., *The* Art of *Computer Programming: Volume 1,* Addison-Wesley, 1968, sect. 1.4.2, pp. 190-196.

[4] Wegner, P., *Programming Languages,* McGraw-Hill, 1968, sect. 4.10, pp. 324-332.

Comments on solutions to problem 4

A solution to problem 4 had essentially two components: the data structure to represent the network and the tradeoff lists, and the algorithms to manipulate the tradeoff lists. The program's space requirement seemed to be its limiting factor; hence the data structure had to be compact and the algorithms efficient, to minimize memory usage (and save time on garbage collection of records).

Most programs created a tree for tradeoff lists: each node in the tree was a record, representing a single tradeoff pair, which contained the delay and cost of that pair along with some links indicating how the pair was computed. The algorithms given in class for combining tradeoff lists yielded programs which needed around 120 pages of memory to build this data structure.

A refinement of the sum algorithm (implemented by Jim Davidson/Carolyn Talcott, Anne Gardner, and Alex Strong) resulted in a significant reduction of the program size, to 50 pages or fewer. Recall that the sum of tradeoff lists L and M is just the list of pairs P+Qfor P in L and Q in M, with absorbed pairs removed. The unimproved algorithm involved computing, for each pair in L, the scalar sum of that pair with M, and then the union of the scalar sum with the union of previous sums; the scalar sum operation created many tradeoff pair records which would be absorbed in later union operations. The improved algorithm avoided this inefficiency. It examined only the "top" of each scalar sum list at any one time, and did not create new records for this examination. It considered the sum lists in parallel, removing the top pair in each list as it was added to the result or absorbed by the top pair in some other list. Thus the only records created were those actually in the sum. The algorithm is outlined in Figure 1.

```
(TOPS is a list of pointers into the scalar sum lists, each pointer indicating the pair currently
 being considered in the list, along with the two tradeoff pairs from which it was derived.  The answer
 it accumulated in RESULT, which starts out null.)
    Initialize TOPS and RESULT! TOPS initially has pointers to the tops of all columns in this table.
    While TOPS is nonempty, repeat the following:
        Find the lexicographically smallest value of the pair COST[P],DELAY[P] among all pointers P in TOPS;
        let BESTP be a pointer for which this minimum occurs.  (Lexicographic order means dictionary-like
        order:  find the smallest cost, and the smallest delay for this cost.)
        Add the tradeoff record (MINCOST,MINDELAY,ancestors of BESTP) to the end of RESULT.
        Update each pointer P in TOPS as follows;
            While BESTP absorbs P, do the following:
                Advance P to the next pair in the corresponding scalar sum.
                If P falls off the end, remove P from TOPS and exit this inner loop.
    He turn RESULT.
```

Figure I – improved sum algorithm

This algorithm is essentially an n-fold union algorithm for taking the union of the n columns of the table; it is a generalization of the 2-fold union algorithm in Figure 5 of the lecture notes from November 16, which almost makes the case of general n appear simpler than the case n=2! To prove that it works, one must only verify that (i) no pair is discarded unless it is absorbed by some other pair (this is obvious) and (ii) the output of the algorithm is a tradeoff list (this is easy to show, since the pointer update operation gets rid of all pairs with cost[P]≤cost[BESTP] and also all pairs with delay[P]≥delay[BESTP]). Conditions *(i)* and (ii) are necessary and sufficient for the validity of a union algorithm.

Two other solutions (those of Bengt Aspvall and, to a lesser extent, Wolf Polak) were organized around coroutines, as mentioned in the notes. Bengt implemented coroutines using Sail processes-his coroutines were recursive procedures, which transferred control using the RESUME and SUSPEND statements. His program proceeded along the following lines. The main program lowered its priority so as not to get in the way of initialization, and then set the initialization process going at the root of the tree. Each node in the tree set up its tradeoff list, sprouted processes for its sons, and then suspended itself. When all nodes were initialized, the main program regained control, and then requested tradeoff pairs from the root, which requested pairs from its descendents as necessary. The coroutine for each node returned its next pair if it was a leaf; otherwise it called a join or sum routine, which cocalled the node's sons. Wolf's program implemented a coroutine-like solution by keeping in each tradeoff list a pointer to the next tradeoff pair to be returned from that list. For "elementary" tradeoff lists-i.e., those corresponding to the edges of the tree-the pointer indicated one of the tradeoff pairs; for other tradeoff lists, the pointer was usually at the end, signalling that the remaining tradeoff pairs had not yet been constructed. (Wolf in addition discovered an interesting algorithm for producing the sum. He generated each pair P' from the previous pair P by searching up and to the right, then down and to the left of P in the table. The search seemed fairly efficient.)

Coroutines in Sail-like code for computing the join of two tradeoff lists, and for generating pairs of a tradeoff list, are given in Figure 2. These routines are based yet another coroutine solution by Knuth. His program kept track of the program counter via a global variable, and represented each node as a record containing the local variables of the corresponding coroutine. The main part of the program was a *case* statement, each of whose component statements represented a state in Join, sum, etc. The most subtle part of this solution was a coroutine to compute the sum of two lists, since this coroutine need not completely generate the two tradeoff lists, even though the result will always contain the sum of the last pairs; the reason is that the full sum need not be computed unless the caller wants to see it.

```
PTR(LIST) COROUTINE JOIN(PTR(LIST) L,M);
    BEGIN "JOIN"
    PTR (PAIR) P,Q;
    P ← NEXT(L); Q ← NEXT(M);
    WHILE P≠sentinel AND Q≠sentinel DO
        IF DELRY [P]<DELAY[Q] THEN
            BEGIN
            CORETURN(NEWPAIR(DELAY[Q], COST[P]+COST[Q]));
            Q ← NEXT(M)
            END
        ELSE
            BEGIN
            CORETURN(NEWPAIR(DELAY[P],COST[P]+COST[Q]));
            IF OELRY [P]=DELAY[Q] THEN Q ← NEXT(M);
            P ← NEXT(L)
            END;
    CORETURN(sentinel)
    END "JOIN";
```

```
PTR (LIST) COROUTINE L INKLIST(PTR(NODE)L);
    BEGIN "LINKLIST"
    WILE L≠NIL OO
        BEGIN
        CORETURN(F IRST(L));
        L ← REST(L)
        END;
    CORETURN(sentinel)
    E N D "LINKLIST";
```

Figure 2 – coroutines for computing LvM and "elementary" tradeoff lists

Other programs contained other interesting features. The solution of Jim Davidson and Carolyn Talcott represented the tree for each tradeoff pair as a string; their program ran fastest of any, and this may have been one of the reasons. (The speed of all these programs is of course dependent on the particular fmplementation of Sail. Running times varied between 13 and 68 seconds; times were measured during a period of light load.) The Marsha Berger/Brent Hailpern/Rich Pattis program used a field in their representation of the network to indicate whether *join* or sum was needed at each node; this eliminated some extra function calls. They also implemented routines which beautifully displayed the implementation of the network for a given maximum delay.

And some programs even had bugs, all the result of carelessness. We encounter here the difficulty of not being able to check the answer by hand; how can we be sure our answer is correct? For problem 4, there were several good debugging techniques to use:

(a) Consistency checks. A -procedure-to see if the result of a join or sum operation is a legal tradeoff list is easy to write, and would have caught bugs in two programs. Furthermore, checks for internal consistency should be designed into any nontrivial program, not just added as afterthoughts.

(b) Test cases which are easy to check by hand-for problem 4, simple trees with large tradeoff lists, and complicated trees with trivial tradeoff lists. A bug in one program only appeared at a node with three or more sons, but it would have been easy to catch with a four-vertex test case. One can also make test cases easier to generate by using flexible data structures; a linked list, for example, would have been easier to initialize for various test tradeoff lists and networks than an array. Once again, the debugging should be designed into the program from the beginning.

(c) Trace information. Useful especially when combined with (a) and (b)-not too useful for a production run.

(d) The execution profile (which was required for this problem). In general, the execution flow summary is most useful for determining if the program is completely tested-has every section of code been executed at least once? Also, execution counts which are abnormally large are sometimes hints of bugs in the program. Sail's PROFIL program unfortunately presented several bugs of its own to some of the people who used it for this assignment.

Notes for November 30, 1976

Topics of today's discussion:

1. approaches to problem 5;
2. shortest path algorithms;
3. application of a generalized shortest path algorithm to problem 5.

The problem of generating optimal machine code to compute arithmetic expressions, while obviously important in practice, is also interesting from a theoretical standpoint. Most likely there are no "fast" algorithms which produce optimal code, even on a simple computer, for arbitrary arithmetic expressions. Today we discussed a heuristic approach to this problem, using an algorithm which is useful for several other applications as well.

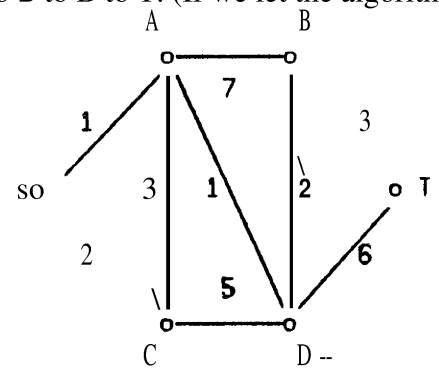
Specifically, we wish to find for problem 5 a code-generation procedure for simple assignment statements which can be applied to machines of various architectures. Aho, Johnson, and Ullman showed in [2] that the optimal code generation problem for machines with one type of register and expressions containing common subexpressions is NP-complete; for this reason, we will restrict ourselves to compiling expressions whose variables are all distinct. However, we will also consider a more realistic machine model: a target machine with two or more different kinds of registers.

Dynamic programming is one method we might use to attack the problem. Code to compute $\alpha \oplus \beta$, where $\oplus$ is some operator, can usually be partitioned into the code to compute (I, the code to compute β, and the instructions to combine them; the problem of generating code for $\alpha \oplus \beta$ can be broken down into the subproblems of generating code for β, then α. in [1], Aho and Johnson describe a dynamic programming algorithm for code generation. With each subtree S of the expression tree in question, they associate an array of costs, each element of which represents the smallest number of instructions necessary to compute S using a program with certain properties. The minimal cost element of the expression tree corresponds to the optimal code for the expression; they then generate it.

However, the dynamic programming approach runs into trouble for machines with two or more different types of registers, say, for floating-point and fixed-point operations. Transfers between registers of different types cause the difficulty: to compute a value v in r_1, we might try to compute it in r_2 and then use the instruction $r_1 \leftarrow r_2$; to compute v in r_2 we would consider computing it in r_1 and then moving it to r_2. Thus we hit a loop, and we must work somewhat harder to reduce the problem to smaller, already-solved subprobkms.

In finding a shortest path in a graph, we would also encounter potential loops (corresponding to cycles in the graph). Shortest-path algorithms, however, typically avoid the problem by working backward from the destination, treating the graph as a tree, since no shortest path will visit a vertex twice. For example, an algorithm by Dijkstra to find the shortest path from a starting vertex s to a -destination vertex t-a nonnegative "distance" value is associated with each edge-assigns to each 'vertex v its mtnimum distance from t. It does this by finding the neighbor of v for which the distance travelled from v through that neighbor to t is minimal. Vertices are evaluated in the order of their proximity to the destination, Dijkstra's algorithm can be generalized; it is a special case of a method described by Knuth in [3]. Knuth's algorithm deals with a context-free language, and it reduces to Dijkstra's algorithm when the context-free language is regular. The terminal strings on right-hand sides of productions roughly correspond in Knuth's algorithm to the "destination" in Dijkstra's; Knuth's algorithm works backward through the nonterminal symbols in the corresponding grammar to find, say, the height of the shortest parse tree in the grammar, or the kngth of the shortest string in the language.

As an example of how Knuth's algorithm works, we consider the shortest path problem of Figure 1. In this example, the edges of the graph are mapped onto the productions shown, and a cost is assigned to the nonterminal on the left hand side of a production whenever that production is used. Each parse tree for the context-free grammar given in Figure 1 represents a path from some vertex of the graph to the destination: S → A → B → T → t represents the path from S to A to B to T, and the cost of this derivation is c(S) = c(A)+ 1 = c(B)+7+1 = c(T)+3+7+ I = I I (the length of the corresponding path). The shortest path from S is then represented by the least costly parse tree rooted at S. It is found by assigning to each nonterminal-here, each vertex of the graph-the minimal cost of a parse tree rooted at that nonterminal, and working backward. In our example, vertex B gets value 3, D gets value 5, A 6, and S 8; hence the desired shortest path goes from S to A to D to B to T. (If we let the algorithm run a little longer, C gets the value 9.)



prod	cost	prod	cost
S → C	c(S)=c(C)+2	C → S	c(C)=c(S)+2
S → A	c(S)=c(A)+1	C → A	c(C)=c(A)+3
A → S	c(A)=c(S)+1	C → D	c(C)=c(D)+5
A → B	c(A)=c(B)+7	D → A	c(D)=c(A)+1
A → C	c(A)=c(C)+3	D → B	c(D)=c(B)+2
A → D	c(A)=c(D)+1	D → C	c(D)=c(C)+5
B → A	c(B)=c(A)+7	D → T	c(D)=c(T)+6
B → D	c(B)=c(D)+2	T a t	c(T)=0
B → T	c(B)=c(T)+3		

Figure 1 – productions for shortest path problem

In. this example, no more than one nonterminal occurs on the right hand side of any production. The same method, however, works in the case of a general context-free grammar, and Knuth's algorithm generalizes Dijkstra's in much the same way as trees generalize linear lists. The formalism is useful in many situations. An application of dynamic programming, which reduces a problem to subproblems, often corresponds to some context-free grammar, where the right-hand sides of the productions represent the subproblems. The context-free formalism extends this to cases allowing loops.

Incidentally, one occasionally runs into applications of the shortest path problem in the real world. It turns out, for instance, that the cheapest way to go the length of the Massachusetts Turnpike is to get off at certain exits and get on again (here we're looking for the cheapest path). One can perhaps make some money in the world currency markets by considering the graph in which the vertices are pounds, dollars, francs, etc., and the edges are the logarithms of the conversion rates between two currencies, and finding negative-sum cycles using known algorithms for shortest paths.

The running time of either shortest path algorithm is proportional to the square of the number of vertices. If the graph contains edges with negative distances, both Dijkstra's and Knuth's algorithms fail. However, if there are no negative-sum cycles, the shortest path can be found in $O(n^3)$ time, where n is the number of vertices. (If there is a negative-sum cycle, the shortest path would of course go round and round it forever.)

Back to problem 5. The code produced for a given expression may be viewed as a string generated by productions of a context-free grammar, and the generalized shortest path algorithm applied to it. Suppose we wish to evaluate the expression a-b•c (whose expression tree has nodes labelled a, representing a-b•c, and β, representing b•c); one of the productions might be the following:

(*) $[\alpha, r_1] \rightarrow [a, r_4]\ [\beta, r_3]$ "$r_1 \leftarrow r_4 - r_3$"

To evaluate α in r_1, first evaluate a in r_4, then β in r_3, then put their difference into r_1.

The set of productions is determined both by the architecture of the target machine, and by the expression being evaluated. Certain of the nodes of the tree might correspond, for example, to multiplications; for each node, there would be productions which represent the ways of computing the product into each of the registers of the target machine using a multiply instruction. With each production we associate a cost in the obvious way; in (*), the cost of computing α in r_1 would be the cost of loading a into r_4, plus the cost of computing β in r_3, plus the cost of doing the subtraction. We also need constraints to avoid clobbering reserved registers-once r_3 is loaded, we don't want to overwrite its contents until α is computed-so (*) would more likely be rewritten as

$$[\alpha, r_1] \rightarrow [a, r_4]\,[\beta, r_3; r_4]\ "r_1 \leftarrow r_4 - r_3"$$

where the notation $[\beta, r_3; r_i, r_j, \ldots]$ means to compute β in r_3 without using the registers r_i, r_j, etc.

Instructions for the IBM 704 are given in the handout for problem 5. (We ignore the CLS, CHS, and FDV instructions for now.) The productions for the expression a-b•c are listed in Table 1. The cost of computing some quantity α in a register or memory location is represented by the function c; c(α,Q), for instance, is the cost of computing a in register Q. The letter M designates any memory location.

[a, M] → [a, A] "STO M"	There are five such productions, 'one for each node in the expression tree. The cost is the cost of computing α in A–c(α,A)–plus 4.36, the time taken by the STO instruction.
[a, M J → [a, Q] "STQ M"	Five such productions. Cost = c(α,Q)+4.36.
[a, A] → [α, M] "CLA M"	Five productions. Cost = c(α,M)+4.36.
[a, Q] → [α, M] "LDQ M"	Five productions. Cost = c(α,M)+4.36.
[α+β, AJ → [a, M] [β, A] "FAD M"	Zero productions, since no additions will be performed. Cost = c(α,M)+c(β,A)+ 13.95.
[α+β, A J → [β, MJ [α, A J "FAD M"	None of these either. Cost = c(β,M)+c(α,A)+ 13.95.
[α–β, A J → [β, MJ [a, A J "FSB M"	One production, since subtraction only takes place at one node in the tree. cost = c(β,M)+c(α,A)+ 13.95.
[α*β, A] → [α, M] [β, Q] "FMP M"	One production, since there's only one multiplication. Cost = c(α,M)+c(β,Q)+25.29.
[α*β, A] → [β, M] [α, Q] "FMP M"	One production. Cost = c(β,M)+c(α,Q)+25.29.
[a, A] → [α, Q] "XCA"	Five productions. Cost = c(α,Q)+2.18.
[a, Q] → [a, Al "XCA"	Five productions. Cost = c(α,A)+4.36.
[variable, MJ → ε	Three productions, one for each variable. Cost = 0.

Table 1– productions to compute a-b•c on the IBM 704

We now start at the bottom of the expression tree, determining for each node the minimal cost of computing the associated quantity into register A, register Q, and memory. We wish eventually to find the cheapest way to compute a-b•c into memory. So we begin: a, b, and c are already in memory, so it costs nothing to put them there and 4.36 to load them into A or Q. The cheapest way to put b•c into A is to load b into A and then multiply; this costs 29.65. Computing the product into Q or memory is most cheaply done by computing it into A first. The same is true for the subtraction. Figure 2 shows the expression tree for a-b•c; the minimal costs found for each node are given, along with the corresponding code to compute a-b•c.

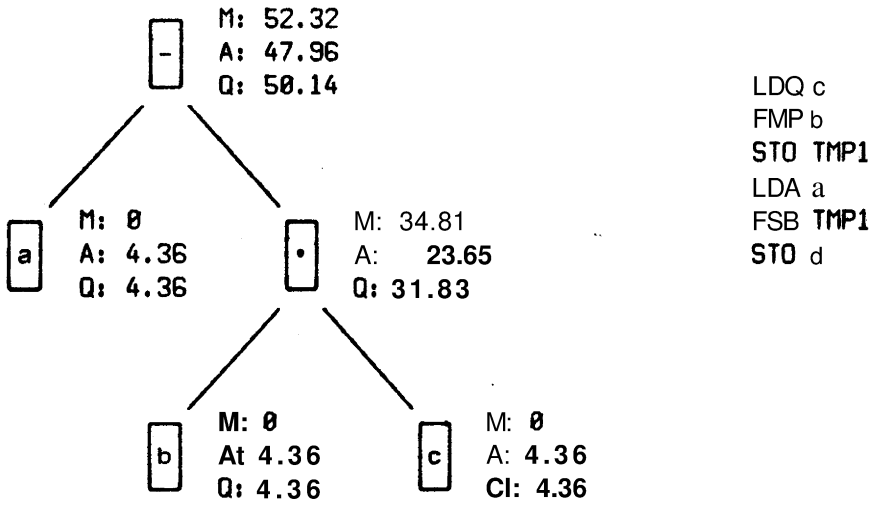


Figure 2 – results of applying the generalized shortest path algorithm to d←a-b•c

The code sequence given in Figure 2 is not the best possible: the sequence CLS c, XCA, FMP b, FAD a, STO d costs only 50.14. For better results we need more productions to work with, representing more programming tricks. Not all programming tricks will be expressible in a context-free grammar; optimization of common subexpressions, for example, is difficult to specify in context-free productions. Other complications arise from different types of registers. We need to keep track of equivalence classes of subsets of registers, and side effects of certain operations (e.g., division) make this task even harder. Although we cannot always be sure we have found the best code, we can be sure that the above approach yields the best program in a fairly large class of possible ones.

References:

[1] A ho, A.V., and Johnson, S.C., "Optimal Code Generation for Expression Trees," *JACM 23,3* (July 1976), pp. 488-501.

[2] Aho, A.V., Johnson, S.C., and Ullman, J.D., "Code Generation for Expressions with Common Subexpressions," *JACM 24,1* (January 1977), pp. 146-160.

[3] Knuth, D.E., "A Generalitation of Dijkstra's Algorithm," *Information Processing Letters,* to appear 1977.

Notes for December 2, 1976

Topics of today's discussion:
1. another case study in problem solving;
2. pinning down the problem;
3. a heuristic algorithm;
4. a dynamic programming algorithm;
5. an improved dynamic programming algorithm.

The problem discussed today was posed by Professor Leland Smith of the Music Department: given a sequence of bars of music, to spread them as evenly as possible on a page of the musical score, in a given number of lines. The class considered various ways to attack the problem, eventually settling on the dynamic programming approach.

Professor Smith's score editing system (implemented on the A.I. Lab computer) contains a "score justification" feature, which takes a sequence of bars of music and fills a page of a score with them. There are three constraints which the justification algorithm must satisfy: (I) a bar of music cannot be spread over two lines of the score; *(ii)* lines in which the notes are squeezed together look bad, so there should be as few of such lines as possible; and *(iii)* all lines of the page must be filled, as evenly as possible. Input to the algorithm will be a sequence of integers, each of which will represent a bar containing the specified number of notes, and the number of lines to fill. Figure 1 contains examples of the desired output. Justifying music is much like justifying text; in both situations, "words" are spread over a line with spaces added in between, and it is easier to expand the words than to compress them. The main difference is that when justifying text, there is usually no chance to compress words, so it is no problem to find the minimum number of lines on which the text will fit comfortably.

3 1 4 1 5	3	3 1 4 1
9 2	1 4 1	5 9
6 5 3	5 9	2 6
	2 6 5 3	5 3
good solution, three lines	bad solution, four lines	better solution, four lines

Figure 1 – possible output, given the input 3 1 4 1 5 9 2 6 5 3

To better understand the problem, we recast it in mathematical terms. We are given n bars to be split over k lines, and we want to split them "optimally". Just what criteria will we use to determine optimality? Well, one property of a solution is its *variance* from a perfect solution; this quantity arises frequently in statistical applications, and is expressed mathematically as $\sum(s_j-\mu)^2$. Here μ is the average number of notes on a line, and s_j is the number of notes on the jth line. The number s_j can itself be expressed as a sum: $\sum(a_i \mid b_j \le i < b_{j+1})$. The numbers b, in the sum specify the positions in the sequence where the line boundaries occur. There are k+1 of these, since there are k lines, and n+1 places at which boundaries can be set, either between two numbers or before the first number or after the last, so $b_1 = 1 < b_2 < \ldots < b_k < b_{k+1} = n+1$. it turns out that numbering the possible boundaries from 0 to n and indexing the b's from 0 to k simplifies matters somewhat, so we have

$$s_j = \sum(a_i \mid b_{j-1} < i \le b_j, \text{ where } b_0 = 0 < b_1 < \ldots < b_{k-1} < b_k = n)$$

What we might look for, then, is the solution with the minimum variance, i.e., the solution for which $\sum(s_j-\mu)^2$ is smallest, where s_j is the sum given above.

We note two things about this formulation of the problem. First, the solution with minimum variance might be a horrible one if the sequence is sufficiently perverse. Second, there may be two or more solutions with the same variance, and so far we have no way of choosing among them. Such solutions may not all be equally good, since some of them may contain more compressed lines than others. Hence we also need some way of expressing our preference for expanding bars. We might consider, Instead of or in addition to the variance, some error function which differentiates solutions better by minimizing the length of the longest line or by maximizing the length of the shortest line.

Expanding the sum for the variance yields the expression $\sum s_j^2 - 2\mu\sum s_j + k\mu^2$. The last two terms in this expression are constant for a given sequence (note that since the sequence is *given,* probability distributions are not relevant), so the only variable quantity is the $\sum s_j^2$. By changing this term, we can distinguish between solutions whose variance values are identical; replacing $\sum s_j^2$ by $\sum s_j^3$, for example, results in higher values for solutions with long lines, and thus minimizing $\sum s_j^3$ would be a good heuristic for minimizing the length of the longest line.

Once we've determined a criterion to use in comparing solutions, we can think about an algorithm for producing them. We might try the *hill-climbing* heuristic of generating an initial layout of the music, then incrementally improving the solution if possible by moving the line boundaries back and forth. The hill-climbing technique produces a locally optimal solution; the best from a set of such locally best solutions may be chosen as an approximation to the optimal answer, or it may be used to generate constraints for more initial guesses to which hill-climbing may be applied. Hill-climbing is one of the basic tools of the heuristic programmer's 'trade.

A heuristic algorithm, however, will not always guarantee us the best possible answer, and so it's worthwhile to explore the problem a bit further. Our choice (at least temporarily) for a "best" solution will be one whose longest line contains as few notes as possible. One way to get the optimal solution, of course, is to examine all the solutions. A "brute force" algorithm is too slow, however, since there are too many ways-binomial(n-l,k-I), to be exact-to split the n bars onto k lines. Another possibility is the *branch-an&bound* method. We choose where to break the first line, then the second and so on, finally coming up with a solution whose longest line contains p notes; we then backtrack ***(branching)***, choosing different line boundaries, rejecting partial solutions which are bound to lead to complete solutions with an interval of more than p notes and updating p ***(bounding)*** whenever we find a complete solution whose longest line contains fewer than p notes. When traversing the tree of possible solutions, however, we would find that many of the nodes represented identical subproblems (e.g., no matter what way we put the first three bars on the first twd lines, we still have to put the remaining n-3 bars on the remaining k-2 lines). Once again, it appears, we've encountered an application for dynamic programming.

So we start from the bottom of the solution tree and work up. We will find it convenient to let subproblems start at the beginning of the input sequence $a_{,}, \ldots, a_{,;}$ hence we let **L(n,k)** be the length of the longest line in the best layout of the first n bars of the sequence on the first k lines of the score. We have

$$\text{(1)} \qquad L(n,k) = \min_{0<b_1<\ldots<b_{k-1}<n} \max_{1\le j\le k} s_j = \min_{1\le m<n} \max(L(m,k-1), a_{m+1}+\ldots+a_n);$$

i.e., for any solution whose final line break is at position m, either the last line $(a_{,,}+_{,} \ldots . a_n)$ is longest, or one of the previous lines is. Computation of, say, **L(3,2)** for the sequence 3 1 4 1 5 9 2 6 5 3 proceeds as follows:

$$L(3,2) = \min\{\max(L(m,1), a_{m+1}+\ldots+a_3)\} \text{ for } m = 1,2$$
$$= \min\{\max(L(1,1), a_2+a_3), \max(L(2,1),a_3)\}$$
$$= \min\{\max(3,5), \max(4,4)\} = \min\{5,4\} = 4$$

Table I contains values of L(n,k) for k = 1,2,3,4 for this sequence; Figure 2 lists a program to compute them.

3 14 15 9 2 6 5 3

3	4	8	9	14	23	25	31	36	39
	3	4	5	8	14	14	17	22	23
-	-	4	4	5	9	11	14	14	14
		-	4	5	9	9	9	11	14

Table 1 – values of L(n,k) for the sequence 3 1 4 I 5 9 2 6 5 3

```
comment L[n,1] and L[k,k] have previously been initialized;
for k ← 2 step 1 until 4 do
    for n ← ktl step 1 until 10 do
        begin
        L[n,k] ← a very large number;
        SUM ← a[n];        comment SUM will be the sum of a,,, through a,,;
        for m ← n-l step -1 until k do
        comment Compute the minimum as in 1;
            begin
            if L[n,k] > max(L[m,k-1],SUM) then L[n,k] ← max(L[m,k-1],SUM);
            SUM ← SUM + a[m]
            end
        end;
```

Figure 2 – code to compute L values

Computing L with the recurrence relation (1) leads to this $O(kn^2)$ algorithm. Can we do better? Well, it would be nice to avoid computing the table entries for layouts that wouldn't ever be selected (e.g., L[n,1] for small a,). There is no obvious way to do this. If we examine the values of Table 1, we might guess that L(n,k) is related in some easy way to L(n-1,k): the L values, are apparently nondecreasing in n, and in some cases L(n,k)=L(n-1,k). If L(n,k) could be computed from L(n-l,k), we would have an O(kn) algorithm instead of $O(kn^2)$. It turns out that this is the case.

First, we can prove that $L(n,k) \le L(n+1,k)$, using induction on k, then n. For some m, L(n,k) = max(L(m,k- 1), $a_{m+1}+\ldots+a_n$). For some m', $L(n+1,k) = \max(L(m',k-1), a_{m'+1}+\ldots+a_{n+1})$. We assume that $L(n,k) > L(n+1,k)$, i.e., that max(L(m,k- I), $a_{m+1}+\ldots+a_n$) > $\max(L(m',k- I), a_{m'+1}+\ldots+a_{n+1})$. There are two cases: either $m<m'$, or $m'<m$. If the former, we have $L(m',k-1) \ge L(m,k-1)$ by induction. That implies that $\max(L(m,k-1), a_{m+1}+\ldots+a_n) = a_{m+1}+\ldots+a_n$, and that $a_{m+1}+\ldots+a_n$ is greater than both L(m',k-1) and $a_{m'+1}+\ldots+a_{n+1}$-but that means that choosing m' would have led to a smaller value for L(n,k), a contradiction. If $m'<m$, we have $a_{m'+1}+\ldots+a_n > a_{m+1}+\ldots+a_n$, so $\max(L(m,k-1), a_{m+1}+\ldots+a_n) = L(m,k- 1)$, and L(m,k-1) is greater than both $L(m',k-1)$ and $a_{m'+1}+\ldots+a_{n+1}$—again that leads to contradiction, since choosing m' would have led to a smaller value for L(n,k).

We see that m is the "crossing point" of the sequence of L values with the sequence of sums. We can show (via case analysis on the max function) that m need never decrease as we go from L(n,k) to L(n+ 1,k). Intuitively this is clear-to go from L(n,k) to L(n+1,k), we add a,,+, to the last line, and if it doesn't fit we move the boundary between (k-1)st and kth lines over to the right. Code for the resulting algorithm is given in Figure 3; the running time is O(kn) as expected. This kind of improvement is not unusual in dynamic programming applications.

```
comment LASTLEFTMOST[n,k] is the position mt 1 of the leftmost number in the kth line, in an
        optimum layout of the first n bars into k lines. LASTLINESUM[n,k] is the number of
        notes in the last line for the best [n,k] layout. The [k,k] entries in these arrays and the L
        array, along with the L[n,1 J entries, are initialized before processing begins:
        LASTLEFTMOST[k,k ]←k, LASTLINESUM[k,k]←a[k ], L[1,1 ]←a[1], L[n,1]←L[n-1,1]+a[n],
        and L[k,k]←max(L[k-1 ,k-1 ],a[k]);
        for k←2 step 1 until kmax do
          for n←k+1 step 1 until nmax do
            begin
            comment Add a[n J to the last line;
            LASTLINESUM[n,k] + LASTLINESUM[n-1,k]+a[n];
            LASTLEFTMOST[n,k] ← LASTLEFTMOST[ n-l, k);
            while LASTLINESUM[n,k]>L[LASTLEFTMOST[n,k]-1,k-1]
              and L[LASTLEFTMOST[n,k],k-1]≤LASTLINESUM[n,k] do
              comment If adding a[n] makes the last line too big, and moving the line boundary
                      doesn't mess things up, then move it over one;
                begin
                LASTLINESUM[n,kJ ← LASTLINESUM[n,k]-a[LASTLEFTMOST[n,k]];
                LASTLEFTMOST[n,k] ← LASTLEFTMOST[n,k]+1
                end;
            L[n,k] ← max(L[LASTLEFTMOST[n,k]-1,k-1], LASTLINESUM[n,k])
            end;
```

Figure 3 – better code to compute L values

The values of LASTLEFTMOST[n,k] can be used to construct the optimum assignments. The space for the other arrays L and LASTLINESUM can be reduced to two rows since row k depends only on row k-l. Similar algorithms would work with other definitions of "best justification".

Notes for December 7, 1976

Topics of today's discussion:
1. some history about the development of optimization techniques in compilers;
2. practical considerations with respect to optimization;
3. current research areas.

Discussion today centered around the philosophy of code optimization in compilers: why it was considered important, why it has been deemphasized in some compilers, and where research in techniques for optimization is headed.

Knuth and Trabb-Pardo [6] and Rosen [9], in their surveys of the development of programming languages, note the concern of the designers and users of computer languages with the object code their compilers produced. In the early days of computing, this concern was the main obstacle to acceptance of high-level languages. Since memory constraints were severe and uninterrupted computing time was a scarce resource, it was thought that tight coding was a necessity, and that the time a programmer spent writing and debugging a program was of minor importance compared to the efficiency of the result. The middle 1950's, therefore, saw the development of compilers which employed sophisticated algorithms for generating efficient object code.

Work in Russia produced a compiler called ПП ("Programming Program") in 1955 for the STRELA computer. This compiler performed local optimizations on source language expressions to minimize the number of instructions generated to compute them. It avoided recomputing common subexpressions within a single formula. It also produced efficient code to evaluate Boolean expressions, using DeMorgan's rules to compute, say, $(p \wedge q) \vee \neg(r \vee \neg s)$ via the tree in Figure 1. Details about the successor to this compiler may be found in [4].

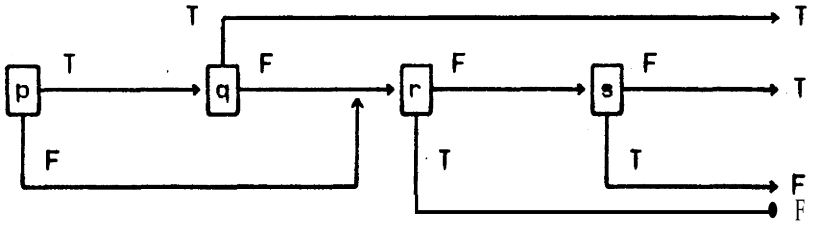


Figure 1 – efficient way to compute $(p \wedge q) \vee \neg(r \vee \neg s)$

IBM in 1957 released the first Fortran compiler (described in [1]), for the IBM 704. Fortran's mathematical notation, its provisions for formatted input and output, and its other programmer conveniences were certainly among the reasons for its enthusiastic acceptance. However, the Fortran I compiler on the 704 also went to great lengths to produce code as good as a human machine-language programmer could produce. It optimized subscript computation for array accessing within nested DO-loops. Its algorithm for optimizing register allocation used information from a flow analysis of the entire program and a simulation of the program's behavior. (Fortran I contained FREQUENCY statements which permitted the programmer to specify how often each branch in an IF statement would be taken; this facility, along with the compiler's simulation phase, was removed from later versions of Fortran.)

Later compilers applied even more ambitious techniques. The Balgol compiler for the Burroughs 220 was the first to use clever data structures for its internal storage; in this way it could compile much faster than had ever been done before, yet it did extensive local optimization. The Fortran compiler for the Univac LARC (Livermore Advanced Research Computer) carried local optimization techniques to the ultimate by, e.g., rearranging expressions via various arithmetic identities, to take maximal advantage of the LARC's 100 arithmetic registers and 99 index registers.

Few current compilers make use of sophisticated local optimization techniques. For one thing, some optimizations may lead to numerically incorrect computations. For example, P*Q/(R/S) might be transformed into P*Q*S/R, to replace a slow division by a faster multiplication; however, different values may arise from the two computations as a result of roundoff error. Another example: sometimes-when X is small-the expression 1.0-X is more accurately computed as .5+(.5-X). Secondly, most programs don't really need local optimizations. Knuth determined, in a study [5] of typical Fortran programs, that complicated expressions rarely appear in programs, and hence sophisticated techniques to optimize long expressions would be infrequently used.

The most important reason for deemphasizing optimization of any kind, however, became obvious as experience with programming was gained. Programs (unlike the mythical goddess Athena) do not burst forth from the mind of the programmer completely correct; a program is compiled many times during the debugging process, and effort spent on optimization of an undebugged program is wasted. Hence most IBM systems, for example, provide compilers like WATFIV and PL/C, which emphasize fast compile time and thorough diagnostic facilities, along with FORTRAN H and the PL/ I Optimizing Compiler.

Features of several current programming systems allow source-level optimizations, which the user may introduce as desired. Sail, for instance, contains both high-level and low-level constructs. It usually turns out that most of a program's processing occurs in around three percent of its code; in a Sail program, this "core" could be converted into low-level STARTCODE routines for maximum speed. Algol 68 provides constructs designed for easy optimization, allowing one, say, to increment an array element with the statement A[I]←*+1. (Unfortunately these constructs are not always reasonably implemented.) Other tools make it easy to determine just where optimizations will be most fruitful. The execution profile, derived either from periodic probes by a monitor into the running program or from explicit trace instructions inserted into the program at compile time, is such a tool, and is provided in the Algol W and Sail systems. One typically notes where a program is spending most of its time, optimizes that section of code, then reruns the program and analyzes it some more. Several such repetitions may yield an impressive improvement in the program's running time; Knuth applied this process to the CDC Fortran compiler and-in two hours-improved its performance by a factor of 4.

Research on methods for code optimization continues. The main shortcoming of today's optimizing compilers is their failure to recognize globally accessed information and keep it in registers throughout execution if possible. Recent progress toward removing this shortcoming was made in Hanan Samet's thesis [IO], for which he implemented a system to determine when two Lisp programs computed the same result; thus the output of two compilers, one an optimizing compiler, can be compared to see if the two sets of code have the same effect. Advances in hardware and software technology create areas for other research. Compilers for computers with parallel and pipelined architectures must produce code which can use these machines efficiently; see, for instance, [7] which describes research in this area at the University of Illinois. Doug Clark, in his thesis work (see [3]), has been one of the few to study how much compilers can benefit from a cache memory. Software-related research areas include development of optimizers for very high-level languages, which need not only to produce efficient code, but also to pick optimal algorithms and data structures to represent high-level constructs in the language. Groups at the University of Rochester and New York University (cf. [8] and [11]) are investigating techniques for data structure selection for very high-level languages. The PSI Program Synthesis System being developed at Stanford (see [2]) incorporates selection of optimal data structures and algorithms for the system's internal very high-level language for modelling programs.

References:

[1] Backus, J.W. et. al., "The FORTRAN Automatic Coding System," *Proc. Western Joint Comp. Con.* (1957), pp. 188- 197; reprinted in *Programming Systems and Languages,* McGraw-Hill, 1967, pp. 29-47.

[2] Barstow, D.R. and Kant, E., "Observations on the Interaction of Coding and Efficiency Knowledge in the PSI Program Synthesis System," *Proc. Second Intl. Conf. on Software Engineering,* Oct. 1976, pp. 19-31.

[3] Clark, D.W. and Green, C.C., "An Empirical Study of List Structure in Lisp," *CACM 20,2,* Feb. 1977, pp. 78-86.

[4] Ershov, A.P., *Programming Programme for the BESM Computer* (translated by M. Nadler), Pergamon Press, 1959.

[5] Knuth, D.E., "An Empirical Study of Fortran Programs," *Software-Practice and Experience J* (1971), pp. 105-133.

[6] Knuth, D.E. and Trabb-Pardo, L., "The Early Development of Programming Languages," Stanford Univ. C.S. Dept. Res. Rpt. CS-562, August 1976.

[7] Kuck, D.J., "Parallel Processing of Ordinary Programs," *Advances in Computers 15* (1976), pp. 119-179.

[8] Low, J.R. and Rovner, P., "Techniques for the Automatic Selection of Data Structures," Univ. of Rochester Res. Rpt. TR4, November 1975.

[9] Rosen, Saul, "Programming Systems and Languages-A Historical Survey," *Proc. Spying Joint Computer Conf.* (1964), pp. I-16; reprinted in *Programming Systems and Languages,* McGraw-Hill; 1967, pp. 3-22.

[10] Samet, H. J., "Automatically Proving the Correctness of Translations Involving Optimized Code," Stanford Univ. C.S. Dept. Res. Rpt. CS-498, May 1975.

[11] Schwartz, J.T., "Automatic Data Structure Choice in a Language of Very High Level," CACM *18,12,* Dec. 1975, pp. 722-728.

Notes for December 9, 1978

Topics of today's discussion:

1. the effects of machine architecture on production schemata to be generated for problem 5;
2. extension of the shortest-path approach to other code generation problems;
3. use of a priority queue in a fast implementation of the shortest-path algorithm;
4. register usage during computation.

Class discussion today covered several topics related to problem 5. One way to apply Knuth's generalized shortest path algorithm to problem 5 involves considering a set of production schemata which represents the instructions of the target computer (this was described in the notes of November 30); we attempted to extend this approach, to cover code generation for conditional expressions and optimization of common subexpressions. We compared the production sets for the two machines given in the problem 5 handout. We also discussed Knuth's fast implementation of the shortest path algorithm using a priority queue (or heap) to store productions yet to be examined, and computed a formula for the number of registers needed to compute a given arithmetic expression.

The schemata for the two machines of problem 5 differ mainly in two ways. For the first machine (the IBM 704), we need never save old values in registers: there are only two In the first place, and there are no instructions which compute a result using both the registers. On the second machine (which resembles the CDC 6000), ail the arithmetic instructions use two registers as operands, so it makes sense to save temporary results therein. Overwriting the temporary values before they're used is a bad idea, of course, and one way to account for registers which contain temporary results is to have a "registers in use" field in each nonterminal. The left hand side of a production for the CDC machine will then have the form

[expression, register, (available registers)],

where the "expression" is merely a pointer to a node in the parse tree for a given expression, and "register" is either X 1, X2, or X3. The set of available registers will include the destination register. Using this representation in general results in twelve nonterminals for each node in the parse tree.

The second difference arises from the structure of the addition and subtraction instructions on the two machines. For the IBM machine, the subtract operation is not symmetric; when the first operand is already in memory, it is faster to compute the negation of the second operand and then add the two. Hence we keep track of the sign of each expression in the productions for the IBM machine: nonterminals are of the form [expression, ±location], where "location" is either memory, register A, or register Q. The CDC machine, on the other hand, requires both operands of a subtract operation to be in registers; if neither operand contains a unary minus, the order in which they are loaded depends only on the number of temporary registers needed to compute each operand, and productions for this machine need not account for the sign of an expression. Unary minus operators do not appear in problem 5; if they were present, we would have to add an extra ± component on our CDC solution, in order to compile optimal code for such expressions as $-((\alpha-\beta)+\gamma)$.

A digression: the unary minus operator has also dirtied up parsing techniques through the years. Floyd, in his article [1] on precedence grammars, was forced to handle the unary minus in an ad *hoc* manner. Wirth, however, combined and extended the notions of precedence grammar and phrase structure grammar in [4], and designed an algorithm for parsing the more general grammar. More recently, Pratt [2] has devised an even nicer way of doing precedence parsing.

We may wonder, having considered an algorithm for optimizing arithmetic expressions, if the algorithm can be adapted in some way to handle conditional expressions (*if* . . . then, . . *else*). This could be more difficult, since a truth value is not specified by the contents of some register but by the position of the program counter or the state of the machine at a given point of execution. We wish to compute the expression *if B* then a *else β* into register A, so our first guess for a production schema might be

[if B then a else *β*, A, . . .]
→ [B, CC] *if* CC then go *to L1*; [*α*, A] *jump L2*; label *L1*: [*β*, AJ *label L2*:

Here CC is a *condition code* and the *label* operator specifies a location which will be the operand of some jump instruction.

Possible problems with this schema lie in two areas. The first are notational: the best notation allows us to see the important concepts clearly, and the use of a condition code, for instance, may obscure some feature of a better production schema. The other difficulties are operational: use of this schema may prevent us from generating optimal code in some cases. A possible notational change would be to introduce a nonterminal which combines "[B,CC] if CC then go *to L1*;" into [B, true to L1]. Operational difficulties arise if we consider the case when a is [if B' then a' else *β'*]; then our method generates a jump to a jump. Associating a jump with each instruction, then making some of the jumps null, is a possible fix. We might also have an implicit label at the end of each production, for example:

[if B then a else *β*, A, L2] → [B, CC] *if* CC then go to *L1*; [*α*, A, go to L2] [*β*, A, L2].

Does this work? 'There will always be two cases: processing either "falls off the end" or exits prematurely from the code we generate. The labels of the target instructions may either be listed in the productions or left implicitly specified. Further consideration of the problem should be interesting; it is therefore left to the reader.

We might also consider using our method to avoid recomputing common subexpressions. This problem in general is NP-complete. However, given an expression which we know we want to check for, we can set up productions whose left hand sides contain the expression and thereby optimize the computation.

Once the production set is decided on, it must be processed. Knuth's algorithm chooses, from a list of unevaluated productions, the production whose cost is minimal. The list may be maintained in various ways; Knuth's impkmentation of the algorithm used a *priority queue*, or heap, in order to access the minimal cost production as quickly as possible. The term "heap" was introduced by J.W.J. Williams in his description [3] of the sorting algorithm "Heapsort". The elements of an array X form a heap if $X[\lfloor k/2 \rfloor] \le X[k]$ for $1 \le \lfloor k/2 \rfloor < k \le n$; hence X[1] is less than or equal to both X[2] and X[3], $X[2] \le X[4]$ and $X[2] \le X[5]$, and so on. The heap represents a binary tree whose root is its minimal element, and whose subtrees are heaps. Keys are entered into the heap at the bottom, then sifted up to their proper position. Deletion from the heap occurs at the root; this leaves a hole at the beginning of the heap which is moved to the end by sifting elements up. (Cf. the Peter -Principle: "In a hierarchy, every employee tends to rise to his level of incompetence.") This sifting operation may be implemented in two ways: either by reconstructing the heap from the top down, leaving a hole somewhere near the end of the heap, filling that hole with the last element and sifting it back up; or by taking the last element and, by starting at the beginning of the heap and comparing in turn with the two descendents, sifting it down. The two methods of deletion are illustrated in Figure 1.

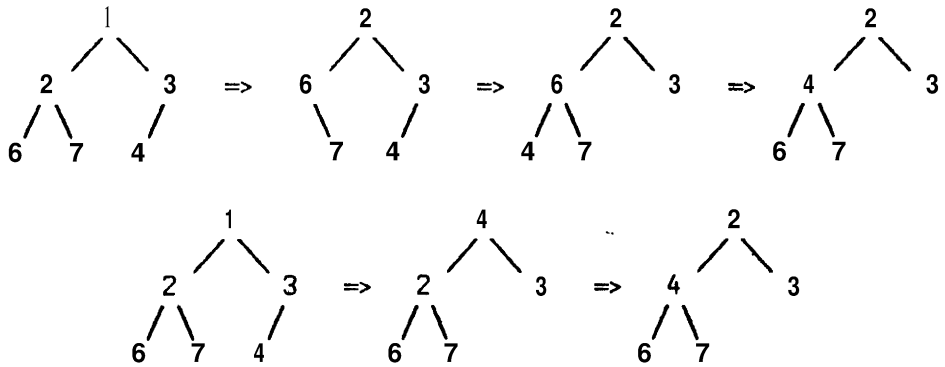


Figure I – deletion of element "1" from a heap

Knuth's solution to problem 5 stored, in a heap, productions whose right hand sides had been totally evaluated. The statement

do production←heapout until value[lhs[production]]>cost[production]

produced the desired minimal cost production (all productions were given an lhs value of ∞, and an lhs value smaller than the current cost indicated that the corresponding nonterminal had already been evaluated).

We may observe that ignoring instruction costs in the program leads, for a given arithmetic **expression, to the solution which computes it in the fewest number of instructions. Furthermore,** if **there is only one instruction that performs a certain operation (e.g. multiplication or** division), then **the running** time **of that instruction does not affect the optimum** code, since we will do the **instruction** a fixed number of times anyway. **We have noted, however, that there** are some ways **to compute b–c•d in four instructions on the IBM machine that are quicker than others.**

The main optimization effort for the CDC machine went toward avoiding the use of 'temporary **storage;** the more quantities we could keep in the three registers, the better, since the machine's arithmetic instructions operate only on register operands. We might ask, then, how many registers are required to compute a given expression. Some examples:

a•b+c•d requires **two registers;**

(a•b+c•d)•(e•f+g•h) requires three registers.

These turn out to be the smallest expressions (i.e. having the least number of operators) which require two and three registers respectively. These expressions also correspond to complete binary trees. We may observe that to force use of four registers, we double the number of variables needed to force three registers. To determine $r(\alpha \text{ op } \beta)$, the number of registers needed to compute an expression a op β, we use the formula

$$r(\alpha \text{ op } \beta) = \max(r(\alpha), r(\beta)) \text{ if } r(\alpha) \neq r(\beta)$$
$$= \max(r(a), r(\beta)) + 1 \text{ if } r(a) = r(\beta).$$

References:

[1] J Floyd, R.W., "Syntactic Analysis and Operator Precedence,,, *JACM 10,3* (July 1963), pp. 316-333.

[2] Pratt, V.R., "Top Down Operator Precedence," *Proc. ACM Symp. on Principles of Programming Languages* (Boston, October 1973), pp. 41-51.

[3] Williams, J.W. J., "Heapsort," *CACM* 7,6 (June 1964), pp. 347-348.

[4] Wirth, N. and Weber, H., "Euler: A Generalization of Algol and Its Formal Definition," *CACM* 9,1 (January 1966), pp. 13-23.

Comments on solution8 to problem 6

The **suggested** method for solving problem 5 involved four parts: (a) construction **of the parse tree** for the input expression; (b) association of productions with the nodes of the parse tree; (c) application of the generalized shortest path algorithm to find the minimal cost computation of **the** expression; and (d) code generation for that implementation. Parts (a) and (d) involved only straightforward tree construction and traversal techniques; these parts were fairly easy.

For part (b), we needed first to determine a set of production schemata which would **represent the** instructions of the target machine; examples of such schemata were discussed in the **notes of** November 30 and December 9. Once a reasonable set of schemata was found, it was a good idea to prune the set of productions that resulted, and this could be done in a variety of ways. **For** instance, all **costs** for the leaf **productions-those which computed the value of** a **variable-could be precomputed,** and the leaf productions discarded. More complicated pruning of the production set required examination and sometimes alteration of the expression parse tree. The number of **registers needed to** compute an expression can be determined from the height of its parse **tree, and it** is normally a good idea to first compute the operand which requires the most **registers, when there is** a choice. For an addition, which is symmetric, we would determine the optimal order of computation of the operands; using this information, we can avoid generating productions which represent evaluation in the opposite order. Also, unary minus signs can be propagated down to the bottom of the parse tree; in this way, some minus signs may be eliminated, and no code efficiency need be lost.

Further reduction of the number of productions depends strongly on characteristics of the target machine like instructions which always operate on specific registers. An example from the IBM computer is the divide instruction; since the register Q always contains the result, the cheapest way to compute the desired quantity in register **A** must be to compute it in Qand transfer **it immediately** afterward (rather than, say, load it from memory). These machine-dependent characteristics could have been input to a solution program by using a more general facility for describing the **target** machine. No one did this, although several programs implemented some of the other optimizations just described.

The shortest path algorithm could be implemented as follows.

I. Initialize the "evaluated list" of nonterminals to empty, and the cost of each nonterminal to ∞.

II. Repeat the following until the top of the parse tree enters the evaluated list:

a. For each production *lhs→rhs* for which *lhs* is not yet in the evaluated list **but all** nonterminals within *rhs* are, repeat the following:

1. Determine the cost associated with *lhs* by summing the costs of the components of *rhs,* **and** let this cost be C.
2. If C < **cost[***lhs***]** then set **cost[***lhs***]** to C.

b. Choose, out of the as-yet-unevaluated nonterminals, the one whose cost is minimal, include it in the evaluated list, note this inclusion in each production whose right hand side contains *lhs,* and remember the production that produced the minimal cost.

Depending on the data structure used, the updating step in **IIb** either merely processed a linear list or else required a search. The other complicated part of the algorithm was the choice (in **IIb)** of the nonterminal to enter the evaluated list. It was worthwhile to maintain the unevaluated nonterminals in a "priority queue" data structure. One such structure was a heap, discussed in the notes of December 9. Examples of optimum code produced by the algorithm appear in **Figure** 1.

(i) a ← b-c•d

IBM 704	CDC 6000
A ← -c	X1 ← c
A ↔ Q	X2 ← d
A ← Q * d	X1 ← X1 * x2
A ← A + b	X2 ← b
a ← A	x3 ← x2 - X1
	a ← x3

(ii) a ← b -(c - (d / (e•f+g•h)) / ((i-j)•(k+l) + m/n))

IBM 704	CDC 6000
A ← m	x2 ← f
Q ← A / n	X1 ← e
t1 ← Q	x3 ← X1 * x2
A ← i	x2 ← g
A ← A - j	X1 ← h
t2 ← A	X1 ← x2 * X1
A ← k	x3 ← x3 + X1
A ← A + l	X1 ← d
A ↔ Q	x3 ← X1 / x3
A ← Q * t2	X1 ← c
A ← A + t1	x3 ← X1 - x3
t1 ← A	t ← x3
Q ← e	X2 ← k
A ← Q * f	X1 ← 1
t2 ← A	x3 ← x2 + X1
Q ← g	X2 ← j
A← Q * h	X1 ← i
A ← A + t2	x2 ← X1 - x2
t2 ← A	x3 ← x2 * x3
A ← d	X2 ← n
Q ← A / t2	X1 ← m
A ↔ Q	X1 ← X1 / x2
A ← A - c	x3 ← x3 + X1
Q ← A / t1	X1 ← t
A ↔ Q	X1 ← X1 / x3
A ← A + b	X2 ← b
a ← A	x3 ← x2 - X1
	a ← X3

Figure 1 – optimal code generated by solutions to problem 5

Appendix 1 -- data for problem 1

Eurasia

36, -6 37,-7 37,-8 39,-9 43,-9 43,-2 46,-1 48,-5 50,1 53,6
54,9 57,8 57,10 55, 11 54,14 55,20 59,23 60,30 60,24 61,21
64,21 66,25 66,22 65,21 64,18 60,18 57,16 56,13 60,11 58,8
59,5 62,5 64,10 70,20 71,30 68,40 69,58 78,105 75,113 72,132
73,145 70,162 70,178 66,-170 65,-173 66,-179 64,177 62,179 60,170 60,167
59,164 54,162 51,157 58,156 60,163" 62,163 59,156 59,143 55,135 53,142
48,140 43,134 43,132 40,127 36,129 35,126 38,126 40,125 39,122 41,122
39,118 38,119 38,123 35,119 30,122 25,119 22,114 21,107 19,106 16,109
12,109 8,105 10,105 14,101 9,99 7,101 6,104 1,104 4,102 6,100
8,98 13,98 17,97 16,94 18,94 23,91 16,80 13,81 10,79 7,77
17,73 22.72 21,70 25,66 25,58 27,56 26,53 30,50 30,48 24,52
24,54 26,56 24,57 22,60 17,57 13,44 21,39 29,35 31,32 32,35
37,36 37,28 41,26 40,23 38,25 36,22 38,21 40,20 42,20 45,13
43,15 40,19 41,17 39.18 38,16 40,16 43,11 44,11 45.9 43,7
44,3 42,4 40,-1 39,0 37,-2 36,-6

Africa

4,-8
-34,18 -17,12 -12,13 -5,12 -1,9 4,10 6,2 31.30- 12,-16 21,-17
28,-14 30,-10 36,-6 37.10 33,11 31,20 33,21 29,33 10,44
12,51 5,49 -3,40 -16,40 -20,35 -25,36 -27,33 -29,32 -34,27 -34,18

Madagascar

-12,49 -25,47 -25,43 -17,43 -12,49

North America

8,-78 9,-79 7,-80 8,-83 13,988 14,-91 16,-95 15,-97 20,-106 21,-105
31,-113 32,-115 23,-109 25,-112 26,-112 28,-115 28,-114 33,-118 34,-121 40,-124
48,-125 58,-137 61,-147 55,-164 58,-158 59,-164 62,-166 64,-161 65,-168 66,-162
68,-166 71,-157 70,-140 69,-115 72,-125 77,-124 83,-65 75,-80 68,-67 66,-61
62,-66 64,-78 65,-74 67,-72 70,-80 67,-80 62,-93 59,-94 57,-92 55,-82
52,-81 51,-79 55,-80 56,-78 59,-79 62,-78 61,-70 59,-70 58,-67 60,-65
52,-55 50,-60 50,-66 46,-59 43,-66 45,-66 43,-70 42,-71 41,-70 41,-73
39,-74 35,-76 32,-81 30,-82 26,-80 25,-81 28,-83 30,-84 30,-85 31,-86
30,-90 29,-89 29,-95 28,-97 22,-97 19,-95 19,-91 21,-90 21,-87 16,-88
15,-83 11,-83 9,-81 10,-79 8,-77

South America

11,-75 12,-71 11,-69 11,-62 7,-58 5,-52 4,-51 0,-50 -3,-40 -5,-35
-8,-35 -13,-38 -20,-40 -23,-42 -26,-48 -28,-48 -35,-53 -35,-58 -37,-56 -39,-57
-39,062 -41,-62 -41,-65 -45,-65 -46,-68 -47,-65 -50,-69 -53,-68 -55,-65 -54,-73
-49,-76 -37,-73 -19,-71 -15,-75 -14,-76 -7,-80 -6,-81 -5,-82 -4,-80 -3,-81
1, -80 3,-78 5,-77 8,-78

Greenland

60,-43 61,-48 67,-53 70,-50 76,-60 76,-68 78,-71 81,-63 84,-30 81,-15
70,-22 68,-27 68,-32 66,-36 65,-40 60,-43

Australia

-11,142 -26,153 -28,154 -38,150 -39,143 -38,140 -33,138 -35,136 -32,134 -32,129
-34,124 -34,120 -35,117 -34,115 -32,116 -22,113 -18,122 -14,127 -15,129 -12,131
-12,137 -15,136 -18,141 -11,142

Great Britain

58,-5 57,-7 56,-5 55,-5 55,-3 54,-4 54,-3 53,-3 53, -5 52,-4
52,-5 51,-3 50,-5 51,1 51,0 52,2 56,-3 57,-2 57,-5 58,-3
58,-5

Ireland
55,-6 55,-8 54,-8 54,-10 53,-9 52,-10 51,-9 52,-6 55,-6

Caspian Sea
48,53 45,57 40,50 37,49 37,55 42,55 45,51 46,54 48,53

Iceland
67,-16 66,-21 67,-23 65,-22 65,-24 63,-19 65,-14 67,-16

Black Sea
47,32 43,27 41,29 42,35 41,38 42,42 45,36 44,34 47,32

Japan
46,142 43,140 39,139 34,130 31,130 35,140 42,141 44,146 46,142

Cuba
23,-81 22,-85 22,-80 20,-77 21,-74 23,-81

Haiti, Dominican Republic
20,-73 19,-72 18,-75 18,-69 20,-70 20,-73

Borneo
2,109 -3,111 -4,116 2,118 7,117 2,109

New Zealand
-46,172 -47,166 -42,169 -41,174 -46,172 -35,173 -39,173 -42,175 -38,178 -35,173

New Guinea
-3,141 -1,131 -9,141 -8,145 -11,151 -7,148 -3,141

Singapore, Indonesia
5,95 -7,106 -8,128 -5,106 -2,107 5,95

Antarctica
-63,59 -73,77 -72,102 -74,100 -73,127 -77,-165 -71,-170 -66,-138 -67,-82
-69,-74 -66,-53 -70,9 -79,52 -63,59

Appendix 2 -- data for problem 4

There are four kinds of tradeoff lists in the network:

#1:	120,13	111,17	92,23	66,29	54,36	40,45	31,58
#2:	186,16	161,25	157,35	123,51	111,68	96,86	65,113
#3:	300,17	278,23	235,37	171,50	151,74	141,97	133,127
#4:	399,24	371,41	313,59	235,92	196,131	152,169	131,222

The network has fifty vertices, numbered 1 thru 50. The root is number 17. The arcs of the network are defined by forty-nine procedure statements of the form ***data(u,v,t)***, meaning that the path from u to the root first goes through v, and the corresponding tradeoff **list** is **#t**. Here are the forty-nine data statements, which you might simply wish to insert into your Sail program:

```
data(1,41,3);
data(2,24,3);
data(3,42,2);
data(4,13,3);
data(5,37,2);
data(6,46,3);
data(7,23,4);
data(8,23,3);
data(9,19,2);
data(10,17,2);
data(11,9,3);
data(12,50,4);
data(13,48,3);
data(14,30,4);
data(15,5,2);
data(16,44,2);
data(18,37,2);
data(19,30,2);
data(20,50,3);
data(21,16,3);
data(22,43,2);
data(23,10,1);
data(24,47,2);
data(25,18,1);
data(26,38,2);
data(27,22,1);
data(28,46,1);
data(29,42,2);
data(30,25,1);
data(31,9,2);
data(32,28,4);
data(33,6,2);
data(34,15,2);
data(35,47,4);
data(36,23,4);
data(37,23,1);
data(38,20,1);
data(39,22,1);
data(40,2,4);
data(41,46,4);
data(42,34,3);
data(43,16,2);
data(44,28,1);
data(45,23,3);
data(46,49,2);
data(47,17,1);
data(48,28,3);
data(49,24,1);
data(50,30,1);
```