ALEC—A user extensible scientific programming language

R. B. E. Napper and R. N. Fisher*

Department of Computer Science, University of Manchester, Manchester M13 9PL

This paper describes the user-extensible high-level scientific programming language, ALEC, the initial proposals for which were first given in 1967. The more interesting properties of the language, in particular the extension mechanism, are illustrated in some detail. Extensibility is achieved by using 'formal' and 'informal' macros. A macro definition causes an extension to the compiler itself and thus macro calls are implemented in the same way as the base language instructions. This alleviates the need to preprocess the source text, gives considerable flexibility and power to the language and enables efficient object code to be produced.

(Received May 1974)

Initial proposals for the language, ALEC, were first given in 1967 by Napper (1967). The name has been altered to ALEC (an acronym for A Language with an Extensible Compiler) to avoid confusion with existing languages. The aim was to apply the full power of the Compiler Compiler (Brooker, 1963), to an ALGOL-like base language to provide an extensible language system that was one-pass and produced efficient object code. Extensions to the language were to be accomplished by the use of 'formal' and 'informal' macros. There was to be no pre-processing of source text: a macro definition would cause an extension to the ALEC compiler to be made and a macro call would be implemented in the same way as any other base language instruction.

The system was to be designed to facilitate language-format extensibility, as opposed to the data-type extensibility which is, for example, available in ALGOL 68, and it would enable both new syntax and new semantics to be added to the base language.

A suitable base language for ALEC has now been designed and a prototype compiler for the system written. The system has been implemented on an ICL 1906A computer and the compiler written using a revised version of the Compiler Compiler, Napper, 1973—henceforth known as RCC. This paper gives an overall description of the facilities available in the language system from the user's point of view. Description of the implementation is only given where necessary, for example to help in the explanation of 'informal' macros.

Many of the features described by Napper (1967) have been included in the current system. Those which have not can be included without much alteration to the current implementation.

The base language

The pre-design decisions which were taken on the base language of ALEC were as follows:

1. It should be adapted from a well-known language (to make it relatively easy to learn), except that
2. Procedure and macro formats were to be designed especially for ALEC, since this was to be the area of exploration in ALEC relative to conventional scientific languages
3. Only a small number of built-in data-types were to be allowed. As explained in Napper (1967), this was to prevent the number of new problems being tackled from growing uncontrollably
4. The language should be a one-pass language, to simplify the implementation of both conventional and unconventional facilities.

Accordingly, a study was made of FORTRAN, ALGOL 60, PL/I and ALGOL 68. FORTRAN was discarded as a possibility because its basic statements were considered too restrictive. ALGOL 68 was rejected primarily because it loses the distinction between an assignment statement and an expression. It is therefore difficult to break down a source program into a set of short statements or clauses, and this was considered essential to the implementation.

Of the remaining two possibilities, it was a subset of PL/I which was eventually adapted as the base language. This seemed to give the best compromise between high level of language and efficiency. Data types are restricted to REAL and INTEGER. However, the logical operators are retained, defined as operating on bit strings the size of the hardware representation of an INTEGER.

To illustrate the basic language and to compare it with PL/I proper, consider the following simple bubble-sort program written first in PL/I and then in ALEC. The program reads 10 integers into an array, sorts and then prints them in increasing order.

PL/I version:

```
BUBBLESORT: PROCEDURE OPTIONS (MAIN);
 DECLARE(A(10),I,K,DUMMY)FIXED BINARY;
 GET LIST(A);
 DO K = 9 BY -1 TO 1;
 DO I = 1 TO K;
 IF A(I) > A(I + 1)THEN
 DO; DUMMY = A(I); A(I) = A(I + 1); A(I + 1) =
   DUMMY; END;
 END;
 END;
 PUT LIST (A);
 END BUBBLESORT;
```

ALEC version:

```
BEGIN PROGRAM;
DECLARE (A(10),I,K,DUMMY)INTEGER;
READARRAY A;
DO K = 9 BY -1 TO 1;
DO I = 1 TO K;
IF A(I) > A(I + 1)THEN
DO; DUMMY = A(I); A(I) = A(I + 1);
  A(I + 1) = DUMMY; END;
END;
END;
WRITEARRAY A;
END OF PROGRAM;
```

*Department of Computational Science, University of St. Andrews, St. Andrews, Fife, Scotland.

Note that delimiter words are underlined and spaces ignored in ALEC. This is to minimise ambiguity problems which arise when procedures and macros are considered. Any standard convention could, in fact, be used to distinguish between delimiter words and identifiers, e.g. writing delimiters in upper case and identifiers in lower case. To be precise, there are no delimiter words in ALEC, only delimiter letters, and these must be distinguishable from identifier letters.

Program structure

A block consists of a **BEGIN**, followed (possibly) by a set of declarations, followed by a set of imperatives, followed by the matching **END**.

A procedure declaration is identical to a block except that the **BEGIN** is replaced by a procedure heading.

The main-program consists of a block, but starting with **BEGIN PROGRAM** and finishing with **END OF PROGRAM.**

Blocks and procedure declarations can be nested within each other as for PL/I or ALGOL 60.

The rules of scope are those of ALGOL 60 with the important restriction that the scope of a declaration starts at the point of declaration and not at the start of the block or procedure it applies to. This means in particular that mutually recursive procedures cannot be declared directly. This restriction enables the ALEC implementation to be one-pass.

Procedures

As with conventional languages, procedures are divided into two categories, viz. routines and functions. A routine call forms a separate imperative statement of the language; a function is called as an operand in an expression.

However, ALEC procedure formats are unconventional in that:

(*a*) the letters used in their names are delimiter letters, and not ordinary letters

(*b*) parameters can be embedded in a procedure name. Moreover, their position in the name helps define it.

A routine name consists of a set of delimiter letters, commas and parameter positions, starting with a delimiter letter. A function name must also include a single pair of brackets which surround all the parameters. This is to avoid syntactic ambiguity between a function call and the rest of the expression in which it is contained. A parameter specification consists of the characteristics of the parameter followed by its name, all enclosed in brackets.

Example 1:

```
ROUTINE ADD (REAL X) TO (REAL VARIABLE Y);
Y = Y + X;
RETURN;
END;
```

with corresponding call, for example,

```
ADD IDEN ADDRESS & TOP BIT MASK TO C(J);
```

Example 2:

```
REAL FN SUM ((REAL X) AND (REAL Y));
RESULT = X + Y;
END;
```

with call, for example, $C(J)$ = **SUM**($A + 4*B$ **AND** $C(J)$);

Example 3:

```
ROUTINE ADDT (REAL X) O (REAL VARIABLE Y);
  :
END;
```

which declares a routine with a different name from Example 1 above.

When required, e.g. as actual parameters, procedure names are specified with a dummy symbol (i.e. %) in each parameter position, e.g. **ADD%TO%** and **SUM(%AND%)** and **ADDT%O%** respectively for the above three examples.

Note that, because of this complication, the value of a function is specified using the imperative

RESULT = ⟨expression⟩;

instead of using the more conventional method of assigning to the function name. This imperative both sets the output value to the given ⟨expression⟩ and carries out a return.

Parameters

There are three characteristics which must be specified for each scalar formal parameter.

1. *Data type* i.e. **REAL** or **INTEGER**
2. *Method of implementation.* The user can choose from three classic methods:
   (*a*) **VALUE** An extension of the call-by-value mechanism of ALGOL 60.
   (*b*) **REF** The call-by-reference mechanism of FORTRAN.
   (*c*) **SUBST** The call-by-name or call-by-substitution mechanism of ALGOL 60.
3. *Input/output characteristic.* The user can specify one of three possibilities:
   (*a*) **EXPRESSION** The formal parameter is input only. Therefore the permitted syntax of the actual parameter is an ⟨expression⟩.
   (*b*) **VARIABLE** The formal parameter is both input and output. Therefore the permitted syntax of the actual parameter is a ⟨variable⟩.
   (*c*) **OUTPUT** The formal parameter is output only. Hence, again, the permitted syntax of the actual parameter is a ⟨variable⟩. This possibility is only relevant in association with **VALUE.**

[The call-by-value mechanism for ALEC is an extension of that for ALGOL 60 in that, not only is the formal parameter initialised to the value of the actual parameter on dynamic entry to a procedure, but the actual parameter is also reset to the value of the formal parameter on dynamic exit. If the parameter is 'input only', the second part of this is suppressed; if 'output only' the first part is suppressed.]

This range of specifications is designed to give the user a choice between power and efficiency. The default I/O characteristic is **EXPRESSION**; the default method of implementation is **VALUE** if the I/O characteristic is **EXPRESSION** or **OUTPUT, REF** if **VARIABLE.**

Array formal parameters have characteristic **REAL ARRAY** or **INTEGER ARRAY** with the choice of implementation **VALUE** or **NAME**—these being the same as in ALGOL 60.

Finally, procedures can be passed as parameters, again with the same effect as for ALGOL 60.

The extension mechanism

The language extension mechanism of ALEC is implemented using formal and informal macro definitions. These occur before the main program, i.e. before **BEGIN PROGRAM** and are considered global to it. Procedure declarations and blocks (but *not* other macro definitions) may be nested within macro definitions, just as in the main program. Thus a program with N macro definitions consists of $N + 1$ ordered, but otherwise independent, units, i.e. the N definitions followed by the main program. Once a macro has been defined, a call on it can occur in any following macro definition, or in the main program.

Macro definitions are logically an extension of the ALEC compiler as implemented using RCC. Therefore they are processed by the RCC implementation. When **BEGIN PROGRAM** is reached, control passes to the ALEC implementation, unwanted sections of RCC are discarded, and the source program is compiled by the ALEC program as for a conventional compiler, but in one pass.

The ALEC compiler works by recognising the source program in units of an ⟨alec piece⟩, or [*ALEC PIECE*] to use the RCC convention of referring to a syntactic element. An [*ALEC PIECE*] is in general a particular imperative statement, if clause, for clause or declaration. To each [*ALEC PIECE*] alternative, e.g. [*VARIABLE*] = [*EXPRESSION*] or **IF** [*EXPRESSION*]**THEN**, corresponds an RCC routine of the ALEC implementation to process it.

Compound statements, blocks and procedure declarations are not recognised as single pieces. Instead, the heading and **END** are recognised as separate pieces and they are matched in the corresponding RCC processing routines.

When a user gives a macro definition, he is effectively adding another alternative to the permitted basic syntax of ALEC, and defining its semantics in the RCC routine. This he does in the same way as the ALEC implementer wrote the ALEC compiler. Thus, in effect, the user is permitted to extend the language and its implementation indefinitely.

However, to use the full power of the mechanism the user must also know RCC and the detail of the ALEC implementation. Moreover, his syntax and corresponding RCC routine must be consistent with the ALEC implementation.

There is a hierarchy of possible ways of extending ALEC in this manner, ranging from the above situation down to relatively trivial extensions requiring minimal knowledge.

The minimum situation has been formalised so that the user need know no RCC or ALEC implementation, but learns a little extra ALEC. These are called 'formal' macros; the other forms are called 'informal' macros.

For technical reasons, the class [*ALEC PIECE*] contains the alternative [*OPEN ALEC PIECE*]. It is to this class of extra pieces that the user can add, thus allowing the pieces of basic ALEC in a source program to be interspersed with pieces of non-basic syntax. There is a further expansion point in the syntax of an ALEC [*EXPRESSION*]. An alternative syntax for [*OPERAND*] is an [*OPEN FUNCTION ALEC PIECE*]. Here the user can insert syntactic forms of his own choosing, thus creating function macros. However, there are more restrictions on his syntax and semantics in this case, since these macros are embedded in expressions.

Formal macros

Syntactically, a formal macro definition is identical to a procedure declaration, except that the word **OPEN** appears at the beginning of the heading.

The effect of a macro definition is that whenever there is a call on the macro, the instructions in the body of the macro are compiled in-line. The effect on the run time program is the same as if the corresponding procedure had been used, but there is now no subroutine entry and exit sequence.

Note that the sequences to carry out actual-formal correspondence are still compiled, e.g. consider:

```
OPEN ROUTINE NEXT
  (INTEGER VARIABLE VALUE V);
V = V + 1;
END;
```

Given a call **NEXT** *J*, the effect is the same as if the programmer had written:

```
BEGIN; DECLARE V INTEGER;
V = J; V = V + 1; J = V;
END;
```

To exploit the full power of macros, further facilities are provided for procedure headings in the case of formal macros. The most important are as follows:

1. A further alternative implementation characteristic for formal parameters is provided, viz **TR SUBST.** 'True substitution' means that for every appearance of the formal parameter, the actual parameter is physically substituted in the body of the macro.
   Thus given:
   ```
   OPEN ROUTINE NEXT
     (INTEGER VARIABLE TRSUBST V);
   V = V + 1;
   END;
   ```
   for call **NEXT** *J*, the effect is as if the programmer had written
   ```
   BEGIN; J = J + 1; END;
   ```
   This, therefore, eliminates the instructions associating actual and formal parameters. Note that, in ALEC, block entry and exit instructions are only compiled if necessary, i.e. only if the block contains any dynamic array declarations.
2. The type characteristic can be omitted, so that a macro is defined for either type of parameter. Thus **NEXT** can be defined for both **REAL** and **INTEGER** variables, i.e.
   ```
   OPEN ROUTINE NEXT (VARIABLE TRSUBST V);
   V = V + 1;
   END;
   ```
3. A declaration is provided to declare the type of a scalar or array to be the same as that of an already-declared scalar or array. Usually the latter is a formal parameter without a given type characteristic, e.g.
   ```
   OPEN ROUTINE INTERCHANGE
       (VARIABLE TRSUBST V1)
         AND (VARIABLE TRSUBST V2);
     DECLARE DUMMY TYPE AS FOR V1;
     DUMMY = V1; V1 = V2; V2 = DUMMY;
     END;
   ```
   Thus for **REAL** array *A*, and call
   ```
   INTERCHANGE A(J) AND A(J + 1);
   ```
   the effective sequence is:
   ```
   BEGIN; DECLARE DUMMY REAL;
   DUMMY = A(J); A(J) = A(J + 1);
     A(J + 1) = DUMMY;
   END;
   ```
   Thus the interchange operation is effectively defined for all four combinations of type. Note again that there are no hidden instructions compiled as a result of the macro call.

Formal macros are particularly useful in encouraging the user to define commonly occurring statements or short sequences in a more descriptive manner without losing any efficiency in the execution of his program. However, when the set of instructions in the macro body gets large, the gain in efficiency may become small in proportion to the time taken to obey the macro body. It may then be better to cut down the size of the object code by using a conventional procedure.

Functions can be declared as formal macros. This has the same implications as for a routine macro, except that the code for the set of instructions yielding the value is embedded in the evaluation of the expression.

Informal macros

It will have been noted that the form of language extension of formal macros is limited. The syntax of the extension is limited to that of the closed procedure call, and the semantics are limited to the set of the existing instructions of ALEC. Note, however, that they can include informal or formal macro calls.

Informal macros allow complete freedom in the syntax of extensions, subject to following conventions that are advisable to minimise ambiguity problems, e.g. using delimiter letters not identifier letters as fixed letters in a format. They also allow full freedom in semantics subject to being compatible with the existing ALEC system.

The extensions are made in RCC. But because RCC is itself a high-level and extensible systems programming language, a 'package' of RCC instructions defined specially for ALEC by the implementer can be used to allow programmers to exploit the system with minimum working knowledge of RCC.

The following examples illustrate the use of informal macros with gradually increasing power.

Example 1:

```
        ROUTINE
[OPEN ALEC PIECE]: NEXT [VARIABLE a];
        BEGIN MACRO BLOCK
        CALL VARIABLE V = [VARIABLE a] BY
          TR SUBST
        COMPILE: V = V + 1;
        END MACRO BLOCK
```

This first example shows the basic construction of many informal macros. It is, in fact, the informal macro version of

NEXT(VARIABLE TR SUBST *V*) ,

and has exactly the same effect.

The first line is the RCC routine heading, which automatically adds another alternative syntactic form to the subclass [*OPEN ALEC PIECE*] of [*ALEC PIECE*]. The syntactic form is **NEXT**[*VARIABLE*]. From now on, any string satisfying it can be included as a piece of ALEC source program and, when the string is to be compiled, control will be passed to this routine.

In RCC terminology [*VARIABLE a*] is a formal parameter of type 'phrase variable'. Its value can only be a symbol string satisfying the syntactic class [*VARIABLE*]. On entry to the routine it will be set to the corresponding string in the actual macro call, e.g. '*J*' for **NEXT** *J*, or '*A*(*J*)' for **NEXT** *A*(*J*). Its effect is that this string is substituted wherever a reference occurs in the body of the routine, i.e. in line 3.

The body of the RCC routine starts with a call on the system routine, *BEGIN MACRO BLOCK*, which, together with *END MACRO BLOCK*, organises the block surrounding the in-line code of the macro.

Then follows an instruction of the form

```
CALL[TYPE?][IO CHARACTERISTIC][NAME] =
    [ACTUAL   PARAMETER]BY[IMPLEMENTATION
                          CHARACTERISTIC],
```

in general one for each ALEC formal parameter. This system routine carries out the association of the actual parameter, e.g. '*J*' or '*A*(*J*)', with the formal parameter name used in the body of the in-line code, i e '*V*' For a **TR SUBST** parameter characteristic, the routine performs the operation of 'declaring' the formal parameter and associating with it the symbol string which constitutes the actual parameter For a conventional characteristic, e g **REF**, it combines the two operations for a conventional routine of compiling code to set the value of the formal parameter from the actual parameter when compiling a cue, and declaring the formal parameter when compiling the routine heading.

This is followed by an instruction of the form *COMPILE*: [*ALEC PIECE*] or, in general, a set of such instructions, one for each [*ALEC PIECE*] in the body of the macro. The effect of this is to call the corresponding [*ALEC PIECE*] compiling routine.

Finally *END MACRO BLOCK* is called.

In fact, formal macros are implemented by doing an automatic transformation from ALEC form to the corresponding informal macro form in true RCC as indicated above. The parameters are dealt with by replacing them in the heading by the implied RCC syntactic element [*VARIABLE*] or [*EXPRESSION*], and putting in the corresponding CALL statements after *BEGIN MACRO BLOCK*.

Note that it is not always suitable to use true substitution for macros. If the formal parameter occurs a number of times in the macro body, and the actual parameter is, say, an array reference or a complicated expression, it may be expensive in instructions compiled or execution time to re-evaluate it on each occurrence. In this case, one of the other methods of implementation should be used. It is even possible to allow the ALEC compiler to choose between **TR SUBST** and another method by programming an RCC sequence to inspect the actual parameter, e.g. implementing by **TR SUBST** only if it is a scalar.

Note that the RCC heading allows any symbols as the delimiter symbols. Thus if the user wanted to use *J* + 1 instead of **NEXT** *J* as an imperative, he could instead use the routine heading:

```
                ROUTINE
    [OPEN ALEC PIECE]: [VARIABLE a] + 1;
```

He would, however, have to keep a wary eye on ambiguity problems!

Example 2:

A further level of complexity in informal macro definitions is to introduce more complicated syntactic forms using the RCC facilities for the automatic definition, recognition and processing of symbol strings.

For example, consider a call **REVERSE ELEMENTS OF** *A* **FROM** *START* **TO** (*N* + 1)/2. This reverses the order of the elements of an array, e.g. *A*, between given limits, e.g. *START* and (*N* + 1)/2. It may, however be considered convenient to have a special form for **FROM** 1, e.g. leaving it out or replacing it by **UP**, e.g. **REVERSE ELEMENTS OF** *B* **UPTO** *N*.

The extra syntax required is defined and a prelude to the standard macro body is written to transform the non-standard syntax into conventional parameters e.g. [*START*] into an [*EXPRESSION*].

```
e.g.                CLASS
      [START] = UP, FROM [EXPRESSION]
                   ROUTINE
[OPEN ALEC PIECE]: REVERSE ELEMENTS OF
     [IDENTIFIER a][STARTs]TO[EXPRESSION c];
  IF [STARTs] = UP: SET [EXPRESSION b] = 1:
     OTHERWISE: RESOLVE [STARTs] INTO FROM
     [EXPRESSION b]
  BEGIN MACRO BLOCK
  CALL ARRAY A = [IDENTIFIER a] BY TRSUBST
  CALL EXPRESSION START = [EXPRESSION b] BY
     TRSUBST
  CALL EXPRESSION FINISH = [EXPRESSION c] BY
     VALUE
  COMPILE: DECLARE J INTEGER;
  COMPILE: DO J = START BY 1 WHILE
     (J < FINISH - J + START);
  COMPILE: INTERCHANGE A(J) AND
     A(FINISH - J + START);
  COMPILE: END;
  END MACRO BLOCK
```

Note here the use of macro call **INTERCHANGE,** just as if it was a routine call or built in to the language. This assumes that its macro definition has occurred previously in the set of macro definitions.

Only basic knowledge of the RCC language processing machinery is required to gain considerable flexibility in instruction formats. For example, library routines with a large number of parameters, many with clear default options, can be defined so that default parameters can be left out, and those retained clothed in self-descriptive language and maybe allowed in any order.

Example 3:
An extension of 2 is to use the control statements of RCC to operate on the *COMPILE* instruction to allow a variable number of instructions to be compiled depending on the information given in the heading. This is particularly useful in allowing a library routine to 'tune' the code compiled depending on what and how much information the user has given.

The simplest example of a variable number of instructions being compiled is where there is a variable list of parameters. For example consider the statement

```
   SET ARRAY STARTING VALUE = 25, B − 3·142, 0,
      ((X + Y)/SQRT(X*Y));
```

This initialises the given array *STARTING VALUE* to the set of given values, e.g. four as shown, starting at index position 1.

```
                         CLASS
[EXPRESSION LIST] = LIST OF [EXPRESSION],
                                      SEPARATED BY,
                        ROUTINE
[OPEN ALEC PIECE]: SET ARRAY [IDENTIFIER a] =
     [EXPRESSION LIST b];
  BEGIN MACRO BLOCK
  CALL ARRAY A = [IDENTIFIER a] BY TRSUBST
  index = 0
  FOR EACH i ON LIST [EXPRESSION LIST b]:
  {index = index + 1
  VALUE OF [INTEGER p] = index
  COMPILE: A([INTEGER p]) = [EXPRESSION b(i)];}
END MACRO BLOCK
```

On the above example, this would generate the in-line sequence:

```
BEGIN;
STARTING VALUE (1) = 25;
STARTING VALUE (2) = B − 3·142;
STARTING VALUE (3) = 0;
STARTING VALUE (4) = (X + Y)/SQRT (X*Y);
END;
```

Example 4:
A new level of complication is to provide facilities that cannot be reproduced in terms of the existing language, in particular with the user planting in-line machine code.

For example, the basic compiling routine of 1900 ALEC compiles the result of an expression evaluation into accumulator 5. A routine available to the user is *COMPILE CODE FOR EXPRESSION*[*EXPRESSION*] and conditional routines are also available to check its operation and the resulting type. It is also given that register 2 is the stack front register of 1900 ALEC and register 3 is available for temporary use.

Consider then a macro to set the contents of an absolute store address to an integer value e.g. $\mathbf{C}(3000 + J) = P + M - 1$; where the value of the second expression, e.g. $P + M - 1$, is placed in the location with address given by the first, e.g. $3000 + J$.

```
                        ROUTINE
[OPEN ALEC PIECE]: C([EXPRESSION v]) =
                                  [EXPRESSION e];
  BEGIN MACRO BLOCK
  COMPILE CODE FOR EXPRESSION [EXPRESSION v]
  IF ERROR IN EXPRESSION; OR IF EXPRESSION
    REAL:
  MAP: ERROR $ IN $ FIRST $ EXPRESSION; GO AND
    terminate
  PLANT 010, 5, 0, 2
  PLANT 101, 2, 1, 0
  COMPILE CODE FOR EXPRESSION [EXPRESSION e]
  IF ERROR IN EXPRESSION; OR IF EXPRESSION
    REAL:
  MAP: ERROR $ IN $ SECOND $ EXPRESSION; GO AND
    terminate
  PLANT 103, 2, 1, 0
  PLANT 000, 3, 0, 2
  PLANT 010, 5, 0, 3
terminate: END MACRO BLOCK.
```

This macro by-passes formal parameters and operates direct on the two ALEC expressions.

Having called the system routines to compile and check the first expression, it plants machine instructions to store its value at the stack front and update the stack pointer. It then calls the routines to compile and check the second expression. Finally it plants machine instructions to decrement the stack pointer, retrieve the destination address from the stack front, and store the second value at the address given by the first.

The MAP instruction is the standard monitoring routine to identify the position in the source program and print the subsequent caption.

A similar matching macro can be defined to extract the value in an absolute store location. This is more conveniently expressed as an integer function macro i.e. [*OPEN FUNCTION ALEC PIECE*]: **C**([*EXPRESSION v*]) with call, e.g. $P = \mathbf{C}(3000 + J)$ to recover the above value.

Given these two macros, the user could operate on his own data organisation in a separate store area. He could define further macros or procedures to set up a sublanguage operating on it.

Scope and textual level

ALEC is a one-pass language and therefore all information must be declared before it is used at any level. Calling the source program string which appears between the **BEGIN PROGRAM** and **END OF PROGRAM** statements the 'static text', it follows that the declaration of any information must physically appear in the static text before any reference to it. However, this is not so for macros. The point is that a macro is not obeyed at compile time until a call for it is encountered either in the static text or from within another macro which is currently being obeyed. Thus, non-local reference to information can be made within a macro definition *so long as* every *call* of the macro appears within the scope of that information.

In a conventional language such as ALGOL 60, the concept of textual level is a static one. It is possible to determine the level of a particular block or procedure declaration simply by looking at its position in the source program. However, the macro facility in ALEC introduces a dynamic element into this concept. To illustrate this, consider the program:

```
OPEN ROUTINE INCREMENT GLOBAL ARRAY BY
     (TR SUBST A);
  DECLARE I INTEGER;
  DO I = 1 TO 10; S(I) = S(I) + A; END;
  END;
OPEN ROUTINE ADD (TR SUBST B) AND RANDOM
     NUMBER TO GLOBAL ARRAY;
  DECLARE I INTEGER;
  GENERATE NEXT RANDOM NUMBER I;
  INCREMENT GLOBAL ARRAY BY B + I;
  END;
BEGIN PROGRAM;
DECLARE (I, S(10))INTEGER;
```

```
INITIALISE S;
   :
INCREMENT GLOBAL ARRAY BY 3;
   :
  BEGIN;
   :
  INCREMENT GLOBAL ARRAY BY 5;
   :
  ADD I AND RANDOM NUMBER TO GLOBAL
    ARRAY;
   :
  END;
   :
END OF PROGRAM;
```

(N.B. The macro **INITIALISE (ARRAY NAME** *A*) and **GENERATE NEXT RANDOM NUMBER (VARIABLE** *V*) are assumed to be predefined.)

The macro **INCREMENT**... is called three times in the program. Two calls occur directly at static levels 1 and 2 of the main program; the third call occurs inside the call of **ADD**..., which is at static level 2 of the main program. Thus it can be seen that the three occurrences of the body of macro **INCREMENT** occur at different textual levels, levels 2, 3 and 4 respectively relative to the main program.

In general, therefore, the textual level of a macro body will vary from call to call of the macro, cf. the level of a conventional procedure body, which is independent of the position of any call of the procedure.

Entry to a block or procedure declaration at compile time in ALEC is thus considered to be entry to a new 'static' textual level (as with conventional languages), but entry to a macro is considered to be entry to a new 'dynamic' textual level.

It is convenient, therefore, to think of textual level as a two-dimensional concept: one dimension giving the number of dynamic levels currently active back up to the level of static text; the other giving the number of static levels currently active within each dynamic level.

A dynamic level is the set of ALEC source instructions effectively produced by a particular formal or informal macro. If that set includes procedure declarations or blocks, then further static levels are formed relative to the dynamic level of the macro. If a macro contains another macro call, a new dynamic level is formed relative to the dynamic and static level at the point of the call.

For example, the textual level of the **DO** statement in the first call on macro **INCREMENT**... is static level 1 in dynamic level 2 (i.e. the level of **INCREMENT**...), which is in static level 1 of dynamic level 1 (i.e. the main program). But the textual level of **DO** inside the call of **ADD**... is static level 1 of dynamic level 3 (i.e. **INCREMENT**...), which is in static level 1 of dynamic level 2 (i.e. **ADD**...), which is in static level 2 of dynamic level 1 (i.e. main program).

The above example also illustrates a problem arising from the use of the **TR SUBST** mechanism, viz that of encountering the same name from different levels in an expression. Consider the macro call

ADD *I* **AND RANDOM NUMBER TO GLOBAL ARRAY**;

which is at static level 2 of the main program. The text at dynamic level 2 after true substitution of *I* for *B* is effectively:

```
DECLARE I INTEGER;
GENERATE NEXT RANDOM NUMBER I;
INCREMENT GLOBAL ARRAY BY (I) + I;
```

Here, the first *I* in (*I*) + *I* derives from the actual-formal association of *B* with *I*. Therefore the expression (*I*) + *I* refers to *I* from **DECLARE** (*I*, *S*(10)) **INTEGER**; and **DECLARE** *I* **INTEGER**; respectively. The problem becomes one level worse when the call of macro **INCREMENT**... is considered. In the body of this macro is the imperative

$$S(I) = S(I) + A;$$

but *A* is a **TR SUBST** formal parameter whose corresponding actual parameter is '*B* + *I*' which, as stated above, is effectively '(*I*) + *I*'. Hence the imperative becomes

$$S(I) = S(I) + ((I) + I);$$

after substitution, where the three *I*'s in the right-hand-side expression refer to three different scalars.

In general, when processing a macro body at dynamic textual level '*n*', any name occurring within it may refer to one of at most '*n*' different scalars, i.e. one per dynamic level.

Note that *S*, which is not declared in **INCREMENT**..., **ADD**..., or static level 2 in the main program, refers direct to **DECLARE** (*I*, *S*(10)) **INTEGER**; in static level 1 of the main program.

It was the sorting out of these problems of scope that was the major addition to the otherwise conventional implementation of the basic language.

It is also the feature of ALEC that would seem to present the biggest problem in implementing macros by pre-processing using a macro-generator (see Napper, 1967). The solution using RCC, as well as being highly efficient, is relatively straightforward.

Conclusions

It can be seen that formal macros enable extensions to the language to be made conveniently, since they themselves are written in ALEC. However, as has been illustrated in the preceding examples, it is with informal macros where the power and flexibility of the language lies. The examples given have all been simple. The possible variety and sophistication of their use is considerable and much work remains to be done to explore their potential.

The extension facilities slow down the operation of the compiler, but not dramatically. Of course, they tend to improve run time efficiency as the associated code is in general in-line.

A one-pass implementation of ALEC seems to be the only practical possibility. Macro definitions must be implemented one-pass to be consistent with RCC machinery. Furthermore, the problems of scope which arise with, say, **TRSUBST** parameters would seem to become much more difficult to overcome with a two (or more)-pass implementation, compared with the present relatively straightforward solution (given standard RCC machinery).

The ALEC system as described in this paper is, in effect, only a prototype. Enhancements which would appear in a subsequent version may for example, include:

(*a*) some of the other facilities mentioned by Napper (1967), e.g. the DUAL procedure declaration and the open-option facility for formal parameters

(*b*) allowing macro definitions in the block structure of an ALEC program, thus allowing the scope rules to apply to the names of macros as well as procedures.

The machine code for the prototype ALEC compiler occupies approximately 25K of 24-bit store. However, when compiling macros, the compiler requires the full RCC system seated behind it. This brings its size up to 65K, with a further minimum of 10K approximately required for working store. When the compiler starts to process an ALEC source program, most of RCC can be discarded, and then the system requires a minimum of about 45K to run in. The store required for macros must be added to the figures above. For technical reasons, this can be quite expensive if they contain many statements.

Compared with writing a compiler with a similar specification, but without macros, there is little extra code in the implementation, say, 20 per cent. Nearly all of this is taken up by the secondary routines available for use in macros, e.g. *BEGIN MACRO BLOCK*, *END MACRO BLOCK*, etc., the rest being used in implementing **TR SUBST** parameters and sorting out the associated problems of scope. The problem of the indefinite extension of syntax is handled automatically by RCC. Again, the hierarchy of use of routines of the ALEC implementation in informal macros is facilitated by the fact that the implementation is written in a high-level (but efficient) language, which itself has macros and free format of routines.

Acknowledgements
The authors would like to thank the staff at the Atlas Computer Laboratory at Chilton for the facilities which were made available and also the Science Research Council for supporting the work.

References

Napper, R. B. E. (1967). Some Proposals for SNAP, A Language with Formal Macro Facilities, *The Computer Journal*, Vol. 10, No. 3, p. 231.

Napper, R. B. E. (1973). *RCC reference manual*, Department of Computer Science, University of Manchester.

Brooker, R. A. *et al.* (1963). The Compiler Compiler, *Annual Review in Automatic Programming*, Vol. 3, (ed. Goodman) Oxford: Pergamon Press.