Using the New Common Lisp Pretty Printer

Richard C. Waters

MIT AI Laboratory
545 Technology Sq.
Cambridge MA 02139
Dick@AI.MIT.EDU

Mitsubishi Electric Research Laboratories
201 Broadway
Cambridge MA 02139
Dick@MERL.COM

Although not part of the initial definition of the language, pretty printing has been an important feature of Lisp programming environments for twenty years or more [1]. By the time Common Lisp was being defined, the importance of pretty printing was clear enough that pretty printing was made a formal part of the language [2]. However, little was done beyond recognizing the least common denominator of the pretty printing facilities available at the time—[2] specifies how pretty printing can be turned on and off, but says very little else. In particular, no provision was made for allowing the user to control what the pretty printer does.

Since the late 1970s, efficient pretty printers that allow extensive user control over the format of the output produced have been a particular interest of mine. In 1989, my XP pretty printer [4] was adopted as part of the proposed Common Lisp standard [5]. This adds a number of very useful facilities to Common Lisp, however, some study and experimentation on the part of the user is required to make the best use of these facilities.

The purpose of this short paper is to show how the user format-control facilities provided by the new Common Lisp pretty printer can be used to advantage. It is intended as an extension to the documentation in [5], rather than a replacement for it. To make the best use of this article, it is advisable to read [5] and obtain a copy of the new Common Lisp pretty printer as outlined at the end of this article, so that you can play with the examples.

Printing Lisp as Pascal

As a convenient context for discussing the format-control facilities provided by the new Common Lisp pretty printer, this article uses the problem of displaying Lisp code using Pascal syntax. In particular, the article shows how the pretty printer can be used to print a simple mathematical subset of Lisp as Pascal. For example, the following Lisp function definition

```
(defun sqt (n &aux sqt)
  (declare (float n) (float sqt))
  (setq sqt 1.0)
  (loop (when (< (abs (- (* sqt sqt) n))
                 1.0E-4)
          (return nil))
        (setq sqt
              (/ (+ sqt (/ n sqt)) 2.0)))
  sqt)
```

is printed as shown below.

```
function Sqt (N: Real): Real;
begin
  Sqt := 1.0;
  while not (Abs(Sqt*Sqt-N) < 1.0E-4) do
    Sqt := (Sqt+N/Sqt)/2.0
end
```

The Lisp-as-Pascal printing system is best viewed as a means for presenting the pretty printing facilities, rather than any kind of serious attempt at program translation. However, it is worthy of note that the system is not totally contrived. A similar system was used as part of the Knowledge-Based Editor in Emacs [3] to display a Lisp-like internal representation as Ada code.

Figures 1–4 contain definitions that cause

```
(in-package "USER")

(defvar *PD* (copy-pprint-dispatch))
(proclaim '(special *B*))

(defun pascal-write (sexpr &rest args)
  (let ((*B* 0))
    (apply #'write sexpr :pretty t
           :pprint-dispatch *PD* args)))

(defun pr-string (s string)
  (setq string (string string))
  (write-char #\' s)
  (dotimes (i (length string))
    (let ((char (aref string i)))
      (write-char char s)
      (when (char= char #\')
        (write-char #\' s))))
  (write-char #\' s))
```

Figure 1: Code for printing atoms.

```
(set-pprint-dispatch 'string
  #'pr-string 0 *PD*)

(set-pprint-dispatch 'character
  #'pr-string 0 *PD*)

(set-pprint-dispatch 'symbol
  #'(lambda (s id)
      (write-string
        (remove #\-
          (string-capitalize
           (string id)))
        s))
  0 *PD*)

(set-pprint-dispatch
  '(and rational (not integer))
  #'(lambda (s n)
      (write (float n) :stream s))
  0 *PD*)
```

the pretty printer to print Lisp as Pascal. The pretty printer operates under the control of a dispatch table that specifies how various kinds of objects should be printed. The second form in Figure 1 defines a variable `*PD*` and initializes it with a copy of the default pretty printing dispatch table. The function `pascal-write` prints a Lisp expression as Pascal by triggering pretty printing and using the dispatch table `*PD*`. (As discussed in conjunction with Figure 2, the variable `*B*` is used to control the printing of parentheses in Pascal expressions.)

The function `pr-string` prints strings as required by Pascal.

```
"Bob's house"   prints as  'Bob''s house'
"say \"Hi\""    prints as  'say "Hi"'
```

The top two forms on the right of Figure 1 cause `pr-string` to be used for printing strings and characters. (The first line of `pr-string` is included so that `pr-string` can be used to print character objects in addition to strings.)

```
#\s                 prints as  's'
```

The third form on the right of Figure 1 specifies how variables and function names should be printed in Pascal. In particular, it specifies that whenever an object of type `symbol` is encountered, it should be pretty printed using the indicated function. This function capitalizes the first letter of each word and removes any hyphens.

```
first-num       prints as  FirstNum
break-level-2   prints as  BreakLevel2
```

(As with most of the code being presented here, this does not guarantee that every relevant Lisp construct will be translated into a valid Pascal construct. However, it is sufficient to translate Lisp constructs that are intended to be displayed as Pascal into valid Pascal.)

A key thing to notice is that the symbol printing function uses `write-string` to print the string it computes, rather than `princ`. The reason for this is that `princ` applies pretty printing dispatching to its argument while `write-string` does not. If the function `princ` were used in the symbol printing function, the symbol `first-num` would be printed as `'FirstNum'`, because the string created by the symbol printing function would be printed as specified by the pretty printing dispatch entry for strings.

The last form in Figure 1 specifies how to print rational numbers. Nothing special has to be said about integers and floating point numbers, because the standard Lisp printer prints them in a way that is compatible with Pascal.

Printing Expressions

Figure 2 shows the pretty printing control code that specifies how expressions should be printed. The most interesting aspect of this code is the way it handles the printing of paren-

```
(defvar *unary*
  '((+ "+" 2) (- "-" 2) (not "not " 4)))

(defun unary-p (x)
  (and (consp x)
       (assoc (car x) *unary*)
       (= (length x) 2)))

(set-pprint-dispatch '(satisfies unary-p)
  #'(lambda (s list)
      (let* ((info (cdr (assoc (car list)
                               *unary*)))
             (nest (<= (cadr info) *B*))
             (*B* (cadr info)))
        (when nest (write-char #\( s))
        (write-string (car info) s)
        (write (cadr list) :stream s)
        (when nest (write-char #\) s))))
  0 *PD*)

(defvar *builtin*
  '((atan "ArcTan") (code-char "Chr")
    (log "Ln") (oddp "Odd")
    (char-code "Ord") (truncate "Trunc")
    (prin1 "Write") (terpri "Writeln")))

(defun builtin-p (x)
  (and (consp x)
       (assoc (car x) *builtin*)))

(defun pr-arglist (s args)
  (when args
    (let ((*B* 0))
      (format s #"~:<~@{~W~^, ~_~}~:>"
              args))))

(set-pprint-dispatch '(satisfies builtin-p)
  #'(lambda (s list)
      (write-string
        (cadr (assoc (car list)
                     *builtin*))
        s)
      (pr-arglist s (cdr list)))
  0 *PD*)
```

```
(defvar *bin*
  '((* "*" 3) (/ "/" 3)
    (mod " mod " 3)
    (round " div " 3)
    (and " and " 3)
    (+ "+" 2) (- "-" 2)
    (or " or " 2)
    (= " = " 1)
    (< " < " 1) (> " > " 1)
    (/= " <> " 1) (<= " <= " 1)
    (>= " >= " 1) (eq " = " 1)
    (eql " = " 1) (equal " = " 1)))

(defun bin-p (x)
  (and (consp x)
       (assoc (car x) *bin*)
       (= (length x) 3)))

(set-pprint-dispatch
  '(satisfies bin-p)
  #'(lambda (s list)
      (let* ((info (cdr (assoc (car list)
                               *bin*)))
             (nest (<= (cadr info) *B*)))
        (pprint-logical-block
            (s (cdr list)
               :prefix (if nest "(" "")
               :suffix (if nest ")" ""))
          (let ((*B* (1- (cadr info))))
            (write (pprint-pop)
                   :stream s))
          (pprint-newline :linear s)
          (write-string (car info) s)
          (let ((*B* (cadr info)))
            (write (pprint-pop)
                   :stream s))))))
    0 *PD*)

(set-pprint-dispatch 'cons
  #'(lambda (s list)
      (write (car list) :stream s)
      (pr-arglist s (cdr list)))
  -1 *PD*)
```

Figure 2: Code for printing expressions.

theses. At each moment, the variable *B* contains the binding strength of the current context on a scale of 0 (weakest) to 4 (strongest). Unless the binding strength of an operator is stronger than the surrounding context, parentheses are printed to specify the proper nesting of expressions.

Consider the top left of Figure 2. The variable *unary* contains information about the relationship between unary operators in Lisp and Pascal. Each triple contains a Lisp function, the corresponding Pascal operator, and the binding strength of the operator in Pascal. The function unary-p tests whether something is a list that is an application of a unary operator.

The printing function for unary operators determines whether the expression should be nested in parentheses by comparing the binding strength of the operator with the binding strength of the surrounding context; changes the value of *B* to reflect the binding strength of the operator; prints parentheses if needed; prints the appropriate Pascal operator; and calls write to print the argument appropriately.

The top three forms on the right of Fig-

ure 2 specify how binary operators should be handled. This is done in a way that is closely analogous to the handling of unary operators, however, two additional complexities have to be handled by the binary printing function.

The left associative nature of Pascal must be taken into account when deciding where to place parentheses. This is done by reducing the binding strength of the context when printing the first argument of a binary operation.

```
(* (+ 1 2) 3)      prints as  (1+2)*3
(* (* 1 2) 3)      prints as  1*2*3
(* 1 (* 2 3))      prints as  1*(2*3)
```

To allow the pretty printer to adjust the output based on the line width available, the printing function for binary operators creates a logical block and introduces a conditional newline. As discussed at length in [4, 5], logical blocks are a central feature of the pretty printing algorithm. Each logical block is printed on a single line if possible. However, if this is not possible, a block is broken across multiple lines as specified by the conditional newlines within it and appropriate indentation is inserted. For example, the Lisp expression

```
(> threshold (+ new-val delta))
```

prints as follows if the line width is sufficient.

```
Threshold > NewVal+Delta
```

If somewhat less space is available it prints as:

```
Threshold
 > NewVal+Delta
```

If even less space is available it prints as:

```
Threshold
 > NewVal
   +Delta
```

Logical blocks and conditional newlines allow a single printing function to produce aesthetic output for a wide range of line widths.

The bottom four forms on the left of Figure 2 support the printing of built-in functions where the name used in Pascal is different from the Lisp name. The most interesting thing here is the function `pr-arglist`. This function prints out zero or more arguments of a Pascal function call. Note that nothing is printed if there are zero arguments and `*B*` is set to 0 reflecting the fact that the printing of the arguments does not have to be sensitive to the binding strength of the outer context.

The function `pr-arglist` and the printing function for binary operators each create a logical block and specify conditional newlines. However, they do so using different forms. The form `pprint-logical-block` is the most general form for creating a logical block. It must be used in situations where complex computation is required to determine what should be printed within the block. In simple situations (e.g., in `pr-arglist`) the format directive "~<...~:>" can be used instead. The directive "~_" is used to specify conditional newlines within a format string.

The last form on the right of Figure 2 supports the printing of user-defined functions and built-in functions where the name is the same in Lisp and Pascal (e.g., `cos` and `round`). Note that the dispatching entry is given a priority of -1 instead of 0 as in the other entries in Figure 2. A different priority is required because the type specifier associated with the entry (`cons`) is not disjoint from the other type specifiers in Figure 2. A lesser priority is used so that the entry will act as a catch-all that only applies in situations where no other entry applies.

As examples of the way function calls are printed, consider the following:

```
(terpri)        prints as  Writeln
(log x)         prints as  Ln(X)
(my-fn a b c)   prints as  MyFn(A, B, C)
```

The logical block introduced by `pr-arglist` causes the Lisp expression

```
(my-fn epsilon (+ end delta) total)
```

to print either as

```
MyFn(Epsilon, End+Delta, Total)
```

or

```
MyFn(Epsilon,
     End+Delta,
     Total)
```

depending on the space available.

```
(set-pprint-dispatch
  '(cons (member setq))
  #"~<~*~1@{~W :=~_ ~W~}~:>"
  0 *PD*)

(set-pprint-dispatch
  '(cons (member progn))
  #"~<~*~1@{begin ~2i~_~@{~W~^; ~_~} ~
    ~I~_end~}~:>"
  0 *PD*)

(defun pr-if (s list)
  (let ((then (caddr list))
        (else (cadddr list)))
    (when
      (and else (consp then)
           (or (member (car then)
                       '(when unless))
               (and (eq (car then) 'if)
                    (null (cdddr then)))))
      (setq then '(progn ,then)))
    (format s #"~@<if ~W ~i~:_~3I~
                  then ~_~W~@[ ~I~_~3I~
                  else ~_~W~]~:>"
            (cadr list) then else)))

(set-pprint-dispatch
  '(cons (member if))
  #'pr-if 0 *PD*)

(defun maybe-progn (list)
  (if (cdr list)
      '(progn ., list)
      (car list)))

(set-pprint-dispatch
  '(cons (member when))
  #'(lambda (s list)
      (pr-if s '(if ,(cadr list)
                    ,(maybe-progn
                       (cddr list)))))
  0 *PD*)

(set-pprint-dispatch
  '(cons (member unless))
  #'(lambda (s list)
      (pr-if s '(if (not ,(cadr list))
                    ,(maybe-progn
                       (cddr list)))))
  0 *PD*)
```

Figure 3: Code for printing simple statements.

Printing Statements

Figure 3 shows the pretty printing control code that specifies how simple statements are printed. The first two forms print assignment statements and `begin...end` blocks. They are specified very compactly by using the extended form of the cons type specifier included as part of the propsed Common Lisp standard and the reader macro `#"..."`, which creates a function corresponding to a format string. Both forms use logical blocks to control the output.

The function `pr-if` is used to print conditional statements. Some complexity is involved, because the function must ensure that nested conditionals print correctly. In particular, the expression

```
(if a (if b c) d)
```

must be printed as

```
if A then begin if B then C end else D
```

instead of

```
if A then if B then C else D
```

to distinguish it from

```
(if a (if b c d))
```

Checking for this problem requires `pr-if` to inspect the `then` clause of the conditional being printed.

The last three forms on the right of Figure 3 specify how to print the Lisp forms `when` and `unless`. This is done by converting them to equivalent `if`s.

The first five forms in Figure 4 specify how to print while and repeat loops. The primary complexity revolves around the need to inspect Lisp `loop` forms and determine what Pascal statements should be used to represent them. (The code shown assumes that every Lisp loop it encounters can be displayed as either a `while` or `repeat` loop in Pascal). An example of the way a `while` loop is printed is shown at the beginning of this paper. The following `repeat` loop

```
(loop (setq result (* result x))
      (setq count (- count 1))
      (when (= count 0) (return nil)))
```

is printed as shown below.

```
repeat
  Result := Result*X;
  Count := Count-1
until Count = 0
```

```
(defun while-loop-p (x)
  (and (consp x) (eq (car x) 'loop)
       (exit-p (cadr x))))

(defun exit-p (x)
  (and (consp x)
       (member (car x) '(if when))
       (equal (cddr x) '((return nil)))))

(set-pprint-dispatch
  '(satisfies while-loop-p)
  #'(lambda (s list)
      (format s "~@<while ~W ~
                 ~:_do ~2I~_~W~:>"
              '(not ,(cadadr list))
              (maybe-progn (cddr list))))
  0 *PD*)

(defun repeat-loop-p (x)
  (and (consp x) (eq (car x) 'loop)
       (exit-p (car (last x)))))

(set-pprint-dispatch
  '(satisfies repeat-loop-p)
  #'(lambda (s list)
      (format s "~@<~<repeat ~2I~
                 ~@{~^~_~W~^; ~}~:> ~I~_~
                 until ~W~:>"
              (butlast (cdr list))
              (cadar (last list))))
  0 *PD*)

(proclaim '(special *decls*))

(defun pr-decl (s var &rest ignore)
  (declare (ignore ignore))
  (format s #"~W: ~W"
          var (declared-type var)))

(defun declared-type (var)
  (cdr (assoc
         (dolist (d *decls* 'integer)
           (when (member var (cdr d))
             (return (car d))))
         '((float . real)
           (single-float . real)
           (integer . integer)
           (fixnum . integer)
           (character . char)
           (string-char . char)))))
```

```
(defun pr-defun (s list)
  (let* ((name (cadr list))
         (args (caddr list))
         (body (cdddr list))
         (*decls* nil)
         (fn? (and (member name args)
                   (eq name
                       (car (last body)))))
         (locals
           (delete name
             (cdr (member '&aux args))))
         (parameters
           (ldiff args
             (member '&aux args)))
         (*B* 0))
    (loop
      (unless (eq (caar body) 'declare)
        (return nil))
      (setq *decls*
            (append *decls* (cdar body)))
      (pop body))
    (pprint-logical-block (s (cdr list))
      (write-string
        (if fn? "function " "procedure ")
        s)
      (write name :stream s)
      (format s #" ~:<~@{~/pr-decl/~^~
                         ; ~:_~}~:>"
              parameters)
      (when fn?
        (format s #": ~W"
                (declared-type name)))
      (format s #";~:@_")
      (when locals
        (format s #"  var ~4I~
                   ~{~:@_~/pr-decl/;~}~
                   ~0I~:@_"
                locals))
      (format s #"begin ~2i~:@_~{~W~^; ~_~}~
                  ~I~_end"
              (if fn?
                  (butlast body)
                  body)))))

(set-pprint-dispatch
  '(cons (member defun))
  #'pr-defun 0 *PD*)
```

Figure 4: Code for printing loops and function definitions.

Printing Function Definitions

The remainder of Figure 4 controls the printing of function definitions. The primary complexity here is that parameters, local variables, and type declarations are specified in Pascal very differently from the way they are specified in Lisp. The function `pr-defun` effectively has to parse the `defun` to be printed and then re-express the information using either a `procedure` or `function` statement in Pascal.

An interesting thing to note is the function `pr-decl`. This function prints a variable followed by its type and is used as a user-defined format directive in `pr-defun`. The type to print is determined by the function `declared-type`, which looks at declaration information stored in the variable `*decl*` by `pr-defun`.

The results produced by `pr-defun` are illustrated by the first example in this paper and the following

```
(defun print-exp (x i &aux count result)
  (declare (integer x count result))
  (setq count i)
  (setq result 1)
  (loop (setq result (* result x))
        (setq count (- count 1))
        (when (= count 0) (return nil)))
  (print result))
```

which is printed as shown below.

```
procedure PrintExp (X: Integer;
                    I: Integer);
  var
    Count: Integer;
    Result: Integer;
begin
  Count := I;
  Result := 1;
  repeat
    Result := Result*X;
    Count := Count-1
  until Count = 0;
  Print(Result)
end
```

Conclusion

You can get a lot of value out of the proposed Common Lisp pretty printer by merely setting `*print-pretty*` to `t`. However, this only scratches the surface of the value that can be obtained. The next level of use comes from defining special pretty printing functions for particular data structures you define. This allows the pretty printer to be much more useful during debugging. However, the usefulness of the pretty printer is not limited to being part of the Lisp programming environment.

An entirely new level of use comes from using the pretty printer as a component of a system you are building, as in the example presented here. The pretty printer's ability to tailor output to fit the space available makes it valuable in a wide variety of situations where output is being produced. In particular, it allows a modular approach to the creation of output.

For example, in the code shown in Figures 1–4, each dispatching entry specifies how to print a single kind of form. Except for a small amount of contextual information (e.g., the information required to decide where parentheses should be printed) each entry operates on a local basis without having to know anything about any other form. However, because each entry specifies how the relevant form should be printed if it will fit on one line and what should be done if it cannot be printed on one line, the pretty printer is able to dynamically combine the entries and automatically adjust the output to fit aesthetically in a wide range of line widths.

Obtaining the Example

The example above is written in Common Lisp and has been tested in several different Common Lisp implementations. The full source is shown in Figures 1–4. In addition, the source can be obtained over the INTERNET by using FTP. Connect to `FTP.AI.MIT.EDU` (INTERNET number 128.52.32.6). Login as "anonymous" and copy the files shown below.

In the directory `/pub/lptrs/`

`xpx-code.lisp`	source code
`xpx-test.lisp`	test suite

Since the example makes use of the pretty printing facilities in the proposed Common Lisp standard, it requires a Common Lisp implementation that supports these facilities. If the Common Lisp implementation you use does not yet support these facilities, you can obtain an implementation over the INTERNET. Connect to `FTP.AI.MIT.EDU`, login as "anonymous", and copy the files shown below.

In the directory `/pub/xp/`

`xp.lisp`	source code
`xp-test.lisp`	test suite
`xp-doc.lisp`	brief documentation

of MIT and/or the author are not used in advertising or publicity pertaining to distribution of the software without specific, written prior permission. MIT and the author make no representations about the suitability of this software for any purpose. It is provided "as is" without express or implied warranty.

MIT and the author disclaim all warranties with regard to this software, including all implied warranties of merchantability and fitness. In no event shall MIT or the author be liable for any special, indirect or consequential damages or any damages whatsoever resulting from loss of use, data or profits, whether in an action of contract, negligence or other tortious action, arising out of or in connection with the use or performance of this software.

References

[1] Goldstein I., "Pretty Printing, Converting List to Linear Structure", MIT/AIM-279, February 1973.

[2] Steele G.L.Jr., *Common Lisp: the Language*, Digital Press, Maynard MA, 1984.

[3] R.C. Waters, "The Programmer's Apprentice: A Session With KBEmacs", *IEEE Transactions on Software Engineering*, 11(11):1296–1320, November 1985.

[4] Waters R.C., *XP: A Common Lisp Pretty Printing System*, MIT AI Laboratory technical memo MIT/AIM-1102a, September 1989.

[5] Waters R.C., "Pretty Printing," in *Common Lisp: the Language*, Second Edition, 748–769, Steele G.L.Jr., Digital Press, Burlington MA, 1990.

From: bclement@cavebbs.gen.nz (Bruce Clement) 17 Jan 92

Nondisclosure agreements limit the amount of precice information I can give, but object oriented CoBOL will have the following enhancements on procedural CoBOL:

1. There will be 3 new DIVISIONS:
   *The CLASS Division where object classes will be declared
   *The MESSAGE Division where all messages which may be used are declared
   *The METHOD Division which associates MESSAGES with CLASSES

2. All existing verbs are deleted from the language. Gone are ADD SUBTRACT COMPUTE (Which should never have been there in the first place, it makes CoBOL too much like ForTran) READ WRITE OPEN CLOSE. There is only one remaining verb SEND which sends messages to objects.

3. Programmers must be careful to avoid using any of the reserved MESSAGE names: ADD SUBTRACT MULTIPLY DIVIDE READ WRITE OPEN CLOSE etc. These are only permitted with built-in object types.

4. The data division has OD declarations to define Object storage.

The following is a brief example of an ADD ONE TO COBOL program:

```
   IDENTIFICATION DIVISION.
       PROGRAM ID. SAMPLE.
       AUTHOR.     BRUCE.
       SOURCE COMPUTER. INTEL-386-REAL.
       OBJECT COMPUTER. INTEL-386-PROTECTED.
       REMARKS.    (C) 1992, Diversity enhancements.
   CLASS DIVISION.
       DEFINE CLASS NUMBER.
       DEFINE CLASS INTEGER EXPANDS NUMBER.
   MESSAGE DIVISION.
       VIRTUAL MESSAGE SORT-OF-ADD APPLIES TO NUMBER
         INTEGER.
       VIRTUAL MESSAGE KIND-OF-PRINT APPLIES TO NUMBER
         INTEGER.

       ENVIRONMENT DIVISION.

       DATA DIVISION.
       OBJECT SECTION.
       OD  NUMBER.
      *    An empty definition, one byte minimum.
       01  FILLER PICTURE X.

       OD  INTEGER.
       01  I-VAL  PICTURE S9(9) COMP.

       WORKING-STORAGE SECTION.
       77  A-NUMBER   CLASS INTEGER.

       LINKAGE SECTION.
       77  VALUE-IN   PIC S9(9) COMP.

       METHOD DIVISION.
       MD CLASS NUMBER.
       SORT-OF-ADD MESSAGE USING VALUE-IN.
          EXIT METHOD.
       KIND-OF-PRINT MESSAGE.
          SEND DISPLAY "Number: " TO SYS-PRINT.
          EXIT METHOD.

       MD CLASS NUMBER.
       SORT-OF-ADD MESSAGE USING VALUE-IN.
          SEND ADD VALUE-IN TO I-VAL.
          EXIT METHOD.

       KIND-OF-PRINT MESSAGE.
      *  Note the power of inheritence of parent methods
          SEND KIND-OF-PRINT OF NUMBER TO SELF.
          SEND DISPLAY I-VAL TO SYS-PRINT.
          EXIT METHOD.

       PROCEDURE DIVISION.
       ONLY SECTION.
           SEND MOVE 1 TO I-VAL OF A-NUMBER.
           SEND SORT-OF-ADD 1 TO A-NUMBER.
           SEND KIND-OF-PRINT TO A-NUMBER.
           STOP RUN.
```

I hope that this simple example of an ADD 1 TO COBOL GIVING COBOL program suffices to show something of the power of the language, and demonstrates the true utility of a modern Business orientated object orienated language.