DATA GENERAL EXTENDED ALGOL 60 COMPILER

Data General's Extended ALGOL is a powerful language which allows systems programmers to develop programs on mini computers that would otherwise require the use of much larger, more expensive computers. No other mini computer offers a language with the programming features and general applicability of Data General's Extended ALGOL.

ALGOL 60 is the most widely used language for describing programming algorithms. It has a flexible, generalized, arithmetic organization and a modular, "building block" structure that provides clear, easily readable documentation. The language is powerful and concise, allowing the systems programmer to state algorithms without resorting to "tricks" to bypass the language.

These characteristics of ALGOL are especially important in the development of working prototype systems. The clear documentation makes it easy for the programmer to recognize and correct program deficiencies and, because of ALGOL's modular structure, the programmer may work independently on separate parts of the prototype program.

Data General's Extended ALGOL is a full implementation of ALGOL 60: recursive procedures are allowed, array declarations may be any arithmetic expression including function calls, and integer labels and conditional expressions may be used. Programs written in standard ALGOL 60 are completely compatible with Data General's Extended ALGOL.

Data General's Extended ALGOL includes facilities for character manipulation, list processing, arithmetic with up to 60 decimal digits of precision, and random or sequential I/O with optional formatting. These extensions complement the basic structure of ALGOL and significantly increase the convenience of ALGOL programming without making the language unwieldy.

FEATURES OF DATA GENERAL'S EXTENDED ALGOL

Character strings are implemented as an extended data type to allow easy manipulation of character data. The program may, for example, read in character strings, search for substrings, replace characters, and maintain character string tables efficiently.

Multi-precision arithmetic allows up to 60 decimal digits of precision in integer or floating point calculations.

Device-independent I/O provides for reading and writing in line mode, sequential mode, or random mode. Free form reading and writing is permitted for all data types, or output can be formatted according to a "picture" of the output line or lines.

Dynamic conversion of parameter types allows one program to process data of several types. Expressions of mixed type or precision are allowed.

Recursive and reentrant object code is generated by the Extended ALGOL compiler. This is particularly important in real-time applications for fast context-switching among programs.

Dynamic storage allocation frees the programmer from many details of data layout and storage assignment.

N-dimensional arrays may be allocated dynamically at run time. Subscripts and array bounds in the array declaration may be any expression, including function references, negative numbers, and subscripted variables.

Direct addressing capabilities are provided by drawing upon some of the powerful addressing features of PL/I, including based and pointer variables. This extended addressing capability provides more efficient code and more easily understood source language notation.

Subscripted labels provide for direct branching to a given label upon evaluation of the subscript expression.

Bit manipulation is provided, using logical operators, octal and binary literals, and built-in functions.

Named constants may be explicitly declared to aid program readability and to simplify program modification.

Condition signalling allows the programmer to set conditions for interrupting normal execution and switching to an interrupt procedure.

Object code optimization is performed for efficient register usage, in-line generation of literals, sub-expression elimination, optimal use of machine instructions, and efficient storage allocation. A commented assembly listing is provided.

Straightforward subprogram linkage conventions, which are well documented, simplify interfacing to assembly language subprograms.

Explicit diagnostics aid debugging at the source level. Compatibility with the Data General symbolic debugger aids run-time debugging.

ELEMENTS OF EXTENDED ALGOL

Declarations

A wide range of characteristics is possible for ALGOL program variables, making them easily adaptable to specific usages. Each program variable has the characteristics of shape, size, data type, and storage class.

For each program variable a declaration of desired characteristics is given. Data type is always declared. Other characteristics have default values if not declared.

The data types are **integer, real, complex, boolean, pointer, string,** and **label**. Data types can be converted to other data types, provided that the shape, size, and storage class are compatible. Integer, real, boolean and pointer data can be converted to and from strings.

The shape of a program variable is scalar, **array, literal, switch, procedure** or **operator**. Scalar is the default shape of a variable. If another shape is to be used, the shape must be defined in a declaration.

The precision of a program variable represents size in words for numeric data (from one to 15 words) and maximum number of characters for string data (maximum of 16,283 characters). Precision is only declared for numeric and string data. Other data types have a fixed length for all cases.

Most ALGOL program variables are allocated and freed dynamically on entry and exit from a portion of the program, called a block. Such variables do not need a storage class declaration. The storage classes that are declared are **own, external,** and **based.**

A **based** variable is merely a template of a program variable, used as described in the section on "Extended Addressing." **own** and **external** variables are held in storage throughout an ALGOL program, instead of being dynamically stored.

Some examples of declarations are:

Characteristic	Example	Description
Data Type	**integer** I, J;	I and J have integer numeric values.
	real K;	K has a real numeric value.
	complex C;	C has a complex numeric value.
	boolean done;	done has a value true or false.
	pointer P;	P has an address as a value.
	string measure;	measure can contain up to 32 characters.
Shape	**real array** MAT[0:4,1:5];	MAT is a 5x5 floating-point array whose first element is MAT[0,1] and last element is MAT [4,5].
	string procedure X(a,b);	X is a function that returns a string value. Its formal parameters are a and b.
Storage Class	**based integer** I,J;	I and J are template variables.
	own real K;	K has a value which is preserved between calls.
	external procedure x;	x is a separately compiled procedure.
Precision	**real** (9) **array** RAY [7];	each of the 8 elements of RAY has 9 word precision.
	string (2000) s;	s can have up to 2000 characters.

Statements

Statements define program action. In ALGOL, there are relatively few different types of statements, but each type is extremely flexible. For example, the basic assignment statement resembles that of most compiler languages:

V := expression;

The expression on the righthand side of the assignment symbol is evaluated, converted if necessary to the data type of variable V, and assigned to variable V.

However, the programmer could have written:

CHAR := A := V := expression;

In this case, the expression is evaluated and assigned to each of the variables V, A, and CHAR, and, in each case, the expression is converted to the data type of the location, each of which might have a different data type. If the expression evaluates to a real number, it might simply be assigned to real V, then truncated and assigned to integer A, and converted to string format and assigned to string CHAR.

The general forms of ALGOL statements are shown below with some examples. In the statement formats following, X is an expression, V is a variable, and S is a statement, compound statement, or block.

Statement Format	Description
for V: = $X_1, \ldots, X_n$ **do** S;	Statements providing loops
for V: = X_1 **step** X_2 **until** X_3 **do** S;	
for V: = X_1 **while** X_2 **do** S;	
V: = X;	Assignment of expression to variable(s).
V_1: = V_2: = . . . V_n: = X;	
if X **then** S;	Conditional statements.
if X **then** S_1 **else** S_2;	X is boolean expression.
go to X;	Unconditional transfer, X is a label expression or label.
NAME;	Call to a procedure NAME.
NAME ($X_1, X_2, \ldots, X_n$);	Each X is a parameter.
	Represents the null statement.

Statements and Blocks

In ALGOL, whenever a single statement may be used, the programmer may choose to use a compound statement or a block. A compound statement is a series of statements surrounded by **begin** and **end.** A block is simply a compound statement which contains declarations for its internal variables, procedures or arrays. Entering a block triggers the dynamic allocation of storage used by the block.

Statement Type	Examples
Basic	a := p + q; **go to** Naples; START: CONTINUE: W := 7.993;
Compound	**begin** x:= 0 ; **for** y:= 1 **step** 1 **until** n **do** x:= x + A[y] ; **if** x > q **then go to** STOP **else if** x > w − 2 **then go to** S ; Aw: St: W := x + bob **end**;

Block	Q: **begin integer** i, k ; **real** w ;
	for i := 1 **step** 1 **until** m **do**
	for k := **i + 1 step 1 until** m **do**
	begin w := A[i, k] ;
	A[i, k] := A[k, i] ;
	A[k, i] := w
	end for i and k
	end block Q

Expressions

An expression is a rule for computing a value. In Extended ALGOL, this rule may result in a value which is numeric, a character string, the address of a computer word, a label, or a truth value. Expressions may contain variables of various precisions and data types; code will be generated to convert variables to a common type and precision whenever necessary. For example, a real number could be multiplied by a character string which contained the ASCII representation of a number. The number in the string would be converted to type real at execution time. Similarly, a subscript or **for** statement variable could be a real, string or integer variable. Expressions can also be used as values passed on procedure calls, and as the dynamic dimensions of an array.

Examples	Description
cos (y + z X 3) (a − 3/y + vu ↑ 8)	Simple expressions. Variables y, z, a and vu may be integers, reals, or strings.
x[sin(n X pi/2), Q[3, n, 4]]	Subscript expressions. The subscripts of the 2-dimensional array x are the result of the function sine and an element of the 3-dimensional array Q.
s := p[0] := n := n + 1 + s	The value n + 1 + s is stored in the order n, p[0] , s. The type of the value will be converted on each step if required.
own integer array A[**if** c < 0 **then** 2 **else** 1 : 20]	Array dimensions. The array A will have a lower dimension of 1 or 2 depending on the value of c.

Procedures

Subroutines and functions in ALGOL are declared in the same manner as variables. An ALGOL **procedure** may be local to a particular block or may be used by the entire program. The declaration of an ALGOL procedure consists of three parts: a procedure header which tells the name of the procedure and the order of its dummy arguments, a list of declarations for the arguments, and a statement (which may be a compound statement or block) defining the action to be performed. The following example shows some declarations from a block. The declarations include a procedure which determines whether or not a character is alphabetic.

```
begin   integer i, j; real array a[100] ;
        boolean procedure alphabetic (c);
                string (1) c;
                alphabetic := (c≥"a")
                   and (c≤"z");
        string (511) str;
```

Extended ALGOL provides an additional **external** declaration to allow a procedure to be compiled separately.

A procedure which is used as a function may return a value and may be of any data type (including **string** and **pointer**). If the procedure does not return a value, then it is a subroutine and is called by means of a procedure statement. The I/O procedures are examples of this kind. If the procedure returns a value, then it is called as part of an expression.

Some procedures are defined by the ALGOL compiler and do not need to be declared by the programmer. These include the I/O procedures, arithmetic procedures such as sin and cos, and various procedures of general utility to the programmer, such as hbound which returns the upper bound of an array dimension.

CHARACTER STRINGS

Data General's Extended ALGOL provides the programmer with facilities for directly accessing character string data to simplify the development of business applications, information retrieval programs, text processing programs, and other applications which depend heavily on the manipulation of sequences of characters.

In ALGOL, a character string is considered to be a sequence of characters, each of which is referenced by its position in the sequence. The sequence may vary in length from zero characters to the maximum declared for the string. ALGOL allows strings up to 16,283 characters long. A program may reference an entire string for purposes of comparison or assignment, for example, or may reference any contiguous sub-sequence of the string.

string is a data type like **real** or **integer** and may be used in the same manner. For example, there may be arrays of strings, functions returning string values, strings passed as parameters, **own** strings which maintain their value between calls, and **based** strings which allow any locations in memory to be accessed as character data. Strings may be used in mixed type expressions and may be converted to or from type **real, boolean, integer** or **pointer** as required.

All strings have an associated current length and maximum length. The maximum length is declared as the precision of the string.

For example:

```
string (300) a, b;
```

declares the strings a and b to have a maximum length of 300 characters. The current length of these strings will be zero until data is moved into the strings by a string assignment such as:

```
a := "abcdefg";
```

The programmer may reference a part of the string by means of the substr built in function. The assignment:

```
b := substr(a, 2, 5);
```

would move the second through fifth characters of a into the first four characters of b. If we then used substr to store into the fifth character of b:

```
substr (b, 5) := ".";
```

The current length of b would be updated to five. If, for this assignment, the current length of b had not been known, the length built in function could have been used:

```
substr(b, length(b) + 1) := ".";
```

The following procedure put appends its parameter, the string s, to string out (which is external to this procedure), separating strings packed into out by blanks.

```
procedure put (s); string s;
begin   integer n;
        n := length (out) ;
        if n > 0 then begin
                substr (out,n+1) := " ";
                n := n+1;
                end;
        substr (out, n+1, n+length (s)) := s;
        end;
```

The procedure put might be called in the following manner. (hundredsstring is another ALGOL procedure and returns a character string value.)

```
if thousands > 0 then begin
   put (hundredsstring (thousands));
   put ("thousand");
   end;
```

Arrays of strings are allowed in which all the elements have the same maximum length but not necessarily the same current length. A string array is declared in the same manner as an array of any other data type. For example:

```
string (10) array [-2: 12];
```

declares a fifteen-element array in which each element has a maximum length of ten characters.

String arrays are particularly convenient for maintaining name or symbol tables. The following procedure looks up symbol in its internal symbol table. If symbol is found its index is returned. If not found, symbol is appended to the table and the new index returned.

```
integer procedure lookup (symbol);
        string symbol;
begin   own string (8) array symboltable[1:1000] ;
        own integer symbolcount;
        integer i;
        for i := 1 step 1 until symbolcount do
           if symboltable [i] = symbol then go to found;
        i := symbolcount := symbolcount+1;
        symboltable [i] := symbol;
found:  lookup := i;
        end
```

The syntax of character strings has been designed to parallel other data types as closely as possible to provide a logical and useable addition to ALGOL 60. String variables, string arrays and string valued procedures allow all the language constructs of variables, arrays and procedures of other data types. The following sort procedure may be modified to sort arrays of any other data type by simply redeclaring the type of array a and temporary w.

```
procedure Shellsort (a);
    string array a;
    begin integer i, j, k, m;
        string w;
      for i := 1 step i until size (a) do m := 2 X i-1;
      for m := m/2 while m ≠ 0 do
        begin k := size (a)-m;
          for j := 1 step 1 until k do
            begin for i := j step -m until 1 do
              begin if a[i+m] ⩾ a [i] then go to 1;
                w := a [i] ; a [i] := a [i+m] ;
                a [i+m] := w;
              end i;
        1 : end j
        end m
    end Shellsort;
```

INPUT/OUTPUT

Input/Output facilities in Data General's Extended ALGOL have been designed to provide flexibility in the formatting of data while retaining a simplicity which allows an efficient, easy-to-use implementation.

The I/O procedures manipulate named data files which may be accessed sequentially or randomly. In order to read or write from a file, a channel number must be associated with the file name and the file positioned to its beginning. This is done by means of the open call. For example, the disk file "newdata" may be associated with channel number 2 by means of the call:

```
open (2, "newdata");
```

The file name may be passed in any legitimate string such as a string variable, substring or the result of a string procedure call. File names corresponding to physical devices will, by convention, have a "$" as the first character. The channel number may be released and I/O terminated by means of the close procedure. For example:

```
close (2);
```

will terminate I/O to channel 2.

The procedures for reading and writing data determine the properties of the data from the call. This allows a small number of simple procedures to be used for all types of data. In the following examples the variables shown will be assumed to be declared as follows:

```
real x;
real (4) y;
integer i;
integer (3) j;
boolean flag;
pointer p;
literal cr ("<15>")
```

Angle brackets may be used to include any ASCII character in a string. The literal cr is defined to contain an ASCII carriage return. All the procedures for writing require that a carriage return be output explicitly. This allows the programmer to use multiple writes to build an output line.

Data may be input in free format by means of the read procedure. A typical call to read:

```
read (0, j, flag, p);
```

could accept the following data from file 0:

```
7777600r8, true, 10400
```

In this case the default radix for integers (base 10) was overridden by specifying the radix for j (base 8). Since p is a pointer it is expected in octal. The items may be separated by commas, carriage returns or spaces. The entire list must be terminated with a carriage return.

String data may be read in either of two forms. If the first character of the string is a quote, then all characters will be included in the string until another quote is reached (the quotes will not be included in the string). Otherwise, all characters will be included until a carriage return is reached (the carriage return will not be included).

The programmer may include a label as the last parameter of read. If a label is passed, a transfer will be made to this label if an end-of-file is encountered. For example:

```
loop: read (1, i, eof);
      go to loop;
eof:
```

will read integers from the file until an end-of-file is encountered.

Data may be written in free format by means of the write procedure. A write call:

```
write (1, "address is", p, cr);
```

would produce the output:

```
address is 10400
```

The data format will be determined according to type and precision. The format will be identical to that used for a conversion to string from the data type in a string assignment such as: s := x;

Frequently it will be desirable to output data in a more controlled fashion, such as columns in a table. For this the output procedure is used; output uses a "picture" of the output line to specify format. Pound signs, "#", are used to indicate positions to be filled in by a variable.

```
output (1, "####  #######r8<15>", i,j);
```

This example would produce the following output:

```
11  7777600
```

In this case a specified radix (r8) was again used to override the default radix for integer. The picture for a number may include a period (".") to indicate the position of a decimal point, "E" to indicate the start of an exponent field, and a plus or minus sign to indicate that the sign is to be output for all values or only for negative values. The statement:

```
output (1,
   "+####.#####E##     +####.#####E##<15>",
   x, y);
```

Can be used repetitively to produce the following table:

```
+0.11947E 5     +0.30576E 5     +0.28890E 4
-0.18810E 5     -0.23770E 4     -0.16468E 5
+0.21621E 5     -0.31646E 5     -0.92130E 4
-0.26072E 5     +0.30945E 5     -0.25986E 5
+0.10895E 5     -0.32028E 5     -0.70270E 4
+0.53380E 4     +0.28251E 5     +0.13792E 5
+0.17273E 5     +0.20342E 5     +0.29543E 5
-0.20708E 5     -0.32347E 5     -0.65740E 4
+0.17843E 5     +0.10904E 5     -0.13551E 5
-0.19090E 5     +0.12607E 5     +0.93000E 4
```

If the data to be output is too long for the field specification, the field will be extended as necessary. If the field is longer than required, the data will be right justified in a field of blanks if it is numeric or left justified in a field of blanks if non-numeric.

```
output (1, "result is #<15>", flag);
```

for a true value of the boolean variable, flag, would output:

```
result is true
```

Free format conversion may be combined with formatted output by means of string assignments. Numeric data assigned to a string will be converted in free format. The string may then be output by a picture. For instance, the declarations and statements following:

```
real (7) array x[0:10];
string sx; integer i;

for i := 0 step 1 until 10 do
output (1, "x[#] = #<15>", i, (sx := x[i]) );
```

could be used to generate:

```
X[0] = 0.28520493777418823300177665235398176944
X[1] = -0.99713870375559424119597552792558286226
X[2] = -0.79670283831695594412018481055358148066
X[3] = -0.51639366713561743460484977706957448 16
X[4] = -0.64659959840557860999425444991 14943005
X[5] = 0.98198455568969611473310622816799746713
X[6] = 0.18640399443552892189139315380972999035
X[7] = -0.61138634269347266747555737286322124189
X[8] = -0.97671678225879973722637929288768514877
X[9] = 0.80174413030837630989026722293576181753
X[10] = -0.99998624434888724876739375734334185606
```

In addition to the previously described procedures, calls are provided to read and write a binary stream of bytes, delete or change the name of a file from program control, and to determine the length of a disk file in bytes.

An ALGOL program operating under the Disk Operating System may use any of the facilities for reading or writing to randomly access a disk file. A DOS disk file may be as long as 33,423,360 bytes. The program may position to any byte of the file before performing an input or output operation by calling the position procedure with the channel number and byte number as arguments. A location counter kept for each channel is updated after each read or write to indicate the next byte to be accessed. This location counter may be read at any time by calling the fileposition procedure.

ARITHMETIC

Data General's Extended ALGOL gives the programmer wide latitude in his choice of arithmetic capabilities. Single precision arithmetic (16 bits) is performed by optimized sequences of machine instructions. Two-word floating point and two-word integer arithmetic may be performed by a software package which combines small size and very fast arithmetic. Generalized multiple precision arithmetic allowing computations with up to 60 decimal digits of precision is provided by another software package.

Constants in the program may specify any radix, and up to 15 words of precision. If a long constant is frequently used, it may be given a name to simplify program modifications and aid documentation.

All the standard ALGOL arithmetic functions are provided and will return results precise to at least 23 digits.

The following program demonstrates the use of 15 word integer arithmetic and a recursive procedure to generate a table of factorials. Note that the procedure, factorial, consists of a single statement.

```
begin

integer (15) procedure factorial (n); integer (15) n;
factorial := if n>1 then n X factorial(n-1) else 1;

integer (15) n;

open (1, "$tto");
for n := 1 step 1 until 50 do
   output (1, "### #<15>", n, factorial(n) );

end
```

```
 1    1
 2    2
 3    6
 4    24
 5    120
 6    720
 7    5040
 8    40320
 9    362880
10    3628800
11    39916800
12    479001600
13    6227020800
14    87178291200
15    1307674368000
16    20922789888000
17    355687428096000
18    6402373705728000
19    121645100408832000
20    2432902008176640000
21    51090942171709440000
22    1124000727777607680000
23    25852016738884976640000
24    620448401733239439360000
25    15511210043330985984000000
26    403291461126605635584000000
27    10888869450418352160768000000
28    304888344611713860501504000000
29    8841761993739701954543616000000
30    265252859812191058636308480000000
31    8222838654177922817725562880000000
32    263130836933693530167218012160000000
33    8683317618811886495518194401280000000
34    295232799039604140847618609643520000000
35    10333147966386144929666651337523200000000
36    371993326789901217467999448150835200000000
37    137[illegible]902400000000
38    [illegible]000000
```

EXTENDED ADDRESSING

Use of pointers and based variables is a programming technique which allows the systems programmer to achieve a very high level of object code efficiency.

In most programming languages, certain information is available to the programmer that is not available to the compiler through the source program. The compiler must always assume the "worst case" in order to generate safe code.

For example, any subprogram call can potentially redefine all external variables. An assignment to any element of an array will force the compiler to assume that all values in the array have been modified. In the case of arrays passed as parameters, the compiler must generate "worst case" code for computing subscripts, since neither the bounds, precision, nor number of dimensions may be known until run time.

Pointers and based variables provide a mechanism for explicitly manipulating machine addresses. Using the facility, the programmer can, for example, force a subscript calculation to be performed only once in a frequently executed part of his program. As another example, if the programmer knows that an external variable will not be modified by a call, he can use pointers and based variables to convey this knowledge to the compiler.

The programmer declares an identifier, called a pointer. The pointer's value is the address of some program variable. Pointer expressions are allowed, so address offsets can be given. When the pointer is used, it points to a based variable with the operator →; in effect, the pointer and based variable have been substituted for the precise address of the program variable that the programmer wants.

The example following shows two procedures that perform the same computation. The first uses subscript values calculated at run time. The second uses pointers and based variables. Note that the four subscript calculations are always required in the first case, since a and b may be the same array or have elements in common. In the procedure using pointers, the programmer has communicated to the compiler his additional knowledge about the two arrays.

```
procedure fn1 (a, b, i);
        integer array a, b;
        integer i;
begin   a[i] := b[i];
        b[i] := a[i+1];
        end;
procedure fn2 (a, b, i);
        integer array a, b;
        integer i;
begin   pointer ap, bp;
        based integer bi;
        ap := address(a[i]);
        bp := address(b[i]);
        ap→bi :=bp→bi;
        bp→bi :=(ap+1)→bi;
        end;
```

Following is coding generated by the last two statements of the procedure containing pointers. To understand the coding, note that register 3, which appears as the third field in some instructions, contains the current stack pointer.

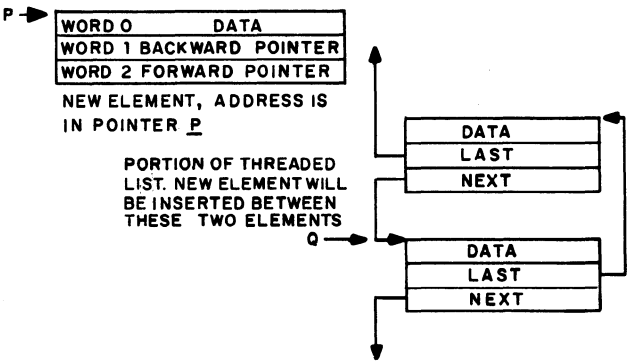
```
                ; AP->BI := BP->BI;

00014'023616            LDA     0,@S+5,3        ;BP
00015'043615            STA     0,@S+4,3        ;AP

                ; BP->BI := (AP+1)->BI;

00016'031615            LDA     2,S+4,3         ;AP
00017'151400            INC     2,2
00020'031000            LDA     2,0,2
00021'053616            STA     2,@S+5,3        ;BP
```

The following example shows a more complicated use of pointers and based variables to produce a very efficient statement which will thread a new element into a forward and backward threaded list. The data structure is organized as shown below.



The new element pointed to by p will be inserted in the list before element q.

```
            ; POINTER P, Q;
            ; LITERAL LAST(1), NEXT(2);
            ; BASED POINTER BP;
            ;
            ; (((P+LAST)->BP := (Q+LAST)->BP)+NEXT)->BP :=

00004'021611          LDA     0,S+0,3         ;P
00005'101400          INC     0,0
00006'025612          LDA     1,S+1,3         ;Q
00007'125400          INC     1,1
00010'131000          MOV     1,2
00011'025000          LDA     1,0,2
00012'041613          STA     0,S+2,3
00013'047613          STA     1,@S+2,3        ;TEMPORARY
00014'034000-         LDA     3,.LP
00015'021601          LDA     0,L+1,3         ;LITERAL
00016'107000          ADD     0,1

            ; (((P+NEXT)->BP := Q)+LAST)->BP := P;

00017'034005$         LDA     3,.SP
00020'031611          LDA     2,S+0,3         ;P
00021'143000          ADD     2,0
00022'031612          LDA     2,S+1,3         ;Q
00023'041613          STA     0,S+2,3
00024'053613          STA     2,@S+2,3        ;TEMPORARY
00025'151400          INC     2,2
00026'021611          LDA     0,S+0,3         ;P
00027'041000          STA     0,0,2
00030'045613          STA     1,S+2,3
00031'043613          STA     0,@S+2,3        ;TEMPORARY
```


DATA GENERAL CORPORATION
Southboro, Massachusetts 01772