```
+444444444444444444444444444444444444444444444444444444444444444444444444444444+

                                   $      $  $$$$
                                   $      $  $   $
                                   $      $  $   $
                                   $      $  $   $
                                   $      $  $   $
                               $   $  $   $  $   $
                                $$$    $$$   $$$$

                        USER=JJD  QUEUE=LPT  DEVICE=@LPB1
               SEQ=5  QPRI=127  LPP=63  CPL=80  COPIES=1  LIMIT=49

                          CREATED:  20-OCT-77  15:45:24
                         ENQUEUED:   6-DEC-77  14:21:06
                         PRINTING:   6-DEC-77  14:31:04

                           PATH=:PDD:MEMO:MEMO$312.LS

   $   $  $$$$$  $   $   $$$     $    $$$$$    $     $$$          $      $$$
   $$ $$  $      $$ $$  $   $   $$$$     $    $$    $   $         $     $   $
   $ $ $  $      $ $ $  $   $  $ $      $    $ $        $         $     $
   $ $ $  $$$$   $ $ $  $   $   $$$    $$$     $      $$          $      $$$
   $   $  $      $   $  $   $    $ $      $    $     $            $         $
   $   $  $      $   $  $   $  $$$$   $   $    $    $        $$   $     $   $
   $   $  $$$$$  $   $   $$$     $     $$$   $$$$$  $$$$$    $$   $$$$$  $$$
+444444444444444444444444444444444444444444444444444444444444444444444444444444+


                                AOS XLPT REV 01.00
```

To: Y'All

From: John Pilat, Michael Richmond

Subject: COBOL S-Language, 1977

Date: 20/Oct/77 Memo No. 312

Abstract:

This memo describes the COBOL S-Language. It supports IBM 370 and Eclipse COBOL data types, as well as the ANSI COBOL standard.

1 Data Types

The data types supported by this COBOL s-language include those mandated by the ANSI (1974) standard, as well as those extensions offered by IBM 370 and the ECLIPSE. Except for some minor storage allocation differences, the extensions offered by both of these potential "market targets" are identical. Should it become desirable (or necessary because of time constraints) this s-language can be stripped of support for these extensions, without effect on the ANSI core, by deleting most instructions containing the word "BINARY" and all instructions containing the word "FLOATING".

1.1 Data Type Considerations Masked by the Compiler

Some aspects of COBOL "typeness" will be removed by the COBOL compiler, and will be realized with explicit instructions or through storage allocation policy. These include:

a) Source operations requiring scaling or decimal point alignment. The compiler will generate explicit instructions to align or scale operands.

b) The determination of radix and the concommitant need for conversion(s) in operations. While this does have the effect of proliferating the number of instructions, it allows the IBM/ECLIPSE extensions to be separated easily from the remaining instructions and does not require a slow secondary decode for programs which use only ANSI standard types.

c) The need for editing, if any. The compiler will generate explicit edit instructions.

d) Synchronization and alignment. These are matters of storage allocation policy.

1.2 Non-numeric Data

1.2.1 ASCII as the NATIVE Character Set

This COBOL s-language supports ASCII as the NATIVE character set. Character translation instructions are included to deal with the alternate character set features of COBOL.

1.2.2 EBCDIC

Support of EBCDIC may be required at all levels of commercial software. The level of support required is something that must be decided at the product planning level before EBCDIC support (beyond the normal translation instructions) can be defined for a COBOL s-language.

1.3 Numeric Data

COBOL defines a data item to be numeric if its picture consists only of the characters 9, S, V, and P. Numeric data items may be combined freely in arithmetic statements. A distinction is made between data representations which must be in standard format (USAGE IS DISPLAY) and those which may be in implementor-defined formats (USAGE IS COMPUTATIONAL).

1.3.1 USAGE IS DISPLAY

All non-numeric data is USAGE IS DISPLAY. Numeric data can also be of this form, which requires that the internal representation be character-oriented decimal digits. The programmer can control the existence and location of an operational sign within a numeric DISPLAY item. In all cases, the space character is a valid substitute for the character zero.

1.3.1.1 Unsigned

This representation is generated when a data item is described by a picture which does not contain the operational sign designator S. The numeric value is represented as a string of the characters '0'..'9'.

1.3.1.2 Separate Sign Leading/Trailing

This is the standard interchange representation for signed data. The sign is either '+' or '-' and is the first/last character position of the data field. The digit values are as above, '0'..'9'.

1.3.1.3 Overpunched Sign Leading/Trailing

In this representation, the first/last character position of the data field implies both a digit and an operational sign. This encoding is a de facto standard. It is presented herewith along with the EBCDIC encoding for comparison. Note that an ASCII/EBCDIC character translation on a character decimal field performs the correct mapping.

Decimal Digit	ASCII +	ASCII -	ASCII unsigned	EBCDIC +	EBCDIC -	EBCDIC unsigned
0	7B	7D	30	C0	D0	F0
1	41	51	31	C1	D1	F1
2	42	52	32	C2	D2	F2
3	43	53	33	C3	D3	F3
4	44	54	34	C4	D4	F4
5	45	55	35	C5	D5	F5
6	46	56	36	C6	D6	F6
7	47	57	37	C7	D7	F7
8	48	58	38	C8	D8	F8
9	49	59	39	C9	D9	F9

(all table entries in hexadecimal)

1.3.2 USAGE IS COMPUTATIONAL Data

The COBOL standard requires no particular representation for COMPUTATIONAL data. In the interest of ECLIPSE/IBM compatibility we have made the bindings presented below.

1.3.2.1 COMPUTATIONAL

These data items will be represented as signed, two's complement binary integers, sized 8, 16, 24, 32, 40, 48, 56, and 64 bits. The compiler will generate MAGNITUDE instructions as needed to handle unsigned binary data (i.e., the interpreter need not worry about it).

1.3.2.2 COMPUTATIONAL-1 and COMPUTATIONAL-2

COMPUTATIONAL-1 data items will be 32 bit system floating point. COMPUTATIONAL-2 data items will be 64 bit system floating point. Note that the interpreter must be prepared to handle operations on mixes of the two precisions.

1.3.2.3 COMPUTATIONAL-3

COMPUTATIONAL-3 data items will be represented as packed decimal.

1.4 Name Table Entries

The COBOL s-language makes extensive use of namespace addressing features. All COBOL subscripting and indexing can be realized directly in the name table. IBM's extension which allows more than one dimension of an array to vary in size based on a run time value (which makes conversion of some IBM COBOL programs difficult on many systems) may also be realized directly.

1.4.1 Starting Address and Fetch Direction

All COBOL data will be addressed from the leftmost (i.e., low address) bit and be fetched to the right. While it might appear that numeric operands should be addressed from the rightmost bit, this is not the case. First, the rightmost bit does not correspond to the low order digit (the address of start of the right most digit is seven less than the address of the rightmost bit). More importantly, an item addressed from the right would require another name table entry in order to be moved without regard to its type.

1.4.2 Length

The length field in the name table contains an entity's bit length. The COBOL s-language uses entity length to distinguish between some source language data types. Single and double precision floating point (COMPUTATIONAL-1 and COMPUTATIONAL-2) are distinguished between each other by 32 or 64 in the length field. Size distinctions specified in a picture are also (obviously) reflected in the length field.

1.4.3 Type

The COBOL s-language uses the name table type field to distinguish among COBOL's cornucopia of decimal data types. Note that of these seven types, only packed decimal is non-standard! The interpreter need only examine the type field when an operand is numeric decimal (something is numeric decimal if the opcode says it is). The encoding of the four bit type field is as follows:

0000:	Packed decimal, unsigned.
0001:	Packed decimal, signed.
0010:	Character decimal, unsigned.
0011:	Character decimal, sign is separate and leading.
0100:	Character decimal, sign is separate and trailing.
0101:	Character decimal, sign is overpunched and leading.
0110:	Character decimal, sign is overpunched and trailing.

The decimal operations are defined to accept any of these decimal formats. Note that when the interpreter stores a signed value (be it decimal, floating or binary) into a decimal datum whose type field indicates "unsigned", only the magnitude of the value is stored.

1.5 Instructions (100)

The notation used to describe instructions is as follows:

Angle brackets < > enclose operands which are names. For example, <source> is the name (index into the name table) of the source operand.

Equal signs = = enclose operands which are literals or offsets. For example, =pc= designates an offset from the program counter.

1.5.1 Moves (4)

* MOVE_CHARACTERS <source>,<destination>

The lengths of the operands as specified in the name table need not be equal. If the length of the source length exceeds the length of the destination, only "destination length" bits are moved. If the length of the destination is greater than the length of the source, the excess destination bits are filled with ASCII blank. Note that the lengths of the operands will always be some multiple of eight.

* MOVE_CHARACTERS_EQUAL <source>,<destination>

Exactly "source_length" bits are moved to the destination from the source.

* MOVE_CHARACTERS_EDITED <source>,<destination>,<picture>

This instruction is described in terms of pointers to the source, destination and picture. These pointers are not part of the macro state. One picture element exists for each eight bit character in the destination. The picture is processed (and its pointer bumped) in syncnrony with the destination. Each pixel is eight bits. Its interpretation is as follows:

> 0001nnnn: Transfer "nnnn" characters from the source to the destination. Both source and destination pointers are bumped by "nnnn". If the source is exhausted, the source pointer is not bumped, and an ASCII blank is transferred.
>
> All others: The pixel character is transferred to the destination and the destination pointer is bumped by one.

* MOVE_CHARACTERS_TRANSLATED
<source>,<destination>,<translate table>

The translate table is a string of eight-bit elements. This move behaves exactly like MOVE_CHARACTERS except that as each character is fetched from the source string, it is used as an index into the translate table. The contents of the translate table at the indexed location are transferred to the destination. Note that if blank padding is required, the blank pad must be translated as well.

1.5.2 Conversions (15)

```
*    ASSIGN_DECIMAL_TO_BINARY <source>,<destination>

*    ASSIGN_DECIMAL_TO_FLOATING <source>,<destination>

*    ASSIGN_DECIMAL_TO_DECIMAL <source>,<destination>

*    ASSIGN_BINARY_TO_DECIMAL <source>,<destination>

*    ASSIGN_BINARY_TO_FLOATING <source>,<destination>

*    ASSIGN_BINARY_TO_BINARY <source>,<destination>

*    ASSIGN_FLOATING_TO_BINARY <source>,<destination>

*    ASSIGN_FLOATING_TO_DECIMAL <source>,<destination>

*    ASSIGN_FLOATING_TO_FLOATING <source>,<destination>

*    ASSIGN_MAGNITUDE_BINARY <source>,<destination>

*    DECIMAL_SCALE_LEFT =scale diff=,<source>,<destination>
*    DECIMAL_SCALE_RIGHT =scale diff=,<source>,<destination>


*    BINARY_SCALE_UP_BY_10 =scale diff=,<source>,<destination>


*    BINARY_SCALE_DOWN_BY_10 =scale diff=,<source>,<destination>


*    DECIMAL_ROUND =rounding position=,<source>,<destination>
```

1.5.3 Numeric Editing (1)

* EDIT_NUMERIC <source>,<target>,<picture>

The source is a decimal number which is edited into the target according to the picture. The precise interpretation of the picture is defined by an SPL procedure in the appendix.

1.5.4 Numeric Comparison Branches (30)

The numeric comparison branches compare two named values, or a single named value and zero, and branch to the indicated pc offset if the indicated comparison is true. The type of the source(s) and the comparison to be performed is specified by each instruction's name.

```
*     DECIMAL_COMPARE_EQUAL <source1>,<source2>,=pc=

*     DECIMAL_COMPARE_NOT_EQUAL <source1>,<source2>,=pc=

*     DECIMAL_COMPARE_GREATER <source1>,<source2>,=pc=

*     DECIMAL_COMPARE_GREATER_OR_EQUAL <source1>,<source2>,=pc=

*     BINARY_COMPARE_EQUAL <source1>,<source2>,=pc=

*     BINARY_COMPARE_NOT_EQUAL <source1>,<source2>,=pc=

*     BINARY_COMPARE_GREATER <source1>,<source2>,=pc=

*     BINARY_COMPARE_GREATER_OR_EQUAL <source1>,<source2>,=pc=

*     FLOATING_COMPARE_EQUAL <source1>,<source2>,=pc=

*     FLOATING_COMPARE_NOT_EQUAL <source1>,<source2>,=pc=
*     FLOATING_COMPARE_GREATER <source1>,<source2>,=pc=

*     FLOATING_COMPARE_GREATER_OR_EQUAL <source1>,<source2>,=pc=

*     DECIMAL_ZERO_TEST_EQUAL <source>,=pc=

*     DECIMAL_ZERO_TEST_NOT_EQUAL <source>,=pc=

*     DECIMAL_ZERO_TEST_GREATER <source>,=pc=

*     DECIMAL_ZERO_TEST_GREATER_OR_EQUAL <source>,=pc=

*     DECIMAL_ZERO_TEST_LESS <source>,=pc=

*     DECIMAL_ZERO_TEST_LESS_OR_EQUAL <source>,=pc=

*     BINARY_ZERO_TEST_EQUAL <source>,=pc=

*     BINARY_ZERO_TEST_NOT_EQUAL <source>,=pc=

*     BINARY_ZERO_TEST_GREATER <source>,=pc=

*     BINARY_ZERO_TEST_GREATER_OR_EQUAL <source>,=pc=
*    BINARY_ZERO_TEST_LESS <source>,=pc=

*    BINARY_ZERO_TEST_LESS_OR_EQUAL <source>,=pc=

*    FLOATING_ZERO_TEST_EQUAL <source>,=pc=

*    FLOATING_ZERO_TEST_NOT_EQUAL <source>,=pc=

*    FLOATING_ZERO_TEST_GREATER <source>,=pc=

*    FLOATING_ZERO_TEST_GREATER_OR_EQUAL <source>,=pc=

*    FLOATING_ZERO_TEST_LESS <source>,=pc=

*    FLOATING_ZERO_TEST_LESS_OR_EQUAL <source>,=pc=
```

1.5.5 Non-numeric Comparison branches (12)

Each non-numeric comparison executes a branch to the pc offset if its specified condition is true. The comparisons operate on eight bit characters in sequence from low to high address ("left-to-right). The CHAR_CLASS_NUMERIC and CHAR_CLASS_NOT_NUMERIC instructions also operate on packed 4-bit digits (if indicated by the name table type field).

When more than one source is specified, they need not be the same length. When the sources are not the same length, the shorter source should be treated as if it were extended on the right by the ASCII blank character. The two-source comparisons effectively proceed by comparing characters in corresponding character positions until either a pair of unequal characters is encountered, or until the end is reached, whichever comes first. The first pair of unequal characters encountered is used to determine if the condition specified by each instruction's name is true. The sources are considered equal if all pairs of characters compare equally through the last pair.

*   CHAR_COMPARE_EQUAL <source1>,<source2>,=pc=

*   CHAR_COMPARE_NOT_EQUAL <source1>,<source2>,=pc=

*   CHAR_COMPARE_GREATER <source1>,<source2>,=pc=

*   CHAR_COMPARE_GREATER_TRANSLATED
<source1>,<source2>,<table>,=pc=

This instruction behaves like CHAR_COMPARE_GREATER except that the sources are translated through "table" as they are compared.

*   CHAR_COMPARE_GREATER_OR_EQUAL <source1>,<source2>,=pc=

*   CHAR_COMPARE_GREATER_OR_EQUAL_TRANSLATED
<source1>,<source2>,<table>,=pc=

This instruction behaves like CHAR_COMPARE_GREATER_OR_EQUAL except that the sources are translated through "table" as they are compared.

*   CHAR_CLASS_ALPHABETIC <source>,=pc=

The condition is true if all the characters in the source are in the set of ASCII characters "A" thru "Z" and the blank. Note that lower case letters are not in the set:

*   CHAR_CLASS_NOT_ALPHABETIC <source>,=pc=

This instruction is the inverse of CHAR_CLASS_ALPHABETIC.

*   CHAR_CLASS_NUMERIC <source>,=pc=

The condition is true if all the characters in <source> contain the digits 0 through 9 and if the sign is valid. The sign is considered valid if its existence and position are as described by the name table type field. Note that the source must be checked 4 bits at a time if the name table type field for <source> indicates "packed".

*   CHAR_CLASS_NOT_NUMERIC <source>,=pc=

This instruction is the inverse of CHAR_CLASS_NUMERIC.

*   CHAR_SPACES <source>,=pc=

The condition is true if all the characters in the source are equal to ASCII blank.

*   CHAR_NOT_SPACES <source>,=pc=

This instruction is the inverse of CHAR_SPACES.

1.5.6 Arithmetics (24)

The type of the source(s) and destination(s) for an arithmetic instruction is specified by the first part of the instruction's name. The operation to be performed is specified by the second part of the instruction's name.

If no preposition follows the first two parts of the instruction's name, then the operation uses two sources distinct from the destination(s). If a preposition does follow the instruction's name, then the second operand is specified by <destination> or <quotient>.

For SUBTRACT instructions, the first source is the subtrahend and the second source (which might also be the destination) is the minuend.

For DIVIDE instructions the first source is the divisor and the second source (which might also be the quotient) is the dividend. Note that DIVIDE instructions produce two results, a quotient and a remainder.

*   DECIMAL_ADD <source1>,<source2>,<destination>

*   DECIMAL_SUBTRACT <source1>,<source2>,<destination>

*   DECIMAL_MULTIPLY <source1>,<source2>,<destination>

*   DECIMAL_DIVIDE <source1>,<source2>,<quotient>,<remainder>

*   DECIMAL_ADD_TO <source>,<destination>

*   DECIMAL_SUBTRACT_FROM <source>,<destination>

*   DECIMAL_MULTIPLY_BY <source>,<destination>

```
*     DECIMAL_DIVIDE_INTO <source>,<quotient>,<remainder>

*     BINARY_ADD <source1>,<source2>,<destination>

*     BINARY_SUBTRACT <source1>,<source2>,<destination>

*     BINARY_MULTIPLY <source1>,<source2>,<destination>

*     BINARY_DIVIDE <source1>,<source2>,<quotient>,<remainder>

*     BINARY_ADD_TO <source>,<destination>

*     BINARY_SUBTRACT_FROM <source>,<destination>

*     BINARY_MULTIPLY_BY <source>,<destination>

*     BINARY_DIVIDE_INTO <source>,<quotient>,<remainder>

*     FLOATING_ADD <source1>,<source2>,<destination>

*     FLOATING_SUBTRACT <source1>,<source2>,<destination>

*     FLOATING_MULTIPLY <source1>,<source2>,<destination>
*     FLOATING_DIVIDE <source1>,<source2>,<quotient>,<remainder>


*     FLOATING_ADD_TO <source>,<destination>


*     FLOATING_SUBTRACT_FROM <source>,<destination>


*     FLOATING_MULTIPLY_BY <source>,<destination>


*     FLOATING_DIVIDE_INTO <source>,<quotient>,<remainder>
```

1.5.7 Size Error Handling (5)

Many arithmetic and conversion operations can produce results which are too long to fit in the specified destination (COBOL lumps division by zero, overflow and underflow into this category). Explicit size error handling, when specified by the programmer, mandates that the destination field remain unchanged when a size error occurs.

The COBOL s-language therefore has two 1-bit size error flags which are part of process data space. The first is called "OP_BAD". OP_BAD is set to one by any operation which causes a size error. All operations except the conditional moves defined below are allowed to change the destination, even when a size error occurs. For those instances where a size error is specified, the compiler will allocate a temporary as the destination, generating a checked assignment after the operation. The second size error flag is called "STMT_BAD". It is set to one if any operation in a statement incurs a size error. OP_BAD is always designated by name 0. STMT_BAD is always designated by name 1.

* BEGIN_CHECKED_STATEMENT

This operation sets OP_BAD and STMT_BAD to zero.

* CHECKED_ASSIGN_DECIMAL_TO_DECIMAL <source>,<destination>

The action of this operation depends on the setting of OP_BAD:

OP_BAD = 1. STMT_BAD is set to 1. OP_BAD is set to 0.

OP_BAD = 0. If the source fits in the destination, then the source is assigned to the destination, as in the unchecked ASSIGN_DECIMAL_TO_DECIMAL. If the source does not fit in the destination, STMT_BAD is set to 1. Note that the condition "fits in the destination" may depend on the number of significant decimal digits in the source, and not just its length!

* CHECKED_ASSIGN_BINARY_TO_BINARY <source>,<limit>,<destination>

The action of this operation depends on the setting of OP_BAD:

OP_BAD = 1. STMT_BAD is set to 1. OP_BAD is set to 0.

OP_BAD = 0. If the source fits in the destination, then the source is assigned to the destination, as in the unchecked ASSIGN_BINARY_TO_BINARY. If the source does not fit in the destination, STMT_BAD is set to 1. The condition "fits in the destination" is true if the magnitude of "source" does not exceed "limit".

* CHECKED_ASSIGN_FLOATING_TO_FLOATING <source>,<destination>

The action of this operation depends on the setting of OP_BAD:

OP_BAD = 1. STMT_BAD is set to 1. OP_BAD is set to 0.

OP_BAD = 0. An ASSIGN_FLOATING_TO_FLOATING is executed.

* IF_NO_SIZE_ERROR =pc=

If STMT_BAD is 0, a branch to the pc offset is taken.

1.5.8 String Instructions (3)

The following instructions support the COBOL STRING, UNSTRING and INSPECT statements. In all these instructions, <source> and <destination> name character (i.e. length is a multiple of eight) data. The data named by <start>,<end> and <index> are binary integers which "index" either the source or the destination. The value one designates the first character position, two the second, etc. (i.e. the strings are entity indexed).

* MOVE_STRING <source>,<start>,<end>,<destination>,=pc=

The characters in "source" from "start" through "end" are moved to "destination". If the number of characters to be moved exceeds the space available in "destination", truncation takes place, and a branch to the pc offset is executed.

* MOVE_TO_STRING <source>,<destination>,<start>,<end>,=pc=

All the characters in "source" are moved to that portion of "destination" bounded by "start" through "end". If the number of characters to be moved exceeds the space in the delimited destination, truncation takes place and a branch to the pc offset is executed.

* SCAN_STRING <source>,<start>,<end>,<index>,<delimiter>,=pc=

The characters in "source" from "start" through "end" are scanned for character(s) in "delimiter". If the delimiter is not found, "index" is set to the "end", plus 1, and a branch to the pc offset is executed. If a delimiter is found, "index" is set to the character position preceeding the start of the delimiter. Note if the delimiter contains more than one character, all the characters must occur in sequence in "source".

1.5.9 Miscellaneous (6)

* GO_TO <pc offset>

This instruction causes replacement of the offset portion of the program counter by "pc offset".

* GO_TO_OFFSET =pc=

This instruction causes a branch to be taken to the pc offset.

The following two instructions are generated as a complimentary pair. The "ending address variable" is the same in both instructions.

* PERFORM_THRU <ending address variable>,=pc=

The address of the instruction following the PERFORM_THRU is assigned to the ending address variable, then a branch is executed to the pc offset.

* END_PERFORM <ending address variable>

The current value of the ending address variable is fetched. The address of the instruction following the END_PERFORM is stored in the ending address variable. A branch to the fetched ending address completes the execution of this instruction. The compiler must ensure that the initial value of the ending address variable is the address of the instruction following the END_PERFORM.

* CALL <procedure>

This instruction invokes the common external call mechanism, with no parameters.

* CALL_USING =n=,<procedure>,<parameter-1> ... <parameter-n>

This instruction invokes the common external call mechanism, with "n" parameters.

Appendix- SPL Description of EDIT_NUMERIC

```
procedure edit_numeric (read only  s: numeric,
                        write only t: alphanumeric,
                        read only  p: picture);


type picture is array [picture_counter] of byte;
     picture_counter is 0..35;

     alphanumeric is array [target_counter] of byte;
     target_counter is 1..30;

     numeric is any of the seven decimal dwarves;
     source_counter is 1..16;

     byte is bit 8 as 0..255;


variable finished:       boolean;
         negative:       boolean;
         triggered:      boolean;

         digit_substitute:      byte;
         negative_insert:       byte;
         positive_insert:       byte;

         si:     source_counter;
         ti:     target_counter;
         pi:     picture_counter;
```

```
function derive_opcode (read only pixel: byte)

                  returns  edit_opcode;


type edit_opcode is (move_numeric,
                     move_numeric_suppressed,
                     move_numeric_floating,
                     insert_character,
                     insert_character_suppressed,
                     insert_sign,
                     end_floating,
                     blank_when_zero,
                     end_edit,
                     set_suppression,
                     set_floating_inserts);


begin

select pixel<0:4> of

   case 1:      return move_numeric;
   case 2..7:   return insert_character;
   case 8:      return move_numeric_suppressed;
   case 9:      return move_numeric_floating;
   case 10..15: return insert_character_suppressed;

   case 0:      select pixel<4:4> of

                   case 0: return set_suppression;
                   case 1: return set_floating_inserts;
                   case 2: return end_floating;
                   case 3: return insert_sign;
                   case 4: return blank_when_zero;
                   case 5: return end_edit;

                   otherwise: nothing;

                end select;

end select;


end function derive_opcode;
```

```
procedure exec_pixel (read only  pixel:             byte)

            inherits (read write triggered:         boolean,
                      read only  negative:          boolean,
                      write only finished:          boolean,
                      read write digit_substitute:  byte,
                      read write negative_insert:   byte,
                      read write positive_insert:   byte);


variable cc:     byte;            % working character
         sc:     byte;            % scratch character
         z:      boolean;         % is source digit zero;


begin


select derive_opcode(pixel) of


   case move_numeric:

        times pixel<4:4> repeat

           get_digit_from_source (cc, z);
           triggered := triggered or not z;
           put_character_into_target (cc);

        end times;



   case move_numeric_suppressed:

        times pixel<4:4> repeat

           get_digit_from_source (cc, z);
           if z and not triggered
              then cc := digit_substitute;
           end if;
           triggered := triggered or not z;
           put_character_into_target (cc);

        end times;
```

```
case move_numeric_floating:

      times pixel<4:4> repeat

         get_digit_from_source (cc, z);
         if not triggered
            then if z then cc := digit_substitute;
                      else if negative
                              then sc := negative_insert;
                              else sc := positive_insert;
                           end if;
                           put_character_into_target (sc);
                 end if;
         end if;
         triggered := triggered or not z;
         put_character_into_target (cc);

      end times;


case insert_char:

      cc := pixel<1:7>;
      put_character_into_target (cc);


case insert_char_suppressed:

      if triggered
         then cc := pixel<1:7>;
         else cc := digit_substitute;
      end if;
      put_character_into_target (cc);


case insert_sign:

      get_next_pixel (sc);
      get_next_pixel (cc);
      if negative then cc := sc; end if;
      put_character_into_target (cc);
```

```
    case end_drifting:

         if triggered
            then nothing;
            else if negative
                    then cc := negative_insert;
                    else cc := positive_insert;
                 end if;
                 put_character_into_target (cc);
         end if;


    case blank_when_zero:

         if not triggered then blank_target; end if;
         finished := true;


    case end_edit:

         finished := true;


    case set_suppression:

         get_next_pixel (digit_substitute),


    case set_floating_insertions:

         get_next_pixel (negative_insertion);
         get_next_pixel (positive_insertion);

end select;

return;

end procedure exec_pixel;
```

```
begin               %-----------------------%
                    %                       %
                    %   MAIN LINE EDIT CODE %
                    %                       %
                    %-----------------------%

      finished := false;
      negative := is_negative(s);
      triggered := false;

      digit_substitute := ASCII_space;
      negative_insertion := ASCII_minus;
      positive_insertion := ASCII_space;

      si := 1;
      ti := 1;
      pi := 0;

      get_next_pixel (target_digits);

      repeat

         get_next_pixel (cc);
         exec_pixel (cc);
         if finished then return; end if;

      end repeat;

end procedure edit_numeric;
```