Controller Compensators with Demo

02/26/96

This document is divided in four parts:

An **Introduction** defines the computational implementation issues behind Control Compensator design in a Control Idiom ICL context. This is elaborated in:
- Discussion of general uses of **Control** (Real Valued or Algebraic) **Compensators**, and
- The three basic **Simple, Shared, and Dual Compensators**, and
- Other cases, **Characterized** and **Discrete (vs. Real) Compensators**.

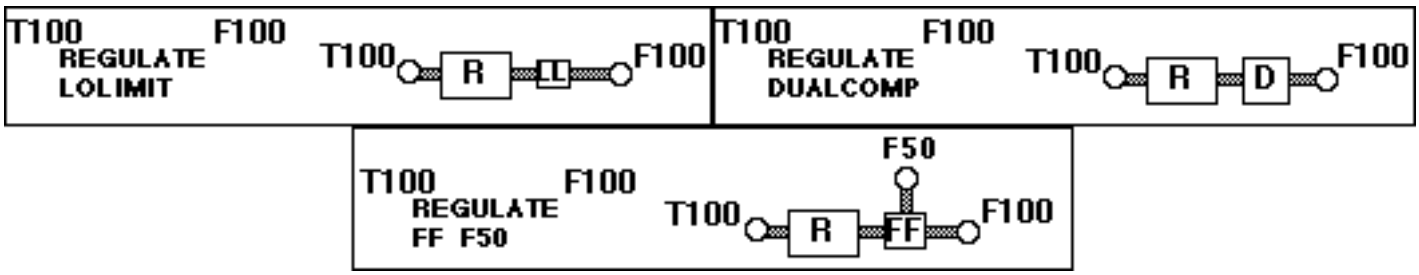
A discussion of the **The Algebraic Compensator Proposal** itself, including the intended variants, and the **Default Inversion Algorithm**.

A description of the **Character Demo** itself, with operating instructions.

An **Appendix** listing the **Demo Code** itself, with line numbers.

Introduction

This document discusses the Compensator as it might be implemented with Idioms or in ICL, first introducing its different roles and syntax, and then discussing and demonstrating its implementation. ICL Compensators are one more case where ICL improves on traditional block control structures, using the simplified Idiom Syntax. Compensator Idioms will be programmed by listing their names after the basic controller (**REGULATE**) Idiom name, modifying its function appropriately. The associated computational statements are defined in a separate compensator definition. No further program connections are needed.



The Compensator Idiom then serves the same documentation goals as the Footnote in a more specialized context. The result allows maximum flexibility with least possible use of memory. At the same time the usage provides disciplined documentation rather than random spaghetti algebraic expressions.

Control Compensators; Simple, Shared, and Dual

Control, as generally addressed with Idioms, covers the operation of real valued sensors and actuators, operating in Degree of Freedom paths to provide feedback control of continuous processes. In this context, the algebraic (or other) statement of the compensation must serve not only to compute the intended change in signal value but to compute associated (e.g. external feedback) signal paths. There are three basic cases (with one variant):

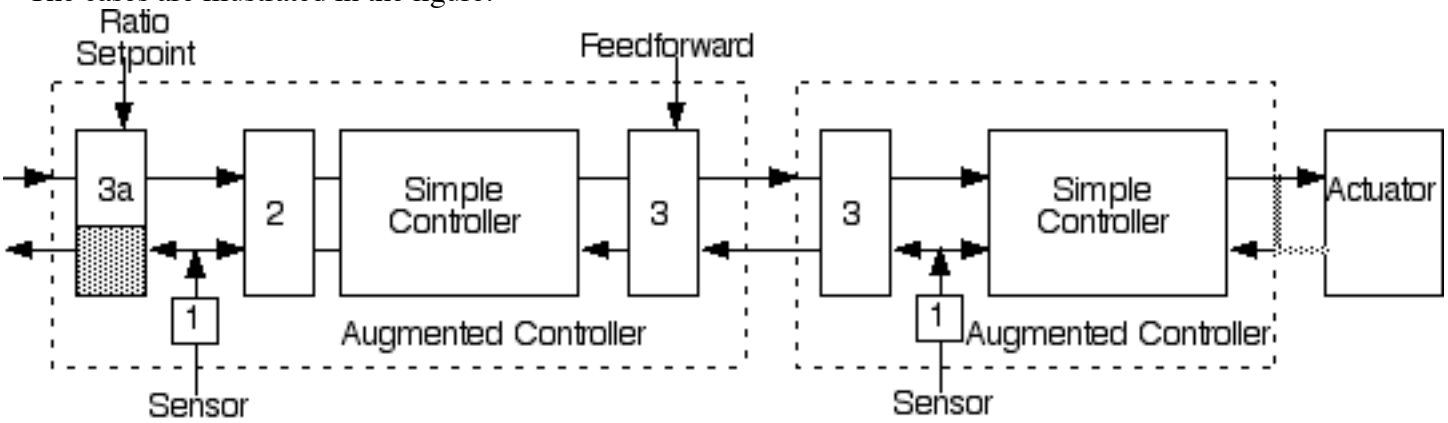
1. Simple; a Compensator applied to a single value, assumed to be the sole source of that data and its representations, as in the case of a sensor, representing the normal flow of measurement data from a sensor.
2. Shared; when a Compensator is applied to two parallel values as in the case of a controller measurement which is modified (e.g. linearized) before being applied, but whose uncompensated value represents the user's operational value. In this case the controller setpoint must also be similarly modified for the controllers use even as the unmodified values are preserved as the basis for all operational use.
3. Dual; when a controller output value is compensated for its use when passed to a following device in such as way that the resulting compensated setpoint or actuator target value is of operational significance to the user and/

or when there is a corresponding true external feedback source of the corresponding actual process value.

The downstream new source may reflect an actual new measurement of the process variable, some constructed value, or in the case of a "dumb" actuator simply as coupling to the original output. In any case, this downstream value must be inverse compensated to provide an actual external feedback value, corresponding to the original uncompensated controller output. The compensation and inverse compensation can include a feedforward measurement or value as shown with the FeedForward (FF) Idiom above.

3a. Ratio; for historical reasons the special case of rationing makes the the operational Setpoint, the Ratio Setpoint, and the controller Setpoint all operationally significant (accessible to the operator).

The cases are illustrated in the figure:



This discussion assumes that the role of Real Valued Compensator is to make single signal or connection transformations, as part of more general control function. This assumes that there will be a single signal or single output variable and a single basic input variable, although the output may be computed from several additional associated (e.g. feedforward) variables or data values. Behind this is the concern that the Compensator never be used to alter the basic Degree of Freedom structure of the controls, simply to compensate it. **The concept can be generalized to express a situation where several basic variables are combined to compute a corresponding number of basic output values.** However the Compensator capability described (in the default inversion algorithm) is not set up for this generalization.

Other Cases: Characterized and Discrete Compensators

Compensators can be based on Characterizers, that is tables of distinct values which are interpolated to approximate some computed or experimentally derived function. In this case both the direct and the inverse Compensator are equally easily achieved by interpolation.

In a second case, Compensators may be useful in discrete (logical) control systems to provide an interface between standardized control signals (e.g. **INITIALIZE/START/HOLD/CONTINUE/STOP/NEXTSTEP** states) and the internal states associated with a special device.

Compensator Proposal

As applied to the Compensator Function, the proposal is that the basic Compensator function be implemented in the ICL, in infix (algebraic) expressions. The inverse function might be solved automatically in algebraic form, entered manually, or alternatively, inverted by an approximation method, expressed using the default inversion algorithm described next.

Default Inversion Algorithm

In a nutshell the approach is to develop a running succession of two most recent guesses of the inverse, interpolate solving on each such pair to generate the next guess. The method controls the divergence possible from

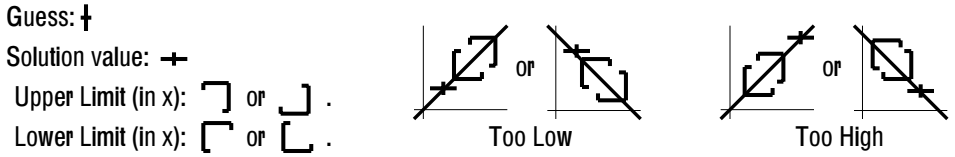
similar methods (Horner's and Newton's methods) by keeping track of a pair of best case bounds about the solution, one above the solution and one below, bounding each guess to be between the bounds, and moving either bound when the latest guess defines a new bound on the same side of the solution as the old bound and closer to the solution. The input arguments to the algorithm are in order (on line 607 , the starting line for the corresponding routine invfunc):

[`607: float invfunc(float xmx, float xmn, float ymx, float ymn, float yans, float x1, float y1, float dl, float dll)`**]**

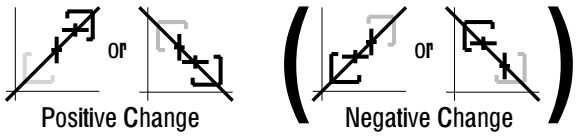
- Maximum value of x (the independent variable to be solved).
- Minimum value of x.
- Value of y (the dependent variable) corresponding to the Maximum x.
- Value of y (the dependent variable) corresponding to the Minimum x.
- y value whose corresponding x value is to be solved.
- Initial guess for x.
- y value corresponding to the initial guess for x.
- Parameter sizing the separation of initial guesses (=.1, in the demonstration).
- Parameter sizing the final guess bound separation (=.06, in the demonstration).

The algorithm follows roughly the following steps (correlated below with the line numbers in the appendix listing).

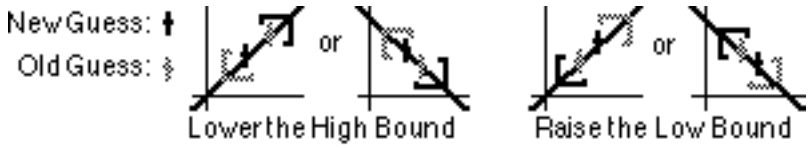
- The internal algorithm bounds are initialized to correspond to the minimum/maximum bounds of the function, and a delta x is computed for these bounds. [Line: 614]
- The initial guess is clipped to fit the bounds. [Line: 615]
- The data is tested to for answer out of bounds, too high (j coded -1) or too low (j coded +1) (in x). [Lines: 617-618]



- Otherwise (j =0), The answer is tested as being high (in x) of starting point, in which case a positive change to get a new guess is imposed. (The new guess is bounded as before.) [Lines: 621-624]
- Or if the answer is low, a negative change is imposed (bounded). [Lines: 625-628]



- Set up the inverse iteration (loop). [Line: 634]
- If new x guess is bounded use the corresponding y guess (to save a recomputation). Otherwise carry out a function calculation to get the y value. [Lines: 635-640]
- Save the absolute difference between new and old guess, and interpolate a new guess. [Lines: 642-643]
- The new guess (for the next iteration) becomes the old guess. [Line: 644]
- Bound the new guess or answer. [Lines: 645-646]
- Lower the high bound to the original old guess if this guess is between the new guess and the upper bound (The previous interpolation drives below the original old guess, placing the answer below it.). [Line: 647]
- Else raise the low bound to this original old guess (for the analogous reason). [Line: 648]



- Iterate as long as the absolute difference (in x guesses) is greater than a set fraction of the x data range. Otherwise return the new guess.

Compensator Demo

This section is in essence an instruction manual for running the demonstration program itself. The program is not expressed as any approximation to a real Compensator Idiom or configurator. Instead it consists of a text interaction demo, demonstrating the key programmed elements in the configuration and computation of such a block. Any work, to create such a Compensator, would require the independent development of the appropriate Extender and Configurator system code. The demo does use compilation to Reverse Polish encoding, and interpretation of that encoding to execute each computation of the entered Compensator function and its explicit inverse or its algorithmic inversion.

The discussion will describe the files needing compilation and running, the command line routine name as expected and the following interaction driven by the program. At key points in the interaction, notes (set off in braces or offset) will describe the action being carried out and lines in the appendix program listing where the described interaction takes place. These notes are provided to allow the reader to get a greater sense of what is going on and locate it in the program if desired. The notes are not needed to demonstrate or test the intended action.

The listing described later is referred to as `invtstt.c`. The opening lines in the program contain a test script for running the compiler and program in the NeXT UNIX environment. in that context the executable code is named `ivtt.out`. On DOS disks the source file will be named `ivtt.c`. When the executable is run the result is the lines shown below:

```
localhost> ivtt.out
---------------------------------------
Program to demonstrate inline compensator function, with word forming, polish translation,
   table print out, inverse guess, inversion table print out, and back compilation
Entered functions are stated in an expression which allows for parentheses, with 20 levels of nesting;
   Add (+), Subtract (-), Multiply (*), Divide (/), and Power (^) Operators; and sin, cos, ln, and exp
   functions. Function parentheses must follow their function name, without intervening spaces. The
   expression can include any named parameter or variable (in addition to the presumed x and y), named
   with any combination of letters, digits, or embedded single spaces, not confused with a number.

Enter function in form y = f(x), x to be solved, or "debug", or "stop":

      y =
```

[Lines: 686-689]

This is a request to enter in an algebraic expression defining the intended value of y as a function of x. The demo supports the normal add (+), minus (-), multiply (*), divide (/), or exponentiation (^) operators as well as parentheses, and sin, cos, ln, and exp functions. The left parenthesis associated with a function call must immediately follow the function without intervening spaces. The command "stop" entered at this point causes the program to exit. The command "debug" causes the program to print additional diagnostic data about its actions. In this case we enter in the following expression:

```
Enter function in form y = f(x), x to be solved, or "debug", or "stop":

      y = (x^2+5)+a
```

The additional "a" in the expression represents an additional variable or parameter. The system then responds as

below:

```
Enter inverse function in form x = f(y), solved for x, or enter "default":

       x =
```

[Lines: 690-694]

At this point the user can enter an algebraic expression expressing the inverse relation defining x as a function of y. A carriage return without expression or the word "default" causes the system to invoke the default inversion algorithm. We will show both cases. With an explicit expression the result is as follows, with intervening offset explanation:

```
Enter inverse function in form x = f(y), solved for x, or enter "default":

       x = (y-5-a)^.5

y =  (x^2+5)+a; x = (y-5-a)^.5
```

[Lines: 695-697]

When an explicit default is executed, Line 697 prints out the original expression and its inverse separated by a semicolon.

```
y = (x^2.000000 +5.000000)+a
```
Print Back-Compiled Expression. **[Line: 747+]**

```
x = (y-5.000000 -a)^0.500000
```
If explicit inverse, print Back-Compiled Expression. **[Line: 757]**

```
***2***
```

[Lines: 766-768]

```
y 0.328571 x 0.471429 )*( a 0.900000
2.000000 5.000000 0.500000
***
```

At this point the system prints out its internal symbol table with initial default values, separating the presumed x and y variables from the additional variables implied by the user in his expression (a above). In the intervening Lines 701-765, the system analyses the function and inverse, grouping of letters into words (Lines 701-719), converting the resulting tokens to Reversed Polish form (Lines 720-738), check demo back compiling the Reverse Polish back into conventional notation (Lines 739-758), and tabulating the resulting functions (Lines 759-765). The results of these intervening actions can be seen if the "debug" command is entered at the earlier point in the interpretation.

```
Enter Working Values, one for each Variable after y:
```

[Line: 769]

The system now requests initial consecutive values for each of the declared variables starting from x, simulating the effect of some control calculation (y will be calculated from x). In this case we enter in the values 4 and 7; If there are more variables with no further entered values the system defaults these values with "random" values.

```
Enter Working Values, one for each Variable after y:
4 7
y = 0.328571; x = 4.000000; a = 7.000000;
```

[Lines: 770-773]

The entered variables are redisplayed. Then the symbol table is redisplayed.

```
***3***
y 0.328571 x 4.000000 a 7.000000
2.000000 5.000000
***
```

[Lines: 774-777]

From this point on, the function and whatever form of inverse desired is analyzed and tabulated. The analysis looks for the worst case reversal in direction (lines 791-793 initialized in lines 779,782) to check the monotonicity of the function. If the function is monotonic this is indicated as below, otherwise the worst case reversal is defined in percent

of full scale with the x value corresponding to this reversal, and a table of function and inverse is displayed. The maximum inverse error in percent of scale is given, with the corresponding x value.

```
***
Fully Monotonic: Positive Monotone.                                    [Lines: 804-805]
+++
 0.0:   12.000;   12.000,  0.0000                                      [Lines: 808-813]
 5.0:   37.000;   37.000,  5.0000
 10.0:  112.000;  112.000,  10.0000
 15.0:  237.000;  237.000,  15.0000
 20.0:  412.000;  412.000,  20.0000
 25.0:  637.000;  637.000,  25.0000
 30.0:  912.000;  912.000,  30.0000
 35.0:  1237.000;  1237.000,  35.0000
 40.0:  1612.000;  1612.000,  40.0000
 45.0:  2037.000;  2037.000,  45.0000
 50.0:  2512.000;  2512.000,  50.0000
 55.0:  3037.000;  3037.000,  55.0000
 60.0:  3612.000;  3612.000,  60.0000
 65.0:  4237.000;  4237.000,  65.0000
 70.0:  4912.000;  4912.000,  70.0000
 75.0:  5637.000;  5637.000,  75.0000
 80.0:  6412.000;  6412.000,  80.0000
 85.0:  7237.000;  7237.000,  85.0000
 90.0:  8112.000;  8112.000,  90.0000
 95.0:  9037.000;  9037.000,  95.0000
 100.0:  10012.000;  10012.000,  100.0000
+++
Maximum error at x = -5.0 of 0.000000% of full scale range.
***
----------------------------------------
Program to demonstrate inline compensator function, with word forming, polish translation,
   table print out, inverse guess, inversion table print out, and back compilation
Enter function in form y = f(x), x to be solved, or "debug", or "stop":

       y =
```

The system then re-loops back to repeat the process, or to accept a "stop" command:

```
Enter function in form y = f(x), x to be solved, or "debug", or "stop":

       y = stop
localhost>
```

The four columns in the above table are, from left to right: the values of x evaluated for 0 to 100 (%) in increments of 5, the corresponding computed values of y, the corresponding values of y computed from the the computed inverse of the previous column, and the computed inverse values. In this case the computed error is displayed as 0%. If a default inverse had been called for the result would have been as below:

```
localhost> ivtt.out
----------------------------------------
Program to demonstrate inline compensator function, with word forming, polish translation,
   table print out, inverse guess, inversion table print out, and back compilation
Enter function in form y = f(x), x to be solved, or "debug", or "stop":

       y = (x^2+5)+a

Enter inverse function in form x = f(y), solved for x, or enter "default":
```

x = **4 7**

y = (x^2+5)+a; Default inverse

y = (x^2.000000 +5.000000)+a

2
y 0.328571 x 0.471429)*(a 0.900000
2.000000 5.000000
***

Enter Working Values, one for each Variable after y:
4 7
y = 0.328571; x = 4.000000; a = 7.000000;

3
y 0.328571 x 4.000000 a 7.000000
2.000000 5.000000
***

***
Fully Monotonic: Positive Monotone.
+++
 0.0: 12.000; **12.000, 0.0000**
 5.0: 37.000; **37.132, 5.0131**
 10.0: 112.000; **107.349, 9.7647**
 15.0: 237.000; **236.381, 14.9794**
 20.0: 412.000; **408.111, 19.9025**
 25.0: 637.000; **638.697, 25.0339**
 30.0: 912.000; **922.294, 30.1711**
 35.0: 1237.000; **1273.102, 35.5120**
 40.0: 1612.000; **1605.237, 39.9154**
 45.0: 2037.000; **2036.861, 44.9985**
 50.0: 2512.000; **2511.914, 49.9991**
 55.0: 3037.000; **3031.769, 54.9524**
 60.0: 3612.000; **3608.624, 59.9719**
 65.0: 4237.000; **4235.047, 64.9850**
 70.0: 4912.000; **4911.007, 69.9929**
 75.0: 5637.000; **5636.568, 74.9971**
 80.0: 6412.000; **6411.850, 79.9991**
 85.0: 7237.000; **7236.962, 84.9998**
 90.0: 8112.000; **8114.523, 90.0140**
 95.0: 9037.000; **9037.336, 95.0018**
 100.0: 10012.000; **10012.000, 100.0000**
+++
Maximum error at x = 35.0 of 0.512001% of full scale range.
***

Program to demonstrate inline compensator function, with word forming, polish translation,
   table print out, inverse guess, inversion table print out, and back compilation
Enter function in form y = f(x), x to be solved, or "debug", or "stop":

y =

In this case , everything associated with the explicit inverse is bypassed and the internal algorithm is executed. The table shows, in each row, the approximated inverted x for the computed y, corresponding to the true x for the row and an initial guess taken as the originally entered value for x (4). This simulates the behavior of a controller output (x=4) being sent through the function and then (after any downstream constraints) being back calculated (inverted) under the initial guess based on the assumption of the original value. The third (y) column shows the affect of this error on the value of y if it is computed from the approximate inverse. This table thus shows, for a given input value to the function, what the worst case effect on the back calculation will be. The inverted function is a single value function with any number of additional parametric variables in the expression.

The affect of selecting the "debug" command, is shown with a function like the initial example, except that a sinusoidal term is added to the function and ignored in the inverse. The differences are shown in bold face:

```
localhost> ivtt.out
---------------------------------------
Program to demonstrate inline compensator function, with word forming, polish translation,
   table print out, inverse guess, inversion table print out, and back compilation
Enter function in form y = f(x), x to be solved, or "debug", or "stop":

      y = debug
-----------------------------------------
Program to demonstrate inline compensator function, with word forming, polish translation,
   table print out, inverse guess, inversion table print out, and back compilation
Enter function in form y = f(x), x to be solved, or "debug", or "stop":

       y = (x^2+100*sin(x) + 5) + a

Enter inverse function in form x = f(y), solved for x, or enter "default":

      x = (y-5-a)^.5

y = (x^2+100*sin(x) + 5) + a; x = (y-5-a)^.5



Lexical processing (Word Forming):
Success:(@!#!#!&(@)!#)!@
Lex:[(][x][^][2.000000] [+][100.000000] [*][sin][(][x][)][+][5.000000] [)][+][a]
Success:(@!#!@)!#
Lex Inverse:[(][y][-][5.000000] [-][a][)][^][0.500000]

Polish Expression Translation:
Success:
Rpol from lexed tokens:[x][2.000000] [^][100.000000] [x][sin][*][+][5.000000] [+][#][a][+]
Success:
Inverse Rpol from lexed tokens:[y][5.000000] [-][a][-][#][0.500000] [^]

Back Compiling Demonstration:
Success:
Back Comp:[(][x][^][2.000000] [+][100.000000] [*][sin][(][x][)][+][5.000000] [)][+][a]
y = (x^2.000000 +100.000000 *sin(x)+5.000000 )+a
Success:
Inverse Back Comp:[(][y][-][5.000000] [-][a][)][^][0.500000]
x = (y-5.000000 -a)^0.500000
```

```
Tabular Function Output:
0.00: 5.9000 inv: 0.0003
5.00: -64.9924 inv: NaN
10.00: 51.4979 inv: 6.7526
15.00: 295.9288 inv: 17.0302
20.00: 497.1945 inv: 22.1652
25.00: 617.6649 inv: 24.7339
30.00: 807.0969 inv: 28.3054
35.00: 1188.0818 inv: 34.3829
40.00: 1680.4114 inv: 40.9208
45.00: 2115.9902 inv: 45.9357
50.00: 2479.6624 inv: 49.7369
55.00: 2930.9243 inv: 54.0835
60.00: 3575.4189 inv: 59.7455
65.00: 4313.5830 inv: 65.6329
70.00: 4983.2891 inv: 70.5506
75.00: 5592.1216 inv: 74.7410
80.00: 6306.5112 inv: 79.3764
85.00: 7213.2925 inv: 84.8964
90.00: 8195.3008 inv: 90.4953
95.00: 9099.2266 inv: 95.3589
100.00: 9955.2637 inv: 99.7465

***2***
y 9955.263672 x 100.000000 )*( a 0.900000
2.000000 100.000000 5.000000 0.500000
***

Enter Working Values, one for each Variable after y:
4 7
y = 9955.263672; x = 4.000000; a = 7.000000;

***3***
y 9955.263672 x 4.000000 a 7.000000
2.000000 100.000000 5.000000 0.500000
***

1MN(0.00;12.0000) 1MX(100.00;9961.3633)
0.00: 1\(12.00) :1
5.00: 1\(-58.89) :1
10.00: 1\(57.60) :1
15.00: 1\(302.03) :1
20.00: 1\(503.29) :1
25.00: 1\(623.76) :1
30.00: 1\(813.20) :1
35.00: 1\(1194.18) :1
40.00: 1\(1686.51) :1
45.00: 1\(2122.09) :1
50.00: 1\(2485.76) :1
55.00: 1\(2937.02) :1
60.00: 1\(3581.52) :1
65.00: 1\(4319.68) :1
70.00: 1\(4989.39) :1
75.00: 1\(5598.22) :1
```

```
80.00: 1\(6312.61) :1
85.00: 1\(7219.39) :1
90.00: 1\(8201.40) :1
95.00: 1\(9105.33) :1
100.00: 1\(9961.36) :1
***
Max Out of Monotonicity at x = 5.0 of -0.712532% of full scale range; Positive Monotone.
+++
 0.0:   12.000;   12.000,  0.0000       -1;0.000000
 5.0:  -58.892;      NaN,  NaN        -1;0.000000
 10.0:   57.598;  102.836,  6.7526       2;-3.247379
 15.0:  302.029;  205.101,  17.0302       2;-3.247379
 20.0:  503.295;  485.981,  22.1652       2;-3.247379
 25.0:  623.765;  584.928,  24.7339       2;-3.247379
 30.0:  813.197;  810.089,  28.3054       2;-3.247379
 35.0:  1194.182;  1211.558,  34.3829       2;-3.247379
 40.0:  1686.511;  1678.511,  40.9208       2;-3.247379
 45.0:  2122.090;  2214.859,  45.9357       2;-3.247379
 50.0:  2485.762;  2435.334,  49.7369       2;-3.247379
 55.0:  2937.024;  2874.424,  54.0835       2;-3.247379
 60.0:  3581.519;  3576.003,  59.7455       2;-3.247379
 65.0:  4319.683;  4353.079,  65.6329       2;-3.247379
 70.0:  4989.389;  5088.477,  70.5506       2;-3.247379
 75.0:  5598.222;  5537.131,  74.7410       2;-3.247379
 80.0:  6312.611;  6238.376,  79.3764       2;-3.247379
 85.0:  7219.393;  7212.062,  84.8964       2;-3.247379
 90.0:  8201.400;  8258.760,  90.4953       2;-3.247379
 95.0:  9105.326;  9194.947,  95.3589       2;-3.247379
 100.0:  9961.363;  9890.720,  99.7465       2;-3.247379
+++
Maximum error at x = 10.0 of -3.247379% of full scale range.
***
---------------------------------------
Program to demonstrate inline compensator function, with word forming, polish translation,
   table print out, inverse guess, inversion table print out, and back compilation
Enter function in form y = f(x), x to be solved, or "debug", or "stop":

      y =
```

The result shows the non-monotonicity as well, as the detailed parsing actions. The algorithm permits inversion even of such functions, but this is beside the point of this application.

Appendix: Demo Code

This Appendix includes the demo listing, with line numbers. Line number cross references to called routines are added in boldface, both in the prototypes below and in central parts of the demo code. This code was developed from a collection of expression processing routines; any extra routines or references to deleted routines stem from this. The original code was some of my first C code, written in Instant C; I apologize for the resulting format. The same line numbers are sited above in the discussion.

```
   1: /*  INFIXTESTDEMO Version 2/21/96 for Compensator Proposal
   2: Edited to support value pointers for variables; invtstt.c is the UNIX file name or ivtt,c for DOS
   3: cd /ICLCntrl/ICLBlkIA/infix
   4: cc -v -o ivtt.out invtstt.c
   5: ivtt.out
```

```
 6: x^2+5
 7:
 8: 4
 9: (x^2)+5
10:
11: -4
12: (x^2+5)
13:
14: -10
15: ((x^2)+5)
16:
17: 96
18: x^2+5
19:
20: 104
21: x^2+5
22:
23: 110
24: x^2+5+sin(10*x)
25:
26: 4
27: ((x^2)+5)+(sin(10*x))
28:
29: -4
30: x^2+5+sin(10*x)
31:
32: -10
33: x^2+5+sin(10*x)
34:
35: 96
36: x^2+5+sin(10*x)
37:
38: 104
39: x^2+5+sin(10*x)
40:
41: 110
42: stop
43: */
44: #include <stdio.h>
45: #include <math.h>
46: /*#define NULL 0*/
47: char opsymb[12][2] = {"(", ")", "=", ".", ",", ":", ";", "+", "-", "*", "/", "^"};
48: char funcsymb[6][4] = {"sin", "cos", "ln", "exp", "#", "~"};     /* Replacing pi (#) and um (~) */
49: char syskeywd[13][5] = {"ramp", "from", "to", "in", "sec", "min", "hr", "wait", "sec", "min", "hr", "in", "on"};
50: char *syskeypt[13];                          /* Initialized in main. */
51: char nlchr[2] = "_";
52: char varsymb[20][20];
53: int precid[5] = {1, 1, 2, 2, 3};
54: int indvar, new_iv;                          /* Indices into Varsymb */
55: float varval[20];
56: float numsymval[20], zero = 0, nlval = 0;
57: char *lexout[50], *lexouti[50], *rpolout[100], *rpolouti[100], *bacompout[100], *bacompouti[100], *odisorout[100];
58: char **plex, **prpol, **pbac, **podis;       /*  Indexing Pointers   */
59: char ffl;                                    /* Initialized to 1 in main command loop */
60: char dfl, ifl;                               /* Debug Flag; Inverse Flag */
61: char *keyfl, *pwdd;                          /* keyfl initialized to NULL in main command loop */
62: char *stack[20];
63: char **pstack;
```

```
 64: char **bstack[20], ***pbstack;
 65: int *istack[20], **pistack, dir[20], *di;
 66: struct vartab
 67:         {
 68:         char *var, **varst, **varnd;
 69:         } vart[10], *varend;
 70: float fstack[20];
 71: float *pfstack;
 72: char word[20], nonword[20], wordd[20];       /* wordd initialized in main command loop */
 73: float fl, *pf;                               /* FP pointer for above char pointer buffers */
 74: char *left_paren, *right_paren, *minus_sign, *equal_sign;          /* Initialized in main */
 75: char *inp_curs, **lex_i_c;                   /* Used in chksys(i), stp_inp_crs, rstr_i_c, lexsys(i), finish(i)
 76:                                                lexword */
 77: /*void printobj();
 78: char *lexvar();
 79: char *lexword();*/
 80:
 81:
 82: /*************** Start of Prototypes*************/
 83: /*
 84: char *defined();       /* line #: 156 */
 85: char *prefix();        /* line #: 173 */
 86: char *operator();      /* line #: 199 */
 87: char *opparen();       /* line #: 217 */
 88: char *operand();       /* line #: 236 */
 89: float *number();       /* line #: 255 */
     float *symbval()      /* line #: 282 */ /* Line added for completeness */
 90: */
 91: int objcmp();          /* line #: 300 */
 92: /*
 93: char *makpref();       /* line #: 317 */
 94: char *chksys();        /* line #: 326 */
 95: char *makvar();        /* line #: 356 */
 96: int prec();            /* line #: 373 */
 97: void printokstr();     /* line #: 387 */
 98: void printstr();       /* line #: 411 */
     void printstrv();     /* line #: 431 */ /* Line added for completeness */
 99: */
100: void printobj();       /* line #: 454 */
101: void printobjv();      /* line #: 468 */
102: /*
103: void printtabls();     /* line #: 487 */
104: void prpolstk();       /* line #: 509 */
105: void clear();          /* line #: 521 */
106: void stp_inp_crs();    /* line #: 539 */
107: void rstr_i_c();       /* line #: 553 */
108: */
109: /*----------------*/
110: int assign();          /* Deleted */
111: /*
     int cvscan();         /* line #: 564 */ /* Line added for completeness */
     float calcfunc();     /* line #: 588 */ /* Line added for completeness */
     float interp();       /* line #: 599 */ /* Line added for completeness */
     float invfunc();      /* line #: 607 */ /* Line added for completeness */
     void main();          /* line #: 656 */ /* "democalculator" Line added for completeness */
     void printstrv();     /* line #: 431 */ /* Line added for completeness */
```

```
112: char *lexpref();       /* line #: 823 */
113: char *lexopand();      /* line #: 865 */
114: */
115: char *lexvar();        /* line #: 888 */
116: /*
117: int lexsys();          /* line #: 911 */
118: int finish();          /* line #: 926 */
119: */
120: char *lexword();       /* line #: 969 */
121: /*
122: char *lextok();        /* line #: 1078 */
123: */
124: int lex();             /* line #: 1095 */
125: /*
126: void push();           /* line #: 1117 */
127: void error();          /* line #: 1131 */
128: */
129: int rpol();            /* line #: 1145 */
130: int backcomp();        /* line #: 1224 */
131: /*
132: float domon();         /* line #: 1302 */
133: float dobin();         /* line #: 1329 */
134: void pushf();          /* Deleted */
135: float popf();          /* Deleted */
136: */
137: int polint();          /* line #: 1354 */
138: int infint();          /* Deleted */
139: /*
140: char *variable();      /* Deleted */
141: char *nonlin();        /* Deleted */
142: char *double_nonlin(); /* Deleted */
143: char *left_nonlin();   /* Deleted */
144: char *right_nonlin();  /* Deleted */
145: char *linear();        /* Deleted */
146: */
147: int distsort();        /* Deleted */
148: /*
149: int opcmp();           /* Deleted */
150: */
151: /*********************End of Prototypes*************************/
152: /*****************************************************************************/
153: /* Called by: prefix, operator, opparen, operand, number, chksys, stp_inp_crs,
154:              rstr_i_c, lexword, nonlin, double_nonlin, right_nonlin, linear.*/
155: /*+++++++++++++++++++++++++++++++++++++*/
156: char *defined(wd)              /*sbrtn*//* Test for lexically accepted symbol */
157:                                          /* but not including NULL */
158: char *wd;
159:         {                                /* defined */
160:         if ((wd >= (char *) funcsymb && wd < (char *) (funcsymb + 6)) ||
161:             (wd >= (char *) opsymb && wd < (char *) (opsymb + 12)) ||
162:             (wd >= (char *) varval && wd < (char *) (varval + 20)) || wd == nonword || wd == nlchr ||
163:             (wd >= (char *) numsymval && wd < (char *) (numsymval + 20)) || wd == (char *) &zero ||
164:             (wd >= (char *) syskeywd && wd < (char *) (syskeywd + 13)))
165:                 return (wd);
166:         else return (NULL);
167:         }                                /* defined */
```

```
168:
169: /*************************************************************************/
170: /* Called by: lexpref(2), rpol(3), backcomp(2), polint, infint(3), distsort(2).*/
171: /*+++++++++++++++++++++++++++++++++++*/
172: char *prefix(wd, ch)               /*sbrtn*//* Test for sin/cos/ln/exp/"pi[(]" */
173: char *wd, ch;                              /* NO Unary_Minus */
174:         {                                      /* prefix */
175:         int i;
176:         char *s;
177:         if ((wd >= (char *) funcsymb && wd < (char *) (funcsymb + 5)) || !wd)
178:                                                              /* wd == NULL) */
179:                 return (wd);
180:         else if (defined(wd))
181:                 return (NULL);
182:         else {
183:                 s = wd;                /* Check for Prefix syntax
184:                                                as noninitial '(' */
185:                 while (*s)
186:                         if (*++s == ch)
187:                                 break;
188:                 if (*s == ch)
189:                         for (i = 0;  i < 4;  i++)
190:                                 if (objcmp(funcsymb[i], wd, ch))
191:                                         return (funcsymb[i]);
192:                 }
193:         return (NULL);          /* No ELSE! */
194:         }                                      /* prefix */
195:
196: /*************************************************************************/
197: /* Called by: rpol, backcomp(2). */
198: /*+++++++++++++++++++++++++++++++++++*/
199: char *operator(wd)              /*sbrtn*//*   Test for + - * / ^   */
200: char *wd;
201:         {                                      /* operator */
202:         int i;
203:         if ((wd >= (char *) (opsymb + 7) && wd < (char *) (opsymb + 12)) ||
204:             !wd)                       /* wd == NULL) */
205:                 return (wd);
206:         else if (defined(wd))
207:                 return (NULL);
208:         else for (i = 7;  i < 12;  i++)
209:                         if (objcmp(opsymb[i], wd, '('))  /* Preceded by lexword; tests match w/wo final paren */
210:                                 return (opsymb[i]);
211:         return (NULL);          /* No ELSE! */
212:         }                                      /* operator */
213:
214: /*************************************************************************/
215: /* Called by: lexword. */
216: /*+++++++++++++++++++++++++++++++++++*/
217: char *opparen(wd)               /*sbrtn*//* Test for nonalphanumeric */
218: char *wd;
219:         {                                      /* opparen */
220:         int i;
221:         if ((wd >= (char *) opsymb && wd < (char *) (opsymb + 12))
222:              || !wd)            /* wd == NULL */
223:                 return (wd);
224:         else if (defined(wd))
225:                 return (NULL);
```

```
226:         else for (i = 0;  i < 12;  i++)
227:                         if (objcmp(opsymb[i], wd, '('))  /* In lexword; tests match w/wo final paren */
228:                                 return (opsymb[i]);
229:         return (NULL);          /* No ELSE! */
230:         }                                       /* opparen */
231:
232: /*************************************************************************/
233: /* Called by: lexopand, lexvar(2), lexword, rpol, backcomp(3), polint, infint(2),
234:               distsort(3). */
235: /*+++++++++++++++++++++++++++++++++++*/
236: char *operand(wd)               /*sbrtn*//*  Test for processed number or defined variable */
237: char *wd;
238:         {                                       /* operand */
239:         int i;
240:         if ((wd >= (char *) varval && wd < (char *) (varval + 20)) || wd == nlchr ||
241:             (wd >= (char *) numsymval && wd < (char *) (numsymval + 20))
242:              || wd == (char *) &zero || !wd)            /* wd == NULL */
243:                 return (wd);
244:         else if (defined(wd))
245:                 return (NULL);
246:         else for (i = 0;  !objcmp(varsymb[i], "", '(') && i < 20;  i++)
247:                         if (objcmp(varsymb[i], wd, '('))
248:                                 return ((char *) &varval[i]);
249:         return (NULL);          /* No ELSE! */
250:         }                                       /* operand */
251:
252: /*************************************************************************/
253: /* Called by: printokstr, lexword. */
254: /*+++++++++++++++++++++++++++++++++++*/
255: float *number(vl, erroutn)      /*sbrtn*//* Test for numeric value */
256: float *vl;
257: char *erroutn;
258:         {                                       /* number */
259:         int i;
260:         if ((vl >= numsymval && vl < numsymval + 20) ||     /* Recognized number or NULL */
261:             vl == &zero || !vl) /* vl == NULL */
262:                 return (vl);
263:         else if (defined(vl))                               /* Recognized symbol */
264:                 return (NULL);
265:         else if (*vl == 0)                                  /* New zero string */
266:                 return (&zero);
267:         else for (i = 0;  i < 20;  ++i)
268:                         {
269:                         if (!numsymval[i])      /* numsymval[i] == NULL */
270:                                 {
271:                                 numsymval[i] = *vl;
272:                                 return (&numsymval[i]);     /* Return number recognized */
273:                                 }
274:                         else if (numsymval[i] == *vl)
275:                                 return (&numsymval[i]);     /* Return recognized number */
276:                         }
277:         strcat(erroutn, " Overflow #Store ");
278:         return (NULL);
279:         }                                       /* number */
280:
281: /*************************************************************************/
282: float *symbval(vl)                /*sbrtn*//* Return symbol reference */
283: float *vl;
```

```
284:        {                                       /* symbval */
285:        int i;
286:        if ((vl >= (float *) numsymval && vl < (float *) (numsymval + 20)) || vl == &zero)
287:                return (vl);
288:        else if (vl >= varval && vl < varval + 20)
289:                {
290:                for (i = 0;  !objcmp(varsymb[i], "", '(');  i++) /* Stop if null symbol or initial paren */
291:                        if (vl == &varval[i])
292:                                return ((float *) varsymb[i]);
293:                }
294:        else if(vl ==(float *) nlchr) return(&nlval);
295:        }                                       /* symbval */
296:
297: /*************************************************************************/
298: /* Called by: prefix, operator, opparen, operand, value, chksys, printtabls */
299: /*+++++++++++++++++++++++++++++++++++*/
300: int objcmp(s1, s2, ch)              /*sbrtn*//*  Compares Object Name (s1) to String (s2) */
301: char *s1, *s2, ch;
302:        {                                       /* objcmp */
303:        while (*s1 == *s2)              /* While Name and String match */
304:                if (*(s1++))
305:                        s2++;           /* If match while Name not ended */
306:                  else return (1);      /* If both end still matching */
307:                                        /* If Name and String don't match */
308:        if (!*s1 && *s2 == ch)          /* If Name ends and odd character matches */
309:                return (1);             /* (prefix object in prefix subroutine) */
310:          else return (0);              /* If Name not ended or odd character fails match */
311: /*  Almost like:        return (!strcmp(s1, s2));  */
312:        }                                       /* objcmp */
313:
314: /*************************************************************************/
315: /* Called by: lexpref. */
316: /*+++++++++++++++++++++++++++++++++++*/
317: char *makpref(wd, erroutp)                     /*sbrtn*/
318: char *wd, erroutp;
319:        {                                       /* makpref */
320:        return (NULL);
321:        }                                       /* makpref */
322:
323: /*************************************************************************/
324: /* Called by: finish, lexword. */
325: /*+++++++++++++++++++++++++++++++++++*/
326: char *chksys(wd)                               /*sbrtn*/
327: char *wd;
328:        {                                       /* chksys */
329:        int i;
330:        if ((wd >= (char *) syskeywd && wd < (char *) (syskeywd + 13))  /* Already recognized Keyword */
331:             || !wd)            /* wd == NULL */
332:                return (wd);
333:        else if (defined(wd))                                           6
334:                return (NULL);
335:        else for (i = 0;  i < 13;  i++)
336:                        if (objcmp(syskeywd[i], wd, '('))               /* New Keyword string */
337:                                {
338:                                if (ffl && syskeypt[i] == syskeywd[i])  /* Primary Keyword */
339:                                        {
340:                                        keyfl = syskeywd[i];
341:                                        return (syskeywd[i]);           /* Return recognized Keyword */
```

```
342:                                    }
343:                                else if (!ffl && syskeypt[i] == keyfl &&
344:                                    keyfl != syskeywd[i])
345:                                        return (syskeywd[i]);
346:                                }
347: /*<*/
348:         else if(*inp_curs == '(') return(prefix(wd, '\0'));
349: /*>*/
350:         return (NULL);          /* No ELSE! */
351:         }                                       /* chksys */
352:
353: /*************************************************************************/
354: /* Called by: lexopand, lexvar. */
355: /*+++++++++++++++++++++++++++++++++++*/
356: float *makvar(wd, erroutv)                      /*sbrtn*/
357: char *wd, *erroutv;
358:         {                                       /* makvar */
359:         if (new_iv < 20)
360:                 {
361:                 strcpy(varsymb[new_iv], wd);
362:                 return (&varval[new_iv++]);
363:                 }
364:         else {
365:                 strcat(erroutv, " Overflow @Store ");
366:                 return (NULL);
367:                 }
368:         }                                       /* makvar */
369:
370: /*************************************************************************/
371: /* Called by: rpol, infint. */
372: /*+++++++++++++++++++++++++++++++++++*/
373: int prec(wd)                                    /*sbrtn*/
374: char *wd;
375:         {                                       /* prec */
376:         int i;
377:         if (wd == funcsymb[5])
378:                 wd = minus_sign;
379:         for (i = 7;  i < 12;  ++i)
380:                 if (opsymb[i] == wd)
381:                         return ((i - 7) / 2);
382:         }                                       /* prec */
383:
384: /*************************************************************************/
385: /* Called by: main(6). */
386: /*+++++++++++++++++++++++++++++++++++*/
387: void printokstr(obj, errout)                    /*sbrtn*/
388: char **obj, *errout;
389:         {                                       /* printokstr */
390:         int j;
391:         while (*obj)
392:                 {
393:                 if (number(*obj, errout))
394:                         {
395:                         pf = (float *) *obj;
396:                         printf("[%f] ", *pf);
397:                         }
398:                 else {
399:                         printf("[");
```

```
400:                        printobjv(*obj);
401:                        printf("]");
402:                        }
403:                *obj++;
404:                }
405:        printf("\n");
406:        }                                       /* printokstr */
407:
408: /*************************************************************************/
409: /* Called by: main. */
410: /*+++++++++++++++++++++++++++++++++*/
411: void printstr(obj, errout)                     /*sbrtn*/
412: char **obj, *errout;
413:        {                                       /* printstr */
414:        int j;
415:        while (*obj)
416:                {
417:                if (number(*obj, errout))
418:                        {
419:                        pf = (float *) *obj;
420:                        printf("%f ", *pf);
421:                        }
422:                else {
423:                        printobj(*obj);
424:                        }
425:                *obj++;
426:                }
427:        printf("\n");
428:        }                                       /* printstr */
429:
430: /*************************************************************************/
431: void printstrv(objv, errout)                    /*sbrtn*/
432: float **objv;
433: char *errout;
434:        {                                       /* printstr */
435:        int j;
436:        while (*objv)
437:                {
438:                if (number(*objv, errout))
439:                        {
440:                        pf = *objv;
441:                        printf("%f ", *pf);
442:                        }
443:                else {
444:                        printobjv(*objv);
445:                        }
446:                *objv++;
447:                }
448:        printf("\n");
449:        }                                       /* printstr */
450:
451: /*************************************************************************/
452: /* Called by: printtabs. */
453: /*+++++++++++++++++++++++++++++++++*/
454: void printobj(obj)                              /*sbrtn*/
455: char *obj;
456:        {                                       /* printobj */
457:        while (*obj)
```

```
458:                {
459:                if (*obj == ' ')
460:                        printf("%c", '_');
461:                else printf("%c", *obj);
462:                obj++;
463:                }
464:        return;
465:        }                                       /* printobj */
466:
467: /*************************************************************************/
468: void printobjv(objv)                           /*sbrtn*/
469: float *objv;
470:        {                                       /* printobj */
471:        char *obj;
472:        if(objv >= varval && objv < (varval + 20)) obj = varsymb[objv - varval];
473:          else obj = (char *) objv;
474:        while (*obj)
475:                {
476:                if (*obj == ' ')
477:                        printf("%c", '_');
478:                else printf("%c", *obj);
479:                obj++;
480:                }
481:        return;
482:        }                                       /* printobj */
483:
484: /*************************************************************************/
485: /* Called by: main(10). */
486: /*+++++++++++++++++++++++++++++++++++*/
487: void printtabls()                              /*sbrtn*/
488:        {                                       /* printtabls */
489:        int j;
490:        for (j = 0;  !objcmp(varsymb[j], "", '(') && j < 20;  ++j)
491:                {
492:                if (j == indvar)
493:                        printf(")*( ");
494:                printobj(varsymb[j]);
495:                printf(" ");
496:                if (j >= indvar)
497:                        varval[j] = (j * j + 2.3) / 7.0;
498:                printf("%f ", varval[j]);
499:                }
500:        printf("\n");
501:        for (j = 0;  numsymval[j] && j < 20;  ++j)
502:                printf("%f ", numsymval[j]);
503:        printf("\n");
504:        }                                       /* printtabls */
505:
506: /*************************************************************************/
507: /* Not presently used. */
508: /*+++++++++++++++++++++++++++++++++++*/
509: void prpolstk(pw)                              /*sbrtn*/
510: char *pw;
511:        {                                       /* prpolstk */
512:        char **pst;
513:        for (pst = stack;  pst <= pstack;  ++pst)
514:                printf("%s,", *pst);
515:        printf(";%s\n", pw);
```

```
516:         }                                          /* prpolstk */
517:
518: /*************************************************************************/
519: /* Called by: main(2). */
520: /*+++++++++++++++++++++++++++++++++++*/
521: void clear()                                      /*sbrtn*/
522:         {                                          /* clear */
523:         int j;
524:         new_iv = indvar = 2;
525:         for (j = 0;  j < 20;  ++j)
526:                 {
527:                 varsymb[j][0] = '\0';
528:                 numsymval[j] = varval[j] = 0;
529:                 varval[j] = (j * j + 2.3) / 7;
530:                 }
531:   varsymb[1][1] = varsymb[0][1] = '\0';
532:   varsymb[1][0] = 'x'; varsymb[0][0] = 'y';
533:         lexout[0] = rpolout[0] = bacompout[0] = odisorout[0] = NULL;
534:         }                                          /* clear */
535:
536: /*************************************************************************/
537: /* Called by: assign, lex, rpol, infint. */
538: /*+++++++++++++++++++++++++++++++++++*/
539: void stp_inp_crs(ins)                              /*sbrtn*/
540: char ***ins;
541:         {                                          /* stp_inp_crs */
542:         if (defined(**ins) || !(**ins)) /* Defined or equal NULL */
543:                 lex_i_c = *ins;
544:         else {
545:                 inp_curs = (char *) *ins;
546:                 lex_i_c = &inp_curs;
547:                 }
548:         }                                          /* stp_inp_crs */
549:
550: /*************************************************************************/
551: /* Called by: assign, lex, rpol, infint. */
552: /*+++++++++++++++++++++++++++++++++++*/
553: void rstr_i_c(ins)                                 /*sbrtn*/
554: char ***ins;
555:         {                                          /* rstr_i_c */
556:         if (defined(**ins) || !(**ins)) /* Defined or equal NULL */
557:                 *ins = lex_i_c;
558:         else {
559:                 *ins = (char **) inp_curs;
560:                 }
561:         }                                          /* rstr_i_c */
562:
563: /*************************************************************************/
564: int cvscan(char *cmnd)                             /* Used to convert a string of values for input in main */
565:   {                                                                  /* */
566:    int i, j, jo;
567:    char fl;
568:    float fb, fa;
569:     i = j = 0;
570:     jo = -1;
571:     while(cmnd[j] != '\0' && i < 20 && j != jo)   {
572:         jo = j;
573:         fb = fa = 0;
```

```
574:         fl = 0;
575:         while(isspace(cmnd[j])) j++;            /* Skip spaces, etc. */
576:         if(cmnd[j] == '-') {fl = 1; j++;}       /* Flag sign */
577:         while(isdigit(cmnd[j])) fb = 10*fb + (cmnd[j++] - '0');  /* Form digits before decimal point */
578:         if(cmnd[j] == '.') j++;
579:         while(isdigit(cmnd[j])) fa =.1*(fa + (cmnd[j++] - '0')); /* Form digits after decimal point */
580:         fb += fa; if(fl) fb = - fb;             /* Form complete number */
581:         varval[++i] = fb;                       /* Count digit, but skip 0th (x) var; store value */
582:         }
583:     if(j == jo) i = -i;
584:     return(i);                                                  /* Return True if legitimate */
585:   }                                             /*  */
586:
587: /*++++++++++++++++++++++++++++++++++++*/
588: float calcfunc(int ix, float x, char **rplt, int *ecode)
589:   {                                             /* Calculates the user compensator function */
590:    char erroutstr[100];
591:    float result;
592:     varval[ix] = x;                            /* Sets up first variable value */
593:     prpol = rplt;                              /* Sets up polish pointer */
594:     *ecode = polint(&prpol, &result, erroutstr);/* Executes Polish interpretation */
595:     return(result);                            /* Returns result */
596:   }
597:
598: /*++++++++++++++++++++++++++++++++++++*/
599: float interp(float yans, float x1, float x2, float y1, float y2, float *del)
600:   {                                             /* Interpolation routine */
601:     if(y1 == y2) *del = 0;                      /* Zeros interpolation if zero denominator */
602:       else *del = (x1 -x2)/(y1 - y2)*(yans - y2);  /* Interpolate */
603:     return(x2 + *del);                          /* Returns a new x2 value */
604:   }
605:
606: /*++++++++++++++++++++++++++++++++++++*/
607: float invfunc(float xmx, float xmn, float ymx, float ymn, float yans, float x1, float y1, float dl, float dll)
608:   {                                             /* Invert given starting guess: x1, y1 */
609:    float xl, xh, yl, yh, x2, y2, del, ndel, odel, xans; /* del = range of x; dl = initial change; */
610:    int j, ecode;                                /* dl = final change */
611:                                 /*************************************/
612:                                 /*         Initialize and Test         */
613:                                 /*************************************/
614:     xl = xmn; xh = xmx; yl = ymn; yh = ymx; del = xmx - xmn; /* Initialize bounds */
615:     if(x1 < xmn) {x1 = xmn; y1 = ymn;}
616:       else if(x1 > xmx) {x1 = xmx; y1 = ymx;}
617:     if((yl < yh && yans < yl) || (yl > yh && yans > yl)) j = -1;  /* y to solve too low */
618:       else if((yl < yh && yans > yh) || (yl > yh && yans < yh)) j = 1; /* y to solve too high */
619:       else   {
620:           j = 0;                                /* y in range; Compute x2 from x1 */
621:           if((yans > y1 && yh > y1)||(yans < y1 && yh < y1))  { /* Go high of starting point */
622:               x2 = x1 + del*dl;                 /* Then Positive del of ans */
623:               if(x2 > xh) x2 = xh;              /* High bounded */
624:               }
625:             else   {                            /* Negative del otherwise */
626:                 x2 = x1 - del * dl;
627:                 if(x2 < xl) x2 = xl;            /* Low bounded */
628:                 }
629:           }
630: if(dfl) printf("%d[%3.2f,%3.4f:%3.4f]; ",j,x1,x2,y1);
631:                                 /*************************************/
```

```
632:                                /*          Loop till converge         */
633:                                /*************************************/
634:     do   {                                        /* Step until convergence */
635: if(dfl) printf("%3.4f]", x2); ecode = 9; /* ecode initialized as a distinct value */
636:         if(x2 == xl) {y2 = yl;                    /* If bounded, save recompute of y */
637: if(dfl) printf("l");}
638:           else if(x2 == xh) {y2 = yh;
639: if(dfl) printf("h");}                                          /* line #: 588 */
640:           else y2 = calcfunc(1, x2, rpolout, &ecode); /* Compute next y2 from x2 */
641: if(dfl) printf("%d(%3.2f) ", ecode, y2);
642:         odel = abs(x1-x2);                                                /* line #: 599 */
643:         xans = interp(yans, x1, x2, y1, y2, &ndel); /* Interpolate to get new x2 */
644:         x1 = x2; y1 = y2;                         /* Move point */
645:         if(xans > xh) xans = xh;                  /* Bound next solution High */
646:           else if(xans < xl) xans = xl;           /* Bound next solution Low */
647:         if(x2 > xans) {xh = x2; yh = y2;}         /* Narrow high bound if answer low */
648:           else {xl = x2; yl = y2;}                /* Narrow low bound if answer high */
649:         x2 = xans;                                /* New second point = next solution */
650:         } while(odel > dll*del);                  /* Do next step until converge */
651:     return x2;
652:   }
653: /*************************************************************************/
654: /* Main Routine, not called. */
655: /*+++++++++++++++++++++++++++++++++++*/
656: void main()                                       /*sbrtn*/
657:         {                                         /* democalculator */
658:         int i, j, nvar, ecode, imax, codct, ssl;
659:         char command[201], *ci, instr[100], instri[100], erroutstr[100];
660:         char *pinstr;           /* Index Pointer */
661:         float result, xvlu, yvlu, xmn, ymn, xmx, ymx, yans, xl, yl, xh, yh, x1, y1, x2, y2, x, y;
662:         float emax, dl, dll, xval[21], yval[21], yansv[21], ytop, xbot, dtb, dbot;
663:         left_paren = opsymb[0];
664:         right_paren = opsymb[1];
665:         equal_sign = opsymb[2];
666:         minus_sign = opsymb[8];
667:         instr[0] = '\0';
668:         dfl = 0;
669:         for (i = 0;  i < 7;  i++)
670:                 syskeypt[i] = syskeywd[0];
671:         for (i = 7;  i < 11;  i++)
672:                 syskeypt[i] = syskeywd[7];
673:         syskeypt[11] = syskeywd[11];
674:         syskeypt[12] = syskeywd[12];
675:         for (i = 0;  i < 100;  ++i)
676:                 {
677:           if(dfl%2) dfl = 1; else dfl = 0;
678:                 clear();
679:                 strcpy(erroutstr, "");
680:                 ffl = 1;
681:           ifl = codct = 0;              /* Debug and Inverse initially turned off */
682:                 keyfl = NULL;
683:                 wordd[0] = '\0';
684:                 indvar = new_iv;
685:                 printf("----------------------------------------\n");
686: printf("Program to demonstrate inline compensator function, with word forming, polish translation,\n");
687: printf("   table print out, inverse guess, inversion table print out, and back compilation\n");
      printf("   table print out, inverse guess, inversion table print out, and back compilation\n");
      printf("Entered functions are stated in an expression which allows for parentheses, with 20 levels of nesting;\n");
      printf("   Add (+), Subtract (-), Multiply (*), Divide (/), and Power (^) Operators; and sin, cos, ln, and exp\n");
```

```
      printf("   functions. Function parentheses must follow their function name, without intervening spaces. The\n");
      printf("   expression can include any named parameter or variable (in addition to the presumed x and y), named\n");
      printf("   with any combination of letters, digits, or embedded single spaces, not confused with a number.\n\n");
688:
689: printf("Enter function in form y = f(x), x to be solved, or \"debug\", or \"stop\":\n\n      y = ");
690:                 fgets(command, 201, stdin);
691:                 command[strlen(command)-1] = '\0'; if(!strcmp(command, "stop")) return;
692:                 if(!strcmp(command, "debug")) {dfl = 1; continue;}
693:                 ci = &(command[strlen(command)+1]);
694: printf("\nEnter inverse function in form x = f(y), solved for x, or enter \"default\":\n\n      x = ");
695:                 fgets(ci, 201-strlen(command)-1, stdin); ci[strlen(ci)-1] = '\0';
696:                 if(!strcmp(ci,"default") || !ci[0]) printf("\ny = %s; Default inverse\n", command);
697:                   else {ifl = 1; printf("\ny = %s; x = %s\n", command, ci);}
698: printf("\n\n");
699:                 strcpy(instr, command);
700:                 if(ifl) strcpy(instri, ci);
701: if(dfl) printf("\nLexical processing (Word Forming):\n");
702:                 pinstr = instr;
703:                 plex = lexout;
704:                 ecode = lex(&pinstr, &plex, erroutstr);                  /* line #: 1095 */
705:                 if (ecode) {codct += 1; if(dfl) printf("Success:");}
706:                   else {printf("LexFailed:"); dfl += 2;}
707: if(dfl) {printf("%s\n", erroutstr);strcpy(erroutstr, "");}
708:                 if (!ecode) continue;
709: if(dfl) {printf("Lex:"); printokstr(lexout, erroutstr);}  /* line #: 387 */
710:                 if(ifl)   {
711:                     pinstr = instri;
712:                     plex = lexouti;
713:                     ecode = lex(&pinstr, &plex, erroutstr);
714:                     if (ecode) {codct += 2; if(dfl) printf("Success:");}
715:                       else {printf("LexInvFailed:"); dfl += 2;}
716: if(dfl) {printf("%s\n", erroutstr);strcpy(erroutstr, "");}
717:                     if (!ecode) continue;
718: if(dfl) {printf("Lex Inverse:"); printokstr(lexouti, erroutstr);}
719:                     }
720: if(dfl) printf("\nPolish Expression Translation:\n");
721:                 plex = lexout;
722:                 prpol = rpolout;
723:                 ecode = rpol(&plex, &prpol, erroutstr);                  /* line #: 1145 */
724:                 if (ecode) {codct += 4; if(dfl) printf("Success:");}
725:                   else {printf("RpolFailed:"); dfl += 2;}
726: if(dfl) {printf("%s\n", erroutstr);strcpy(erroutstr, "");}
727:                 if (!ecode) continue;
728: if(dfl) {printf("Rpol from lexed tokens:"); printokstr(rpolout, erroutstr);}
729:           if(ifl)   {
730:                     plex = lexouti;
731:                     prpol = rpolouti;
732:                     ecode = rpol(&plex, &prpol, erroutstr);
733:                     if (ecode) {codct += 8; if(dfl) printf("Success:");}
734:                       else {printf("RpolInvFailed:"); dfl += 2;}
735:                     if(dfl) {printf("%s\n", erroutstr);strcpy(erroutstr, "");}
736:                     if (!ecode) continue;
737:                     if(dfl) {printf("Inverse Rpol from lexed tokens:"); printokstr(rpolouti, erroutstr);}
738:               }
739: if(dfl) printf("\nBack Compiling Demonstration:\n");
740:                 prpol = rpolout;
741:                 pbac = bacompout;
742:                 ecode = backcomp(&prpol, &pbac, erroutstr);              /* line #: 1224 */
743:                 if (ecode) {codct += 16; if(dfl) printf("Success:");}
```

```
744:                   else {printf("BackCompFailed:"); dfl += 2;}
745:                 if(dfl) {printf("%s\n", erroutstr);strcpy(erroutstr, "");}
746:                 if (!ecode) continue;
747:                 if(dfl) {printf("Back Comp:"); printokstr(bacompout, erroutstr);}
                     printf("y = "); printstrv(bacompout, erroutstr);
748:                 if(ifl)   {
749:                     prpol = rpolouti;
750:                     pbac = bacompouti;
751:                     ecode = backcomp(&prpol, &pbac, erroutstr);
752:                     if (ecode) {codct += 32; if(dfl) printf("Success:");}
753:                       else {printf("BackCompInvFailed:"); dfl += 2;}
754:                     if(dfl) {printf("%s\n", erroutstr);strcpy(erroutstr, "");}
755:                     if (!ecode) continue;
756:                     if(dfl) {printf("Inverse Back Comp:"); printokstr(bacompouti, erroutstr);}
757:                     printf("x = "); printstrv(bacompouti, erroutstr);
758:                     }
759: if(dfl) {printf("\nTabular Function Output:\n");
760:                 for(i=0; i<21; i++)   {         /* Compute function table */
761:                     x = i*5.0; printf("%3.2f: ", x);                    /* line #: 588 */
762:                     printf("%3.4f", y = calcfunc(1, x, rpolout, &ecode));
763:               if(ifl) printf(" inv: %3.4f", calcfunc(0, y, rpolouti, &ecode));
764:               printf("\n");
765:                     }}
766:                 printf("\n***%d***\n",indvar);
767:                 printtabls();                   /* Print system data/variable tables */
768:                 printf("***\n");
769: printf("\nEnter Working Values, one for each Variable after y:\n");
770:                 fgets(command, 201, stdin);
771:                 command[strlen(command)-1] = '\0';                      /* line #: 564 */
772:                 if((nvar=cvscan(command)) < 1) {printf("Failed:"); continue;}    /* Enter a nvar consecutive values of
*/
773:                 for(i=0; i<nvar+1; i++) printf("%s = %f; ", varsymb[i], varval[i]); /*   varval */
774:                 printf("\n");
775:                 printf("\n***%d***\n",indvar = new_iv);
776:                 printtabls();                   /* Print system data/variable tables */
777:                 printf("***\n\n");                                      /* line #: 588 */
778:                 yvlu = calcfunc(1, xvlu=varval[1], rpolout, &ecode); /* Initial input and output */
779:                 xmn = xbot = 0.; xmx = 100.; dl = .1; dll = .06; dbot = 0; /* Set up x min and max scale values */
780:                 ymn = ytop = calcfunc(1, xmn, rpolout, &ecode); /*   and delta parameters; Calculate */
781: if(dfl) printf("%dMN(%3.2f;%5.4f) ", ecode, xmn, ymn);
782:                 ymx = calcfunc(1, xmx, rpolout, &ecode); /*   y min and max scale values */
783:           if(ymx > ymn) ssl = 1;
784:             else if(ymx < ymn) ssl = -1;
785:             else ssl = 0;
786: if(dfl) printf("%dMX(%3.2f;%5.4f)\n", ecode, xmx, ymx);
787:                 for(i=0; i<21; i++)   {         /* Iterate a table of solution values */
788:                     x = i*5.0;                  /* Compute x solution and corresponding */
789: if(dfl) printf("%3.2f: ", x);                  /*   y answer value */
790:                     yans = calcfunc(1, x, rpolout, &ecode);
791:               dtb = yans - ytop;
792:               if(dtb*ssl>0) ytop = yans;        /* Find new Top y */
793:                 else if(ssl*(dtb-dbot) < 0) {dbot = dtb; xbot = x;}
794: if(dfl) printf("%d\\(%3.2f) ", ecode, yans);
795:                     x1 = xvlu; y1 = yvlu;       /* ReSet up initial guess */
796:               if(ifl) x2 = calcfunc(0, yans, rpolouti, &ecode);  /* Compute any entered inverse */
797:  /* line #: 607 */  else x2 = invfunc(xmx, xmn, ymx, ymn, yans, x1, y1, dl, dll); /* or Solve function for x2
giving */
798:                     xval[i] = x2; yansv[i] = yans; /*   yans; Fill table values */
799:                     yval[i] = calcfunc(1, x2, rpolout, &ecode); /* Fill table values for actual y correspond- */
```

```
800: if(dfl) printf(":%d\n", ecode);                /*   ing to solved x2 */
801:                     }
802:
803:                printf("***\n");
804:           if(xbot>0) printf("Max Out of Monotonicity at x = %3.1f of %f%% of full scale range; ", xbot, 100*dbot/(ymx-
ymn));
805:             else printf("Fully Monotonic: ");
               if(ymx > ymn) printf("Positive Monotone.\n");
                 else if(ymx , ymn) printf("Negative Monotone.\n");
                 else printf("Non-Monotone. \n");
806:                printf("+++\n");
807:                emax = 0.0; imax = -1;
808:           for(i=0; i<21; i++)   {              /* Print function/solution tables */
809:                           if(fabs(xval[i]-i*5) > fabs(emax)) {imax = i; emax = xval[i] - i*5;}
810:                     printf("% 4.1f: % 8.3f; % 8.3f, % 4.4f", i*5., yansv[i], yval[i], xval[i]);
811:                     if(dfl) printf("       %d;%f", imax, emax);
812:                     printf("\n");
813:                    }
814:                printf("+++\n");
815:           printf("Maximum error at x = %3.1f of %f%% of full scale range.\n", 5.*imax, emax);
816:                printf("***\n");
817:                }
818:         }                                      /* democalculator */
819:
820: /*************************************************************************/
821: /* Called by: lexvar, lextok, rpol, infint. */
822: /*+++++++++++++++++++++++++++++++++++*/
823: char *lexpref(wd, erout)                       /*sbrtn*/
824:                                    /* lexically and/or syntactically
825:                                        recognizes prefixes with
826:                                        following "(".  Makes any allowed
827:                                        user created prefixes, as
828:                                        determined by makpref.  Assumes prefix
829:                                        word already grouped by lexword. */
830: char *wd, *erout;
831:         {                                      /* lexpref */
832:         char *pw, *s;
833:         if (pw = prefix(wd, '('))
834:                 {
835:                 if (wd == word)
836:                         strcat(erout, "%");
837:                 return (pw);
838:                 }
839:         else {
840:                 if (wd == word) /* Check for raw word */
841:                         {
842:                         s = wd;         /* Check for Prefix syntax
843:                                            as noninitial '(' */
844:                         while (*s)
845:                                 if (*++s == '(')
846:                                         break;
847:                         }
848:                 else return (NULL);     /* Not a raw word */
849:                 }
850:         if (*s)                         /* Prefix? */
851:                 {
852:                 if (pw = makpref(wd, erout))
853:                         {
854:                         strcat(erout, "%");
```

```
855:                         return (pw);
856:                         }
857:                 else *s = '\0'; /* Failed to make prefix; try variable. */
858:                 }
859:         return (NULL);          /* No legal prefix */
860:         }                                       /* lexpref */
861:
862: /*************************************************************************/
863: /* Called by: lextok, rpol, infint. */
864: /*+++++++++++++++++++++++++++++++++++*/
865: char *lexopand(wd, erout)                       /*sbrtn*/
866:                                                 /* returns lexically/syntactically
867:                                                         recognized operand, making new
868:                                                         variable with makvar when needed. */
869: char *wd, *erout;
870:         {                                       /* lexopand */
871:         char *pw;
872:         if (wd == word)
873:                 strcat(erout, "@");
874:         if (pw = operand(wd))                   /* Processed number or defined variable */
875:                 return (pw);
876:         else if (wd == word)
877:                 {
878:                 if (pw = (char *) makvar(wd, erout))     /* New variable */
879:                         return (pw);
880:                 else strcat(erout, " No operand ");
881:                 }
882:         return (NULL);
883:         }                                       /* lexopand */
884:
885: /*************************************************************************/
886: /* Called by: assign. */
887: /*+++++++++++++++++++++++++++++++++++*/
888: char *lexvar(wd, erout)                         /*sbrtn*/
889:                                                 /* returns lexically/syntactically
890:                                                         recognized operand, making new
891:                                                         variable with makvar when needed. */
892: char *wd, *erout;
893:         {                                       /* lexvar */
894:         char *pw;
895:         if (wd == word)
896:                 strcat(erout, "@");
897:         else return (NULL);
898:         if (lexpref(wd, erout))
899:                 return (NULL);
900:         else if (pw = operand(wd))
901:                 return (pw);
902:         else if (pw = (char *)makvar(wd, erout))
903:                 return (pw);
904:         else strcat(erout, " No operand ");
905:         return (NULL);
906:         }                                       /* lexvar */
907:
908: /*************************************************************************/
909: /* Called by: finish, lexword. */
910: /*+++++++++++++++++++++++++++++++++++*/
911: int lexsys(wd)                                  /*sbrtn*/
912: char wd[];
```

```
913:         {                                          /* lexsys */
914:         int i;
915:         i = 0;
916:         wd[0] = *(inp_curs++);
917:         while (isalpha(*inp_curs) || isdigit(*inp_curs))
918:                 wd[++i] = *(inp_curs++);
919:         wd[++i] = '\0';
920:         return (i);
921:         }                                          /* lexsys */
922:
923: /*************************************************************************/
924: /* Called by: lexword(2). */            /* 'finish' finishes out a potential name, which is currently    */
925: /*+++++++++++++++++++++++++++++++++++*/ /* scanned to a letter or digit, testing for further letters,    */
926: int finish(j)                   /*sbrtn*/ /* digits, isolated spaces or _, and for terminating System_Name.*/
927: int j;                                    /* Also packs on final '(', for later function prefix check.     */
928:         {                                          /* finish */
929:         int k;
930:         while (isalpha(*inp_curs) || isdigit(*inp_curs) || *inp_curs == ' ' || *inp_curs == '_')
931:                 {                                  /* 4 */
932:                 if (*inp_curs == ' ' || *inp_curs == '_')
933:                         {                          /* 3 */
934:                         if (*(inp_curs++) == ' ')
935:                                 {                  /* 2 */
936:                                 if (isalpha(*inp_curs))
937:                                         {          /* 1 */
938:                                         k = lexsys(wordd);    /* Form tentative system word */
939:                                         if (*inp_curs != '_' && (pwdd = chksys(wordd)))
940:                                                 break;  /* Break to end ####################*/
941:                                            else { /* 0 */
942:                                                 word[++j] = ' ';
943:                                                 word[j + 1] = '\0';
944:                                                 strcat(word, wordd);
945:                                                 j = j + k;
946:                                                 wordd[0] = '\0';
947:                                                 goto spend; /* Loop Back ####################*/
948:                                                 } /* 0 */
949:                                         }          /* No 'else' */  /* 1 */
950:                                 }                  /* No 'else' */  /* 2 */
951:                         if (*inp_curs == ' ' || *inp_curs == '_')
952:                                 break;             /* Break to end ###################*/
953:                         word[++j] = ' ';           /* 'else' permitted but not needed */
954:                         }                          /* 3 */
955:                     else word[++j] = *(inp_curs++);
956: spend:          ;
957:                 }                                  /* 4 */
958:         if (word[j] == ' ')
959:                 --j;
960:         else if (*inp_curs == '(')
961:                 word[++j] = '(';           /* xxx( No Spaces */
962:         word[++j] = '\0';
963:         return (j);
964:         }                                          /* finish */
965:
966: /*************************************************************************/
967: /* Called by: assign(2), lexpref, lextok, rpol(6), infint(6). */
968: /*+++++++++++++++++++++++++++++++++++*/
969: char *lexword(erout)                            /*sbrtn*/
970: char *erout;
```

```
 971:        {                                       /* lexword */
 972:        float nmbr, *pnmbr;
 973:        char *pw, spfl;
 974:        int i, j, k;
 975:        if (wordd[0])           /* wordd[0] != NULL */
 976:                {
 977:                wordd[0] = '\0';
 978:                if(pwdd >= (char *) syskeywd) strcat(erout, "$");
 979:                     else strcat(erout, "&");
 980:                return (pwdd);
 981:                }
 982:        else if (!*lex_i_c)     /* *lex_i_c == NULL */
 983:                return (NULL);
 984:        else if (defined(*lex_i_c))
 985:                return (*(lex_i_c++));
 986:        else spfl = 1;                          /* Start of Symbol counts as prior Space */
 987:        while (*inp_curs == ' ' || *inp_curs == '_')
 988:                {
 989:                if (*inp_curs == ' ')           /* (spfl initialized above) */
 990:                     {if(!spfl) return(nlchr);} /* Prior '_'s isolated by Space */
 991:                  else spfl = 0;                /* Flag '_' */
 992:                inp_curs++;
 993:                }
 994:        if (!*inp_curs)         /* *inp_curs == '\0' */
 995:                {if(spfl) return (NULL);
 996:                    else return (nlchr);}
 997:        else if (isalpha(*inp_curs))
 998:                {
 999:                j = lexsys(word) - 1;           /* Form tentative system word */
1000:                if (spfl && *inp_curs != '_' && (pw = chksys(word))) /* Keyword and Preceding or following _ */
1001:                        {
1002:                        if(pw >= (char *) syskeywd) strcat(erout, "$");   /* Keyword */
1003:                             else strcat(erout, "&");                     /* Function */
1004:                        ffl = 0;                        /* Primary Keyword flag */
1005:                        return (pw);
1006:                        }
1007:                ffl = 0;
1008:                finish(j);
1009:                return (word);
1010:                }
1011:        else ffl = 0;
1012:        if (isdigit(*inp_curs) || (*inp_curs == '.' && isdigit(*(inp_curs + 1))))
1013:                {
1014:                i = 1;                                  /* No decimal point */
1015:                j = 0;
1016:                k = 1;
1017:                word[0] = *inp_curs;
1018:                if (*(inp_curs) == '.')
1019:                        {
1020:                        inp_curs++;                     /* Next character */
1021:                        nmbr = 0;                       /* Initialize number */
1022:                        i = 0;                          /* Flag decimal point */
1023:                        }
1024:                else
1025:                        nmbr = *(inp_curs++) - '0';     /* Number = first digit */
1026:                while (isdigit(*inp_curs) ||
1027:                    (*inp_curs == '.' && i))            /* Scan digits after decimal point */
1028:                        {
```

```
1029:                           if (!i)
1030:                                   k = k * 10;
1031:                           if (*inp_curs == '.')
1032:                                   i = 0;
1033:                           else nmbr = 10 * nmbr + *inp_curs - '0';  /* Combine digits */
1034:                           word[++j] = *(inp_curs++);      /* Combine characters */
1035:                           }
1036:                   nmbr = nmbr / k;                        /* Include decimal effects */
1037:                   if (i && *inp_curs != '(' && finish(j) > j + 1) /* Bypass 'finish' '(' test;with this,finish */
1038:                           return (word);                  /* tests sp, _, or letter, and completes     */
1039:                   else {                                  /* resulting word.                           */
1040:                           if (word[j] != '\0')
1041:                                   word[++j] = '\0';
1042:                           strcat(erout, "#");
1043:                           if (pnmbr = number(&nmbr, erout))
1044:                                   return ((char *) pnmbr);
1045:                           else {                          /* Should occur only on memory overload */
1046:                                   strcpy(nonword, " No operand ");
1047:                                   return (nonword);
1048:                                   }
1049:                           }
1050:                   }
1051:           else {
1052:                   word[0] = *inp_curs;
1053:                   if (word[0] == ':')
1054:                           ffl = 1;
1055:                   word[1] = '\0';
1056:                   if (pw = opparen(word))
1057:                           {
1058:                           if (pw == left_paren || pw == right_paren ||
1059:                               pw == equal_sign)
1060:                                   strcat(erout, pw);
1061:                           else strcat(erout, "!");
1062:                           ++inp_curs;
1063:                           return (pw);
1064:                           }
1065:                   else
1066:                           {
1067:                           strcat(erout, "?");
1068:                           nonword[0] = *inp_curs;
1069:                           strcpy(&nonword[1], "  /??/");
1070:                           return (nonword);
1071:                           }
1072:                   }
1073:           }                                       /* lexword */
1074:
1075: /*************************************************************************/
1076: /* Called by: lex. */
1077: /*+++++++++++++++++++++++++++++++++++*/
1078: char *lextok(erout)                                /*sbrtn*/
1079: char *erout;
1080:           {                                       /* lextok */
1081:           char *pw;
1082:           if ((pw = lexword(erout)) == word)
1083:                   {
1084:                   if (pw = lexpref(word, erout))
1085:                           return (pw);
1086:                   else return (lexopand(word, erout));
```

```
1087:                 }
1088:         if (pw == nlchr) strcat(erout, "_ ");
1089:         return (pw);
1090:         }                                       /* lextok */
1091:
1092: /*************************************************************************/
1093: /* Called by: main(2). */
1094: /*+++++++++++++++++++++++++++++++++++*/
1095: int lex(pins, lxout, errout)                    /*sbrtn*/
1096: char ***pins, ***lxout, *errout;
1097:         {                                       /* lex */
1098:         char *pw, **plxout;
1099:         plxout = *lxout;
1100:         stp_inp_crs(pins);
1101:         while (pw = lextok(errout))             /* While not NULL */
1102:
1103:                 if ((*(plxout++) = pw) == nonword)
1104:                         {
1105:                         *plxout = NULL;
1106:                         return (0);
1107:                         }
1108:         rstr_i_c(pins);         /* Update pointers if success */
1109:         *lxout = plxout;
1110:         *plxout = NULL;
1111:         return (1);
1112:         }                                       /* lex */
1113:
1114: /*************************************************************************/
1115: /* Called by: rpol(4), infint(4). */
1116: /*+++++++++++++++++++++++++++++++++++*/
1117: void push(ps, errout)                           /*sbrtn*/
1118: char *ps;
1119:         {                                       /* push */
1120:         if (pstack >= stack + 10)
1121:                 {
1122:                 strcat(errout, " Overflow Stack ");
1123:                 pstack--;
1124:                 }
1125:         *(pstack++) = ps;
1126:         }                                       /* push */
1127:
1128: /*************************************************************************/
1129: /* Called by: rpol(4). */
1130: /*+++++++++++++++++++++++++++++++++++*/
1131: void error(pw, str, pout, errout)               /*sbrtn*/
1132: char *pw, *str, **pout;
1133:         {                                       /* error */
1134:         strcat(errout, str);
1135:         strcpy(nonword, "  /??/ ");
1136:         *(pout++) = nonword;    /* transfer */
1137:         if (pw)                         /* pw!= NULL */
1138:                 *(pout++) = pw; /* transfer */
1139:         *pout = NULL;
1140:         }                                       /* error */
1141:
1142: /*************************************************************************/
1143: /* Called by: main(2). */
1144: /*+++++++++++++++++++++++++++++++++++*/
```

```
1145: int rpol(pins, rplt, errout)                    /*sbrtn*/
1146: char ***pins, ***rplt, *errout;
1147:         {                                       /* rpol */
1148:         char *pw, *pw1, **pout;
1149:         stp_inp_crs(pins);
1150:         pout = *rplt;
1151:         pstack = stack;
1152: un_min:
1153:         pw = lexword(errout);
1154:         if (pw == minus_sign)
1155:                 {
1156:                 push(funcsymb[5], errout);
1157:                 pw = lexword(errout);
1158:                 }
1159: l_p:
1160:         if (pw == left_paren)
1161:                 {
1162:                 push(funcsymb[4], errout);
1163:                 goto un_min;
1164:                 }
1165:         else if (pw1 = lexpref(pw, errout))
1166:                 {
1167:                 push(pw1, errout);
1168:                 pw = lexword(errout);   /* Skip ( */
1169:                 goto un_min;
1170:                 }
1171:         if (pw1 = lexopand(pw, errout))
1172:                 {
1173:                 *(pout++) = pw1;        /* transfer */
1174:                 pw = lexword(errout);
1175:                 }
1176:         else {
1177:                 error(pw, " Missing operand ", pout, errout);
1178:                 return (0);
1179:                 }
1180:         while (pw == right_paren)
1181:                 {
1182:                 do      if (pstack == stack)
1183:                                 {
1184:                                 error(pw, " Missing ( ", pout, errout);
1185:                                 return (0);
1186:                                 }
1187:                 while (!(prefix(*(pout++) = *(--pstack), '(')));
1188:                                                 /* pop */
1189:                 pw = lexword(errout);
1190:                 }
1191:         if (!pw)                        /* pw==NULL */
1192:                 {
1193:                 do      if (pstack == stack)
1194:                                 {
1195:                                 rstr_i_c(pins); /* Update pointers if success */
1196:                                 *rplt = pout;
1197:                                 *pout = NULL;
1198:                                 return (1);
1199:                                 }
1200:                 while (!(prefix(*(pout++) = *(--pstack), '(')));
1201:                                                 /* pop */
1202:                 error(pw, " Missing ) ", pout, errout);
```

```
1203:                 return (0);
1204:                 }
1205:         else if (pw1 = operator(pw))    /* At this point, pw1==pw */
1206:                 {
1207:                 while (!(pstack == stack || prefix(*(pstack - 1), '(')) &&
1208:                     prec(pw1) <= prec(*(pstack - 1)))
1209:                         *(pout++) = *(--pstack);        /* pop */
1210:                 push(pw1, errout);
1211:                 pw = lexword(errout);
1212:                 goto l_p;
1213:                 }
1214:         else
1215:                 {
1216:                 error(pw, " Missing operator ", pout, errout);
1217:                 return (0);
1218:                 }
1219:         }                                       /* rpol */
1220:
1221: /*************************************************************************/
1222: /* Called by: main. */
1223: /*+++++++++++++++++++++++++++++++++*/
1224: int backcomp(polins, bcmpt, errout)             /*sbrtn*/
1225: char ***polins, ***bcmpt, *errout;
1226:         {                                       /* backcomp */
1227:         char **pins, *pw, **pout;
1228:         pins = *polins;
1229:         pout = *bcmpt;
1230:         pistack = istack;
1231:         di = dir;
1232:         while (*pins)
1233:                 {
1234:                 if ((*pins >= (char *) varval && *pins < (char *) (varval + 20)) || *pins == nlchr ||
1235:                                 (*pins >= (char *) numsymval && *pins < (char *) (numsymval + 20)) ||
1236:                                   *pins == (char *) &zero) /* Operand: Variable, Number, zero, Null */
1237:                         {
1238:                         *(pistack++) = di;      /* push directory pointer */
1239:                         *(di++) = 0;
1240:                         }                       /* incr dir pointer, zero entry */
1241:                 else if (*pins >= opsymb[7] && *pins <= opsymb[11]) /* Test for operator: + - * / ^ */
1242:                         (*(*(--pistack)))++;    /* pop and incr entry */
1243:                 else if (*pins == funcsymb[4] || *pins == funcsymb[5]) /* #(Dummy paren Function) ~ */
1244:                         (*(*(pistack - 1)))++;  /* incr entry */
1245:                 else if (*pins >= funcsymb[0] && *pins < funcsymb[4])  /* Test for sin cos ln exp */
1246:                         *(*(pistack - 1)) += 2; /* incr entry by 2 */
1247:                 else {
1248:                         strcat(errout, "code error");
1249:                         *pout = NULL;
1250:                         return (0);
1251:                         }
1252:                 pins++;
1253:                 }
1254:         pins = *polins;
1255:         pout = *bcmpt;
1256:         pbstack = bstack;
1257:         di = dir;
1258:         while (*pins)
1259:                 {
1260:                 if ((*pins >= (char *) varval && *pins < (char *) (varval + 20)) || *pins == nlchr ||
```

```
1261:                                (*pins >= (char *) numsymval && *pins < (char *) (numsymval + 20)) ||
1262:                                  *pins == (char *) &zero) /* Operand: Variable, Number, zero, Null */
1263:                         {
1264:                         pout += *(di++);        /* output spaces */
1265:                         *(pbstack++) = pout;    /* push */
1266:                         *(pout++) = *pins;
1267:                         }                       /* output operand */
1268:                 else if (*pins >= opsymb[7] && *pins <= opsymb[11]) /* Test for operator: + - * / ^ */
1269:                         *(--(*(--pbstack))) = *pins;    /* pop, decr entry and output op */
1270:                 else if (*pins == funcsymb[4])  /* To #; dummy paren function */
1271:                         {
1272:                         *(--(*(pbstack - 1))) = left_paren;
1273:                                                 /* decr entry and output ( */
1274:                         *(pout++) = right_paren;        /* output ) */
1275:                         }
1276:                 else if (*pins == funcsymb[5])  /* To -; unary minus */
1277:                         *(--(*(pbstack - 1))) = minus_sign;
1278:                                                 /* decr entry and output - */
1279:                 else if (*pins >= funcsymb[0] && *pins < funcsymb[4]) /* Test for sin cos ln exp */
1280:                         {
1281:                         *(--(*(pbstack - 1))) = left_paren;
1282:                                                 /* decr entry and output ( */
1283:                         *(--(*(pbstack - 1))) = *pins;  /* decr entry and output pref */
1284:                         *(pout++) = right_paren;
1285:                         }                       /* output ) */
1286:                 else {
1287:                         strcat(errout, "code error");
1288:                         *pout = NULL;
1289:                         return (0);
1290:                         }
1291:                 pins++;
1292:                 }
1293:         *polins = pins;
1294:         *bcmpt = pout;
1295:         *pout = NULL;
1296:         return (1);
1297:         }                                       /* backcomp */
1298:
1299: /*************************************************************************/
1300: /* Called by: polint, infint(4). */
1301: /*+++++++++++++++++++++++++++++++++++*/
1302: float domon(op, vl)                             /*sbrtn*/
1303: char *op;
1304: float vl;
1305:         {                                       /* domon */
1306:         switch ((op - funcsymb[0])/4)
1307:                 {
1308:         case 0:
1309:                 return (sin(vl));
1310:         case 1:
1311:                 return (cos(vl));
1312:         case 2:
1313:                 return (log(vl));
1314:         case 3:
1315:                 return (exp(vl));
1316:         case 4:
1317:                 return (vl);
1318:         case 5:
```

```
1319:                 return (-vl);
1320:         default:
1321:                 printf("op errof");
1322:                 return (0);
1323:                 }
1324:         }                                       /* domon */
1325:
1326: /*************************************************************************/
1327: /* Called by: polint, infint(3). */
1328: /*+++++++++++++++++++++++++++++++++++*/
1329: float dobin(op, vl1, vl2)                      /*sbrtn*/
1330: char *op;
1331: float vl1, vl2;
1332:         {                                       /* dobin */
1333:         switch ((op - opsymb[7])/2)
1334:                 {
1335:         case 0:
1336:                 return (vl1 + vl2);
1337:         case 1:
1338:                 return (vl1 - vl2);
1339:         case 2:
1340:                 return (vl1 * vl2);
1341:         case 3:
1342:                 return (vl1 / vl2);
1343:         case 4:
1344:                 return pow(vl1, vl2);
1345:         default:
1346:                 printf("op errof");
1347:                 return (0);
1348:                 }
1349:         }                                       /* dobin */
1350:
1351: /*************************************************************************/
1352: /* Called by: main. */
1353: /*+++++++++++++++++++++++++++++++++++*/
1354: int polint(polins, rslt, errout)               /*sbrtn*/
1355: char ***polins, *errout;
1356: float *rslt;
1357:         {                                       /* polint */
1358:         char **pins;                            /* interpreter pointer (pointing to operator strings) */
1359:         float vl;
1360:         pins = *polins;                         /* polins points to interpreter string of op strings */
1361:         pfstack = fstack;
1362:         while (*pins)
1363:                 {
1364:                 if ((*pins >= (char *) varval && *pins < (char *) (varval + 20)) || *pins == nlchr ||
1365:                                 (*pins >= (char *) numsymval && *pins < (char *) (numsymval + 20)) ||
1366:                                   *pins == (char *) &zero)  /* Operand: Variable, Number, zero, Null */
1367:                                     *(pfstack++) = *(float *)(*pins); /* pushf */
1368:                 else if (*pins >= opsymb[7] && *pins <= opsymb[11])    /* Test for operator: + - * / ^ */
1369:                         {
1370:                         vl = *(--pfstack);                              /* popf */
1371:                         *(pfstack++) = dobin(*pins, *(--pfstack), vl); /* pushf, popf */
1372:                         }
1373:                 else if (*pins >= funcsymb[0] && *pins <= funcsymb[5]) /* Test for sin cos ln exp # ~ */
1374:                         *(pfstack++) = domon(*pins, *(--pfstack));     /* pushf, popf */
1375:                 else {
1376:                         strcat(errout, " Code error ");
```

```
1377:                           return (0);
1378:                           }
1379:                   pins++;
1380:                   }
1381:           *polins = pins;
1382:           *rslt = *(--pfstack);                   /* popf */
1383:           return (1);
1384:           }                                       /* polint */
```