ACM SIGSOFT SOFTWARE ENGINEERING NOTES Vol 9 No 5 Oct 1984 Page 28

PROGRAMMING WITH small BLOCKS

Mark K Joseph

System Development Corporation
A Burroughs Company
Santa Monica, CA.

ABSTRACT

Programming style is a highly personalized characteristic of programs. Modular and block structured programming techniques provide many standards for good programming. These techniques are used to develop well designed but only marginally readable code. Similar organizational techniques are not typically applied to code inside subroutines, which is either too small or too specific to be further broken down into subroutines. The result is often obscure subroutine code that makes the maintenance programmer's job extremely difficult. Presented here is a styling technique for intra-subroutine code that groups program statements into "small BLOCKS" of function or conditional constraints. It is shown that this style of formating subroutine code can greatly improve the readability of the average program. The technique is demonstrated in Pascal, C, and Lisp.

Key Terms and Phrases: Software Engineering, program readability, and programming style.

1. Introduction

Structured programming, modular programming, and data encapsulation [3, 4, 5] are the current software engineering techniques used in developing software. If used properly these techniques can lead to well designed systems. Such software systems will be understandable at the module and subroutine level, because these techniques use the logical structure of the problem to provide a clear design.

However, these techniques do not guarantee readability of code inside a subroutine. Confusing subroutine code is due partly to the fact that programming style is very personalized, and that at this level no good formating guidelines exist. Structured programming does deal with code inside subroutines, however, it does not provide enough guidelines to make a whole subroutine readable.

A clear understanding of subroutine internals is needed in order to be able to modify subroutine code. Thus it is desirable to have a technique to make subroutine code more readable by ensuring that its format convey its logical structure. This paper presents a formating technique, called small BLOCKS, that solves this problem.

Many of the foundations and justifications of small BLOCKS can be found in [6, p.177-191]. However, small BLOCKS goes much further by enlarging the scope of what small

segments of code to separate and on how to do the actual separation. It does this by clearly defining the idea of separating function and control flow, as well as pointing out aesthetic in addition to logical statement groupings. The last major difference is that small BLOCKS frequently separates the code of a "program unit", i.e. a construct of a programming language such as IF-THEN-ELSE, where in [6] this is not done. The small BLOCKS technique follows the suggestion given in [6, p.182]: "A program is well-presented if its structure is clearly and quickly apparent to the reader".

The rest of this paper consists of four sections. Section two defines the small BLOCKS concept. Section three describes how this technique increases the readability of programs. Section four explains how small BLOCKS can be used as a development tool. Lastly in section five, several examples of code before and after the use of small BLOCKS are presented. It is hoped that these examples will solve many of the problems that the reader has encountered in trying to make his/her programs more readable.

2. What are small BLOCKS?

Small BLOCKS are used in subroutines, that are approximately 30 to 150 lines long. This range provides the degree of detail in which the internal workings of a subroutine can become confusing. These lines can contain subroutine calls, but the remainder are either too small or too specific to be grouped into subroutines themselves. Further, it should be noted that some applications cannot afford the overhead of many subroutine calls. This may lead to the use of macros, but can still result in complex code. The small BLOCK technique can be applied to many programming languages.

A small BLOCK is defined to be a grouping of program statements, which have some logical action or aesthetic quality associated with them. "Aesthetic" is used here to mean that grouping certain statements together will increase the readability of the code from the concerned author's point of view. These statement groupings are made into small BLOCKS by using several blank lines both before and after, a beginning comment describing its function, and indentation when needed. This is shown in Figure 5.2. The grouped statements are separated from the surrounding code, so that upon looking at the whole subroutine at once the viewer can see several separate block-like structures of code.

The size of a small BLOCK is not a fixed value. It depends upon the amount of complexity that it contains. Very simple but long actions can be a single small BLOCK, yet a few program statements may need to be separated in order to be comprehensible. If a small BLOCK is not easily understandable, then it is probably too long. In general, the programmer's judgement will need to be used here.

There are two forms of logical action which make natural small BLOCK groupings. The first is a sequence of simple program statements which together accomplish one logical function. The other is a nesting or clustering of control constructs such as IF-THEN-ELSE, WHILE loops, and FOR iteration loops. Small BLOCKS may nest, and thus these two forms can be contained within each other. These two forms introduce the idea of separating control flow and function by placing each into different small BLOCKS. Control flow small BLOCKS deal with the control of execution between small BLOCKS not individual language statements. Such organization allows the program reader to concentrate on the control flow and functions separately as the program is read. It is the author's opinion that this will increase the comprehension of programs.

```
{ Determine if action A is to be performed          }
{ This small BLOCK is a nesting of IF-THEN-ELSEs }
IF condition C1
   THEN some action
        IF condition C2                    <CONTROL>
           THEN some action
                IF condition C3
                   THEN { Perform action A: }
```

```
{ Action A is ~~~~~~~~~~~~~~~~ }
BEGIN
   IF condition C4
      THEN ~~~~~~~~          <FUNCTION>
      ELSE ~~~~~~~~
   some actions
END
```

```
{ What to do if action A is not done }
                  ELSE ~~~~~~~~~
            ELSE ~~~~~~~~~          <FUNCTION>
   ELSE ~~~~~~~~~~
```

Figure 3.1-A CODE TEMPLATE I Control Flow VS. Function
Each box represents a small BLOCK. The "<control>", and "<function>" symbols indicate the action of a small BLOCK. The boxes are used here for emphasis.

Before showing some concrete code examples, several code templates which exhibit the concepts in the above paragraphs are shown in Figures 3.1A-C. These represent real code examples extracted from some of the author's past programming projects. The templates are presented in a Pascal-like pseudo-language.

3. Increases Readability

One typical impediment to readability of a piece of code is complexity. Small BLOCKS can greatly improve the readability of the average program by breaking up the details of a subroutine into small understandable units. The reader can then examine the code at several levels of abstraction.

The reader first examines what each small BLOCK does by its heading comment, then how each block interacts with the others, and lastly how each works at the statment level. This is very similar to how modular programming makes software systems readable at the module level. First, each module's function is determined, then how each work together by their interfaces, and finally how each is organized internally.

Look at code template I, in Figure 3.1-A, for an example of how this works. First, a program reader looks at the heading comments of each small BLOCK. Next, by concentrating on the first small BLOCK the program reader learns under what conditions action A is to be performed. The

```
{ Are initial conditions met?                                    }
{ This small BLOCK is a clustering of IF-THEN-ELSEs }
  IF conditional C1
     THEN RETURN error              <CONTROL>
  IF conditional C2
     THEN RETURN error
```

```
{ Perform the first sequential action }
  IF conditional C3
     THEN action A1                 <FUNCTION>
     ELSE action A2
```

```
{ Swap A and B }
  temp <- A
  A    <- B
  B    <- temp                     <FUNCTION>
```

Figure 3.1-B CODE TEMPLATE II
A sequence of small BLOCKS, where each perform an individual logical action.

reader is not bothered with how action A works, but instead is just aware of its basic function from its heading comment.

Next, the program reader deciphers the second small BLOCK, which is action A itself. While determining how this does the stated function he no longer has to worry about all the details of the previous small BLOCK. The program reader knows how it works and will abstract away the details as he turns his attention toward action A. However, the reader will occasionally need to refer back to previous small BLOCKS in order to check on important interrelationships, such as a common variable being assigned the proper value.

By trying to understand the internals of only one small BLOCK at a time the reader has limited the amount of detail that it is necessary to deal with at any one time. It is the separation of the small BLOCKS of code which allows a reader to concentrate on a function, for example, without having distracting conditionals immediately around it. The conditionals are instead several lines above separated by enough blank lines to make it seem like an unrelated piece of code.

Finally, the reader gets to the last small BLOCK in Figure 3.1-A. Now, he ignores how action A is implemented, and determines what the subroutine does when action A is not performed.

4. Use small BLOCKS From The Beginning

Small BLOCKS can be added onto programs which are already in a target language (as is done in the next section). However, using them from the beginning of pseudo coding will result in readable and well organized pseudo-code. Later translation into real code will be much easier.

An exception is that aesthetic small BLOCKS are created both during and after pseudo or real coding.

```
{ The conditional takes several lines }
 IF complex conditional
   where C1 is TRUE
    THEN some small action                <CONTROL>
    ELSE BEGIN

      { search for the element in the list }
      FOR I <- L to K
          some action                     <FUNCTION>

      some action
      IF conditional C2 is TRUE
        BEGIN
          some action1                    <FUNCTION>
          some actionn
        END
END
```

Figure 3.1-C CODE TEMPLATE III
An IF-THEN-ELSE language construct can be broken up into several small BLOCKS in order to emphasize separate logical actions.

Aesthetic small BLOCKS are frequently added when a programmer is reviewing subroutine code and finds that it's still too complex with the current small BLOCKS. Also, the programmer might have missed a logical action small BLOCK and could add that on while reviewing it.

The translation of pseudo-code into a target language can be a complex task with many distracting details. To help simplify this task the programmer can code one small BLOCK of pseudo-code at a time. This will allow him to concentrate on only a few program statements, while ignoring all the remaining pseudo-code yet to do. The programmer will undoubtedly make frequent references to the target code that has already been produced. But now, making these inter-block references should be more manageable.

It is important to notice that more small BLOCKS will appear in the target language program than in the pseudo-code. This occurs because the translation process will expand one pseudo-code statement into possibly several target language statements. In this added detail more subfunctions and/or control structures will appear, thus warranting additional small BLOCKS.

Translating pseudo-code into a target language can also be a very tiring endeavor. Programmers are more prone to make mistakes as they get tired. However, stopping in the middle of coding a subroutine, in order to take a break, can cause a programmer to lose his train of thought. Small BLOCKS provide natural stopping places since they are organized around one basic function. Now, a programmer will not be in the middle of coding a complex action when getting up from his desk to take a break, but may still be in the middle of a subroutine.

5. A Few Examples

Presented here are four examples of the small BLOCK technique used in Pascal, C, and Lisp. The first example demonstrates the use of aesthetic small BLOCKS. The remaining three examples are presented in their reformated and original forms. This is done so the reader can make a comparison as to which form is preferred.

In preparing these examples, no variable names were changed and in most cases only the original comments were used. It was necessary to add a few comments since they were needed for any added small BLOCKS. Overall, the changes to the original code were kept to a minimum, in order to show only the effects of adding small BLOCKS.

In Figure 5.1 the first small BLOCK is an aesthetic grouping. In standard formating techniques the second "WHILE" statment would be connected with the rest of its loop. Here we instead want to emphasize the functions of the two small BLOCKS inside the loop. This example is just one of many varied situations in which aesthetic groupings can be used. This aspect of small BLOCKS is more of an art than a method.

If the separated "WHILE" statement contained a complicated conditional then it would basically be a variant of code template III, Figure 3.1-C. It would therefore, more strongly represent a separation of control flow and function.

Now looking at Figure 5.2, the first thing that strikes the reader is all that white space. Small BLOCKS spreads complexity out and gives the code some structure. The reader is not scared off when he first looks at Figure 5.2, but he is when he sees it in its original form, shown in Figure 5.3! At a glance to Figure 5.2, the reader sees two major components of "PROCEDURE Underflow". These are two small BLOCKS head by the comments: "{ b := page to the right of a }", and "{ b := page to the left of a }". Next, it is easily seen that each is composed of two internal small BLOCKS, and their functions are clear.

It is not an easy task to see this code's structure from its original formating style. In general, "the more white space the better" [9].

Turning attention to Figure 5.4, we notice that some comments start with an upper-case letter while others start with an arrow and a lower-case letter. This convention is used to help distinguish the heading comment of a small BLOCK from all of its internal subcomments. In some sense, the heading comment is similar to that of a procedure since it describes a high-level function. Whereas, the internal comments describe how that high-level function is accomplished. A minor point is that a blank line or two may appear inside a small BLOCK, as in the last small BLOCK in Figure 5.4. This can be helpful when a reader is focusing on statement-level actions.

An interesting observation can be made after a programmer develops a program using stepwise decomposition [5,6]. First, he breaks a problem into subproblems which tend to become small BLOCKS, when at the subroutine level. However, in the final code he tends to cram all the program statments together and the previously derived logical separation disappears. By using small BLOCKS from the beginning, the logical structure inside subroutines can be preserved, aided by the ideas of separation of control flow and function and aesthetic groupings.

It is interesting to study the reformated Lisp code in Figure 5.6. It is essentially the combination of small BLOCK groupings from code templates II & III, shown in Figure 3.1-B&C.

Function, Control Flow, and Aesthetic Groupings

```
* * *
{ Produce a cross reference for the input file }
WHILE NOT eof(f) DO
   BEGIN
      IF n = c4
        THEN n := 0;
      n := n + 1;
      WRITE(n:c3);
      WRITE(' ');
      WHILE NOT eoln(f) DO
            BEGIN



               { scan non-empty line }
               IF f^ IN ['A'..'Z'] THEN
                  BEGIN
                     k := 0;
                     REPEAT
                        IF k < c1
                          THEN BEGIN
                                  k    := k + 1;
                                  a[k] := f^;
                               END;
                        WRITE(f^); GET(f)
                     UNTIL NOT( f^ IN ['A'..'Z', '0'..'9'] );
                     * * *
                  END



               { check for quote or comment }
               ELSE BEGIN
                       IF f^ = '''' THEN
                                REPEAT
                                   WRITE(f^); GET(f)
                                UNTIL f^ = ''''
                       ELSE IF f^ = '{' THEN
                                REPEAT
                                   WRITE(f^); GET(f)
                                UNTIL f^ = '}';
                       WRITE(f^); GET(f)
                    END
            END
   END
```

FIG 5.1 [7, p.271]

small BLOCKS in Pascal

```
PROCEDURE Underflow( c,a: ref;  s: INTEGER;  VAR h: BOOLEAN );
   { a = Underflow page,  c = ancestor page }
   VAR b: ref;
       i,k,mb,mc: INTEGER;
   BEGIN  { h = TRUE,  a^.m = n-1 }
      mc := c^.m;



      { b := page to the right of a }
      IF s < mc
        THEN BEGIN
                  s  := s + 1;
                  b  := c^.e[s].p;
                  mb := b^.m;
                  k  := (mb - n + 1)  DIV  2;
                  { k = no. of items available on adjacent page b }
                  a^.e[n]   := c^.e[s];
                  a^.e[n].p := b^.p0;



                  { Move k items from b to a }
                  IF k > 0
                    THEN BEGIN
                            FOR i := 1  TO  k-1  DO
                                 a^.e[i+n] := b^.e[i];
                            c^.e[s]   := b^.e[k];
                            c^.e[s].p := b;
                            b^.p0     := b^.e[k].p;
                            mb        := mb - k;
                            FOR i := 1  TO  mb  DO
                                 b^.e[i] := b^.e[i+k];
                            b^.m := mb;
                            a^.m := n-1+k;
                            h    := FALSE;
                         END;



                  { Merge pages a and b }
                    ELSE BEGIN
                            FOR i := 1  TO  n  DO
                                 a^.e[i+n] := b^.e[i];
                            FOR i := s  TO  mc-1  DO
                                 c^.e[i] := c^.e[i+1];
                            a^.m := nn;
                            c^.m := mc-1;  {dispose(b)}
                         END
      END
```

```
    { b := page to the left of a }
    ELSE BEGIN
            IF s = 1
              THEN b := c^.p0
              ELSE  b := c^.e[s-1].p;
            mb := b^.m + 1;
            k   := (mb-n) DIV 2;



            { Move k items from page b to a }
            IF k > 0
              THEN BEGIN
                      FOR i := n-1 DOWNTO 1 DO
                           a^.e[i+k] := a^.e[i];
                      a^.e[k]   := c^.e[s];
                      a^.e[k].p := a^.p0;
                      mb        := mb-k;
                      FOR i := k-1 DOWNTO 1 DO
                           a^.e[i]  := b^.e[i+mb];
                      a^.p0     := b^.e[mb].p;
                      c^.e[s]   := b^.e[mb];
                      c^.e[s].p := a;
                      b^.m      := mb-1;
                      a^.m      := n-1+k;
                      h         := FALSE
                    END



            { Merge pages a and b }
              ELSE BEGIN
                      b^.e[mb]   := c^.e[s];
                      b^.e[mb].p := a^.p0;
                      FOR i := 1 TO n-1 DO
                           b^.e[i+mb] := a^.e[i];
                      b^.m  := nn;
                      c^.m  := mc-1;    {dispose(a)}
                    END
          END
END; { Underflow }
```

FIG 5.2 [7, p.254]

```
procedure underflow( c,a: ref;  s: integer;  var h: boolean );
    { a = Underflow page,  c = ancestor page }
    var b: ref; i,k,mb,mc: integer;
    BEGIN mc := c^.m;  { h = true,  a^.m = n-1 }
        IF s < mc THEN
        BEGIN { b := page to the right of a}  s := s + 1;
            b := c^.e[s].p; mb := b^.m; k := (mb-n+1) DIV 2;
            {k = no. of items available on adjacent page b}
            a^.e[n] := c^.e[s]; a^e[n].p := b^.p0;
            IF k > 0 THEN
            BEGIN {move k items from b to a}
                FOR i := 1 TO k-1 DO a^.e[i+n] := b^.e[i];
                c^.e[s] := b^.e[k]; c^.e[s].p := b;
                b^.p0 := b^.e[k].p; mb := mb - k;
                FOR i := 1 TO mb DO b^.e[i] := b^.e[i+k];
                b^.m := mb; a^.m := n-1+k; h := FALSE;
            END ELSE
            BEGIN {merge pages a and b}
                FOR i := 1 TO n DO a^.e[i+n] := b^.e[i];
                FOR i := s TO mc-1 DO c^.e[i] := c^.e[i+1];
                a^.m := nn; c^.m := mc-1;  {dispose(b)}
            END
        END ELSE
        BEGIN {b := page to the left of a}
            IF s = 1 THEN b := c^.p0 ELSE b := c^.e[s-1].p;
            mb := b^.m + 1; k := (mb-n) DIV 2;
            IF k > 0 THEN
            BEGIN {move k items from page b to a }
                FOR i := n-1 DOWNTO 1 DO a^.e[i+k] := a^.e[i];
                a^.e[k] := c^.e[s]; a^.e[k].p := a^.p0; mb := mb-k;
                FOR i := k-1 DOWNTO 1 DO a^.e[i] := b^.e[i+mb];
                a^.p0 := b^.e[mb].p;
                c^.e[s] := b^.e[mb]; c^.e[s].p := a;
                b^.m := mb-1; a^.m := n-1+k; h := FALSE
            END ELSE
            BEGIN {merge pages a and b}
                b^.e[mb] := c^.e[s]; b^.e[mb].p := a^.p0;
                FOR i := 1 TO n-1 DO b^.e[i+mb] := a^.e[i];
                b^.m := nn; c^.m := mc-1;  {dispose(a)}
            END
        END
END  { Underflow };
```

Figure 5.3 [7, p.254] The original formating style of Figure 5.2.

small BLOCKS in C

```
#include <stdio..h>

/* Allocate and fill input buffer */
_fillbuf( fp )
register FILE *fp;
{
     static  char  smallbuf[_NFILE];  /* for unbuffered I/O */
     char   *calloc();

     /* Is the file open for reading? */
     if  ((fp->_flag&_READ) == 0  ||  (fp->flag&(_EOF | _ERR)) != 0)
          return(EOF);



     /* Find a buffer to perform buffered I/O */
     while  (fp->_base  ==  NULL)
       /* -> perform unbuffered I/O  */
       if  (fp->_flag  &  _UNBUF)
                fp->_base  =  &smallbuf[fp->_fd];
       /* -> try to get a big buffer */
       else  if  ((fp->_base = calloc(_BUFSIZE, 1))  ==  NULL )
                fp->_flag   |= _UNBUF;
       /* -> got a big buffer       */
       else         fp->_flag   |= _BIGBUF;



     /* Fill in the buffer and set up the count and pointers */
     fp->_ptr  =  fp->_base;
     fp->_cnt  =  read(fp->_fd,  fp->_ptr,
                      fp->_flag  &  _UNBUF  ?  1  :  _BUFSIZE);

     /* -> hit eof while reading in the last buffer? */
     if  (--fp->_cnt  <  0) {
          if  (fp->_cnt  ==  -1)
                  fp->_flag  |=  _EOF;
          else     fp->_flag  |=  _ERR;
          fp->_cnt  =  0;
          return(EOF);
     }
     return(*fp->_ptr++  &  0377);  /* make char positive */
}
```

FIG 5.4 [2, p.168]

```
#include <stdio..h>

_fillbuf( fp )     /* allocate and fill input buffer */
register FILE *fp;
{
   static char smallbuf[_NFILE];  /* for unbuffered I/O */
   char *calloc();

   if ((fp->_flag&_READ) == 0 || (fp->flag&(_EOF | _ERR)) != 0)
        return(EOF);
   while (fp->_base == NULL) /* find buffer space */
        if (fp->_flag & _UNBUF) /* unbuffered */
             fp->_base = &smallbuf[fp->_fd];
        else if ((fp->_base = calloc(_BUFSIZE, 1)) == NULL )
             fp->_flag  |= _UNBUF; /* can't get big buf */
        else
             fp->_flag  |= _BIGBUF; /* got big one */
   fp->_ptr = fp->_base;
   fp->_cnt = read(fp->_fd, fp->_ptr,
                  fp->_flag & _UNBUF ? 1 : _BUFSIZE);
   if (--fp->_cnt < 0) {
        if (fp->_cnt == -1)
             fp->_flag |= _EOF;
        else
             fp->_flag |= _ERR;
        fp->_cnt = 0;
        return(EOF);
   }
   return(*fp->_ptr++ & 0377);  /* make char positive */
}
```

Figure 5.5 [2, p.168] The original formating style of Figure 5.4.

small BLOCKS in LISP

```
(DEFUN inf-to-pre (ae)
       (PROG  (OPERANDS OPERATORS)                ; The two lists
              ; Test beginning parameters
              (COND ((ATOM ae)  (RETURN ae)))
              (SETQ OPERATORS  (LIST 'dummy))



              ; Scan for operand, ends on operator
              stuff
              (COND  ((NULL as) (RETURN 'unexpected-end)))
              (SETQ  OPERANDS
                     (CONS (COND ((ATOM (CAR ae)) (CAR ae))
                                 (t (inf-to-pre (CAR ae))
                           )
                           OPERANDS)
                     ae (CDR ae)
              )



              ; Scan for operator
              scan
              (COND ((AND (NULL ae)
                          (EQUAL (CAR OPERATORS) 'dummy)     ; ae & list empty
                     ) (RETURN (CAR OPERANDS)))
              )



              ; End of ae or nesting order
              (COND ((OR (NULL ae)
                         (NOT (GREATERP (weight (CAR ae))
                                        (weight (CAR OPERATORS)))))


                        ; Construct code
                        (SETQ OPERANDS
                              (CONS (LIST (OPCODE (CAR OPERATORS))
                                          (CADR OPERANDS)
                                          (CAR  OPERANDS))
                                    (CDDR OPERANDS)          ; POP two operands
                              )
                              OPERATORS (CDR OPERATORS)      ; POP operator
                        )
                        ; Look for operator
                        (GO scan)
                     )
```

```
                ; Push operator, peel off operator
                (t (SETQ OPERATORS
                         (CONS (CAR ae) OPERATORS)
                         ae (CDR as))
                   (GO stuff)
                )
            )
))
```

FIG 5.6 [8, p.158]

```
(DEFUN inf-to-pre (ae)
       (PROG (OPERANDS OPERATORS)                              ;The two lists
             (COND ((ATOM ae) (RETURN ae)))                    ;Special case
             (SETQ OPERATORS (LIST 'dummy))                    ;Dummy terminator
             stuff
             (COND ((NULL as)                                  ;Scan for operand
                    (RETURN 'unexpected-end)))                 ;Ends on operator
             (SETQ OPERANDS (CONS (COND ((ATOM (CAR ae)) (CAR ae))
                                        (t (inf-to-pre (CAR ae)))      ;Recurse
                                  OPERANDS)                    ;PUSH operand
                   ae (CDR ae))                                ;Peel off operand
             scan
             (COND ((AND (NULL ae)                             ;Scan for operator
                         (EQUAL (CAR OPERATORS) 'dummy))       ;ae & list empty
                    (RETURN (CAR OPERANDS))))                  ;Return result
             (COND ((OR (NULL ae)                              ;End of ae or
                        (NOT (GREATERP (weight (CAR ae))       ;nesting order
                                       (weight (CAR OPERATORS)))))


                    (SETQ OPERANDS                             ;Construct code
                          (CONS (LIST (OPCODE (CAR OPERATORS))
                                      (CADR OPERANDS)
                                      (CAR  OPERANDS))
                                (CDDR OPERANDS))               ;POP two operands
                          OPERATORS (CDR OPERATORS))           ;POP operator
                    (GO scan))                                 ;Look for operator
                   (t (SETQ OPERATORS (CONS (CAR ae)
                                            OPERATORS)         ;PUSH operator
                            ae (CDR as))                       ;Peel off operator
                      (GO stuff)))))                           ;Look for operand
```

FIG 5.7 [8, p.158] The original formating style of Figure 5.6.

Lastly, there are no hard and fast rules in using small BLOCKS. Which small BLOCK organization is used depends upon what the code looks like and what the programmer wants to emphasize. This paper has presented the general principles of this formating technique.

6. Conclusion

Program readability is a requirement for good programming. The current tools used to accomplish this are meaningful comments and variable names, as well as structured and modular programming. However, none of these directly addresses the problem of complex straight-line subroutine code. This paper has presented a new formating technique which solves this problem.

Many existing formating techniques work at too low a level. They are concerned only with making a small sequence of language statements easier to read, for example reformating nested IF-THEN-ELSEs into a CASE-like statement. [1, p.146-148] Such techniques are necessary, but do not convey the statements' higher level function in the subroutine. Small BLOCKS bridges this gap by grouping statements by logical action, while also allowing aesthetic groupings of code. It has also been shown how small BLOCKS are useful in organizing code to make programming easier.

As with all formating techniques experience and experimentation is needed to perfect them. The reader should not expect small BLOCKS to be easy to use at first. However, after some practice the programmer will notice the difference in his code, and will get a feel for how to proceed.

Programmers need to take program formating more seriously, and not only be concerned with making their programs work. An unreadable program only works until a modification is required. Many programmers unknowingly use concepts from small BLOCKS already. They do so because they seem natural to use.

References

[1] B.W. Kernigham and P.J. Plauger, *The Elements of Programming Style*, Second Edition. McGraw-Hill, New York, 1978.

[2] B.W. Kernighan and D.M. Ritchie, *The C Programming Language*. Prentice-Hall, New Jersey, 1978.

[3] G.J. Myers, *Composite / Structured Design*. Van Nostrand Reinhold, New York, 1978.

[4] B. Liskov and S. Zilles, "Programming With Abstract Data Types". Proc. of a Symposium on Very High Level Languages, SIGPLAN Notes 9, 4 (April 1974), p.50-59.

[5] C.L. McGowan and J.R. Kelly, *Top-Down Structured Programming Techniques*. Van Nostrand Reinhold, New York, 1975.

[6] R. Conway, D. Gries, and C.E. Zimmerman, *A Primer On Pascal*. Winthrop, Mass, 1976.

[7] N. Wirth, *Algorithms + Data Structures=Programs*. Prentice-Hall, New Jersey, 1976.

[8] P.H. Winston and B.K.P. Horn, *LISP*. Addison-Wesley, Mass, 1981.

[9] M.J. Bennett, Conversation on program readability, SDC, Santa Monica, CA. June 1984.