Some proposals for SNAP, a language with formal macro facilities

By R. B. E. Napper*

This paper describes some proposals to apply the full power of the Compiler Compiler to an ALGOL-like base language, to provide an extendable language system that is one-pass and produces efficient object code. At its simplest SNAP permits easily defined "formal macros" which allow conventional routine calls to be coded as open sequences tailor-made to the actual parameters. The paper discusses the problems created by this facility, in particular to ensure efficient code. It finishes with an illustration of the more powerful "informal macros" which use phrase structure notation in the routine headings and Compiler Compiler instructions in the routine bodies.

Introduction

The name SNAP is derived from a "System for NAtural Programming". The aim of the system is to provide a simple basic source language together with a definition mechanism that will allow the language to be extended indefinitely. In order to implement the system the compiler will be written using a revised version of the Compiler Compiler (Brooker *et al.*, 1963).

The key to the extensions are "formal" and "informal" macros, which allow extensions to be made to the existing language easily and without loss of efficiency in the compiled code.

A *formal macro*, which will be defined more precisely later, is a conventional routine that is compiled as an open routine tailor-made to the actual parameters. The definition is essentially the same as that of a conventional routine, and so formal macros can be used easily and safely by programmers with no further knowledge or experience beyond that required for using conventional routines.

An *informal macro* is a routine containing instructions in the language of the Compiler Compiler which are obeyed at compile-time, as well as any source instructions that are to be compiled into the source program. Such definitions require a minimum knowledge of Compiler Compiler theory, e.g. such as is contained in Napper (1965). With this minimum knowledge comparatively simple definitions can be written which are simply transformations into the corresponding set of instructions of the existing language of an instruction with a very free format (with options or indefinite repetitions of clauses, phrases, and/or parameters), using phrase-handling instructions to break down the informal phrases used in the format into the formal syntactic units of the basic language. More advanced informal macros will require more detailed knowledge of the language of the Compiler Compiler and of the implementation of SNAP.

Notes

(i) The most important and interesting extensions to the language are derived from informal macros, which though they may be non-trivial to write are easy to use. There is not space in this paper to describe informal macros in any detail; the description of the formal macros covers ground which is necessary for describing informal macros, and a fuller description of informal macros will be given in a later paper which will describe the implementation of SNAP in more detail.

(ii) It is more appropriate to refer to SNAP as a "system" rather than a "language". The basic language could be an adaptation of an existing language, e.g. Atlas Autocode (AA) or ALGOL, or it could be a simple language designed independently. SNAP then adds a Compiler Compiler capability to the basic compiler and demands a certain internal organization of the basic language's implementation in order to provide a smooth interface with the language processing machinery.

(iii) CPL, AA, and ALGOL will be used for comparison purposes as "conventional" languages. As a convenient general rule it can be assumed that facilities not mentioned in the paper follow the pattern of one or other of these languages.

(iv) Napper (1967) describes a system such as SNAP as a *2nd-order* language (or compiler), where an assembly language is of order 0, and a conventional language like ALGOL is of order 1. A language of order 0 has just one basic format (a machine-code instruction), a language of order 1 has a fixed set of formats, and a 2nd-order language has an indefinite set of formats derived from a fixed set comprising the basic language of the system and a set of language-defining formats.

If ALGOL is regarded as the definitive first-order first-generation language, then PL/1 is potentially a second-generation language, but not necessarily a second-order language. Its direction of progress is complementary to SNAP in that the main emphasis of PL/1 is in providing power, flexibility, and generality to the data-type facilities available in the language; on the other hand SNAP aims at transforming the power of a programming language within the range of data-type facilities available.

Initially SNAP will not be designed to facilitate the expansion of the data-type facilities: i.e. whereby a new type can be introduced by adding special informal macros to define the relevant conversions between a new type and the existing types, and to define the action

* *Dept. of Computer Science, University of Manchester, Manchester*, 13.

of the existing operators on it; or whereby a new operator can be introduced by defining its action on the existing types. Using informal macros in SNAP it will be possible to define a package of formats to manipulate new types or use new operators, and to correlate them with the existing facilities, but it will not be possible to use them as operators and operands of the basic language, except where the function facility can be adapted for this purpose.

It is considered that operator-operand expansion and language-format expansion are two major growth areas beyond conventional languages that are best explored independently at first, in order to keep the number of new problems to be tackled within bounds.

For this reason, and because CPL, AA and ALGOL are more simply defined, PL/1 is not being used as a frame of reference in this paper.

(v) Although earlier work by the author, e.g. Napper (1966), has concentrated on "Natural English", and specifications of a revised English Mode syntax have been drawn up (see next section), this work has been pushed into the background while effort is concentrated on an extension of conventional languages through macros. It is believed that the use of English conventions will become acceptable in appropriate contexts once programmers have experienced the greater freedom of expression SNAP will allow. The emphasis at this time is simply on "natural language", i.e. on permitting the programmer to express himself freely within a neutral framework, without forcing him to mould his expression into a purely mathematical or purely English framework.

Basic language of SNAP

It is hoped to provide eventually two alternative modes for the basic language, as shown in this example taken from a specimen program and autocode used for illustration purposes in Napper (1965).

[Note that function **sqdf**$(a,b)\equiv(a-b)^2$.]

Mathematical Mode

```
  p = sqdf( Y[2], Y[1])
  q = 0
cycle e = 3,1,n
      x = sqdf(Y[e], Y[e−1])
      y = sqdf(Y[e−1], Y[e−2])
      p = p + x
      q = q + sqdf (x,y)
repeat
  print ( Y[n] − Y[1])/n pr 5, 6
  print sqrt(p)/(n−1)    pr 10,6
  print sqrt(q)/(n−2)    pr 14,6
```

English Mode

Set $p =$ *the* **squared-difference** *between the YEAR of the 2nd event and the YEAR of the 1st event, and set q to* 0.

Then REPEAT for each event from the 3rd event up to the final-event:

Set $x =$ *the* **squared-difference** *between the YEAR of the* **current** *event and the YEAR of the* **previous** *event, and set* $y =$ *the* **squared-difference** *between the YEAR of the* **previous** *event and the YEAR of the* **last-but-one** *event. Then add x to p, and add the* **squared-difference** *between x and y to q.*

WHEN each event has been dealt with:

Print the YEAR of the final-event—the YEAR of the 1st event **DIVIDED-BY** *the number-of-events* ((≡ *final-event*)) (*precision* : 5 *places before*, 6 *after*).

Then print the **square-root** *of the sum-of-squared-differences* ((≡p)) **DIVIDED-BY** 1-**less** *than the number-of-events* (10,6), *and print the* **square-root** *of the sum-of-squared-differences-of-squared-differences* ((q)) **DIVIDED-BY** 2-**less** *than the number-of-events* (14,6).

In Mathematical Mode instructions are separated by ";" followed by a space (or new line), "." followed by a new line, or just a new line. Identifiers are in the form of single letters, maybe followed by a prime. Expressions are in the form of algebraic formulae with implicit multiplication allowed in all cases (provided there is no space between the two operands). Spaces are usually optional.

The syntax of the initial basic language of SNAP has not been decided upon yet. It may be decided to make a direct adaptation of an existing language as the basic language. If not, it is expected that initially a *Neutral Mode* will be used. This will be based on the Mathematical Mode, with a relaxation on the form of identifiers to allow multi-symbol names (with a corresponding restriction on implicit multiplication), and with words that are parts of a routine name allowed without underlining provided there is a space before and after them. Studies will be made on an ideal basic syntax in parallel with experimental work on the implementation of SNAP using the existing Compiler Compiler on Atlas.

For the sake of simplicity and clarity Mathematical Mode will be used for illustration purposes in this paper.

It must be remembered in the examples given of declarations that the language-defining mechanism of the Compiler Compiler will allow alternatives to the syntax to be made easily. It can be assumed for example that parameter characteristics will have standard abbreviations, e.g. REN for REAL EXPRESSION NAME. However, simply for clarity of description in this paper, words will be used instead of abbreviations both in the basic syntax and in the choice of routine names.

NOTE. In a program where both Mathematical Mode and English Mode were being used, the two modes could be mixed together freely. In particular there would be "where" clauses to give temporary names in one mode to variables or expressions in another, e.g.:

Add 4 *to the counter, and where c is the counter, and s is the sum, form:*

$$p = (Q[c]+Q'[c-2])/3(a+h)$$
$$s = s + pR[c]$$

General structure of a SNAP program

It is proposed to break up the formal block structure of conventional languages with respect to routines. A

program will consist of a number of independent sections prefaced by *master declarations*. In particular there will be a **MAIN PROGRAM** and **ROUTINE** and **FUNCTION** declarations. There will also be master declarations for giving global information about the characteristics of identifiers, and for giving **ROUTINE FORMATS** and **FUNCTION FORMATS,** and one for **PRESET LISTS** to give information according to various conventions that would otherwise have to read in as data.

Routines can be given serial numbers for reference, and information about a routine can be declared outside it by using these serial numbers, e.g. in the global data-type declarations, or in the local declarations of other routines.

There are local declarations which allow a routine to use variables declared as local to other routines, using a different identifier in the routine if required.

A routine as a whole can be declared to be within the scope of another routine. This will have the usual effect, but there will be no need to group within a formal block all the routines within the scope of a routine.

However, the system will be basically one-pass, so all information must be declared before it is otherwise referred to. Thus if a routine is declared to be within the scope of another which has not been defined, this will be queried. But if a routine uses a variable from another routine that has not been defined, this will not be in error if its characteristics have already been declared or are included in the declaration; the variable need not then be declared in its own routine. Similarly a routine can only be called after its format has been declared; but if a routine is defined before it is called there is no need to give the format specification separately.

The reasons for proposing this informality of scope derive from the author's experience with large programs. Natural language is at its greatest advantage in this context, and there is more opportunity for using macros since there is a greater possibility of the frequent repetition of small groups of instructions (or even single instructions) which have a clear function in the context of the job which would be obscured if they were described in the basic language. The greater use of self-description in instructions and the increased readability of the program make it far easier for the programmer to coordinate the program in his mind.

To reinforce the readability it is convenient to set out the program so that the sections of the program are of a uniform level of relevance and detail. In a large context this means that it is convenient to group routines which are at the same level together, and it is therefore inconvenient to intersperse them with any more detailed subroutines which are within the scope of individual routines. Within higher level routines, it is sometimes convenient to use a subroutine for a small group of instructions simply because they describe operations so detailed in relation to the rest of the routine that they impede the flow of the description. It is more convenient to cover such sections with a general comment on their function in the context of the routine. Such subroutines will tend to be called only once, and if they are in a frequently-used section of the program of course they will have to be defined as macros.

The reason for allowing variables to be declared in one routine as belonging to another is to allow a group of routines at the same level to share the same set of variables.

An important reason for designing a one-pass language is that when on-line instruction-by-instruction compiling becomes possible it will be easy to adapt the compiler to request undeclared information, add it to its compile time organization, and compile correctly on the spot.

Together with the use of implicit blocks with conditional instructions (see below) it will be possible to remove the concept of a block from the beginner's armoury. Long programs can be written without any need for an explicit block declaration.

It will, however, still be necessary to have a full formal block structure for routines in the case where different sections of a program are written as independent packages. Regarding each such unit as a "subprogram", subprograms can be included as a unit of program description, and can be nested within each other in the usual way; but within a subprogram a one-level program description as described above is encouraged. The facilities associated with subprograms will be restricted in comparison with the analogous routine facilities; in particular it will not be possible to define a subprogram as a formal macro.

Use of the stack

Scalar variables and arrays will not use a stack unless specifically requested or unless its use is implicit, e.g. because a routine has been declared recursive or uses dynamic storage allocation. Thus scalar variables will in general occupy fixed global locations. Also the addresses of dynamic arrays will be held in fixed global locations.

Thus all variables declared local to a routine except for those used recursively can always be accessed directly from any other routine in the program. A further location of global store is used to hold the current value of the stack pointer for each routine that uses a stack. This will be set to zero when the routine is not active, and whenever a routine uses a stacked variable of another routine a check will be made on entry to the routine that the other routine is currently active.

Features derived from the Natural English system

Some of the features of the author's original Natural English system have proved useful enough to propose carrying them over into standard compiler practice.

(*a*) *Implicit Blocks.* In English Mode, instructions are separated by punctuation , ; : . and paragraph. These instruction separators are given an order 0 to 4 respectively and are used as implicit block markers. For example, if an "if" or a "where" clause, or a cycle instruction, is used to qualify a set of following instruc-

tions, the end of this local block is taken to be the next instruction separator of order higher than that terminating the qualifying instruction.

This is adapted to Mathematical/Neutral Mode by allowing two levels of punctuation, ";" with order 1 and "." (or just a new line) with order 3. Qualifying punctuation is restricted to ":", and this qualifies all instructions up to the first instruction separator of order 3.

(*b*) *Conditional Routines.* Routines can be defined as conditional. In this case exits from the routine are specified by one of the two instructions **condition satisfied** or **condition not satisfied** instead of the **finish** (or **return**) of ordinary routines. Conditional routines replace the "**if** ⟨boolean expression⟩ **then** . . ." of a conventional language.

(*c*) *Cycle Control.* There are two jump instructions that can be used in the text covered by a cycle instruction. Where for example *i* is the controlling parameter these are **finish current** *i* and **finish all** *i*. These have the obvious effect in relation to the controlling parameter, and they can be used anywhere from inside any nested cycles.

(*d*) *Organization of Control.* The organization of control is achieved by manipulating "control lists" of machine-code instructions requiring addresses to be filled into certain points, for example the next instruction separator of a certain order, or the points indicated by the instructions of (*b*) and (*c*). In general addresses are not filled in until the final destination is known. Thus there is no jumping to an unconditional jump instruction except maybe via an explicit label reference. This control organization works in liaison with the routine and conditional routine mechanism, and in all its forms including macros. Regardless of the complexity of definition the control compiled is completely efficient.

A set of consecutive conditional clauses is treated specially. Each conditional instruction (basic or routine call) is compiled separately, and as each new conditional is met the control lists are updated as appropriate to the and/or connectives and the bracketing implied by the punctuation order (where ";" is allowed between conditional instructions)—see the example given in the definition of the function "middle-value" below. Normal mode is resumed when the first non-conditional instruction is met.

Formats of routines and functions

The format of a routine format is an alternating string of underlined symbols* and formal parameter specifications, starting with an "underlined small word" (i.e. a string of underlined symbols starting with two small letters) and ending in "." for ordinary imperatives or ": . . ." for conditionals.

A formal parameter specification consists of a set of characteristics of the formal parameter followed by the name by which it is referred to in the body of the routine, all enclosed in square brackets. In the simplest case, the specification of a parameter should contain the data type and should indicate whether the permitted syntax of each actual parameter is that of a variable (which is taken to imply that it is reset by the routine) or an expression.

Thus a set of routine formats appearing as a global declaration might be:

ROUTINE FORMATS
interchange [*REAL VARIABLE v*] **and** [*REAL VARIABLE v'*].
convert [*REAL EXPRESSION e*] **to int**[*INTEGER VARIABLE i*] **and frac** [*REAL VARIABLE r*].
if [*INTEGER EXPRESSION e*] **divisible by** [*INTEGER EXPRESSION d*]: . . .
if [*INTEGER EXPRESSION y*] **leap year**: . . .

In general, there is an optional space in the call wherever there is a space in the format and on either side of each parameter. Some symbols are allowed in the format without being underlined, e.g. "=" and ",".

The format of a function format is an underlined small word, followed by alternations of underlined symbol string and formal parameters enclosed in round brackets. The format is preceded by a formal parameter to specify the result parameter.

For example:

FUNCTION FORMATS
[*REAL r*] = **middle-value** ([*REAL EXPRESSION a*], [*REAL EXPRESSION b*], [*REAL EXPRESSION c*])
[*REAL s*] = **sum (for** [*DUMMY INDEX i*] **from** [*INDEX EXPRESSION s*] **to** [*INDEX EXPRESSION f*] **of** [*REAL FUNCTION f*(*i*)])
[*INTEGER i*]=**nearest-integer** ([*REAL EXPRESSION e*])

Routine and function definitions

A routine comprises a master declaration, the routine heading (identical to the format specification) and then the routine body terminated by the next master declaration.

For example:

ROUTINE
interchange[*REAL VARIABLE v*] **and** [*REAL VARIABLE v'*].
REAL t
$t = v$
$v = v'$
$v' = t$

ROUTINE
if [*INTEGER EXPRESSION e*]**divisible by**[*INTEGER EXPRESSION d*]: . . .
if $e = d$**intpt**(e/d): **condition satisfied** : **otherwise** : **condition not satisfied.**

ROUTINE
if [*INTEGER EXPRESSION y*] **leap year** : . . .
if *y* **divisible by** 400 : **condition satisfied.**

* Printed in bold in this paper, as are all other words and symbols that would be underlined in the typewritten form of a program.

if y **divisible by** 100 : **condition not satisfied.**
if y **divisible by** 4 : **condition satisfied** : **otherwise** :
condition not satisfied.

A conditional routine is the same as an ordinary routine definition except that instead of using **finish** return must be specified by either **condition satisfied** or **condition not satisfied.** And whereas in an ordinary routine body if the final instruction is not an unconditional jump a final **finish** is implicit, in a conditional routine the last instruction must be an unconditional jump. Further, there must be at least one such return in the routine for each branch.

A function definition comprises a master declaration and the function heading as for the function format, and then the body of the routine. The formal parameter at the beginning of the heading is taken to declare the name of the result. Then the conventions are exactly the same as for an imperative routine; that is, exit is indicated by **finish,** or is implicit at the end.

Thus:

FUNCTION
[*REAL r*] = **middle-value** ([*REAL EXPRESSION a*], [*REAL EXPRESSION b*], [*REAL EXPRESSION c*])
if $a \leqslant b$; **and if** $a \geqslant c$: **or if** $a \leqslant c$; **and if** $a \geqslant b$: $r = a$; **finish.**
if $b < a$; **and if** $b \geqslant c$: **or if** $b \leqslant c$; **and if** $b > a$: $r = b$: **otherwise** : $r = c$.

Note in this example of the use of multiple conditional instructions that the instructions are separated by punctuation; **and** and **or** are not instruction delimiters but "connectives". The level of punctuation gives the implicit bracketing. Within a pair of implicit brackets the choice of the connectives between units must be consistently **and** or **or,** but which it is is independent of the structure outside the brackets or of that inside any conditional groups that may be bracketed within them at a lower level.

Using **and, or** and **then** as instruction delimiters, the implied bracketing in the example above is as follows :

((**if** $a \leqslant b$ **and if** $a \geqslant c$) **or** (**if** $a \leqslant c$ **and if** $a \geqslant b$)) **then** ($r = a$; **return**)
((**if b** $< a$ **and if** $b \geqslant c$) **or** (**if** $b \leqslant c$ **and if** $b > a$)) **then** ($r = b$) **else** ($r = c$).

Note also that the individual conditional instructions that make up the multiple conditional are quite independent; they could have been conditional routine calls as well as basic conditional instructions.

Formal macros

A conventional routine call associates the actual parameters of the call with the corresponding formal parameters of the routine heading. Then the effect of the routine mechanism is as if the call had been replaced by the instructions of the routine with each formal parameter reference replaced by the corresponding actual parameter. If the compiler carries out this substitution itself, so that an open routine is compiled that is unique to the actual parameters, then this is defined as a *formal macro.*

However, it is usually beyond the resources of a compiler to carry out this substitution. Further, whether or not it can, it is frequently economically unjustified to compile open routines in place of routine calls since the time saved in program execution compared with a cue-subroutine mechanism does not compensate for the extra time and store used in compiling and the extra store used for the compiled program.

A conventional compiler therefore compiles for each routine definition a fixed subroutine which will be used to process all calls on the routine. In place of each call it compiles a cue which converts each actual parameter of the call into a single general form that can be used by the subroutine whenever the associated formal parameter is referred to. The cue also organizes the change of control between the routine it is contained in and the subroutine. Thus the general subroutine combines with the particular cue that is unique to the call to simulate the action of the corresponding formal macro.

The definition of a macro routine is the same as that of a conventional routine. The only difference is in the implementation implied by true substitution as opposed to a cue-subroutine mechanism. So in SNAP the routine definition of a formal macro is the same as for an ordinary routine except for the master declaration. Thus:

CLOSED ROUTINE or **CLOSED FUNCTION** for a conventional subroutine implementation, and **OPEN ROUTINE** or **OPEN FUNCTION** for a formal macro.

However, there is frequently a need when using formal macros to be discriminatory for any particular call as to whether an open or closed form should be used. For a particular macro it may be desirable to compile a call open if it is in an often-used section of the program but closed if it is not. Therefore there is the further master declaration:

DUAL ROUTINE or **DUAL FUNCTION**

In this case a special formal parameter can be inserted after the first underlined small word of the format, either [*OPEN* *] or [*CLOSED* *]. This means that if in a call there is an asterisk at this point the call will be compiled *OPEN* or *CLOSED* respectively. If there is no such parameter, [*OPEN* *] will be assumed.

If the subspecification **OPEN, CLOSED,** or **DUAL** is left out of the master declaration, **CLOSED** is assumed; if the **ROUTINE** or **FUNCTION** is left out the correct alternative is deduced from the format of the format.

Correlation of parameter calls and local declarations

Independent of whether or not it is suitable to compile an open or closed form of a routine in place of any particular call, there are further considerations with respect to each of the formal parameters as to how their calls should best be implemented. For example, if for

each call on a routine it is known that the actual parameter for a particular formal parameter will always be an expression whose value remains constant throughout the execution of the routine, then it is uneconomic to recalculate the value of the expression each time the formal parameter is used in the routine.

Accordingly conventional compilers provide alternative implementations of parameter calls to allow the programmer to specify the most economic combination of implementations for the formal parameters of a routine. The classical choices, for parameters that are variables or expressions, are as follows (using CPL terminology):

(i) *Call by Value.* The value of the actual parameter is calculated in the cue and a local variable of the subroutine is set to this value before entry. Whenever the formal parameter is referred to in the routine it is treated as a reference to this local variable. If the actual parameter is to be reset by the routine (i.e. it has the characteristic *VARIABLE*), the cue also arranges, on return from the subroutine, to copy back the final value of the local variable to reset the actual variable.

(ii) *Call by Reference.* (Call-by-simple-name of AA.) If there is a possibility that an actual parameter that is to be reset is referred to during the execution of the routine by using its actual name instead of its formal name, it is better that each reference to its formal name should deal direct with the actual parameter instead of with a local copy as in (i); otherwise it might happen that for certain actual parameters at certain times during the execution of the routine the values referred to by the actual name and the formal name could be different (i.e. a "side-effect"). If it is known that the *address* of the element to be reset remains constant throughout the execution of the subroutine, then the cue can calculate this address and pass it on to the subroutine. The subroutine keeps this address in a local location associated with the formal parameter and whenever the formal parameter is referred to it is treated as a reference to the variable with this associated address. Thus the value of the actual variable can both be referred to and reset from the subroutine by specifying these operations on the formal parameter.

(iii) *Call by Substitution.* (Call-by-name of ALGOL.) If an actual expression can contain operands whose values can be altered during the execution of the subroutine, or if an actual variable (e.g. an array element) can have an address that can vary, then to ensure generality there is no alternative but to recalculate the value of the expression or variable address on each reference to the formal parameter. In this case a *secondary subroutine* is compiled as part of the cue to carry out this calculation, and its address is passed on to the subroutine. Then whenever the formal parameter is referred to control is passed back to the secondary subroutine which will provide the required value or address.

Note that it is equally valid to use implementations (i) or (ii) for parameters of type (iii) if it is required that the value or address referred to by the formal parameter should remain frozen at its entry value.

However, these devices and others are also applicable within a routine or block. For example, if an expression containing operands that can vary is often used in a block, it may be convenient, to save both source program and object program space, to declare, e.g. *REAL NAME* $e = (a-b)(a-c)/2bc$. Then whenever the parameter e is used in the block the value of the associated expression is recalculated from a secondary subroutine compiled on meeting the declaration. The effect of the use of e is the same as if it had appeared as a formal parameter calling an expression by substitution.

It is this generalization of the classical routine call implementations into the field of local declarations that accounts for the wide set of such facilities in CPL compared with e.g. ALGOL. SNAP continues this generalization further because the formal macro facilities allow further alternative implementations. In particular *true substitution* is possible; that is where call-by-substitution is implemented by open sequences instead of by secondary subroutines.

For the reasons just stated there is a very close correspondence in SNAP between the variety of local definitions provided and the possible formal parameter specifications. The implementation of the corresponding facilities will tend to be done by the same compiler routines. However, there are differences in format for the user in the two cases as some of the required specification will be implicit from the context. Thus for example [*REAL EXPRESSION NAME e*] corresponds to the local declaration *REAL NAME e*=[EXPRESSION], where [EXPRESSION] is used here in the true Compiler Compiler sense as referring to the current symbol string value corresponding to this compile-time phrase-variable, i.e. to the symbol string of the actual parameter of the call. In fact there is a set of *call-declaration* instructions used behind the scenes in formal macros which can be used explicitly in informal macros (as illustrated later). For example the appropriate call-declaration for [*REAL EXPRESSION NAME e*] is *CALL REAL* [EXPRESSION] *e BY SUBSTITUTION.*

It might be worth mentioning at this point that in SNAP the recognition machinery automatically "remembers" the dynamic textual level of each identifier, and hence it knows the block to which the identifier belongs however many levels of true substitution it may have been passed through before its characteristics are required for compiling code. Further, any variable of any routine can be referred to direct from any other routine, except that for recursive routines special action has to be taken if there is a call-by-substitution of a stacked variable through a level of the recursion. For non-local variables that are stored on the stack of another routine a test that the routine is active is made once only, on entry to the block declaring the variable; therefore this test need not be repeated on each reference to the variable either in that routine or if it is passed by substitution into other routines. Note that for example in ALGOL a call-by-name implies temporary return to the block containing the cue and so requires the corre-

sponding reorganization of the stack on entry to and exit from the secondary subroutine.

Local declarations

When discussing formal parameters and local declarations for SNAP it must be remembered that there will be great freedom in the syntax of formats since the full power of the Compiler Compiler is available to define it. It will be general practice to allow information to be left out where a strong option can be used safely. The syntax used in the examples given below will be essentially illustrative in the context of the discussion rather than being a specification of the permitted syntax.

The following is a selection of the more obvious declarations that would be useful for scalar data elements. These must occur at the beginning of a block and must not be prefaced by labels or interspersed with imperative instructions except by using declaration (12). Note that declarations (6) to (11), and some forms of (5), are versions of implementations of parameter calls for closed routines. Declaration (2), and the true-substitution forms of (8) to (11), are versions of open routine implementations. Note that not all the declarations illustrated are essential to the system; some are provided because they are by-products of the parameter call machinery that are easily understood and sometimes useful.

(1) ⟨TYPE⟩ *a*, *b*. Defines the name and type of a new local variable.

(2) *VARIABLE NAME* $i = j$, $k' = k$ (*R*50). This declares a new name for an existing scalar variable. The scope within which the existing variable is defined is that of the associated routine, or if none given, the current routine.

(3) *NONLOCAL VARIABLE n'* (*R*8). Declares that for this block the given identifier refers to the local variable with the same name from the given routine. This is just a special case of (2).

(4) *VARIABLE a TYPE AS FOR q* (*R*28). Defines a local variable with type the same as another; the latter must be currently declared. This is more commonly used without a routine given, i.e. where the parent variable given is a formal parameter name that refers dynamically (at compile time) to an actual variable from another routine or block.

(5) *PRESET* ⟨TYPE?⟩ *VARIABLE* $a=b$, $x=b(b+c)/2$. Declares the name of a local variable, and its type either explicitly or implicitly from the type of the expression. It then initializes the variable to the dynamic value of the expression.

(6) ⟨TYPE?⟩ *CONSTANT* $a = b$, $x = b(b + c)/2$. Has the same effect as (5) except that the variable cannot be subsequently reset.

(7) *ARRAY ELEMENT REFERENCE* $a=P[i+j+k]$, $r = Q[i, j]$. Here the address is calculated on initialization; all subsequent references to e.g. *a* and *r* will be direct to the scalar element so defined.

(8) *ARRAY ELEMENT NAME* $a = P[i + j + k]$, $r = Q[i, j]$. Any subsequent reference to *a* and *r* will be replaced by a cue to a secondary subroutine to calculate the address of the element.
ARRAY ELEMENT NAME'. As above but true substitution is used.

(9) ⟨TYPE?⟩ *EXPRESSION NAME* $x=4\mathbf{sqrt}(a+b)$. Here any subsequent reference to x will be treated as a call-by-substitution for the expression using a secondary subroutine. It cannot be used as a variable, and is in fact a local parameterless function.
⟨TYPE?⟩ *EXPRESSION NAME'* $x=4\mathbf{sqrt}(a+b)$. As above but true substitution is used.

(10) ⟨TYPE⟩ *FUNCTION NAME* $\mathbf{f}([\langle\text{TYPE/1}\rangle\ x], [\langle\text{TYPE/2}\rangle\ i]) = 2P[i](x + 1)/(a + b)$. The function name must be a single underlined small letter (cf. the function routine name which is an underlined small word).
This sets up a local definition equivalent to:
$[\langle\text{TYPE}\rangle\ r] = \mathbf{f}([\langle\text{TYPE/1}\rangle\ \textit{EXPRESSION}\ x], [\langle\text{TYPE/2}\rangle\ \textit{EXPRESSION}\ i])$
$r=2P[i](x+1)/(a+b)$.
There can be any number of formal parameters. A subroutine is compiled to calculate the value and it is called whenever the function is used.
⟨TYPE⟩ *FUNCTION NAME'* etc. . . . is the same except that true substitution is used.

(11) ⟨TYPE⟩ *FUNCTION VALUE* etc. . . . is similar to (10) but here the values of all operands other than those containing the formal parameters are frozen at their value on initialization.

(12) *OBEY* $a = \mathbf{sqrt}(2n)$; **if** $i = 0$: $a = 2a$. Here a line of imperative (and conditional) instructions can be written, but it must not contain a label or a label reference.

The instructions of (12) and the dynamic elements of other local declarations are obeyed serially on entry to the block and can only be obeyed on entry (i.e. the dynamic elements of declarations can only be obeyed once per entry into the block). If such dynamic instructions call other (closed) program routines this will be queried to remind the programmer to check that he does not refer to any stacked information not yet declared in the block.

Formal parameter specifications

It will have been noted that *VARIABLE* and *EXPRESSION* have been used in the specifications of formal parameters. As well as specifying the permitted syntax of the actual parameter this is also taken to specify whether or not the formal parameter (i.e. each actual parameter) is reset by the routine. Any further information as to the implementation of the parameter call is given in a separate specification, e.g. *VALUE*, *REFERENCE*, or *NAME* (i.e. substitution) or a serial number, e.g. *C*18. If this specification is not given, *VALUE* is assumed.

The specifications of formal parameters for **CLOSED** routines and functions include the following possibilities:

(13) [⟨TYPE⟩ *VARIABLE NAME v*]. Call-by-substitution on the variable.
(14) [⟨TYPE⟩ *VARIABLE REFERENCE v*]. Call-by-reference on the variable.
(15) [⟨TYPE⟩ *VARIABLE VALUE v*]. Call-by-value on the variable, actual parameter reset to the final value of the local variable on exit.
(16) [⟨TYPE⟩ *VARIABLE RESULT v*]. Variable declared local but *not* preset; actual parameter set to final value of local variable on exit.
(17) [⟨TYPE⟩ *EXPRESSION VALUE e*]. Call-by-value on the expression.
(18) [⟨TYPE⟩ *EXPRESSION NAME e*]. Call-by-substitution on the expression.
(19) [*DUMMY* ⟨TYPE?⟩ *i*]. This can be used in connection with (20). It must be matched in the call by a scalar identifier. If a ⟨TYPE⟩ is given, the declaration has the dual effect of declaring a formal parameter [*DUMMY i*] and a local variable ⟨TYPE⟩ *i*.
(20) [⟨TYPE⟩ *FUNCTION NAME* **f**(⟨TYPE?⟩ *i*)]. There can be any number of parameters (separated by commas). Each formal parameter of the function must have been declared as a previous formal parameter of the routine heading. If the previous declaration contains a ⟨TYPE⟩ there is no need to respecify it.

For each particular call on the routine a local function is set up in the cue; this function corresponds to the actual expression matching the parameter, with those scalar identifiers which match the actual parameters of the appropriate previous formal parameters treated as the formal parameters of the function. If a previous formal parameter is not a dummy a check is made that the actual parameter is a scalar, but there is not necessarily any connection between it and the formal parameter of the local function. For example:

FUNCTION
[*REAL s'*] = **sum** (**for**[*DUMMY INDEX i*]**from**[*INDEX EXPRESSION s*] **to** [*INDEX EXPRESSION f*] **of** [*REAL FUNCTION NAME* **g**(*i*)])
  s' = 0
  cycle *i* = *s*, 1, *f*
    s' = *s'* + **g**(*i*)
  repeat

The function could be called, for example, by:

P[*k*] = **sum** (**for** *j* **from** 1 **to** *n* **of** *Q*[*j*, *k*]/(**sqdf**(*j*, *k*) + 1))

and the local function created by the cue is in effect:

[*REAL r*] = **g**([*INDEX EXPRESSION j*])
  r = *Q*[*j*, *k*]/(**sqdf**(*j*, *k*) + 1)

Note that since *j* matches a *DUMMY* formal parameter, if it happened that *j* had been declared as a local variable of the routine containing the call there would be no connection at all between this variable and the *j* of the call.

Closely related to formal parameter (20) is:

(21) [⟨TYPE⟩ *FUNCTION VALUE* **f**(⟨TYPE?⟩ *i*)]. In this case the local function compiled by the cue is such that all operands other than those containing the formal parameters are frozen at their value on initialization.

Variations for formal parameters of open routines

If a routine is declared as **OPEN** or **DUAL** all the above specifications are equally valid. However, for open routines there are further variations on the set, and in the case of dual routines standard assumptions are made for the corresponding open versions of the closed variations, or specific alternatives can be given explicitly.

The variations possible when open routines are being used can be classified under three general headings:

(*a*) *True substitution.* Where call-by-substitution is specified, call-by-true-substitution can be specified instead. Note that in particular a *FUNCTION* call can be specified as a true substitution; so in the above example, if the function **sum** is defined as open and the *FUNCTION NAME* call specified as a true substitution, for the given call **g**(*i*) would be replaced by

Q[*i*, *k*]/(**sqdf**(*i*, *k*) + 1).

(*b*) *Free type.* The type of the formal parameter can be left out, so that the definition is valid for all types.

For example:

OPEN ROUTINE
interchange [*VARIABLE v*] **and** [*VARIABLE v'*].
VARIABLE t TYPE AS FOR v
  t = *v*
  v = *v'*
  v' = *t*

This routine therefore defines the interchange operation for all types and combinations of data. If the types are incompatible this will emerge in the usual way when the open routine is being compiled.

(*c*) *Open-option.* Where an expression or a variable is not being called by substitution, a variation can be specified so that if a particular actual parameter is such that its value can be accessed or reset as easily as the corresponding local variable, then the call will be implemented by true substitution instead.

For example if in the interchange routine an actual parameter is a scalar it will be uneconomic to call it by value. This entails copying the value of the scalar into a local location before entering the routine and copying it back at the end. It would be better to call the scalar by true substitution and avoid the two copies.

Thus where the required variant is parameter call number *C*41, say:

OPEN ROUTINE
interchange [*VARIABLE C*41 *v*] **and** [*VARIABLE C*41 *v'*].
VARIABLE t TYPE AS FOR v

$t = v$
$v = v'$
$v' = t$

This routine now not only defines the interchange operation for all types and combinations of data, but for the standard data-types automatically implements the parameter calls in the most efficient manner.

Further options that might be convenient for a compiler to implement are to choose automatically between implementing call-by-substitution by subroutine or by open sequences depending on the actual parameter. And in the case of functions an option could be provided to allow the compiler to substitute an appropriate output parameter for the result variable, in particular a program variable if the function is called simply to set a variable, e.g. as in the example of a call given in (20).

Restriction of variations

It might appear that the choice of alternatives and the variety of syntax possible in routine headings is unmanageable. These problems are inevitable in all aspects of programming as the power of available facilities and alternative language increases. The beginner must be provided with a simple subset of facilities and language to learn, and he can feel himself outwards in those directions dictated by his needs. In those cases where there is a strong common option amongst facilities the variant can be left out of the syntax.

In particular, for the formal macro facilities, a very small subset of alternative formal parameter specifications will provide a workable subset of facilities. There is no need for the programmer to be aware of the others until he needs to be aware of the practical considerations that necessitate them; then of course the categorizing of the alternatives will help to clarify the considerations.

In the case of the experienced programmer, it is expected that he will also tend to write his programs using the basic subset, and will use closed routines everywhere except where it is obvious that he will be needing open or dual routines. Then only when he reaches the point where his program is near the production stage need he alter the format specifications and bring forward any routines that should be open or dual to the front of the program. The alterations to the program need only be the addition of serial numbers to specify the precise implementation required for a formal parameter, and the addition of asterisks throughout the program where necessary if duals are being used. The addition of these symbols will not alter the description of the program, or interfere with its execution (except perhaps where side-effects need to be considered). The detail of how to achieve the optimum implementation can be largely isolated from the problem itself.

Particularly where the detail of the formal parameter implementation can be specified by adding a serial number to the small subset of common formal parameters, it should be possible for the additions to be made only to the format specifications, which will tend to be grouped together at the beginning of the program. The subsequent routine heading will then only be checked for consistency (not identity) against the format specification.

It is hoped that, contrary to what might be expected, the introduction of a powerful language and compiler will tend to cause an increase in the efficiency of the compiled program. This is achieved by the precision of parameter specification, the efficiency of the control mechanism, the avoidance of the stack, the use of open routines and duals defined specially for particular programs, and the use of library informal macros to deal with special situations maybe by direct use of machine code.

Implementation of macros

Textual Levels. It has been stated that SNAP is a one-pass language, i.e. all information must be declared before it is otherwise referred to. It is equally true that the implementation of the formal macros is one-pass. The source program string is read and dealt with in the usual way, except that when open or dual routines are encountered the compiler switches from "compiler mode" to "Compiler Compiler mode" (see Napper, 1965 and 1967) and the routines are stored as an extension of the compiler, i.e. as compile-time routines.

The only action taken when reading a macro is similar to that taken on reading an ordinary routine heading: check if the format has been declared; if so check the routine heading for consistency, and otherwise, open a file for this new routine. No further action is taken with respect to compiler mode activity except perhaps to note information given in the local declarations. If the next master declaration is another macro (or a set of macro formats) the system remains in Compiler Compiler mode, but otherwise it returns to compiler mode and deals with the source string that follows in the usual way. The program source string with the macros removed will be referred to as the *static text*. It is this text that the compiler processes in compiler mode.

A macro is not obeyed in compiler mode until an occurrence of a call for it in the static text, or until it is activated by such a call from a macro which in turn calls it. Thus any variables it may use from another routine need only have been declared before the first call on a macro (or the first call on a macro calling it); they do not have to be declared before the macro occurs in the original program text. The macro will be obeyed in compiler mode every time it is called. However, the first call is distinctive in that it will give any compile-map information that it was not possible to give when the routine heading was read. In the case of the first occurrence of the closed form of a dual routine a cue and subroutine will be compiled, but subsequently only a cue will be compiled.

As has been stated before, all true substitution is dynamic; there is no preprocessing of source program

symbol string and then compiling from the processed string. Where the static text calls macros, or where such macros call further macros there will be more than one dynamic level of routines current in the compile-time stack. So a standard Compiler Compiler mechanism is used to keep a record of the dynamic textual level of each identifier in its analysis record. Note that this dynamic textual level stack is not the same as the organizational compile time stack of a conventional compiler. This *static* textual level stack has one level for each nesting of blocks (and routines) current in the program text. The property of any identifier is found by working through this stack from bottom to top until the name is found.

However, with formal macros operating there is a 2-dimensional stack: one dimension gives the set of macros currently active back up to the level of static text, and the other gives any nesting of blocks current in each macro and the static text. When the characteristics of an identifier are required the compiler first finds the dynamic textual level of the identifier. In the case of a macro it then proceeds to search the property lists of any blocks created by this call on the macro, and for the static text it searches the lists of any blocks current within the routine or the top level program. Then in either case it searches the property list of the macro or routine, then any routine it is within the scope of, and so on back to the list of global information.

NOTE. There is a special case for certain macros where it is desirable to allow the sequence of levels searched to pass from the dynamic level of the routine on to the dynamic level of the block calling it, and from then up as described above. This is required for "correction" macros which can be inserted at the beginning of the program, and for other small non-parametric once-called macros, e.g. where a routine has only been defined to make the description neater. In these cases it may not be powerful enough to define the macro within the scope of the calling routine since some of the identifiers it uses may only be defined within a local block (e.g. covered by a "where" clause). And it may be inconvenient, or in the case of corrections impossible, to include all such identifiers as parameters of the routine call. This case will be covered by a **CORRECTION** facility, although a special form with a different master declaration will cover the case where the macro is part of the intended program description.

NOTE. It might be thought that the 2-dimensional stack is largely an academic concept; open routines will generally be too short to require any significant block structure. However, there are two classes of macro where this is not the case. First is the case of large routines that are called once only, where it may be desirable for various reasons to change them from closed routines to open routines when the final compilation of a production program is made. Second is the case of informal macros which compile into large open routines. These derive from instructions with complicated formats, with many options of syntax and parameters, including indefinite lists of parameters or clauses, for example the "reorder" instruction illustrated in Napper (1966 and 1967), which has been used to compile tailor made open routines with length ranging from 25 to 75 basic instructions of the language. Note that even if it was possible to simulate the action of such a macro with a closed routine it would be uneconomic as it would have to be executed interpretively.

Dynamic Substitution. It should be observed that a two-pass macro-generation of a program—i.e. one that, given a set of macros and a source program string including macro calls, generates a secondary string and then compiles this secondary string—loses the information as to which dynamic textual level substituted identifiers belong.

For example, consider the macros:

```
OPEN ROUTINE
convert [REAL VARIABLE NAME' a]
INDEX i
    i = intpt(a/2)
    add (a+C[i+1])/2
    a = C[i]

OPEN ROUTINE
add [REAL EXPRESSION NAME' a]
INDEX i
    i = intpt(4(a+c)/c)
    S[i] = S[i] + 1
```

If these had been called for example by the instruction: **convert** $Y'[i]$, then the dynamic substitution (enclosing each macro call in a block) is:

```
BEGIN          ((formal parameter a = actual Y'[i]))
INDEX i
    i = intpt(Y'[i]/2)
    BEGIN              ((now a = (Y'[i]+C[i+1])/2))
    INDEX i
        i = intpt(4((Y'[i]+C[i+1])/2 + c)/c)
        S[i] = S[i] + 1
    END
    Y'[i] = C[i]
END
```

Here i in the 6th line should really be referring in turn to the local variable i of three different routines, but it will be interpreted as the local i of routine "**add** . . ." in each case.

This problem can be overcome by injecting declarations at the beginning of the block enclosing the macro to associate the formal parameters with names one level up; then the formal names only are used in the textual level of the routine.

The above sequence now becomes:

```
BEGIN
REAL NAME' a = Y'[i]
INDEX i
    i = intpt(a/2)
    BEGIN
    REAL NAME' a = (a+C[i+1])/2
```

```
    INDEX i
    i = intpt(4(a+c)/c)
        S[i] = S[i] + 1
    END
    a = C[i]
END
```

However, the simple substitution model is now lost and the compiler will have to contain quite sophisticated string-handling and analysis machinery to recreate the true substitution. In other words it is doubtful whether formal macros can be implemented conveniently except by using a first-order language to write the basic compiler in; a 2-pass system comprising a macro-generator and an assembly language compiler would seem to be inadequate.

In fact, although it is a one-pass system, SNAP implements formal macros in a similar way. Anyone familiar with the Compiler Compiler will have noticed that open routines do not contain phrase-identifiers, the special "phrase-variables" that implement dynamic substitution; it might be expected that they would be used in place of the formal parameter names. Similarly, the routine (and routine format) declarations of SNAP do not satisfy the basic formats of the language-defining language; in particular the routine heading should contain the names of the formal phrase-variable parameters. In fact these declarations are read in compiler mode, any required information is extracted from them (as for closed routines), and then they are converted into format routine declarations of the Compiler Compiler language. In addition the appropriate call-declarations are inserted after the routine heading to implement the parameter calls, e.g. *CALL REAL* [VARIABLE] *a BY TRUE SUBSTITUTION*, and *CALL REAL* [EXPRESSION] *a BY TRUE SUBSTITUTION*.

Usually the effect of these declarations is to make a call by value, reference, or ordinary substitution, in which case the characteristics of the formal name will be recorded as those of the local object set up by the call. Where an open-option call-declaration makes a choice between one of these calls and true substitution, it is only likely to choose true substitution if the parameter is a simple object, in which case it can store its characteristics as the characteristics associated with the formal name. It is only in the less common cases where true substitution is specified (e.g. because the formal name is only referred to once in the routine) that it will be necessary to associate with the formal name (just) the analysis record of the phrase-variable (i.e. of the actual parameter). Inside such an analysis record, the record of each identifier will contain a reference to the dynamic textual level.

Thus it would be less efficient always to use a phrase-variable instead of a formal name to implement formal parameter calls in open routines. Where dynamic substitution is required it is little extra trouble (in the context of the standard Compiler Compiler machinery for analysing variables and expressions) to return to processing an analysis tree at any point where on inspection an identifier (apparently a "fruit" of the tree) turns out to refer to a further subtree. However, in informal macros phrase-identifiers can be used instead of formal parameter names if this is more convenient or efficient.

Informal macros

There is not space to give a detailed account of the uses of informal macros and their implementation. However, the following example of a simple informal macro for a "choose random" instruction should give some indication.

In English Mode a call on such a routine might be, for example:

choose random direction left (*with weight p'*), *right* (*q'*), *forwards* (2(10−*p'*−*q'*)/3), *or backwards* ((10−*p'*−*q'*)/3).

Consider the form in Mathematical Mode:

choose random *d* = *l* **wt** *p'*, *r* **wt** *q'*, *f* **wt** 2(10−*p'*−*q'*)/3, *b* **wt** (10−*p'*−*q'*)/3.

The object of the instruction is to set a variable, e.g. *d*, to one of an indefinite set of expressions, e.g. 4 : *l*, *r*, *f*, *b*, at random in proportion to the corresponding weights, e.g. *p'*, *q'*, 2(10−*p'*−*q'*)/3, and (10−*p'*−*q'*)/3.

The format of the instruction, in Compiler Compiler notation, is:

```
choose random [VARIABLE] = [EXPRESSION]wt
        [EXPRESSION] [MORE-CHOICES?]
Where PHRASE [MORE-CHOICES?] = [,]
        [EXPRESSION] wt [EXPRESSION]
        [MORE-CHOICES?], NIL
```

And the corresponding routine is:

```
INFORMAL ROUTINE
choose random [VARIABLE] = [EXPRESSION/1]wt
        [EXPRESSION/2][MORE-CHOICES?].
CALL [VARIABLE] c BY RESULT/TRUE
    SUBSTITUTION
CALL [EXPRESSION/2] BY VALUE/TRUE
    SUBSTITUTION
IN OPTIONAL LIST [MORE-CHOICES?]:
  FOR 2ND [EXPRESSION] IN EACH ITEM, CALL
    [EXPRESSION] BY VALUE/TRUE SUBSTITU-
    TION
REAL x
  LET [MORE-CHOICES?/1] = [MORE-CHOICES?]
  LET [EXPRESSION] = [EXPRESSION/2]
1) → 2 UNLESS [MORE-CHOICES?/1] ≡ [,]
          [EXPRESSION/3]wt [EXPRESSION/4]
          [MORE-CHOICES?/1]
       LET [EXPRESSION] = ([EXPRESSION])
         + ([EXPRESSION/4]);    → 1
2)     set random x between 0 and [EXPRESSION].
3) → 4 IF NO [MORE-CHOICES?]
       x = x − [EXPRESSION/2]
       if x < 0 : c = [EXPRESSION/1]; finish.
       LET [MORE-CHOICES?] ≡ [,]
         [EXPRESSION/1]wt[EXPRESSION/2]
         [MORE-CHOICES?];    → 3
4)     c = [EXPRESSION/1]
```

Note that the routine heading contains no formal parameter specifications of SNAP, but only the true phrase-variables of the Compiler Compiler. The first job of the body of the routine is to comb the informal phrase structure of the format and carry out all the call-declarations required by the phrases corresponding to the formal SNAP parameters, e.g. [VARIABLE] and [EXPRESSION].

Thus *CALL* [VARIABLE] *c BY RESULT/TRUE SUBSTITUTION* is the call-declaration of formal parameter [*VARIABLE C*42 *c*], where *C*42 is the open-option version of parameter specification no. 16 (cf. *C*41 which is the open-option version of no. 15). In fact [*VARIABLE C*42 *c*] could have been written in the routine heading instead of [VARIABLE], and the standard routine-heading processing machinery would have changed it into [VARIABLE] and then inserted the call-declaration at the beginning of the routine automatically.

CALL [EXPRESSION/2] *BY VALUE/TRUE SUBSTITUTION* is the call-declaration of the open-option version of parameter specification no. 17. Note here that no formal name is given. This is because the first weight is subsequently treated in the same way as the rest of the list of weights and it is not convenient or necessary to give them all explicit formal names. If the option is taken which sets up a local object (i.e. call-by-value), an "artificial name" is set up behind the scenes to refer to this new object and the phrase-variable is reset to it. Then in either case all references to the parameter are made using the phrase-variable, i.e. by dynamic substitution.

The third instruction of the routine is a special cycle control construction provided by SNAP (for technical reasons) to process the list of choices carrying out the appropriate calls. Note that the first expression in each item, i.e. the choice, can be left as a dynamic substitution since it is only referred to once in the open routine generated by the macro (this is also true of the last weight).

The declarations are completed by *REAL x*. Then follows a sequence in basic Compiler Compiler phrase-handling instructions which runs through the list of choices explicitly and forms an expression which represents the sum of the given weights. Finally, starting at Compiler Compiler label 2, comes the actual generation of the imperative instructions of the open routine. Note that "set random" is assumed to be already defined as part of the existing (nonbasic) language.

The effect of this informal macro for the given call is therefore to generate the following open routine. Note that call-declarations that have had no effect have been omitted, and for those that have, the effect is shown by the equivalent local declaration used by the option exercised. The "artificial names" are shown as *e***1** and *e***2**.

BEGIN
VARIABLE NAME $c = d$
CONSTANT $e\mathbf{1} = 2(10-p'-q')/3$
CONSTANT $e\mathbf{2} = (10-p'-q')/3$
REAL x
  set random x **between** 0 **and** $(((p')+(q'))+(e\mathbf{1}))+(e\mathbf{2})$.
    $x = x - p'$
    if $x < 0 : c = l$; **finish.**
    $x = x - q'$
    if $x < 0 : c = r$; **finish.**
    $x = x - e\mathbf{1}$
    if $x < 0 : c = f$; **finish.**
    $c = b$
END

Thus the informal macro compiles an open sequence that is close to the optimum coding for all possible combinations of actual parameters. If the programmer was forced to write the sequence out himself the compiled code would tend to be less efficient as he would not bother to give thought to the optimum coding, e.g. to use local variables where necessary for efficiency. Conversely, if a closed routine was available to carry out this operation, any parameters that did not have to be set to local values would have to be copied unnecessarily, and the cue required for the subroutine would not be significantly shorter than the actual open sequence.

To illustrate this latter point, consider the following comparison between an open version of this instruction and an equivalent closed version. Also consider the comparison with an informal macro that has been written in a more careful way, using simple "micro-programming" to allow the accumulator to be referred to as an operand (e.g. $d = r$ becomes $A = r$; $d = A$). Assume that a closed subroutine exists that allows up to 5 choices (zeros being used for choices and weights not required)—note of course that this restricts the data types of the variable being set and of the possible choices. Assume that the subroutine has been written using the same simple micro-programming but using a closed sequence for the instruction "set random" instead of the open sequence used for the informal macro. Assume that the routine changing sequence required in the cue is 3 machine instructions in addition to the parameter calls (which require 10 copies on entry and one on return). Assume that the random generation sequence is 5 instructions.

Then if all the actual parameters are simple (i.e. accessible in one instruction), and there are 4 choices, assuming a one-address code and one accumulator we get the following figures:

	M/C INSTRUCTIONS COMPILED	AV. M/C INSTRUCTIONS OBEYED
Open with micro-programming:	25	20
Open without micro-programming:	36	25
Cue for closed routine:	25	(in all, about) 60

Note that in a more sophisticated version of the informal macro it could be arranged that the weights

could be optional; if any were missing, **wt** 1 would be assumed. Further refinements that could be included are to add together any preset weights at compile time instead of at execution time, or to check if all the weights were left out, in which case a multiway switch could be used, or to check if there were only two equal choices, in which case a simpler yes/no output from the random generator could be used.

This example should give some idea of the potential power of informal macros. With the freedom given by the use of phrase structure notation it should be possible to allow the source programmer to describe larger sections of program in a single instruction. This is of advantage to him since it means that he can write at a higher level of description and therefore in a language closer to the program-independent formulation of his job. At the same time the expert compiler-writer can acquire a larger block of source program within which he can exercise his experience to optimize the object code efficiency. In doing so he should more than offset the losses inherent in more generalized instructions; and in the case where there is more than one way of carrying out the operation, options can be introduced into the instruction format to allow the programmer to give further information if he wants to indicate the most suitable method.

A further large area of informal macro usage is to define instructions (with simple syntax) to carry out operations not provided by the basic language, e.g. to deal with new data-types. A library of macros can be built up, as pure/relocatable routines of the basic compiler, and when a program is being translated the quick access store need only contain the basic compiler and those macros actually required by the programmer. This means that features that might otherwise be permanent features of a language can be left as options in the library. Note that in the case of a simple informal macro like "choose random", unoptimized versions of a compiler routine can be written without expert knowledge, to be used until the need for a more expert version is established.

Acknowledgements

I should like to thank Mr. R. A. Brooker for his continuing advice, and the Science Research Council for supporting my work for the previous two years.

References

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

BARRON, D. W. *et al.* (1963). The main features of CPL, *The Computer Journal*, Vol. 6, p. 134.

NAPPER, R. B. E. (1965). An Introduction to the Compiler Compiler, Technical report, Dept. of Computer Science, Manchester University.

NAPPER, R. B. E. (1966). A System for Defining Language and Writing Programs in "Natural English", *Formal Language Description Languages for Computer Programming* (Ed. Steel), Amsterdam: North Holland.

NAPPER, R. B. E. (1967). The Third-Order Compiler: a Context for Free Man Machine Communication, *Machine Intelligence* 1 (Ed. D. Michie), Edinburgh: Oliver & Boyd.

BROOKER, R. A. *et al.* (1966). The main features of Atlas Autocode, *The Computer Journal*, Vol. 8, p. 303.