Programming the Evans & Sutherland PS 300*: I. A preprocessor with macro facilities

Arthur M Lesk†

Medical Research Council, Laboratory of Molecular Biology, Cambridge, UK

The Evans and Sutherland PS 300 is a powerful computer graphics device, programmable in a specialized line-drawing graphics language processed by internal firmware. We describe here a preprocessor to facilitate the generation of programs for the PS 300. It converts programs from a concise and easy-to-write style into a form acceptable to the PS 300 command interpreter. In addition to the expansion of abbreviations, the preprocessor supports macro definitions with substitutable parameters, and symbol definitions and replacements. All special features are embedded transparently in the PS 300 language itself. These features create a programming environment appropriate and convenient for generating function networks for the PS 300. It has been useful in producing robust and supple (i.e., easy to modify) programs for molecular graphics, but is not limited to this area of applications.

Keywords: preprocessor, Evans & Sutherland PS 300, programming

received 6 December 1983, revised 6 November 1984

The design of programming languages for powerful interactive graphics devices presents certain special problems arising from the realtime interactions they must support, the need to drive specialized hardware or firmware to process 3D lists of vectors, and the establishment of effective communications and cooperation between the graphics computer and a host. In the Evans & Sutherland PS 300, an internal graphics control processor accepts statements in a special command language to define, transform, and interact with 2D and 3D line drawings[1].

Two aspects of this command language and the execution of programs written in it are conceptually different from the corresponding aspects of more familiar numerically-oriented programming languages such as Fortran. One of these is the hierarchical nature of the data structures. The other is the nonsequential order of execution of statements: programs can contain networks of functions, each with multiple input and output connections. Only on receipt of initial impulses — from an interactive input device, or a clock — will the entire network 'fire', generating a concatenation of activities dependent on the connectivity and the logic of the network. The programmer creates function networks by assembling, initializing, and interconnecting instances of a fixed set of elementary functions defined by the language. (Conceptually, the PS 300 function network is a 'data flow machine', implemented in software on a M68000 computer.)

The general features of the preprocessor presented here, such as symbol and macro definitions, are useful in generating the data structure. It is in the generation of function networks, however, that the power of the preprocessor most explicitly aids the programmer by providing easy generation of modular packages of subnetworks, and providing fairly flexible control over connections.

The preprocessor decribed here was developed partly as a kind of exploratory training exercise for a new and unfamiliar machine. However, it has already proved effective in generating useful software; for example, Dr P R Evans is using it* in his recoding of the popular molecular graphics program FRODO, written originally by Dr T A Jones[2]. Nevertheless, it should be regarded as merely the first step towards defining a truly higher-level graphics language, preferably embedded in a general-purpose host language such as Fortran[3,4]. Inquiries from colleagues who wish to use the preprocessor should be directed to the author.

The programmer considering the use of this preprocessor should be aware of alternative facilities developed and distributed by the Evans & Sutherland Computer Corporation themselves, notably D J Schlegel's Interactive Network Editor, and the Graphics Support Routines.

THE PREPROCESSOR

In order to facilitate the generation, debugging, and modification of programs for the PS 300, we have created software that accepts a more general set and style of statements and produces a PS 300 program. This software is intended to be run in the host, in

*PS 300 is a trademark of the Evans and Sutherland Computer Corporation

†Permanent address: Fairleigh Dickinson University, Teaneck-Hackensack Campus, Teaneck, NJ 07666, USA

 0263-7855/85/010025-07 $03.00

our case a DEC VAX 11/780. It is written entirely in Fortran, to provide host-to-host portability. The PS 300 programming language is of course unique, and therefore the preprocessor is not portable to any other graphics device, although it might provide the basis for 'homologous' software for other graphics languages or even in other contexts of similar logical design.

Like other preprocessing software[5], this program permits:

- a more concise, readable, transparent and supple specification of a PS 300 program,
- the possibility of defining higher-level languages in terms of macros, a potential that we are currently exploring.

The preprocessor distinguishes eleven types of statements, or blocks of statements:

- instances of functions
- function network interconnections
- symbol definitions, and a special command to dump the current symbol table
- macro definitions
- macro expansions with parameter substitution
- definitions of input and output 'pin' connections to *modules* (which can be thought of as macros or clusters of macros with defined input and output nodes) and instructions to 'plug' one module into another
- comments
- an end-of-input flag
- other statements, assumed to be in the PS 300 command language itself, and which are transcribed without alteration

Precise definition of the syntax of these statements, and illustrative examples, follow. In general each statement treated by the preprocessor occupies a single line. However, the sequence &: serves as a continuation signal. If the last two nonblank characters on any line are &:, the program will strip off the &: and any trailing blanks, and concatenate the next line in the input stream.

1 Instances of functions

A typical PS 300 function is F:DXROTATE, which accepts an input signal, e.g., from a dial, to specify a differential angle of rotation. Other inputs to the function, which specify the accumulated angle value and a scale factor, can be initialized with suitable constants. As its output the function creates a 3 × 3 rotation matrix specifying a rotation around the *X*-axis. To use this function, the programmer must generate and name an instance of this function, initialize some and connect other of its inputs, and connect its outputs to one or more destinations. In PS 300 command language, such a sequence might appear as follows:

```
ROTX:= F:DXROTATE;
CONNECT DIALS(1):(1)ROTX;
SEND 0          TO (2)ROTX;
SEND 180        TO (3)ROTX;
CONNECT              ROTX(1):(1)S.ROT;
```

Here ROTX is the name of the instance of the function F:DXROTATE. The first input is accepted from dial 1, one of the interactive input devices. The second input (the accumulator) and third input (the scale factor) are initialized with constants 0.0 and 180.0. The rotation angle is taken to be the sum of the initial accumulator value and the product of the scale factor and the first input. The first output, a 3 × 3 matrix, is connected to the rotate node of a structure named S.

The preprocessor would generate this block of PS 300 code from the following statement:

```
ROTX = DXROTATE  DIALS(1) 0.0 180.0 // (1)S.ROT
                                          (2)ROTX //
```

The general form is:

```
instance = function  input1  input2 ... // output1  output2 ... //
```

The equals sign (at the end of, or after, the first character string) is the diagnostic of a function instance. The string following the equals sign names the function. (Because all PS 300 functions begin with F:, the software prefixes this automatically). Input and output connections follow. The // is an obligatory break sequence terminating the list of inputs and the list of outputs. A function instance with no first input connection and no output connection [at least none established by the statement] could be generated as follows:

```
EQ1 = EQC ,1 // //
```

which would be translated to:

```
EQ1 := F:EQC;
SEND FIX(1) TO (2)EQ1;
```

The arguments need not appear on the same line. Thus the example ROTX could have equivalently been written:

```
ROTX = DXROTATE
DIALS(1)  0.0      180  //
     (1)S.ROT           //
```

In many cases, input or output channels of a function require multiple connections. The preprocessor will accept multiple inputs or outputs separated by the character &. For example:

```
EQ1 = EQC   ,1 // (3)OUTA&(3)OUTB //
```

would be translated to:

```
EQ1 := F:EQC;
SEND FIX (1) TO (2)EQ1;
CONNECT          EQ1(1):(3)OUTA;
CONNECT          EQ1(1):(3)OUTB;
```

As an example of multiple inputs, consider initializing a network that is driven by a dial during execution. The preprocessor will translate the statement:

```
DIAL1MULC = MULC 0.0&DIAL1ACCUM(1) 180.0 // (1)ROTX //
```

into:

```
DIAL1MULC := F:MULC;
SEND 0.0 TO                  (1)DIAL1MULC;
CONNECT DIAL1ACCUM(1):(1)DIAL1MULC;
SEND 180.0 TO                (2)DIAL1MULC;
CONNECT                        DIAL1MULC(1):(1)ROTX;
```

Notice that the PS 300 command interpreter assumes all numbers to be real unless FIX is specified. Thus, for example,

```
SEND 180.0 TO ⟨3⟩ROTX;
```

and

```
SEND 180   TO ⟨3⟩ROTX;
```

are equivalent in PS 300 command language. If an integer is desired, one must specify this to the command interpreter, thus:

```
SEND FIX(1) TO ⟨1⟩EQ1;
```

The preprocessor, by contrast, assumes that numbers without decimal points are integers, and it equips them with a FIX in the generated code. Therefore real numbers *must* contain a decimal point‡.

Carat and double carat refer to current and previous function instances. Two useful bits of shorthand are:

```
  ^ = the name of the current function, and
^^ = the name of the previous function.
```

Thus a network to calculate:

```
OUTPUT = 4.0*INPUT + 1.0
```

could be written either as:

```
MULTIPLY = MULC      INPUT⟨1⟩  4.0 //            //
ADD      = ADDC  MULTIPLY⟨1⟩  1.0 // ⟨1⟩OUTPUT //
```

or as:

```
MULTIPLY = MULC  INPUT⟨1⟩  4.0 //            //
ADD      = ADDC     ^^⟨1⟩  1.0 // ⟨1⟩OUTPUT //
```

both of which translate to:

```
MULTIPLY := F:MULC;
CONNECT      INPUT⟨1⟩:⟨1⟩MULTIPLY;
SEND 4.0 TO              ⟨2⟩MULTIPLY;
ADD := F:ADDC;
CONNECT            MULTIPLY⟨1⟩:⟨1⟩ADD;
SEND 1.0 TO                    ⟨2⟩ADD;
CONNECT                          ADD⟨1⟩:⟨1⟩OUTPUT;
```

A statement that is useful in handling the function keys follows. Whenever function key N is pressed, a TRUE will be sent to anything connected to the output node FCNKEYS⟨N⟩:

```
FCNKEYS = ROUTEC(36) FKEYS⟨1⟩ TRUE //  //
```

which expands to

```
FCNKEYS :=:ROUTEC(36);
CONNECT FKEYS⟨1⟩:⟨1⟩FCNKEYS;
SEND TRUE TO       ⟨2⟩FCNKEYS;
```

Any networks to be driven by a specific function key can be CONNECTed to the appropriate output node of FCNKEYS (see the section on macro expansions).

2 Extra connections

The symbol ⟩ at the beginning of a line allows the programmer to establish additional initializations or connections in a function network or to a display structure. Its effect is similar to a combination of input and output parameters in an instance of a function.

Thus,

```
⟩ 180.0 ⟨1⟩ROTX
```

will be translated to

```
SEND 180.0 TO ⟨1⟩ROTX;
```

and

```
⟩ FALSE ⟨1⟩PRINTA&⟨1⟩PRINTB&⟨1⟩PRINTC
```

will be translated into three statements:

```
SEND FALSE TO ⟨1⟩PRINTA;
SEND FALSE TO ⟨1⟩PRINTB;
SEND FALSE TO ⟨1⟩PRINTC;
```

A statement containing n multiple inputs and m multiple outputs would generate n×m SEND or CONNECT statements.

Also, compare:

```
⟩ ROTX ⟨1⟩DLABEL5
```

which is translated to:

```
SEND 'ROTX' TO ⟨1⟩DLABEL5;
```

with

```
⟩ ROTX⟨1⟩ ⟨1⟩S.ROT
```

which is translated to:

```
CONNECT ROTX⟨1⟩:⟨1⟩S.ROT;
```

The program takes numbers, and strings that do not contain ⟨...⟩ (that is, that do not contain the character ⟨ appearing before the character ⟩), to be constants for initializations of function instances. The program assumes that strings which do contain ⟨...⟩ specify connections.

3 Symbol definitions

The PS 300 preprocessor recognizes symbol definitions as statements of the form:

```
[SYMBOL] REPLACEMENT
```

The character string which will *be replaced* must begin with a [, must end with a] and may contain no more than ten characters, including the [and]. Its replacement is taken to be the first character string

‡In SEND statements, the question arises whether a parameter should be enclosed in single quote marks. Character strings are to be delimited by single quotes, but numbers, logical variables, and vectors must not be. Most cases can be handled straightforwardly by analysis of the sequences of characters that specify a parameter. Enclosure of a parameter in single quotation marks forces it to be treated as a character string (e.g. '100' is not an integer, and the single quotation marks would be retained in the translated statement. The problem is that some variable types require embedded punctuation marks or blanks, e.g., V3D(1.0, 2.0, 3.0). In order for the parser to recognize these as a single parameter rather than as several parameters separated by delimiters, it would be convenient to enclose them in single quotes; but that would force them to be treated as character strings! The convention has been adopted that if a parameter is enclosed in *double* quotation marks, e.g. "V3D (1.0, 2.0, 3.0)", it will not be broken by the internal blanks or punctuation marks, but neither single nor double quotes will appear in the SEND statement.

following the]; this must be enclosed in single quote marks if it contains embedded blanks or special characters.

Symbols are replaced in each input line immediately upon reading it (before macro expansion, for example). Because execution of the preprocessor is a one-pass operation, a symbol is active when its definition is first processed but not before. Thus the definition of a symbol must appear in the input file before it can be replaced. If a symbol definition appears within a macro definition (a treacherous procedure), an *expansion* of the macro must occur before the symbol enters the symbol table.

To suspend symbol substitution (for any single line only), an input line containing a symbol may be enclosed in double quotes. Using this device (necessarily), a symbol may be redefined.

A line consisting of [] is a special command that will cause the current contents of the symbol table to be listed.

As an example of the use of symbols, consider setting up a network with multiple dial assignments. Let us refer to a full set of dial assignments as a panel. We define separate accumulators for each dial in each panel, in order to switch smoothly from one panel to another. To organize these definitions, we create a symbol [PANEL] to distinguish their instance names. In the following example, the input from dial 3 will be routed to either of two rotate functions, WSROT or OSROT, depending on the constant in ⟨1⟩ROUTEDIAL. (This constant is initialized here to 1; it could be reset under control of a function key, for example.) Thus, showing the functions for dial 3 only:

```
[PANEL] 1
DIALACC[PANEL]3 = ACCUMULATE
      DIALS⟨3⟩ 0.0  0.001  90.0  1000.0  −1000.0 // ⟨1⟩WSROT //
"[PANEL] 2"
DIALACC[PANEL]3 = ACCUMULATE
      DIALS⟨3⟩ 0.0  0.001  90.0  1000.0  −1000.0 // ⟨1⟩OSROT //
ROUTEDIAL       = ROUTEC(2)  1  DIALS⟨3⟩ //
                          ⟨1⟩DIALACC13 ⟨1⟩DIALACC23 //
```

would translate to:

```
DIALACC13 := F:ACCUMULATE ;
CONNECT          DIALS⟨3⟩:⟨1⟩DIALACC13 ;
SEND 0.0               TO ⟨2⟩DIALACC13 ;
SEND 0.001             TO ⟨3⟩DIALACC13 ;
SEND 90.0              TO ⟨4⟩DIALACC13 ;
SEND 1000.0            TO ⟨5⟩DIALACC13 ;
SEND −1000.0           TO ⟨6⟩DIALACC13 ;
CONNECT                     DIALACC13⟨1⟩: ⟨1⟩WSROT ;



DIALACC23 := F:ACCUMULATE ;
CONNECT          DIALS⟨3⟩:⟨1⟩DIALACC23 ;
SEND 0.0               TO ⟨2⟩DIALACC23 ;
SEND 0.001             TO ⟨3⟩DIALACC23 ;
SEND 90.0              TO ⟨4⟩DIALACC23 ;
SEND 1000.0            TO ⟨5⟩DIALACC23 ;
SEND −1000.0           TO ⟨6⟩DIALACC23 ;
CONNECT                     DIALACC23⟨1⟩: ⟨1⟩OSROT ;



ROUTEDIAL := F:ROUTEC(2) ;
SEND FIX(1)          TO ⟨1⟩ROUTEDIAL ;
CONNECT     DIALS⟨3⟩:⟨2⟩ROUTEDIAL ;
CONNECT              ROUTEDIAL⟨1⟩: ⟨1⟩DIALACC13 ;
CONNECT              ROUTEDIAL⟨2⟩: ⟨1⟩DIALACC23 ;
```

One could generate this code using substitutable parameters in macros. However, the use of symbols makes it easier to edit the program. To move the connection of dial 3 to WSROT to another panel we could simply lift the two lines containing the accumulate function and move them into the range of a different assignment of the value of [PANEL], where they would automatically be translated properly.

4 Macro definitions

A statement sequence

```
$MACNAME
...
$MEND
```

defines the block of lines between $MACNAME and $MEND lines to be a macro named MACNAME. This block of lines may contain the items *1, *2, . . . *9, *A, *B, . . . *K to represent substitutable parameters. Each macro has a different MACNAME but every macro definition is terminated by the character string $MEND.

Names of macros are truncated to eight characters if longer, and may contain alphanumeric characters only. All letters in names of macros are translated to upper case for both definition and expansion.

5 Macro expansions

The statement

```
@MACNAME A B 180 3.0 B23
```

will cause transcription into the PS 300 program being generated of the block of lines defined as MACNAME, with the character strings A, B, 180, etc, substituted for parameters *1, *2, *3, . . . respectively. Function instances within macro definitions are expanded as described above in the section on instances of functions. The definition of a macro must precede any instance of its expansion.

Macros may call upon other macros within themselves (but not define them); the nesting level may be arbitrarily deep. However, macros must not call themselves recursively (or reciprocally). This will cause an infinite loop of transcription and expansion (and the program does not check for this).

Examples of macros follow. It is assumed in these examples that the statement:

```
FCNKEYS = ROUTEC(36)  FKEYS⟨1⟩  TRUE
```

is present in the network (see end of the section on instances of functions):

Example 1. To create a Boolean variable, flipped by a function key. Defining:

```
  $TOGGLE
{ THE VALUE OF A LOGICAL VARIABLE CORRESPONDING
                                   TO FUNCTION KEY *1,
  INITIALIZED TO *2, IS FLIPPED EACH TIME KEY *1 IS
                                              PRESSED}
  VARIABLE FK*1;
  ⟩ *2  ⟨1⟩FK*1
  FETCHFK*1  = FETCH  FCNKEYS⟨*1⟩ FK*1 //          //
  INVERTFK*1 = NOT  FETCHFK*1⟨1⟩        //⟨1⟩FK*1 //
  $MEND
```

then

```
@TOGGLE 3 FALSE
```

would expand to:

```
{ THE VALUE OF A LOGICAL VARIABLE CORRESPONDING
                                    TO FUNCTION KEY 3,
  INITIALIZED TO FALSE, IS FLIPPED EACH TIME KEY 3 IS
                                    PRESSED}
  VARIABLE FK3;
  SEND FALSE TO ⟨1⟩FK3;
  FETCHFK3 := F:FETCH;
  CONNECT FCNKEYS⟨3⟩:⟨1⟩FETCHFK3;
  SEND 'FK3' TO          ⟨2⟩FETCHFK3;
  INVERTFK3 := F:NOT;
  CONNECT FETCHFK3⟨1⟩:⟨1⟩INVERTFK3;
  CONNECT                INVERTFK3⟨1⟩:⟨1⟩FK3;
```

What would be the effect of including the following statement in the macro definition, and how would the statement calling for the expansion of the macro have to be modified to use it?

```
FK*1MSGE = BOOLEAN_CHOOSE INVERTFK⟨*1⟩ *2 *3 //
                                    ⟨1⟩FLABEL*1 //
```

Example 2. To create a counter, driven by a function key, that cycles through the integers 1, 2, . . . N (note that *1 is the number of the function key; *2 is the maximum value of the counter; and observe the use of ^^ to refer to the previous function):

```
$COUNTER
VARIABLE COUNT*1VAL;
› 0 ⟨1⟩COUNT*1VAL
GETCOUNT*1 = FETCH FCNKEYS⟨*1⟩ COUNT*1VAL //
CFK*1BUMP  = ADDC ^^⟨1⟩       1  //⟨1⟩COUNT*1VAL //

CFK*1MOD   = MODC ^^⟨1⟩       *2 //              //

COUNT*1    = ADDC ^^⟨1⟩       1  //              //

$MEND
```

Then the statement

```
@COUNTER 5 3
```

would be translated to the following:

```
VARIABLE COUNT5VAL;
SEND FIX(0) TO ⟨1⟩COUNT5VAL;

GETCOUNT5 := F:FETCH;
CONNECT          FCNKEYS⟨5⟩:⟨1⟩GETCOUNT5 ;
SEND 'COUNT5VAL'          TO ⟨2⟩GETCOUNT5 ;

CFK5BUMP := F:ADDC ;
CONNECT        GETCOUNT5⟨1⟩:⟨1⟩CFK5BUMP ;
SEND FIX(1)                TO ⟨2⟩CFK5BUMP ;
CONNECT                   CFK5BUMP⟨1⟩:⟨1⟩COUNT5VAL ;

CFK5MOD := F:MODC ;
CONNECT          CFK5BUMP⟨1⟩:⟨1⟩CFK5MOD ;
SEND FIX(3)                TO ⟨2⟩CFK5MOD ;

COUNT5 := F:ADDC ;
CONNECT          CFK5MOD⟨1⟩:⟨1⟩COUNT5 ;
SEND FIX(1)                TO ⟨2⟩COUNT5 ;
```

Other parts of a network could gain access to the current value of COUNT5VAL by fetching it; or initiate actions whenever the valuc of COUNT5 changes by connecting COUNT5⟨1⟩ to the appropriate input node.

Example 3. Printing debugging information. The following macro has proved useful for debugging function networks, by reporting the value of a selected variable:

```
$DEBUG
DBG*1 = BEGIN_STRUCTURE
CSCALE := CHARACTER SIZE 0.05;
DBG*1TEXT := CHARACTERS *3,*4 ' ';
END_STRUCTURE;
DBG*1PRT = PRINT *2 // ⟨2⟩DBG*1CCT //
DBG*1CCT = CCONCATENATE '*1' //
                          ⟨substitute⟩DBG*1.DBG*1TEXT //
DISPLAY DBG*1;
$MEND
```

The first parameter, to be substituted for *1, is merely to ensure that the different expansions of the macro create unique names. (We shall shortly provide a facility whereby a special character will be converted to a different, unique, string each time it appears.) To report the value of dial 6, at the center of the screen, the programmer could include the following:

```
@DEBUG D6 DIAL6⟨1⟩ 0.0 0.0
```

6 Modules

Any block of PS 300 code may be logically defined as a module by specifying input and output 'pin connections'.

Thus the special command

```
!IPIN SCALER 1 ⟨1⟩ADD4
```

would define ⟨1⟩ADD4 as logical *input* pin 1 of module SCALER. (Module names are, if necessary, truncated to 8 characters and alphabetic characters are converted to upper case.)

Similarly:

```
!OPIN GETDIAL 1 DIAL4ACC⟨2⟩
```

would define DIAL4ACC⟨2⟩ to be logical *output* pin 1 of module GETDIAL. Then the statement:

```
!PLUG GETDIAL INTO SCALER
```

would generate the statement:

```
CONNECT DIAL4ACC⟨2⟩:⟨1⟩ADD4;
```

The preprocessor will generate as many CONNECT statements as necessary to plug all output pins of the upstream module into the corresponding input pins of the downstream module. It is assumed that the corresponding connections of the two modules are compatible in number and type.

7 Comments

The preprocessor follows the PS 300 convention of assuming that any material enclosed in curly brackets {...} is a comment. Comments may be nested (this is useful to suspend temporarily an entire block of code which may itself contain comments).

It is recommended that comment material be set apart on separate lines from material to be translated. However, if a line begins with a comment, it will be split into two separate lines after the } and processed appropriately.

Comments may be interspersed with parameter values in macro expansions, but will not be transcribed. They can, however, be included in the parameters. An example will clarify this (this example makes use of the COUNTER network of example 2 of a previous section).

Defining:

```
{ SEND THE VALUE OF DIAL *1 TO ONE OF THREE
                                        DESTINATIONS
  ACCORDING TO THE VALUE OF A COUNTER DRIVEN BY
                                        FUNCTION KEY 5}
          $ROUTEDIAL
          SENDDIAL*1 = CROUTE(3)  COUNT5(1)
                                        DIALS(*1)// //
          )               ^^(1)       (1)*2
          )               ^^(2)       (1)*3
          )               ^^(3)       (1)*4
          $MEND
```

then:

```
@ROUTEDIAL 2 'ROTX;' {DIAL 2 TO SUBUNIT X
                                        ROTATIONS}&:
           'ROTY;' {DIAL 2 TO SUBUNIT Y
                                        ROTATIONS}&:
           'ROTZ;' {DIAL 2 TO SUBUNIT Z ROTATIONS}
```

would be translated to:

```
SENDDIAL2:=F:CROUTE(3)
CONNECT        COUNT5(1):(1)SENDDIAL2 ;
CONNECT          DIALS(2):(2)SENDDIAL2 ;
CONNECT                   SENDDIAL2(1):(1)ROTX;
CONNECT                   SENDDIAL2(2):(1)ROTY;
CONNECT                   SENDDIAL2(3):(1)ROTZ ;
```

without the comments. But:

```
@ROUTEDIAL 2 'ROTX; {DIAL 2 TO SUBUNIT X
                                        ROTATIONS}' &:
           'ROTY; {DIAL 2 TO SUBUNIT Y
                                        ROTATIONS}' &:
           'ROTZ; {DIAL 2 TO SUBUNIT Z ROTATIONS}'
```

would be translated to:

```
SENDDIAL2 := F:CROUTE(3)
CONNECT COUNT5(1):(1)SENDDIAL2 ;
CONNECT   DIALS(2):(2)SENDDIAL2 ;
CONNECT                 SENDDIAL2(1):(1)ROTX;
                      {DIAL 2 TO SUBUNIT X ROTATIONS}
CONNECT                 SENDDIAL2(2):(1)ROTY;
                      {DIAL 2 TO SUBUNIT Y ROTATIONS}
CONNECT                 SENDDIAL2(3):(1)ROTZ;
                      {DIAL 2 TO SUBUNIT Z ROTATIONS}
```

8 Flag for end of input

A line beginning with the symbol # signals the end of the input data set. An end of file does the same thing.

OUTPUT FROM THE PREPROCESSOR

The program produces the following four files:

- A record of its work showing the effect of the processing of each statement, the listing of symbol tables as requested, and the flow of macro definitions and expansions.
- A version of the PS 300 output code that includes comments and is laid out in a way that is easy to read. This is legitimate PS 300 code, which *could* be sent to the command interpreter, but is useful primarily for human inspection.
- A compressed version of the PS 300 program that contains no comments and no unnecessary blank characters.
- Another version of the compressed command file. This is sorted so that all CONNECT and SEND statements follow all statements of other types. It is this file that should be sent to the PS 300 command interpreter, particularly if a large program is to be sent over a slow line.

So far, the debugging aids provided by the preprocessor are rudimentary but not useless. A table, listing (in alphabetical order) the function names generated and the lines at which they are defined is useful in detecting and pinpointing duplicate function names, a common source of problems, especially when using macros.

CONCLUSIONS AND DIRECTIONS FOR FUTURE WORK

Using the programs described here as a starting point, the author and his colleagues have been engaged in a number of projects related to software development for molecular graphics. Foci of current activities include:

- applications software
  - linking the program of Lesk and Hardman that draws schematic diagrams of proteins[6] to the PS-300. A simple data structure and network permit line drawings produced by that program to be transmitted and displayed on the PS 300, with interactive control over viewpoint and scaling, and colour and intensity control over individual parts of the structure. This system is in routine use by several colleagues at the Laboratory of Molecular Biology.
  - the revisions of FRODO by Dr P R Evans, already mentioned.
- the provision of a set of debugging tools for PS 300 program development. These will be particularly useful in multiuser situations in which access to the host is more readily available than access to the PS 300 itself. Such tools will include the collation and printing of the function network interconnections, and checking for missing or illegal interconnections.
- preparing a version of the output that contains calls to the graphics support routines, rather than PS 300 code itself. Formatted PS 300 code must be interpreted in the PS 300. The graphics support routines send coded information directly to the PS 300 command interpreter, eliminating the parsing step, and thereby speeding up the process of loading networks.
- Development of higher-level graphics languages which can produce input to this preprocessor as their output.

Descriptions of the results of this work will be published in due course.

ACKNOWLEDGMENTS

We are very grateful to Dr P R Evans for many helpful suggestions, and to R Spitz of Evans and Sutherland for a critical reading of the manuscript. The work was supported in part by the US National Science Foundation (PCM83-20171).

REFERENCES

1 *PS300 User's Manual* Evans & Sutherland Computer Corp., Salt Lake City, UT, USA (1983)
2 **Jones, T A** in **Sayre, D (ed)** *Computational Crystallography*. Clarendon Press, UK (1982) pp 303–317
3 **Lesk, A M** 'Generation of interactive displays from Fortran using the LDS-1 computer graphics system' *Software — Pract. Exper. No. 2* (1972) pp 259–273
4 For a discussion, see **Lesk, A M** *Case studies in chemical computing* COMPress, Hanover, NH, USA (1981) Chapter 5
5 For a general reference, see: **Kernighan, B W and Plauger P J** *Software tools* Addison-Wesley, USA (1976)
6 **Lesk, A M and Hardman, K** 'Computer-generated schematic diagrams of protein structures' *Science* No. 216 (1982) pp 539–540