An index directed compiler

By R. A. Brooker* and S. R. Clark†

This paper describes a system for discriminating between many possible combinations of conditions and invoking a routine to deal with them. It is illustrated with reference to a situation which may arise in a compiler, namely the treatment of different combinations of operands and operators.

We describe the technique used for organizing the system of assembly routines which provide the special arithmetic facilities embedded in Atlas Autocode (Clark and Lunnon, 1966). These amount to a language within a language and consist of declarations describing the types and names of the data, and a single form of imperative statement for processing the data. Examples of declarations are

dc array *H*, *K*(1 : *n*)
mr(*I*) *a,b,c*

These declare and reserve space for two double-length complex arrays *H*, *K* of dimensions (1 : *n*) and three multi-length real numbers of length *I* (previously computed) *a,b,c*.

In general declarations take the form

⟨scalar type⟩⟨array type?⟩⟨name list⟩

where the ⟨array type⟩, if not present, signifies scalar.

The scalar types are

sr (single length real) **sc** (single length complex)
dr (double length real) **dc** (double length complex)
mr (multi-length real) **mc** (multi-length complex)
sb (bit pattern)

The precision (**s,d,m**) and mode (**r,c,b**) are effectively separate factors although they are not themselves recognized delimiter symbols.

An arithmetic statement is an infix expression with binary and right unary operators (i.e., placed on the right of the operand they refer to). All operators have uniform precedence, and the expression is processed from left to right. For example

[*a* **sqrt** + *b* * [*c* **chs** + (1)] + *H*(1) **im** . . .]

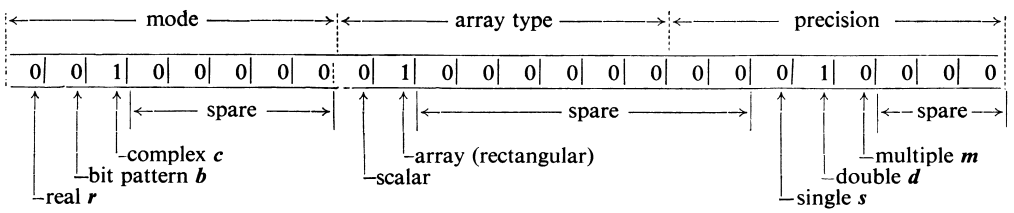
means: fetch *a*, take its square root, add *b*, multiply by [*c* **chs** + (1)], add *H*(1), take the imaginary part, etc.

At each stage the result is left in the "accumulator" (a dynamically varying location). Evaluation reduces therefore to a series of unary and binary operations in which the value and type of the result depend on the operator and the value and type of the operand (or operands) involved. Thus if *t* and *v* denote type and value and α and β denote unary and binary, we have

$$(A\alpha)_t = \alpha_t(A_t) \qquad (A\alpha)_v = \alpha_v(A_v,A_t)$$
$$(A\beta B)_t = \beta_t(A_t,B_t) \qquad (A\beta B)_v = \beta_v(A_v,B_v,A_t,B_t)$$

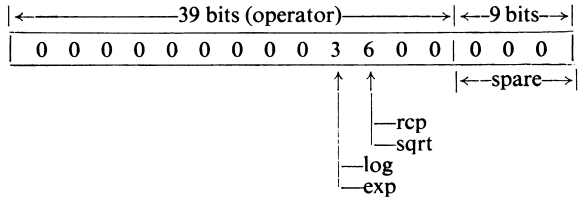
The functions α_t α_v β_t β_v are described in Brooker and Clark (1966): they amount to a definition of the language. For each combination of operator and operand types the compiler calls in a specified routine (although a single routine may deal with several combinations) to compile the appropriate instructions, either an open sequence or a call for a run-time subroutine. The mechanism by which the compiling routine is selected will now be described.

The type of an operand is represented in the store of the machine as follows



For each of the 3 components of the type, namely, precision, mode, and array type there are 8 possibilities, some of which are spare. The full type of an operand is therefore given by a single bit in each of the 3 sections of the word. The digits shown correspond to **dc array** [The other components of the type, the dimensionality of the array and the associated bound pairs, are treated separately.]

An operator is represented internally as follows (in octal digits)

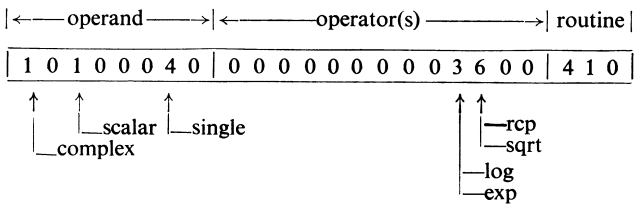


* *Department of Computer Science, The University, Manchester*, 13.
† *Commonwealth Scholar, Department of Computer Science, The University, Manchester*, 13.

Downloaded from https://academic.oup.com/comjnl/article/10/1/60/375987 by guest on 19 April 2024

each bit corresponding to one of 39 unary (or 39 binary) operators. We always know whether it is a unary or binary operator we are dealing with. The reason for the 9 spare digits will become clear shortly.

A unary operation is described by a 72-bit pattern, the code for the operand and the code for the operator, 3 half-words in all. The routine for dealing with a unary is found by comparing this 72-bit pattern* with an ordered table of 72-bit entries, each of which specifies, by means of suitable bit pattern, a class of operations and (in the 9 "spare" bits) a routine to deal with it. The entry selected is the first one which has bits in the same position as those of the unary code on hand (although it may have other bits as well). Thus the entry (in octal digits)



traps all applications of these operators on single-length complex numbers, and calls in the routine R264 which is reproduced in **Fig. 1.** This, like all the routines, is in symbolic machine language (ABL) but nevertheless its meaning should be clear from the accompanying notes.

A similar scheme involving a table of 96-bit patterns is used for the binary operations.

The interpretation of multiple digits in the operator section of a table entry is fairly clear: they mean "if any of these operators are present . . .". Multiple digits in the operand section define a combination of types. For example digits corresponding to single, double, real, complex and scalar will trap the operands **sr**, **dr**, **sc** and **dc.**

A system for adding entries to and deleting entries from the table allows the facilities to be built up in a piecemeal way. The easier cases (e.g., the one described) can be provided fairly quickly; the more difficult cases can be diverted into dummy routines which simply monitor the fact that these facilities are not yet available.

Existing facilities can be made more efficient by introducing new routines to deal with special cases. All that is necessary is to insert an entry to trap the particular combination of operator/operands before the entry which would otherwise catch them. (This is done automatically by the system.)

Finally the operations not trapped by any of the entries correspond to illegal situations (e.g., **sb sin**) and fall through the bottom of the table, there to be monitored.

* Atlas does not permit us to compare 72-bit words in a single operation and so we discriminate first on the 1st half word, and when agreement is reached we proceed to match the 2nd and then the 3rd half words. Similarly when scanning the "binary" table we discriminate in 4 stages.

```
R264                                        | sc unary
101     99    62    −7A1                    |fetch extracode
113     99    1     0                       |and plant with
114     50    1     0                       |address of operand
113     51    1     0·4                     |(B62 = operator number)
121 ]   99    0     J1456u2 + 83u4          |plant "dump" extracode
113     99    1     1                       |with address
114     54    1     1                       |of result
113     55    1     1·4
124     1     0     2                       |advance instr. pointer
121     60    58    0                       |set type of result
121     56    54    0                       |set pointer to
121     57    55    2                       |end of "accumulator"
121     −1    90    0                       |exit
1) 1414 83    0     0                       |rcp (7)
   1410 83    0     0                       |sqrt (8)
   1400 83    0     0                       |log (9)
   1402 83    0     0                       |exp (10)
   Z
```

The significance of this particular class of operations is that they can be implemented by extracode instructions. The function of this routine is to select the appropriate extracode, insert the address of the "accumulator" (which is held in the stack) and compile the instruction. Not all routines are as simple as this of course, and the writer must understand the strategy of the stack and the role of the relevant index registers.

Fig. 1.—Routine R264

```
delimiter   act,16
delimiter   arccos,46
delimiter   arcsin,45
delimiter   arctan,47
delimiter   arg,37
delimiter   cos,43
delimiter   chs,32
delimiter   cvt,18
delimiter   conj,54
delimiter   dc,204
delimiter   dr,203
delimiter   exp,41
delimiter   fracpt,34
delimiter   intpt,33
delimiter   int,35
delimiter   im,52
delimiter   log,40
delimiter   load,31
delimiter   mc,206
delimiter   mr,205
delimiter   mod,36
delimiter   mdv,27
delimiter   nl,50
delimiter   norm,58
delimiter   prt,48
delimiter   rarray,208
delimiter   rcp,38
delimiter   re,51
delimiter   rd,53
delimiter   rcm,30
delimiter   scalar,207
delimiter   sb,200
delimiter   sr,201
delimiter   sc,202
delimiter   sin,42
delimiter   sqrt,39
delimiter   sp,49
delimiter   shift,19
delimiter   to,15
delimiter   tan,44
delimiter   trp,60
```

Fig. 2.—Delimiters

Downloaded from https://academic.oup.com/comjnl/article/10/1/60/375987 by guest on 19 April 2024

Setting up the tables

The source language uses underlined† "delimiter" words to denote scalar types, array types, unary and binary operators (except for the conventional $+ - * / \uparrow < \leqslant > \geqslant = \neq \pm$ most of which are also used by the host language). Each delimiter is associated with an item number which serves as its internal representation in the machine.

The system provides for up to 30 type delimiter words and 39 operators of each kind with item numbers in the ranges

⟨scalar type delimiter⟩ or ⟨array type delimiter⟩	200–229
⟨unary⟩	31–60, 239–247
⟨binary⟩	1–30, 230–238

The operators $+ - * / \ldots$ take fixed item numbers (as a result of their use in Atlas Autocode) but otherwise they can be assigned arbitrarily.‡ This is done by a statement of the form

delimiter ⟨name⟩ ⟨item number⟩

For example

delimiter *sin*, 42
delimiter *sr*, 201

allows **sin** and **sr** to be used as a unary operator and a scalar type respectively.

After introducing a scalar type delimiter such as **sr** the statement

type sr, *40000040, 215

arranges for a routine R215 to be called in to deal with all statements starting with **sr**. These will in fact take the form

sr ⟨name list⟩

*40000040 is the internal representation of the type in octal form.

Similarly **type rarray,** *00040000, 302

arranges for a routine R302 to be called in as a subroutine during the processing of statements of the form ⟨scalar type⟩ **rarray** ⟨dimensional structure⟩ ⟨name list⟩ ⟨scalar type⟩ may consist of more than just the scalar type delimiter as e.g., in

mr (1) **rarray** *A*, *B*, *C* (1 : *n*)

scalar type — array type

To write routines such as R215, R302 requires of course a knowledge of the structure of property lists and the strategy of storage reservations.

† Printed in bold type in this paper.

‡ The actual item number is irrelevant but if two or more names are given the same item number then they will be equivalent (a fact which is sometimes useful). The system will, on request (see Monitoring), print a list of currently assigned item numbers. Delimiters such as ***delimiter***, ***type***, ***unary***, ***binary*** and certain others are built-in to the system.

Finally, operations are introduced by listing the operators and operand involved. For example

unary (rcp, sqrt, log, exp) (sc, scalar) (264)

defines the table entry discussed earlier.

Note: replacing **sc** by **sr**, **sc** would trap only **sr** and **sc** operands but replacing **sc** by **sr, dc** would result in the same bit pattern as **sr**, **sc**, **dr**, **dc** and would in fact trap all these types. In other words it is as if we had written **(s, r, d, c)**, but single character delimiters are not permitted.

Making changes to the tables

To facilitate changes in the system unary, binary, and delimiter entries may be removed from their respective tables. For example

delete delimiter *sin*, 42

will remove **sin** from the delimiter table, and

delete unary (sqrt, rcp, log, exp) (sc, scalar) (264)

will remove from the unary table the entry which corresponds to the listed operators/operands. Entries in the type table are located according to their item numbers, and thus if a new type meaning is given to an existing delimiter it simply cancels the previous definition.

Monitoring

Monitoring statements exist which allow one to obtain an up to date documentation of the facilities implemented in a particular version of the compiler. Thus

monitor unary
monitor binary
monitor type
monitor delimiter

will print out the state of the relevant tables.

Figs. 2, 3, 4, 5 show how the system has grown in practice. Fig. 2 introduces the delimiters, Fig. 3 the type statements, Fig. 4 the unary operations, and Fig. 5 the binary operations. Together they provide an "index" to the system which is particularly useful in locating faults. The monitoring statements enable us to keep up to date. Moreover the users can to some extent keep abreast of developments without waiting for Computer Service bulletins.

Limitations of the type description

It is interesting to examine the limitations to operand type description imposed by the simple system we have adopted, which provides $8 \times 8 \times 8$ alternatives. Firstly all "structures" must be homogeneous, that is the real and imaginary parts of a complex number, the elements of an array, etc., must all be of the same individual type. The designation of the 3 levels as precision, mode, array type is arbitrary. Thus for example suppose we wish to introduce interval arithmetic. It would be convenient

Downloaded from https://academic.oup.com/comjnl/article/10/1/60/375987 by guest on 19 April 2024

```
type    sb,      *20000040, 215
type    sr,      *40000040, 215
type    sc,      *10000040, 215
type    dr,      *40000020, 215
type    dc,      *10000020, 215
type    mr,      *40000010, 216
type    mc,      *10000010, 216
type    rarray,  *00040000, 34
type    scalar,  *00100000, 215
```

Fig. 3.—The type statements

to describe interval as a mode in which case the components (the limits of the interval) could be of any designated precision. Alternatively if we **fixed** the precision of its components (and there are good reasons for limiting it, say, to multiprecision) an interval could be introduced at the precision level, e.g., **mi r** (1). Now the complex arithmetic routines are so arranged that the real and imaginary parts can be of any precision (provided they are both the same) so that once the routines for interval arithmetic are written, complex interval arithmetic (should it ever be needed) is automatically available. The same principles would apply if we were to introduce other modes. For example defining **polynomial** as a mode would enable the coefficients to be of any precision (including **mi**). They could not however be complex (**c**). This would only be possible if **polynomial** were an "array", in which case we would forgo the possibility of handling genuine arrays of polynomials.

The semantics

We have already mentioned that the routines which make up the system are written in assembly languages (i.e., machine code). This is for reasons of space economy and efficiency (particularly for the more primitive types). It is almost certainly possible to write some of the routines in a higher level language, probably ABC itself, and this is being looked into.

Acknowledgement

The authors wish to acknowledge many useful conversations with Mr. W. F. Lunnon of the Department of Computer Science.

```
unary    (sp,nl) (sb,sr,sc,dr,dc,mr,mc,scalar,rarray) (260)
unary    (prt,rd) (dr,mr,scalar) (259)
unary    (prt,rd) (sc,dc,mc,scalar) (245)
unary    (re,im) (sr,sc,dr,dc,mr,mc,scalar) (258)
unary    (load,chs,intpt,fracpt,int,mod,arg,rcp, c
         sqrt,log,exp,sin,cos,tan,arcsin,arccos, c
         arctan,prt,rd) (sr,scalar) (262)
unary    (load,chs,intpt,fracpt,int,mod,arg,rcp, c
         sqrt,log,exp,sin,cos,tan,arcsin,arccos, c
         arctan) (dr,mr,scalar) (263)
unary    (load,chs,intpt,fracpt,int,mod,arg,rcp, c
         sqrt,log,exp,sin,cos,tan,conj) (sc,scalar) (264)
unary    (load,chs) (dc,mc,scalar) (237)
unary    (mod) (dc,mc,scalar) (238)
unary    (arg) (dc,mc,scalar) (239)
unary    (rcp) (dc,mc,scalar) (240)
unary    (sqrt) (dc,mc,scalar) (241)
unary    (log) (dc,mc,scalar) (242)
unary    (exp) (dc,mc,scalar) (243)
unary    (sin) (dc,mc,scalar) (231)
unary    (cos) (dc,mc,scalar) (232)
unary    (tan) (dc,mc,scalar) (233)
unary    (conj) (dc,mc,scalar) (250)
unary    (norm) (sr,sc,dr,dc,mr,mc,rarray) (247)
unary    (trp) (sr,sc,dr,dc,mr,mc,rarray) (249)
unary    (load,chs,intpt,fracpt,int,mod,arg,rcp, c
         sqrt,log,exp,sin,cos,tan,arcsin,arccos, c
         arctan,prt,rd,conj) (sb,sr,sc,dr,dc,mr, c
         mc,rarray) (265)
```

Fig. 4.—The unary operations

```
binary   (+,−,*,/) (sr,dr,mr,scalar) (sr,dr,mr,scalar) (203)
binary   (≤,≥ <,>,=, ≠ ) (sr,dr,mr,scalar) (sr,dr,mr,scalar) (214)
binary   (to) (sr,dr,mr,scalar) (sr,dr,mr,scalar) (223)
binary   (act) (sr,dr,mr,scalar) (sr,dr,mr,scalar) (234)
binary   (cvt) (sr,dr,mr,scalar) (sr,dr,mr,scalar) (228)
binary   (to,cvt) (sr,dr,mr,scalar) (sb,scalar) (253)
binary   (to,cvt) (sb,scalar) (sr,dr,mr,scalar) (254)
binary   (shift) (sb,scalar) (sr,dr,mr,scalar) (255)
binary   (+,−,and,or, ≢ to,cvt) (sb,scalar) (sb,scalar) (256)
binary   (+,−,*,/) (sr,dr,mr,scalar) (sc,dc,mc,scalar) (221)
binary   (to,cvt) (sr,scalar) (sc,scalar) (225)
binary   (+,−,*,/) (sc,dc,mc,scalar) (sr,dr,mr,scalar) (220)
binary   (+,−,*,/) (sc,scalar) (sc,scalar) (213)
binary   (to,cvt) (sc,scalar) (sc,scalar) (224)
binary   (to) (sr,dr,mr,scalar) (sc,dc,mc,scalar) (227)
binary   (cvt) (sr,dr,mr,scalar) (sc,dc,mc,scalar) (230)
binary   (+,−,*) (sc,dc,mc,scalar) (sc,dc,mc,scalar) (222)
binary   (/) (sc,dc,mc,scalar) (sc,dc,mc,scalar) (212)
binary   (to) (sc,dc,mc,scalar) (sc,dc,mc,scalar) (226)
binary   (cvt) (sc,dc,mc,scalar) (sc,dc,mc,scalar) (229)
binary   (rcm) (sr,sc,dr,dc,mr,mc,rarray) (sr,sc,dr,dc,mr, c
         mc,rarray) (246)
binary   (mdv) (sr,sc,dr,dc,mr,mc,rarray) (sr,sc,dr,dc,mr, c
         mc,rarray) (251)
binary   (+,−,*,/,↑,≤,≥,<,>,=, ≠ ,and,or, ≢ ,to,act,cvt,shift) c
         (sb,sr,sc,dr,dc,mr,mc,scalar,rarray) (sb,sr,sc,  c
         dr,dc,mr,mc,scalar,rarray) (257)
```

Fig. 5.—The binary operations

References

BROOKER, R. A., ROHL, J. S., and CLARK, S. R. (1966). "The main features of Atlas Autocode", *The Computer Journal*, Vol. 8, p. 303.

CLARK, S. R., and LUNNON, W. F. (1966). "Multiple precision arithmetic in Atlas Autocode", Letter to *The Computer Journal*, Vol. 9, p. 174.

BROOKER, R. A., and CLARK, S. R. (1966). "Notes on the Special Arithmetic Statements in Compiler ABC", Manchester University.

BROOKER, R. A. (1964). "A Programming package for generalized arithmetic", *Comm. A.C.M.*, Vol. 7, p. 119.

Downloaded from https://academic.oup.com/comjnl/article/10/1/60/375987 by guest on 19 April 2024