A system program generator

D. Morris, I. R. Wilson and P. C. Capon

Department of Computer Science, The University, Manchester

This paper describes a self expanding program generator aimed at the general area of system programming. A syntax directed scan is used for statement recognition. The set of statements is extended by additions to the syntax tables.

(Received May 1968)

This system is intended for any application where close control is required over the code compiled and the storage organisation. System programming is in this category. The basic instructions have one to one correspondence with machine orders, being deliberately made machine dependent for reasons of efficiency. However, there is a macro facility which enables the user to define the syntax and semantics of his own instructions. This is in some ways similar to the facility for defining [AS] formats in the Compiler Compiler (Brooker *et al.*, 1963). Some macros of general application are predefined within the basic system. Careful choice of macros in programs of universal interest (e.g. compilers) should result in a high degree of machine independence. The current implementation is on Atlas but a version for the I.C.T. 1900 is under development. It is felt that the system is a powerful programming tool and that through its use ideas for more formal systems programming languages will evolve. However, we realise that the system may not be easy to learn and that it is little more than an assembly language if used unimaginatively.

Symbols and editing conventions

The Atlas implementation will accept the input from either cards or 7-track tape, but backspace is not permitted on the latter. Some of the required symbols are not available and are represented by symbol pairs, e.g. $\neq$ is represented by /= and $\not\equiv$ by —=. This description observes the conventions of the Atlas implementation.

'Newline' usually terminates instructions, although continuation may be indicated by a π symbol in column 80 of a card and immediately before the newline on paper tape. In this case the π and newline are both edited out.

Comments are introduced by the symbol pair :: and these symbols and the rest of the line are always edited out. Continuation cannot be used on the same line as a comment. Columns 1 to 8 on cards are also edited out, but a check is made that the number they represent is not less than the one for the previous card.

Erase symbols are edited out but spaces are not (although multiple spaces and tabs are condensed to single spaces) and can therefore be specified in the syntax as separators. Redundant spaces are ignored by the scanning routine.

The structure of a program

Programs have a textual block structure similar to ALGOL, thus any program can be enclosed in any other program. This block structure delimits the scope of names (e.g. labels) and macros. The scope of a label is the block in which it is defined and any enclosed block in which it is not redefined Thus non-local reference to labels which are not redefined more locally is permitted. The scope of macros and names used for syntactic elements (synes) is similar but it extends into sub-blocks up to the textual point where they are redefined. They must also be defined before any instructions (i.e. macros) whose syntax they describe are used.

Because the basic language contains no dynamic declaratives, control transfers across block boundaries are not treated specially, but entry to a block is normally at its beginning since its internal labels are inaccessible. It is felt that this restriction will improve readability and will be of value when dynamic declarations are introduced as macros. A block is entered after executing the preceding instruction unless this is a control transfer, and no instructions are assembled for an 'END'.

Using Backus Naur Form (Naur, 1963) the syntax of a program is as follows:

⟨PROGRAM⟩ ::= ⟨STATEMENTS⟩ ENTER ⟨EXPR⟩ ⟨NL⟩
⟨STATEMENTS⟩ ::= ⟨STATEMENT⟩|⟨STATEMENTS⟩ ⟨STATEMENT⟩
⟨STATEMENT⟩ ::= ⟨BASIC DECLARATIVE⟩| ⟨INSTRUCTION⟩
⟨BASIC DECLARATIVE⟩ ::= ⟨MACRO DEFN⟩| ⟨SYNE DEFN⟩|⟨COSYNE DEFN⟩
⟨INSTRUCTION⟩ ::= ⟨LABEL⟩|⟨BASIC INSTRUCTION⟩|⟨BLOCK⟩|⟨MACRO⟩
⟨BLOCK⟩ ::= BEGIN ⟨NL⟩⟨STATEMENTS⟩ END ⟨NL⟩
⟨LABEL⟩ ::= ⟨NAME⟩ :

The value associated with a name used as a label is the usual one of the run time address of the next instruction.

⟨NAME⟩ ::= ⟨LETTER⟩|⟨NAME⟩⟨NAME SYMBOL⟩
⟨LETTER⟩ denotes any upper or lower case letter
⟨NAME SYMBOL⟩ ::= ⟨LETTER⟩|⟨DIGIT⟩|.
⟨DIGIT⟩ denotes any decimal digit
⟨NL⟩ denotes 'newline'
⟨EXPR⟩ is defined later, in the above context it specifies the entry point of the program.

An example of a program structure is

```
    START: - - - -  :: THIS IS THE ENTRY POINT
           - - - -
           BEGIN
    L1: L2: - - - -
           - - - -
INT. STEP: BEGIN
           - - - -
           - - - -
           END
   FINISH: END
           - - - -
           ENTER START
```

The basic instructions are highly machine dependent and we shall discuss first the basic declaratives. Although there are only three of these an important use of the macro facility will be to define further declaratives.

Syntactic elements

Syntactic elements are defined by means of metalinguistic formulae similar to those which have been used above. Their function is in condensing the syntactic part of macro definitions and steering the scanning algorithm which identifies statements. The names used for synes have the same structure as label names and must be distinct from all label names within their scope. Syne definitions differ from the notation of the Backus Naur form, used earlier, in the following ways:

(1) Stylistic difference—The word SYNE must precede each definition and ::= is replaced by =

(2) Differences in ordering—The order of alternatives and of elements within alternatives are both different. Syne formulae are used by a left to right scanning algorithm which requires that:

(*a*) Any alternative which is a stem of another comes after it.

(*b*) If one alternative is a special case of another it must come first.

(*c*) In recursive definitions there must be at least one leftmost element not recursive.

(3) Metalinguistic Bracketing—several alternatives may be specified as an element of another by enclosing them in square brackets. This device has been used by Iverson (1964), Burkhardt (1965) and others.

For example the definition of ⟨STATEMENTS⟩ given earlier might take the form

SYNE ⟨STATEMENTS⟩=⟨STATEMENT⟩ [⟨STATEMENTS⟩|⟨NULL⟩]

Where ⟨NULL⟩ indicates an empty string.

For both semantic and syntactic reasons it is convenient to be able to interrupt the scanning algorithm at defined points and execute code provided by the user. This is achieved by inserting an element into the syntax which is apparently a syne but is in fact defined as the name of a section of code (COSYNE) to be executed when the scanning algorithm reaches that point. A cosyne definition has the form,

COSYNE ⟨NAME.IN.SHARP.BRACKETS⟩⟨NL⟩
⟨STATEMENTS⟩ END ⟨NL⟩

where the ⟨STATEMENTS⟩ are treated as a block. To make this facility more flexible the cosyne routine may have one numeric parameter the value of which is defined explicitly wherever the cosyne is used thus:

⟨cosyne name (numeric parameter)⟩

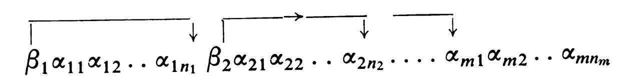
One example of the use of a cosyne is illustrated by the cosyne ⟨SYM⟩ which is preloaded into the system. This checks that the next symbol to be scanned is equal to its parameter. Thus ⟨SYM((A))⟩ is equivalent to A.* For the metalinguistic symbols ⟨ ⟩ [ ] | which cannot be used to represent themselves, it is essential to use ⟨SYM((⟨))⟩, ⟨SYM((⟩))⟩, ⟨SYM(([))⟩, ⟨SYM((]))⟩, ⟨SYM((|))⟩ or ⟨⟨⟩, ⟨⟩⟩, ⟨[⟩, ⟨]⟩, ⟨|⟩, which are acceptable shorthand for these five cases. The conventions which a cosyne routine must observe are described later.

A description of the internal form of syne definitions and the scanning algorithm will clarify the above. The general form of the syne definition is

$$\text{SYNE}\ \langle\text{NAME}\rangle = \alpha_{11}\alpha_{12}\ldots\alpha_{1n_1}|\alpha_{21}\alpha_{22}\ldots\alpha_{2n_2}|\ldots|\alpha_{m1}\alpha_{m2}\ldots\alpha_{mn_m}$$

where m is the number of alternatives.
n_i is the number of elements in alternative i.
α_{ij} is element j in alternative i,

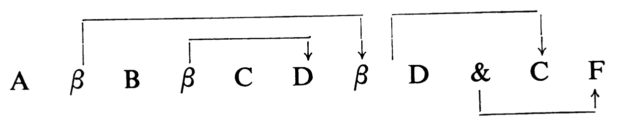
The α's are symbols, synes, cosynes or compound elements comprising further sets of alternatives within sharp brackets. A branch element (β) is introduced internally in order to represent the definition linearly. It indicates the position of the next alternative if the scan fails to match on subsequent elements. Thus the internal form of a syne definition is


$\beta_1\alpha_{11}\alpha_{12}\ldots\alpha_{1n_1}\ \beta_2\alpha_{21}\alpha_{22}\ldots\alpha_{2n_2}\ldots\ \alpha_{m1}\alpha_{m2}\ldots\alpha_{mn_m}$

where β_i is the address of β_{i+1} (or α_{m1} if $i = m - 1$) Any α which is compound will have the same structure provided it is the last element of an alternative. When it is not the last element of an alternative, an additional element is added at the end of the first $m - 1$ alternatives of the compound element, to link them to what follows. It is a merge element (&). Note that, for this purpose, the element ceases to be the last element if an & is added. For example,

SYNE ⟨X⟩ = A[B[C|D]|[D|C]F]

meaning ABC, ABD, ADF or ACF would be represented by

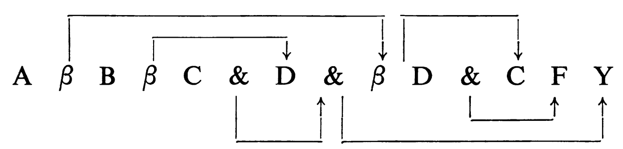


If a further element Y(say) were added to the definition of ⟨X⟩ giving,

SYNE ⟨X⟩ = A[B[C|D]|[D|C]F]Y

* (A) is a form of numeric constant, more fully described in a later section. Its value is the internal code for A.

its internal form would be



For the purpose of scanning therefore a syne definition consists of a string of five types of element which cause the scanning routine to behave as follows:

(1) SYMBOL—The action of the scan in this case is to compare the symbol in the source string with that in the syne. If they are the same the pointers in both are advanced and the next element is examined. If they differ all points are reset from the branch stack and the next symbol is examined. An exception to this occurs if the source string contains a space. In this case the space is skipped and the comparison is repeated.
(2) BRANCH (*β*)—All relevant points are stacked in case the alternative branch has to be explored.
(3) SYNE—When a syne is encountered the position in the current syne definition is noted and the syne string pointer is reset to the new syne definition. The scan is then continued.
(4) COSYNE—Control is transferred to the associated subroutine.
(5) MERGE (&)—The pointer in the syne is moved to the new position.

Obviously in the above no record is kept of the way in which the source string fits the syne definitions. Also there is no way of recognising the end of an alternative. Both these functions are performed by cosynes. A terminal cosyne is required at the end of every alternative in a syne definition. This should generate the required record and end by transferring control to a specified point in the scanning routine, where the stack will be adjusted and the scan continued in the syne definition one level up. Three terminal cosynes are built into the system, they may be used as models for others. The first is UP which causes the scan to continue at the syne definition one level up and generates no analysis record. The other two, VAL and EXVAL generate an analysis record which has a tree structure similar to the one produced by the Compiler Compiler (Brooker *et al.*, 1962). VAL, if used consistently, will produce a structure identical to that of the Compiler Compiler where the parameter supplied to VAL represents the category number of the alternative it terminates. EXVAL, will extend the value into a triplet of words but otherwise generates the same kind of tree structure. Here one word is the value of the parameter and the other two are the addresses of the first and last symbols in the current line, which have been identified as matching the syne. In the first example quoted above terminal cosynes should be inserted as follows

SYNE ⟨X⟩ = A[B[C⟨VAL(0)⟩|D⟨VAL(1)⟩]|
[D|C]F⟨VAL(2)⟩]

It is interesting to note that when compound elements appear in non-terminal positions there are fewer terminal values than alternatives. If it is necessary to distinguish between such alternatives, cosynes may be introduced which make entries in the analysis record but do not perform terminal functions. Two such cosynes which may prove to be of general use are IN and SQ. IN inserts the number given as a parameter into the *current* level of the analysis record and comparison is continued with the current syne definition. SQ is used for optional symbols. It inserts (in the same manner as IN) a '1' or a '0' according to whether the symbol whose code is specified by its parameter is recognised or not.

Terminal cosynes have been omitted from the syne definitions which appear in the body of this text where only the syntax is relevant.

Macros

A macro definition consists of the word MACRO followed by a heading which defines its syntax. This has the same form as an alternative in a syne definition. The current set of macros are the alternatives in the definition of the pseudo syne ⟨MACRO⟩ which was mentioned earlier in defining the instructions which may appear in a program. When the end of a block is reached all macros defined in that block are deleted. The ordering of the macros in the syne ⟨MACRO⟩ is the reverse of the order in which they are defined. This means that if macros are redefined the most recent definitions are used. An example of a macro heading is

MACRO OUTPUT (⟨SYMBOL STRING⟩) ON
OUTPUT ⟨NUMBER⟩ ⟨NL⟩

The synes which appear in a macro heading are called parameters. If they have some semantic meaning, as the first two above obviously have, they are termed functional. Non-functional parameters may be used to permit stylistic variants in a macro, as ⟨JUMP⟩ is below.

SYNE ⟨JUMP⟩ = JUMP TO ⟨UP⟩ | GO TO ⟨UP⟩|
— ⟨⟩⟩ ⟨UP⟩
MACRO ⟨JUMP⟩⟨LABEL⟩⟨NL⟩

The choice of ⟨UP⟩ as the terminal cosyne above means that no record will be generated for ⟨JUMP⟩ during the scan of instructions in which it appears, such a distinguishing record being unnecessary. A better way of defining the syntactic description of the macro would be

MACRO [JUMP TO | GO TO | —⟨⟩⟩] ⟨LABEL⟩⟨NL⟩

The meaning of a macro is defined by a subroutine to which control is transferred whenever the scan matches an instruction with the macro heading. This routine is written immediately after the heading and is terminated by an END. Like cosyne routines macro routines are recorded as a part of the assembler stack and are distinct from any object code which is being assembled. They are both regarded as blocks.

Preloaded macros

Some macros perform a declarative function, and their associated routines record information to be used by other macro routines. This creates a requirement for a set of global variables common to all macro routines. Cosyne routines may also use these same variables. The names B0 to B255 are reserved for use in this way and must not be redefined. Some of these are important pointers in the system and have alternative names. On Atlas B-lines 0 to 127 correspond to names B0 to B127 and the rest of the global names are allocated space in the core store.

Macro routines also have a requirement for local variables. Since recursion is permitted they must occupy space in the main working stack. The following preloaded macro meets this need.

MACRO ASSIGN ⟨ASS.LIST⟩⟨NL⟩

where

SYNE ⟨ASS. LIST⟩ = ⟨S.VAR⟩ [⟨ASS. LIST⟩| ⟨NULL⟩]

⟨S. VAR⟩ denotes a name, possibly labelled (see below), which must be enclosed in sharp brackets.

The effect of this macro is partly declarative in so far as it adds the names to the name list and assigns values to them which are the addresses of the associated stack positions, relative to SB. SB is a stack base pointer which points to the start of the area of the main working stack allocated to the current block or macro. The ASSIGN macro also causes instructions to be compiled which move a stack front pointer (SF) forwards by the appropriate amount. The scope of these names is confined to the macro in which they are defined. Any name used as an ⟨S.VAR⟩ may be labelled with an integer, thus:

⟨name/integer⟩
e.g. ⟨Z/2⟩

The label on an ⟨S.VAR⟩ used in this context causes several additional stack positions to be allocated to the name. Thus

ASSIGN ⟨X⟩ ⟨Y⟩ ⟨Z/9⟩

assigns one position to X, the next to Y, the next to Z and the next nine also to Z. The label in the context described below is used to identify individual positions in this nine.

The ASSIGN macro must not be used inside a cosyne routine.*

In order to describe the preloaded macros for register manipulation and control functions it is necessary to give a detailed description of their main constituents. These are ⟨VAR⟩ and ⟨EXPR⟩, where the definition of ⟨VAR⟩ is

SYNE ⟨VAR⟩ = ⟨G.VAR⟩|⟨[⟩⟨ADDR⟩⟨]⟩|⟨S.VAR⟩

⟨G. VAR⟩ denotes the register associated with any of the predefined global register names (B1, SF, etc.), and [⟨ADDR⟩] denotes the store register whose address is given by the value of ⟨ADDR⟩. ⟨ADDR⟩ is a restricted form of expression and is implemented by using B-modification. Its definition is

SYNE ⟨ADDR⟩ = ⟨VAR⟩ [[+|−] ⟨LIT⟩|⟨NULL⟩]| ⟨SQ((−))⟩⟨LIT⟩

where ⟨LIT⟩ is a literal defined later. The third alternative ⟨S. VAR⟩ is the method of referring to stack variables defined in the ASSIGN macro. The notation for these is again the name (possibly labelled) in pointed brackets, as in the ASSIGN statement, for example, ⟨X⟩ ⟨Z/3⟩. Their meaning is

$$\langle X\rangle \equiv [SB + X]$$
$$\langle Z/3\rangle \equiv [SB + 1.4Z]$$

An expression is essentially a sequence of operators (OPR) and operands (OP), defined as:

SYNE ⟨EXPR⟩ = ⟨OP⟩ ⟨REPEAT.IF.OPR⟩

where ⟨REPEAT.IF.OPR⟩ is a cosyne which causes the scan to search for another ⟨OP⟩ if an operator is recognised and terminates the scan for an ⟨EXPR⟩ if not. The possible types of operator symbols and the corresponding meanings are given in Fig. 1.

To avoid possible ambiguities the recognition mechanism requires that double symbol operators must be juxtaposed. Use of these operators does not invoke any precedence rule, and an expression is evaluated from left to right. For example,

$$B1 \times B2 - B3/B4 + B5$$

is evaluated as

$$(((B1 \times B2) - B3)/B4) + B5$$

The three kinds of operand (OP) may appear in an expression, variables (VAR), constants (CONST) and a very restricted form of subexpression. These are defined by:

```
SYNE ⟨OP⟩     = ⟨VAR⟩|⟨LIT⟩|(⟨VAR⟩[+|−]
                ⟨CONST⟩)
SYNE ⟨LIT⟩    = ⟨CONST⟩|⟨NAME⟩|'⟨EXPR⟩'
SYNE ⟨CONST⟩ = ⟨N⟩|⟨N⟩.⟨OD⟩|⟨OD⟩|
                (⟨SYMBOL⟩)|*⟨OW⟩|*
```

When an expression is used as a literal (e.g. '⟨EXPR⟩') it is evaluated at assembly time. The various forms for ⟨CONST⟩ follow the usual Atlas terminology:

⟨N⟩ denotes a decimal integer, its scaling will be such that it addresses words of instruction size (i.e. unity is three bits up on Atlas).

⟨OD⟩ denotes an octal digit which goes into the bottom three bits.

⟨OW⟩ is a sequence of octal digits which will be left justified.

⟨SYMBOL⟩ is any available symbol and (⟨SYMBOL⟩) denotes the number whose value corresponds to the code for the symbol.

A '*', not followed by any octal digits is taken to be

* The reason for this is that cosynes are blisters of the scanning routine and the next free stack space relative to SB for the scanning routine would not be known at assembly time. A suitably dynamic version could be defined for use in this context.

+ addition	− subtraction	& logical and
V logical or	−= logical non equivalence	× multiplication
/ division	← logical shift up	→ logical shift down
<+ arithmetic shift up	+> arithmetic shift down	⇐ circular shift up
⇒ circular shift down		

Fig. 1. The meanings of the operator symbols

a label whose value is the address of the instruction being assembled.

Any ⟨NAME⟩ used in a ⟨LIT⟩ is replaced by its assembly time value. It will not normally be the name of a syne or cosyne, but if it is the value will be the address of the encoded definition. If it is an S.VAR name, which is again unlikely, the value is the relative address of the associated register. Because ⟨VAR⟩ and hence ⟨G.VAR⟩ appear before ⟨LIT⟩ a global name cannot appear as a form of constant except in '⟨EXPR⟩'. Thus the most common form of ⟨NAME⟩ in ⟨LIT⟩ is a label which will have an address as its value. In general these may be forward references (i.e. labels not yet set) except when they are part of a literal expression (i.e. '⟨EXPR⟩').

Expressions usually appear in a context which implies the assembly of code to evaluate the expression at run time. This assembled code will in general require the use of two registers for partial results and addresses. The Atlas implementation uses B98 and B99.

We can now define the macros for register manipulation and control functions. The first is for assigning the value of an expression to a variable. It is

MACRO ⟨VAR⟩ ⟨ASS.OPR⟩ ⟨EXPR⟩ ⟨NL⟩

where ⟨ASS.OPR⟩ is a cosyne which accepts any of the above operators or = or = −.

When the ⟨ASS.OPR⟩ is an operator the macro is interpreted as an expression and the left to right evaluation is taken to include the specified ⟨VAR⟩ and ⟨ASS.OPR⟩. The resultant value of the expression is assigned to the ⟨VAR⟩. Examples are

B1 + B2 × 4 B1 = − B3 + 2 B1 = B4/B5

meaning

B1 = (B1 + B2) × 4 B1 = − (B3 + 2) B1 = B4/B5

In some forms of this macro it is assumed that the register on the left-hand side can be used to accumulate the result. This means it must not appear in the expression. For example

B1 = B2 + B1

is not allowed. Instead it should be written

B1 + B2

The second macro involving EXPR permits conditional control transfers. Control is passed to the address specified by an EXPR or to the following instruction according to whether or not the given condition is satisfied. Since the limitations of the character set do not permit a full range of conditions, the condition may be qualified by IF or UNLESS. The second of these reverses the meaning of the condition. The macro is

MACRO⟨IU⟩⟨EXPR⟩⟨CONDITION⟩, −⟨⟩⟩
⟨EXPR⟩⟨NL⟩

where

SYNE ⟨IU⟩ = IF | UNLESS

and

SYNE ⟨CONDITION⟩ = ⟨⟩⟩0|⟨⟨⟩0| = 0|/ = 0

There is also a macro which provides an absolute control transfer. It is not recommended that control (B127) be changed using the first (register setting) macro, as this may use the L.H.S. as local workspace. The control macro transfers control to the address given by an expression. It is

MACRO − ⟨⟩⟩⟨EXPR⟩⟨NL⟩

Those control transfers which require special mention are

(*a*) exit from a macro routine
(*b*) 'true' exit from a cosyne routine
(*c*) 'false' exit from a cosyne routine
(*d*) 'up' exit from a terminal cosyne routine

The values of the predefined labels EXIT, TRUE and FALSE are the required addresses. The labels should not be redefined. Thus the instructions take the form

→ EXIT
→ TRUE
→ FALSE
→ UP

The last instruction transfers control to the cosyne UP.

The macros above suffice for manipulating the analysis records generated by the 'VAL' cosynes but an additional macro is preloaded to facilitate the association of ⟨S.VAR⟩'s with the subtrees of this type of analysis record.
This is

MACRO SET ⟨ASS.LIST⟩ FROM ⟨EXPR⟩

It assumes that the value of the ⟨EXPR⟩ is the address of a tree having the structure of Fig. 2.
The specified ⟨S.VAR⟩'s are set to the addresses of consecutive subtrees. There is no check for illegal use. An example of the use of this macro is given below, but it is necessary first to describe a preloaded macro which enables macro routines to call dynamically the routines associated with other statements (e.g. in order to define the macro in terms of simpler ones). This is achieved by using the macro

MACRO * ⟨STATEMENT⟩

When a * macro is assembled the ⟨STATEMENT⟩ is processed by the scanning routine and an analysis record is generated. The ⟨STATEMENT⟩ need not be explicit; stack variables of the *same* name may be substituted for any of the synes appearing in the definition of the ⟨STATEMENT⟩. This is also true of cosynes provided they generate the standard tree form of analysis record. Wherever a stack variable appears the analysis record

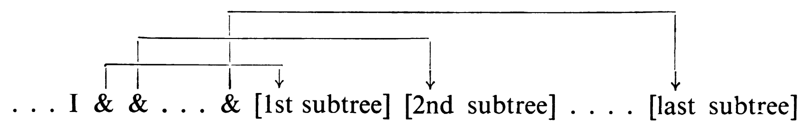

. . . I & & . . . & [1st subtree] [2nd subtree] . . . . [last subtree]

Fig. 2. Assumed value of ⟨EXPR⟩

will be incomplete. When the * statement is executed the analysis record is copied into the stack, it is relocated and completed by substitution of the current value of the S.VAR. Then the associated routine is called. To facilitate the relocation of the analysis record the ⟨STATEMENT⟩ is scanned twice and the two analysis records are compared. It is assumed that all words which differ are dependent on the position of the analysis record and a bit pattern is generated containing this information. In this macro all forms of ⟨STATEMENT⟩ are permissible substitutions but none are treated specially. Thus if it is required to call all of the routines associated with the statements of a block each one must be 'starred'. The example, Fig. 3. we now give is intended to illustrate the less obvious features of the system, and is probably not typical of the type of macro the user would write. It is a more general conditional macro which either obeys a ⟨STATEMENT⟩ or transfers control to the following instruction, according to the result of comparing two expressions. In this macro the only variants of ⟨STATEMENT⟩ allowed are ⟨BASIC.INST⟩, BEGIN and ⟨MACRO⟩. The associated routine deals with BEGIN in a special manner in order that the following block may be treated as the ⟨STATEMENT⟩ and not executed if the condition fails.

This example makes use of the fact that, on entry to a macro routine, the analysis record is in the stack behind the links. Also, it assumes that the definitions of the synes in the macro heading have been terminated by cosynes of the VAL type, which generate a branch for each syne at the top level. Thus the SET macro would be translated into,

```
    ⟨IU⟩             = [[SB—.4] + .4]
    ⟨EXPR⟩           = [[SB—.4] + 1]
    ⟨COMPARATOR⟩     = [[SB—.4] + 1.4]
    ⟨EXPR/1⟩         = [[SB—.4] + 2]
    ⟨STATEMENT⟩      = [[SB—.4] + 2.4]
```

B131, B132 and B139 contain the addresses of the BEGIN, ⟨BASIC INSTRUCTION⟩ and ⟨MACRO⟩ routines, respectively.

Two statements which appear in the above example require further discussion. FAULT PRINT represents an instruction sequence which would cause appropriate monitoring. SET 'END' LINK TO L4 changes the link associated with 'BEGIN' so that control will go to L4 when the matching 'END' is encountered.

Thus the sequence is entered when the END of the block appearing in the conditional macro is reached. Both of these statements could be defined as macros and as an example of a simple macro we define the second.

```
MACRO SET 'END' LINK TO ⟨EXPR⟩⟨NL⟩
    ASSIGN ⟨EXPR⟩
    SET ⟨EXPR⟩ FROM [SB—.4]

    * [B189—.4] = ⟨EXPR⟩
      :: THIS INST ASSUMES A KNOWLEDGE
      ::— OF THE POSN OF THE 'END' LINK
    → EXIT
END
```

Finally three preloaded macros are provided for output of integers, input of integers and output of symbol strings. These are

```
MACRO PRINT ⟨EXPR⟩ TO ⟨EXPR⟩ PLACES ⟨NL⟩
MACRO READ INTEGER TO ⟨VAR⟩
```

and

```
MACRO OUTPUT —⟨P.STRING⟩
```

where

⟨P.STRING⟩ denotes any string of symbols and ⟨S.VAR⟩'s

The action of the first of these is to print out the number given by the first expression, preceded by a 'minus' sign if negative and a space if positive. The

```
SYNE ⟨COMPARATOR⟩ = ⟨⟨⟩|⟨⟩⟩|/ = | =
MACRO ⟨IU⟩⟨EXPR⟩⟨COMPARATOR⟩⟨EXPR⟩, ⟨STATEMENT⟩
        ASSIGN ⟨IU⟩⟨EXPR/1⟩⟨COMPARATOR⟩⟨STATEMENT⟩⟨CONDITION⟩
        SET ⟨IU⟩⟨EXPR⟩⟨COMPARATOR⟩⟨EXPR/1⟩⟨STATEMENT⟩ FROM [SB-.4]
               ⟨CONDITION⟩ = ⟨COMPARATOR⟩
               * BEGIN
               * B99 = ⟨EXPR/1⟩
               * B99 = — B99
               * B99 + ⟨EXPR⟩
               * ⟨IU⟩ B99 ⟨CONDITION⟩,          →L1 :: PLANT CONDITIONAL JUMP
               IF [⟨STATEMENT⟩] — B131  = 0, →L1 :: JUMP IF 'BEGIN'
               IF [⟨STATEMENT⟩] — B132  = 0, →L2 :: JUMP IF ⟨BASIC. INSTR⟩
               IF [⟨STATEMENT⟩] — B139 /= 0, →L3 :: JUMP IF NOT ⟨MACRO⟩
        L2:    * ⟨STATEMENT⟩
               → L4
        L1:    * BEGIN
               SET 'END' LINK TO L4
               → EXIT
        L3:    FAULT PRINT
        L4:    * L1:                                        :: END CAUSES RE-ENTRY HERE
               * END
               → EXIT
        END
```

Fig. 3. Example of a more general conditional macro

second expression gives the field size (−1 for the sign). The action of the second macro is to read the next integer into the specified variable. The first appearance of either the PRINT or the READ macros in the program being assembled causes the insertion of a subroutine. Subsequent use causes only a call for the subroutine to be assembled. If they are used in macros a call for the SPG copy of the READ or PRINT routine is inserted.

When an OUTPUT macro is assembled a packed symbol string is generated. Wherever a stack variable appears in the ⟨P.STRING⟩ a marked code word is stored. On execution of the macro the string is examined and, if necessary, the symbol strings associated with the stack variables are substituted for the code words before the string is output. One use of this macro would be to output object program (perhaps in the code of some other machine), rather than assembling it for 'load and go'. All synes (i.e. S.VAR's) which appear in an OUTPUT macro must be defined using the EXVAL terminal cosyne. The additional value words this produces are used by the OUTPUT macro routine to locate the strings to be substituted. A similar macro for use in *input* of symbol strings may prove to be useful. This would generalise the * macro by storing the symbol string instead of the analysis record of the statement, and the scanning would be carried out at execution time of the enclosing macro.

The basic instructions

In the Atlas implementation these have the form:

⟨FD⟩ ⟨B⟩ ⟨B⟩ ⟨ADDRESS⟩

⟨FD⟩ is the function digits of the required instruction and

SYNE⟨B⟩ = ⟨N⟩|'⟨EXPR⟩'

The ⟨B⟩'s are the B-line modifiers of the instruction and the B-line number specified is taken to be the number ⟨N⟩ or the assembly time value of the ⟨EXPR⟩ (modulo 127).

The ⟨ADDRESS⟩ specifies the address half of the instruction. It consists of a sequence of literals separated by + or − and is defined as,

SYNE ⟨ADDRESS⟩ = ⟨LIT⟩ ⟨REPEAT.IF.P.OR.M⟩

where ⟨REPEAT.IF.P.OR.M⟩ is a cosyne which causes the scan to search for another ⟨LIT⟩ if a + or − is found and terminates the scan for an ⟨ADDRESS⟩ if not.

Preloaded macros in source program

The above description of the preloaded macros is mainly relevant to the use of macros within macro or cosyne definitions. However, use of the following macros is also permitted in the source program:

MACRO ⟨VAR⟩ ⟨ASS.OPR⟩ ⟨EXPR⟩ ⟨NL⟩
MACRO ⟨IU⟩ ⟨EXPR⟩ ⟨COND⟩, −⟨⟩⟩ ⟨EXPR⟩ ⟨NL⟩
MACRO −⟨⟩⟩ ⟨EXPR⟩ ⟨NL⟩
MACRO ASSIGN ⟨ASS.LIST⟩ ⟨NL⟩

Conclusion

The aim has been to produce a system which is compact and efficient whilst being easily expandable in a number of directions. Its size on Atlas is 1·5K words and compilation speeds of 20K instructions per minute have been achieved. An additional set of macros specifically aimed at generating syntax directed compilers is available and the system has been implemented using itself as a test of its flexibility. Currently ALGOL and COBOL compilers are being written.

Acknowledgements

We are indebted to our colleagues, and friends at the SRC Atlas Lab., Chilton, for helpful criticism and comments.

One of us (I. R. Wilson) wishes to acknowledge the financial support of SRC during the period of research which led to this paper.

References

Brooker, R. A., MacCallum, I. R., Morris, D., and Rohl, J. S. (1963). The Compiler Compiler, *Annual Review in Automatic Programming*, Vol. 3, London: Pergamon.
Brooker, R. A., Morris, D., and Rohl, J. S. (1962). Trees and Routines, *The Computer Journal*, Vol. 5, p. 33.
Burkhardt, W. H. (1965). Metalanguage and Syntax Specification, *CACM*, Vol. 8, No. 5, p. 304.
Iverson, K. E. (1964). A Method of Syntax Specification, *CACM*, Vol. 7, No. 10, p. 588.
Naur, P. (Ed.) (1963). Revised Report on the Alogrithmic Language ALGOL 60, *The Computer Journal*, Vol. 5, p. 349.