AUG 5 1976

The person charging this material is responsible for its return to the library from which it was withdrawn on or before the Latest Date stamped below.

Theft, mutilation, and underlining of books are reasons for disciplinary action and may result in dismissal from the University.

UNIVERSITY OF ILLINOIS LIBRARY AT URBANA-CHAMPAIGN

ENGINEERING

L161 — O-1976
PECESPOL: A Metaprogramming Language for Operating Systems

by

David M. Grothe

December 1973
PEESPOL: A Metaprogramming Language
for Operating Systems

David M. Grothe*

Center for Advanced Computation
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801
December 1973

* This work was supported in part by the Advanced Research Projects Agency of the Department of Defense and was monitored by the U.S. Army Research Office-Durham under Contract DAHC04-72-C-0001. The views and conclusions contained in this document are those of the author and should not be interpreted as necessarily representing the official policies, either expressed or implied, of ARPA or the U.S. Government.
<table>
<thead>
<tr>
<th>TABLE OF CONTENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract ........ 1</td>
</tr>
<tr>
<td>Introduction ..... 1</td>
</tr>
<tr>
<td>The Base Language 2</td>
</tr>
<tr>
<td>Data Types ....... 2</td>
</tr>
<tr>
<td>Arrays .......... 3</td>
</tr>
<tr>
<td>Arithmetic Expressions 3</td>
</tr>
<tr>
<td>General ........ 4</td>
</tr>
<tr>
<td>The Metaprocessor 4</td>
</tr>
<tr>
<td>Metaoperations ... 8</td>
</tr>
<tr>
<td>Control Metaoperators 8</td>
</tr>
<tr>
<td>Synthesis Metaoperators 9</td>
</tr>
<tr>
<td>Numeric Metaoperators 10</td>
</tr>
<tr>
<td>Defines - Metaprocedures 11</td>
</tr>
<tr>
<td>DEFINES within DEFINES 13</td>
</tr>
<tr>
<td>List Processing ... 14</td>
</tr>
<tr>
<td>Data Structures ... 15</td>
</tr>
<tr>
<td>Measurements .... 19</td>
</tr>
<tr>
<td>Concluding Remarks 19</td>
</tr>
<tr>
<td>References ........ 20</td>
</tr>
</tbody>
</table>
Abstract

An operating system implementation language is presented which allows the system programmer to specify the disciplines to which he must adhere in order to produce a working system in a reasonable length of time. The language is oriented towards the Digital Equipment Corporation PDP-11 series of machines and has been acronymed PEESPOL (PDP Eleven Executive System Programming Oriented language). PEESPOL taken as a whole consists of a programming language (the base language) and a metaprogramming language. Facilities and features of the base language relieve the programmer from the tedium of assembly language programming by providing higher level constructs such as conditionals, loop control, control switches (i.e. CASE statements), arithmetic assignments, procedure invocations and interrupt declarations; at the same time the programmer may code on an instruction-by-instruction basis where critical time constraints demand the highest possible efficiency. The metaprogramming language allows one to impose upon oneself whatever measure of discipline is necessary if not to ensure, at least to expedite the production of a working system, the meta program being a program which executes at compile time and generates code in the base language, which code is known to be consistent and correct within the context of the system as a whole. The metaprogram can be halted and resumed at a later time, a facility which lends itself to the production of layered systems a la Dijkstra.

Keywords: language, compiler, macro, compile-time, system, implementation-language.

Introduction

PEESPOL (PDP Eleven Executive System Programming Oriented Language) is a programming language for use in writing operating systems, or other "stand alone" programs, for the Digital Equipment Corporation PDP-11 series of machines. The language is organized in two distinguishable parts: the base language, which bears a passing resemblance to ALGOL, and the meta (or macro) language, which gives the programmer facilities for performing computations at compile time and for "generating" strings of the base language as input to the parser of the compiler.
Figure 1 illustrates the organization of the PEESPOL compiler with regard to the stages of processing of its input. The metaprocessor first examines each token of the input stream and determines whether that token is a construct of the base language or a construct of the meta (or macro) language. Base language constructs are simply passed on to the compiler proper. Metalanguage constructs are interpretively "executed" by the metaprocessor and transformed into base language constructs.

For example, the construct '&LENGTH("ABC")' is a metafunction which evaluates a simple argument and generates a number whose value is the length of the argument, in this case 3. The number is then passed on to the compiler as a base language construct.

The focus of this paper will be on the metalanguage. For the purpose of this presentation, the nature of the base language itself is relatively unimportant (for example, a very similar metalanguage and metaprocessor exists for the ILLIAC IV assembler [1]). It is of greater importance that the metaprocessor stand in the relation to the compiler as illustrated in figure 1.

The Base Language

The base language of PEESPOL[2] is pretty much a garden-variety programming language. The language is modelled after ALGOL 60: it obeys ALGOL scoping rules; it is a statement language as opposed to an expression language; it includes syntactic forms similar to those of ALGOL. We will quickly sketch just enough of the base language to provide a basis for the exposition of the metaprocessor.

Data Types:

There are two data types in PEESPOL: Word and Byte. Variables of these types are introduced by way of declarations:

WORD A, B, C=A, D;
BYTE X, Y=B, Z=B+1;
The compiler allocates storage for A, B, D, and X; C is given the same address as A; similarly, Y has the same address as B, and Z addresses the next location after B.

Arrays:

Arrays are single dimensional and of type Word or Byte. Arrays can be address-equated in much the same way as Words and Bytes:

```plaintext
BYTE ARRAY BA[10];
WORD ARRAY WA[*]=A;
WORD ARRAY MEMORY[*]=0;
```

Array BA is allocated by the compiler, WA is unsized ([*]) and addresses the same location as A. MEMORY is unsized and addresses memory location zero.

The PDP-11 is a 16-bit machine. Addresses on the PDP-11 are addresses of 8-bit bytes (two per word). If an instruction accesses a word, the effective address must be even. Therefore, the Ith word of array WA is located 2*I bytes from the base of the array.

Arithmetic Expressions:

Without going into any detail about the various operators and their relative precedences in arithmetic expressions, we will simply look at some forms of arithmetic primaries in order to show how one accesses variables in PEESPOL. The examples will refer to the above declarations.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>The contents of the location in memory named by A. Same as: @([A]).</td>
</tr>
<tr>
<td>.A</td>
<td>A accessed as a Byte.</td>
</tr>
<tr>
<td>@A</td>
<td>Indirect through A. May also be written: @(A).</td>
</tr>
<tr>
<td>.@A</td>
<td>Indirect Byte through A (one level of indirect).</td>
</tr>
<tr>
<td>@@A</td>
<td>Two levels of indirect through A.</td>
</tr>
<tr>
<td>.@@A</td>
<td>Two levels of indirect through A with</td>
</tr>
</tbody>
</table>
the last access fetching a Byte.

[A]
Address of A.
BA[I]
Same as: .@[BA]+I
WA[I]
Same as: @[WA]+2*I) that is, the
subscripting is in units of Words.
WA<I>
Same as: @[WA]+I) that is,
the subscripting is in units of
bytes, but a word is accessed.

Arithmetic expressions are evaluated strictly from left to right.

General:

The base language bears a close enough resemblance to ALGOL that
further description is unnecessary. The illustrations of the various
forms of variable accessing were given in order to clarify the examples
in the next section.

The Metaprocessor

One can think of the metaprocessor as a machine which is
interpretively executing a compile-time program and generating strings
in the base language. The constructs of the language interpreted by the
metaprocessor are text processing oriented since the design criterion
for the metaprocessor was that it be able to generate text in the base
language.

We will describe the constructs of the metalanguage with a concrete
example of their use in mind, developing a body of metaconstructs which
implements data structures as an extension to the base language. The
goal is to be able to program the metaprocessor in such a way that it
will generate base language constructs which will allow the programmer
to name fields of data structures and access those fields by name.

To illustrate the mechanism of the accessing of data structures,
let us suppose that we would like to define a linked list, each of whose elements denotes an element of a two dimensional array. Using only constructs of the base language we could accomplish this with the following set of ARRAY declarations (the character period [.] is allowed in identifiers):

```plaintext
WORD ARRAY ELEM.FLINK[*]=0;
WORD ARRAY ELEM.BLINK[*]=2;
WORD ARRAY ELEM.X       [*]=4;
WORD ARRAY ELEM.Y       [*]=6;
```

Referring to the definitions of array accessing given above, if such a list element is located at some memory address A, then

```plaintext
ELEM.FLINK<A> = @(ELEM.FLINK)+A
              = @(0+A)
              = @A
```

accesses the FLINK field of the element, and

```plaintext
ELEM.X<A> = @(ELEM.X)+A
           = @(4+A)
```

accesses the X field of the element.

With this example in mind we will now describe the constructs of the metalanguage.

**Metavariables:**

We call metavariables those variables with which the metaprogram (i.e. the compile-time program) computes. There are two kinds of metavariables; one to be used to store arithmetic values computed by the metaprocessor, and the other to be used in the storage and manipulation of text by the metaprocessor. First the arithmetic
variable:

CELL I;
CELL J=0;

The CELL declaration introduces a name to the metaprocessor and gives it an initial value, if desired. One can assign into these variables (the assignment is performed by the meta processor):

cellname = compile-time-expression;

Or, to give a specific example:

I = J+1 ;

The elements of compile time expressions can be constants, CELLS, or the name of any variable which denotes a location in memory. A variable name, when used in a compile time expression, denotes its associated memory address.

Next the compile-time text variable:

TEXT T[100] = "XYZ"+"ABC";

The TEXT declaration declares the TEXT variable, gives its maximum size, and initializes it to the value of a TEXT-expression. The symbol "+" is the text concatenate operator. A TEXT variable is a compile-time repository for text.

One can access partial fields of TEXT variables according to the following syntax:

T[leftchr:numberofchrs] specifies a field of characters. Characters are numbered from left to right with the leftmost character having index zero.

T[*] the first character off the right hand end of T.

T[leftchr:*] a field from leftchr to the
T[leftchr]
end of T.
same as T[leftchr:0], i.e.,
a zero-length field.
the entire text variable.

One can either access or assign into partial fields of a TEXT variable. In the text assignment,

T[leftchr:memberofchrs] = "string";

"memberofchrs" characters are deleted from T starting at "leftchr"; the "string" is then inserted to the left of "leftchr". In this way, the index of the first character of the "string" will always be "leftchr". A few examples will illustrate this process:

T = "ABCDEF";
T[*]="GH" ;
T[2:3]="1234";
T[6:*]=T[8:1] ;

T now equals "ABCDEFGH"
T now equals "ABACDEFGH"
T now equals "AB1234EFGH"
T now equals "AB1234G"

Having built something in a TEXT variable, one can present the text to the compiler for processing as input. Schematically, one can insert the contents of the TEXT variable into the input tape for the metaprocessor such that its text becomes the next input for the metaprocessor. One indicates this by following a TEXT variable by an apostrophe; i.e. T'.

Note that the text inserted into the input may or may not be acted upon by the metaprocessor. The actual nature of the text itself determines whether it is metaprogram or simply program. For example, one could write the word BEGIN in a round-about way by the following section of metaprogram:

T = "BEGIN" ;
T'

-7-
Metaoperations:

The metaoperators are the instructions to the metaprocessor. They are distinguished from the text of the base language program by always starting with the character ampersand (&). The result of a metaoperation is always to make an insertion into the input stream of the metaprocessor. If the text so inserted is a metaoperator, the metaprocessor is invoked again. This process continues until some base language construct is generated, at which point it is simply passed on to the compiler proper. Sometimes the text inserted is the empty string.

We will distinguish three different classes of metaoperators: Control, Synthesis, and Numeric.

Control Metaoperators:

These are the operators which allow for the conditional transfer of control of the metaprogram and loops within the metaprogram. It is important to stress that the entire mechanism is interpretive so that "transfer of control" means interrupting the sequential order of the input stream to the compiler, just as transfer of control in a computer means interrupting the sequential order of instructions fetched into the CPU for execution. With this point firmly in mind, we introduce the two "transfer of control" metaoperators:

&IF compile-time-expression &THEN text1 &FI or,  
&IF compile-time-expression &THEN text1 &ELSE text2 &FI

This is the compile-time analog of the run time IF-THEN-ELSE-FI construct. The compile-time-expression is evaluated. If it is odd (true), the input is switched to text1; if it is even (false), the input is switched to text2 if it is present, otherwise it is switched to the point just beyond the &FI.

&WHILE compile-time-expression &DO text &OD

The input is switched to text if and as long as the compile-time-
expression evaluates to odd (true). This construct is restricted to be used only in a compile-time procedure (called a DEFINE) which we will discuss below.

By way of example, let us suppose that we wanted to declare an array GEORGE and initialize it to the numbers 0, 3, 6, 9, ..., 81:

```
CELL CTR=0;
BYTE ARRAY GEORGE[28]:= 0
&WHILE (CTR=CTR+3) LEQ 81
&DO
  ,CTR
&OD
```

The effect of the above example is as if the following declaration had been written:

```
BYTE ARRAY GEORGE[28]:=0, 3, 6, 9, 12, ..., 81;
```

**Synthesis Metaoperators:**

The synthesis metaoperators take a sequence of tokens as an argument and return the concatenation of them as the indicated type of item (identifier, string, or number). The sequence of tokens is treated specially in that a TEXT identifier which occurs as one of the tokens has its associated text substituted for it. The synthesis metaoperators themselves are:

- `&STRING(token-sequence)` builds a string
- `&ID(token-sequence)` builds an identifier
- `&NUMBER(token-sequence)` builds a number
For example, an alternate way to give the declaration:

```
WORD ARRAY ELEM.X[*]=4;
```
might be:
```
CELL LC;
TEXT STRUCTNAME[64];
STRUCTNAME = "ELEM";
LC = 4;
WORD ARRAY &ID(STRUCTNAME . X)[*] = LC ;
```

This seemingly laborious and round about way of writing an array declaration will be seen to be quite useful in the subsequent presentation. We will shortly introduce metaprocedures (which we call DEFINEs) and show how one can use them to generate such declarations automatically.

**Numeric Metaoperators:**

The numeric metaoperators generate numbers in the input stream. This is to be taken as quite literally true: a numeric metaoperator which evaluates to the result 5 will actually generate the numeral "5" in the input stream.

These metaoperators are:

- **&CLASS(token)**
  - the compiler's internal classification code for the token.

- **&LENGTH(token)**
  - the length of the token. If the token is a TEXT identifier, the length of its corresponding text is generated.

- **&EMPTY(Define-parameter)**
  - one or zero depending upon whether the parameter contains any text or not. DEFINEs are explained in the next section.
Defines - Metaprocesses:

Since the metaprocessor is interpreting text, the logical form for a metaprocess procedure to take is that of a piece of text. When the metaprocessor identifier (or DEFINE identifier) is encountered by the metaprocessor it switches the input stream to the text of the DEFINE. A DEFINE declaration introduces an identifier as the name of a DEFINE and specifies its associated text. The simplest form of a DEFINE declaration is:

```
DEFINE define-identifier = define-text ##;
```

The text which appears between the equals sign (=) and the double-sharp (##) is the text, or body, of the DEFINE.

DEFINES can be declared with parameters. The parameter names begin with the character ampersand (&). For example:

```
DEFINE BUMP(&X) = &X:=&X+1 ##;
DEFINE STRUCTELEM(&TYPE,&NAME) =
   &TYPE ARRAY &ID(STRUCTNAME . &NAME)[*] = LC ;
   LC = LC+2
##;
```

The second example shows a further parameterization of the declaration of WORD ARRAY ELEM.X. This example makes reference to the TEXT variable and CELL used in the example of the previous section. The DEFINE STRUCTELEM would be invoked in the following way:

```
STRUCTELEM(WORD,X);
```

It is not necessary to separate the parameters of a DEFINE by commas. Any punctuation mark or identifier will perform the same function. The invocation of the DEFINE must conform to its declaration with regard to the punctuation of the parameter list. For example, we could just as easily have written DEFINE STRUCTELEM in the following way:

```
DEFINE STRUCTELEM: &TYPE &NAME; =
   &TYPE ARRAY &ID(STRUCTNAME . &NAME)[*] = LC ;
   LC = LC+2 ;
##;
```
In this case, the invocation would look like:

\[
\text{STRUCTELEM: WORD X;}
\]

The rule according to which actual text is associated with corresponding DEFINE parameters is: The text of a parameter is all text which appears on the calling line from, but not including, the terminating symbol for the previous parameter up to, but not including, the first unbound occurrence of the terminating symbol for the parameter whose text is being associated. A symbol is said to be bound if it occurs between properly nested pairs of (), [], <>, or BEGIN END. The terminating symbol for a parameter is the same symbol which followed the formal parameter name in the DEFINE declaration. At invocation time, once the terminating symbol has been found it is simply disregarded.

In the example above the terminating symbol for the DEFINE name is a colon (colon), the terminating symbol for the parameter &TYPE is a blank, and the terminating symbol for the parameter &NAME is a semicolon (;). Thus, the general form of a legal invocation of this DEFINE is:

\[
\text{STRUCTELEM text : text-p1 text-p2 ;}
\]

The text between the DEFINE name (STRUCTELEM) and the terminator for the DEFINE name is discarded; the first terminator serves only to punctuate the start of the text of the first parameter.

A special form of DEFINE parameter exists which consists of just a single token from the input stream of the invocation. One signifies that a parameter is to have this property by naming it with the characters "&TOKEN" as the first six characters of the parameter name. Token parameters, as they have come to be called, do not have an associated terminating symbol (since the corresponding text is a single token).

We may rewrite DEFINE STRUCTELEM once more using token parameters:

\[
\text{DEFINE STRUCTELEM &TOKENTYPE &TOKENNAME =}
\text{&TOKENTYPE ARRAY &ID(STRUCTNAME . &TOKENNAME) [ ]=LC;}
\text{LC = LC+2}
\#
\#
\]

-12-
In the invocation:

STRUCTELEM WORD X;

the associated text of the parameters is:

&TOKENTYPE = "WORD"
&TOKENNAME = "X"

At this point we could easily declare our linked list elements as follows:

We write:

STRUCTNAME = "ELEM";
LC = 0;
STRUCTELEM WORD FLINK;
STRUCTELEM WORD BLINK;
STRUCTELEM WORD X;
STRUCTELEM WORD Y;

The compiler sees:

WORD ARRAY ELEM.FLINK[*]=0;
WORD ARRAY ELEM.BLINK[*]=2;
WORD ARRAY ELEM.X [*]=4;
WORD ARRAY ELEM.Y [*]=6;

DEFINES within DEFINES:

A DEFINE can also be used to declare another DEFINE. For example let us suppose that we would like to declare a number of DEFINES which, when invoked, will add one to a variable. That is, we would like to declare a number of DEFINES like the following:

DEFINE BUMP.A = A:=A+1 ##;
DEFINE BUMP.X = X:=X+1 ##;
etc.

We could write a DEFINE to do this for us:

DEFINE INCR &TOKENNAME; =
    DEFINE &ID("BUMP" . &TOKENNAME) =
        &TOKENNAME:=&TOKENNAME+1
            ##;  
        end of inner DEFINE
##;
    end of outer DEFINE
Then when we write:

    INCR A;

we get:

    DEFINE BUMP.A = A:=A+1 ##;

and when we write:

    BUMP.A;

we get:

    A:=A+1 ;

List Processing

With the aid of the following observation, we will be able to develop the technique for doing list processing with DEFINEs.

Observation:

    If a DEFINE of the form: DEFINE X(&P,&Q)=...##; is invoked as follows: X(A,B,C,D), then the parameter associations will be:

    &P = "A"
    &Q = "B,C,D"

We shall now extend the STRUCTELEM DEFINE in such a way that it can
be invoked with a list of field types and names separated by semicolons.

```
DEFINE STRUCTELEMS(&ELEMS); = 
   STRUCTELEM(&ELEMS;)
##;
DEFINE STRUCTELEM(&TYPE &NAME;&REST) = 
   &TYPE ARRAY &ID(STRUCTNAME . &NAME)[*] = LC;
   LC = LC+2;
   &IF NOT &EMPTY(&REST) &THEN
      STRUCTELEM(&REST)
   &FI
##;
```

If we now follow an invocation of the STRUCTELEMS DEFINE, the mechanism for accomplishing list processing will become apparent:

```
STRUCTELEMS(WORD FLINK;WORD BLINK);
Recursion level 0 of STRUCTELEM:
   &TYPE    = "WORD"
   &NAME    = "FLINK"
   &REST    = "WORD BLINK;"
Recursion level 1:
   &TYPE    = "WORD"
   &NAME    = "BLINK"
   &REST    = ""                &EMPTY(&REST ) = True
```

Data Structures

As a concluding example, we will show a set of DEFINEs which will declare the entire structure for us. We will write the DEFINEs in such a way that the fields can be either of type WORD or BYTE. The process will also generate a declaration of a TEXT variable whose contents will summarize the names of the fields and their respective types (for future reference). We will also arrange that a CELL gets declared in such a way that its value will give the size of the structure.

An invocation will look like this:

```
STRUCTURE ELEM
   (WORD FLINK;
    WORD BLINK;
```
WORD X;
WORD Y);

The invocation will generate these declarations:

WORD ARRAY ELEM.FLINK[*]=0;
WORD ARRAY ELEM.BLINK[*]=2;
WORD ARRAY ELEM.X [*]=4;
WORD ARRAY ELEM.Y [*]=6;
TEXT   ELEM.FIELDLIST[the right size] =
   "WORD FLINK;WORD BLINK;WORD X;WORD Y;";
CELL   ELEM.SIZE=8;

Proceeding to the example itself, we will wish to declare some compile time working storage for use in the DEFINEs:

CELL LC;
TEXT STRUCTNAME [64],
FIELDLIST [500];

location counter
name of the structure (ELEM)
list of field types and names
(WORD FLINK;WORD BLINK;...)

The DEFINE STRUCTURE:

DEFINE STRUCTURE &STRUCTNAME(&FIELDS); =
  LC = 0; zero location counter
  STRUCTNAME=&STRING(&STRUCTNAME); save structure name
  FIELDLIST=&STRING(); set FIELDLIST to empty string
  STRUCTELEMS(&FIELDS;) declare the structure elements
  (The declaration for DEFINE STRUCTELEMS is given below.)

  TEXT &ID(STRUCTNAME . "FIELDLIST")
  &LENGTH(FIELDLIST) =
  &STRING(FIELDLIST);

  CELL &ID(STRUCTNAME . "SIZE") = LC; save the size
  of the structure

##;
and DEFINE STRUCTELEMS:

```
DEFINE STRUCTELEMS(&TYPE &NAME; &REST) =
    &IF &CLASS(&TYPE) EQUALS &WORDCLASS word field?
    &THEN
        &IF LC &THEN LC=LC+1; &FI
        declare the array
        &TYPE ARRAY &ID(STRUCTNAME . &NAME)[*]=LC;
        add the type and name to list of fields
        FIELDLIST[*]=&TYPE+" "+&NAME+";
        bump location counter for next field
        &IF &CLASS(&TYPE) EQUALS &WORDCLASS &THEN
            LC=LC+2;
        &ELSE
            LC=LC+1;
        &FI
        recurse to the next element
    &ELSE
        &IF NOT &EMPTY(&REST) &THEN
            STRUCTELEMS(&REST)
        &FI
    ##;
```

Each recursion level of DEFINE STRUCTELEMS declares an ARRAY whose name is synthesized from the structure name and the name of the particular field. It adds the type of the field and the name of the field to the field list. It then steps the location counter by the amount appropriate to the type of the field and recurses to the next field definition in list processing fashion.

For the sake of emphasis, we call attention to the fact that the ARRAYS declared by the STRUCTURE DEFINE do not cause any storage to be allocated. The relations amongst the addresses of the arrays declared define the form or structure of a hypothetical section of memory. If at this point we wished to be able to declare objects whose structure has been defined via the STRUCTURE DEFINE, the saved field list would allow us to do so quite easily. If, for example, we wished to give a "declaration" of the form:

```
ELEM A;
```

in order to declare an object of "type" ELEM whose name is A, we need only write a DEFINE called ELEM which would generate the following declarations:

```
WORD ARRAY A.SPACE[(ELEM.SIZE+1)/2];
WORD A.FLINK = A.SPACE+ELEM.FLINK;
WORD A.BLINK = A.SPACE+ELEM.BLINK;
WORD A.X = A.SPACE+ELEM.X;
```
In fact, a general purpose DEFINE can be written which if given a structure name (ELEM) and the name of an object (A) will generate just those declarations by making reference to the field list of the structure (ELEM.FIELDLIST). The DEFINE ELEM would then simply invoke the general purpose DEFINE (which we may suppose is called DECLAREOBJECT) in the following way:

DECLAREOBJECT(ELEM,A);

Thus DEFINE ELEM would look like:

DEFINE ELEM &TOKENNAME; =
    DECLAREOBJECT(ELEM,&TOKENNAME);
    
The form of this DEFINE would be the same for any structure declared via the STRUCTURE DEFINE, the only specific difference between one structure and another being the name of the structure itself. Thus it would be quite a simple addition to make to the STRUCTURE DEFINE to let it declare the DEFINE which will, in turn, declare objects of the given structure type. We would simply add the following text to DEFINE STRUCTURE:

DEFINE &STRUCTNAME &TOKENNAME; =
    DECLAREOBJECT(&STRUCTNAME,&TOKENNAME);
    
With this addition (plus the writing of DEFINE DECLAREOBJECT) we can declare objects of any structure which is declared via the STRUCTURE DEFINE.
Measurements:

The metaprocessor, in general, processes more text than the compiler proper; also, the compiler, in general, processes more text than the programmer originally wrote. The ratio of the number of tokens processes by the metaprocessor to the number of tokens processed by the parser of the compiler (we call these last "syntactic items") is an indication of the amount of work the metaprocessor is doing. The ratio of the number of syntactic items to the number of tokens written by the programmer (we call these last coded items) is an indication of the amount of coding the programmer is spared as a result of letting the metaprocessor generate portions of his program for him.

For the inner level[3] of the ANTS[4] system, these ratios are:

\[
\begin{align*}
\text{total items/syntactic items} &= 232405/35974 = 6.5 \\
\text{syntactic items/coded items} &= 35974/21754 = 1.7
\end{align*}
\]

Concluding Remarks:

It would seem in order to address the restrictions of the metaprocessor, i.e. to speak about what it is NOT.

The metaprocessor can, in one sense, be said to be of Turing Machine power. That is, one can write a set of DEFINES which will accept an encoding of a Turing Machine together with its initial tape and simulate its action (subject only to finitude restrictions). It is, however, in the preparation of the "tape" that we find fairly severe restrictions. The preparation of the tape corresponds to the specification of parameters to DEFINES. The machine that associates actual text with DEFINE parameters is equivalent to a language which is essentially a regular language augmented by a parenthesis counting facility. The language is strictly less powerful than context free, which brings us to what the metaprocessor is NOT. The metaprocessor does not constitute an extensible language system in the conventional use of the term. There is nothing there that allows the syntax of the base language (which is context free) to be extended. What one CAN do, however, is extend the declarative power of the base language, since
any DEFINE whose invocation is of the form: DECLARATOR &TEXT ; can bring the power of a Turing Machine to bear in processing the actual text of the "declaration". Thus far, this has been found to be quite satisfactory for the development of operating systems.

References

1. Grothe, D., REFERENCE MANUAL FOR ILLIAC IV ASSEMBLER(ASK), Burroughs Corp., IL4-PM2, March 1969.
**Title and Subtitle**
PEESPOL: A Metaprogramming Language for Operating Systems

**Author(s)**
Center for Advanced Computation / David M. Grothe

**Performing Organization Name and Address**
Center for Advanced Computation
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

**Sponsoring Organization Name and Address**
Advanced Research Projects Agency
1400 Wilson Boulevard
Arlington, Virginia

**Type of Report & Period Covered**
Research report

**Abstracts**
See page 1 of CAC Report No. 124

**Key Words and Document Analysis**

17a. **Descriptors**
Language
compiler
macro
compile-time
system
implementation-language

17b. **Identifiers/Open-Ended Terms**

17c. **COSATI Field/Group**

18. **Availability Statement**
See Item 9

19. **Security Class (This Report)**
UNCLASSIFIED

20. **Security Class (This Page)**
UNCLASSIFIED

21. **No. of Pages**
20

22. **Price**
USCOMM-DC 14952-P72
REPORT DOCUMENTATION PAGE

<table>
<thead>
<tr>
<th>1. REPORT NUMBER</th>
<th>4. TITLE (and Subtitle)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAC Document No. 124</td>
<td>FEESFOL: A Metaprogramming Language for Operating Systems</td>
</tr>
</tbody>
</table>

5. TYPE OF REPORT & PERIOD COVERED
Research Report

6. PERFORMING ORG. REPORT NUMBER
CAC 124

7. AUTHOR(s)
David M. Grothe

8. PERFORMING ORG. REPORT NUMBER
CAC 124

9. PERFORMING ORGANIZATION NAME AND ADDRESS
Center for Advanced Computation
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

10. CONTRACT OR GRANT NUMBER
DAHC04-72-C-0001

11. CONTROLLING OFFICE NAME AND ADDRESS
Advanced Research Projects Agency
1400 Wilson Boulevard
Arlington, Virginia

12. REPORT DATE
December 1973

13. NUMBER OF PAGES
23

14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)
U. S. Army Research Office
Box CM, Duke Station
Durham, North Carolina 27706

15. SECURITY CLASS. (of this report)
UNCLASSIFIED

16. DISTRIBUTION STATEMENT (of this report)
No restrictions. Copies may be requested from the address in (9) above.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)
Approved for public release; distribution unlimited.

18. SUPPLEMENTARY NOTES
The findings in this report are not to be construed as an official Department of the Army position, unless so designated by other authorized documents.

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

- FEESFOL
- PDP-11 computer
- language
- compiler
- macro
- compile-time
- system
- implementation-language

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)
An operating system implementation language is presented which allows the system programmer to specify the disciplines to which he must adhere in order to produce a working system in a reasonable length of time. The language is oriented towards the Digital Equipment Corporation PDP-11 series of machines and has been acronymed FEESFOL (PDP Eleven Executive System Programming Oriented Language).