

RCC—A user-extensible systems implementation language

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

RCC, an acronym for the Revised Compiler Compiler, was designed as a comprehensive revision of the original Brooker-Morris Compiler Compiler. Because of the inherent ability of the Compiler Compiler to generate an extensible compiler, implementing RCC in RCC has meant that it is itself extensible. The language provided by RCC has been designed at as high a level as possible consistent with the generation of efficient object code. Although the language processing facilities which are available mean that using RCC as a compiler compiler is still a major application area, the emphasis has now changed to that of providing a more general systems implementation language in which the user can make powerful extensions to the syntax and semantics with comparative ease and without significant loss of efficiency.

(Received July 1978)

The Revised Compiler Compiler, RCC, was designed in the late 1960s as a comprehensive revision of the Compiler Compiler of Brooker and Morris (Brooker *et al.*, 1963). The main aims were to turn the basic language of the Compiler Compiler into a more conventional and high level form, and to redesign the implementation to remove certain inefficiencies in the system (Napper, 1968). In fact the basic language has been completely replaced, and the special-purpose language-processing machinery has been revised and extended considerably, so that only the basic ideas are easily recognisable as deriving from the Compiler Compiler.

RCC provides a Systems Implementation Language (SIL) suitable for general systems and non-numeric applications, together with special facilities designed to help compiler writing, i.e. it is a classical compiler compiler. In particular it provides a formal language for defining, recognising, analysing and synthesising syntactic elements, based on an extended BNF, using a new data type *phrase*.

However RCC is implemented in RCC, using its formal language-processing facilities, and as a result any user who understands the use of *phrase* variables can make powerful extensions to the syntax and semantics of RCC with comparative ease and without significant loss of efficiency.

Therefore the emphasis of RCC has changed from that of providing a compiler compiler to that of providing a SIL with a high degree of user extensibility, which enables highly descriptive programs to be written with minimal loss of efficiency compared with hand coding. This feature of RCC is of course independent of the application for which it is being used. Moreover RCC provides special facilities for writing compilers, and for carrying out general manipulation and transformation of language.

The most familiar mechanism of extensibility is the 'primary' routine, which adds a new statement to the RCC language, with its own syntax and semantics. At its simplest such a routine reads like a syntax macro. However it is possible for a user to write powerful extensions to the language by using the language-processing facilities, but without needing to know about the detail of the RCC implementation. The implementation automatically integrates such primary routines into the compiler in such a way that the generation of the in-line code corresponding to an occurrence of a new statement is carried out as simply and efficiently as if the statement had been part of the original language.

An unusual mechanism of extensibility is the unified organisation of the RCC system, in particular using 'master sections',

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

†Department of Computational Science, University of St Andrews, St Andrews, Fife, Scotland KY16 9SX

whereby the user's program and even the user's data are formally compiled as an extension to the implementation of RCC itself. Coupled with primary routines, this mechanism provides a continuous spectrum of extensibility techniques, permitting more and more sophisticated extensions, leading to redefinition of parts of the RCC language and compiler.

A second unusual feature is the ability to make purely syntactic extensions in the conventional subroutines of the user program, i.e. 'secondary' routines. Since formal parameters can be of type *phrase*, the user can define new syntax specifically for a particular routine, thereby significantly enhancing the level of language used in his program, but without significantly losing efficiency.

RCC has been implemented on a few machines, and has been used for a number of years by a number of people, mostly in connection with the authors' research. Despite its old design (mostly 1968), it still seems to have a number of features that are unusual and interesting, and the system has been shown to work in practice as it was designed to do. This paper gives a general description of the language and in particular of its more interesting features, and summarises the experience gained from it so far and its relationship with other systems. Knowledge of the Compiler Compiler is not assumed.

The basic language

The basic language of RCC is at a moderate level, leaving out various features that are now commonplace, and including features that are unusual.

Variables are untyped, being effectively integers, bit patterns, or addresses as convenient to the user. An identifier can be declared as a constant or as a variable. The storage of a scalar can be allocated automatically, or it can be specified by the user to be in a specific store address or in a central register. Thus, if necessary, the user can have complete control over storage allocation by using machine dependent options in declarations.

There is a notation for indexed addresses, e.g. $(a + i)$ and $a(i)$, which (both) refer to the location whose address is given by adding i to a . There is no formal type *array*. Statements are provided to specify static, dynamic and preset arrays, but the associated identifier (e.g. a above) is treated as a scalar constant or variable whose value is preset to the address of the base of the array.

Expressions have no subexpressions, no user defined functions, and no operator precedence. The operators are $+$ and $-$, together with those logical and shift operators available on the machine of implementation. Evaluation is left to right.

Multiplication, division, and any shift operators not available are provided by separate statements. Whilst these restrictions are severe for a user familiar with more powerful forms of expressions, they enable the implementation to be efficient and to avoid use of the stack.

Control

The organisation of control in RCC is unusual, with a formal clause and sentence structure being used to specify implicit control, replacing the lowest level of compound statement bracketing. Non-declarative statements are divided into imperative statements, conditional clauses, and FOR clauses. There are two levels of statement terminator, 'major' (newline) and 'minor' (; and :, possibly followed by newline). A sentence consists of a set of statements separated by minor terminators and finishing in a major terminator, and there are a number of rules specifying the permitted clause structure within a sentence.

Compound statement brackets '{' and '}' are provided to allow a set of sentences to be treated as a single imperative statement within a sentence.

A conditional sentence consists of one or more conditional clauses followed by one or more imperative statements. If there is more than one conditional clause, they must be linked consistently by either AND or OR. The conditional clause (or combined set of clauses) qualifies all imperative statements up to the end of the sentence. However the imperatives may contain a single OTHERWISE, with obvious effect. There is no formal type *Boolean* in RCC. Instead, a number of basic conditional clauses are provided in the language, and further clauses can be defined by the user, either as 'secondary' routines (effectively *Boolean* functions) or 'primary' (macros).

For example, given that

```
IF <expression> IS BETWEEN <expression> AND
    <expression>:
```

is a user defined conditional clause:

```
IF  $a = b + 1$ ; AND IF  $q > r$ ; AND IF  $d(i)$  IS BETWEEN
     $q - x$  AND  $q + x$ :
```

```
 $d(i) = d(i) + a$ ;  $d(i + 1) = 0$ :
```

```
    OTHERWISE:  $d(i + 1) = d(i) - d(i - 1)$ ;  $d(i) = 0$ 
cf. ALGOL 60:
```

```
if  $a = b + 1$  and  $q > r$  and between ( $d[i]$ ,  $q - x$ ,  $q + x$ ) then
begin  $d[i] := d[i] + a$ ;  $d[i + 1] := 0$  end
else begin  $d[i + 1] := d[i] - d[i - 1]$ ;  $d[i] := 0$  end
```

The brackets '{' and '}' are also used to enable a general 'Boolean expression' of conditional clauses to be written as a single clause, e.g. the above combined set of conditional clauses could be written as a single clause:

```
IF { $a = b + 1$ ; AND  $q > r$ ; AND  $d(i)$  IS BETWEEN  $q - x$ 
    AND  $q + x$ }:
```

Or, avoiding the IS BETWEEN statement:

```
IF { $a = b + 1$ ; AND  $q > r$ ; AND
    {{ $d(i) > q - x$ ; AND  $d(i) < q + x$ };
    OR { $d(i) < q - x$ ; AND  $d(i) > q + x$ }}}:

```

A FOR sentence consists of a FOR clause followed by a set of imperative statements, which form the 'body' of the cycle. A number of FOR clauses are provided in the language, and further ones can be added by users (using 'primary' routines only). If the key word CYCLE is used instead of FOR, the body of the cycle is specified explicitly as all following sentences up to the statement END OF CYCLE <name>, where the <name>, e.g. *i*, is the controlling variable of the cycle. Also imperative statements FINISH CURRENT <name> and FINISH EACH <name> can be used anywhere within the body of a FOR or CYCLE statement (including within nested cycles) to mean

respectively jump to the end of the current iteration and jump to the next statement after the cycle body.

For example consider a sequence which, given a vector of vectors, counts the number of vectors whose positive elements add up to less than 1000, but which stops altogether if a very large or small number is found.

```
CYCLE  $i = 1$  TO  $n$ :
 $a = \text{iliffe}(i)$ ;  $t = 0$ 
FOR  $j = 1$  TO length( $i$ ):
{IF  $a(j) < \text{min}$ ; OR IF  $a(j) > \text{max}$ : FINISH EACH  $i$ 
  IF  $a(j) < 0$ : FINISH CURRENT  $j$ 
   $t = t + a(j)$ 
  IF  $t > 1000$ : FINISH CURRENT  $i$ }
 $\text{count} = \text{count} + 1$ 
END OF CYCLE  $i$ 
```

A less formal cycling method is provided by REPEAT or REPEAT UNTIL <condition> as the last statement in a compound statement. This causes control to be returned to the start of the compound statement (only if the 'condition' is false in the latter case).

There is no case statement in RCC, only a simple FORTRAN-like switch. Labels and GO TO statements are permitted.

Subsequences are provided, i.e. local parameterless sub-routines implemented very efficiently. A subsequence is labelled by SUBSEQUENCE <name> with dynamic return FINISH <name>, and it is called by PERFORM <name>.

In general the implementation of control is very efficient, even when 'primary' routines are being used. Conditional control is organised in terms of control operations and not the evaluation of Boolean expressions. The operation code of a conditional machine code jump is not filled in till the next machine code instruction is known, to ensure that the correct test (e.g. = or \neq , \geq or $<$) is chosen. In general control addresses are not filled in until the final destination is known (i.e. there is no jumping to unconditional jumps). Thus if the above piece of code was followed by 'GO TO tidy up', the control jumps for ' $a(j) < \text{min}$ ' and ' $\text{max} < a(j)$ ' would both be direct to label 'tidy up'.

Therefore given the attention paid to control in both language and implementation, together with the restrictions on the complexity of expressions, a user can write a program at a high level knowing that the efficiency of the underlying code will be extremely close to hand coding.

Routines

There are a number of different types of routine in RCC. The type that corresponds to the conventional routine in a conventional programming language is called a 'secondary' routine. Other types are 'master', 'class', 'format class' and 'primary', and these will be described later.

A secondary routine does not have an identifier as its name, but a 'format', i.e. a string of fixed symbols interspersed with parameters in fixed positions. In the routine heading the complete specification of the formal parameter is given in the appropriate position, enclosed in square brackets.

```
e.g. ROUTINE
    INTERCHANGE [V  $x$ ] AND [V  $y$ ].
    LOCAL dummy
    dummy =  $x$ ;  $x = y$ ;  $y = \text{dummy}$ 
```

Here e.g. [V x] specifies a formal parameter of type *integer* with local name ' x '. There are three ways in which parameters can be passed, all by 'value'. E means actual parameter is copied to formal on entry to the routine, R means formal parameter is copied to actual on exit, and V combines the two copies.

A routine call comprises the given set of symbols interspersed with appropriate actual parameters, i.e. an *integer* expression for E or variable for V and R. Note that any symbols can be

used in the fixed part of a format, but capital letters are recommended rather than small letters, to avoid ambiguity with identifiers.

Thus a call on the above routine might be

INTERCHANGE $a(i)$ AND $a(i + 1)$

A secondary routine can have one of three characteristics to specify the way it is called (default STANDARD):

BASIC means that no stacking/destacking takes place on entry/exit.

STANDARD means that certain information, including values of central registers and the system stack front, is stacked/destacked on entry/exit.

RECURSIVE provides the STANDARD facilities, but in addition formal parameters and automatically allocated local scalars are allocated storage on the stack and not in unique static storage (i.e. effectively *own*) as is the usual case.

A secondary routine can have one of two characteristics to specify its control type (default IMPERATIVE):

IMPERATIVE means that a call of the routine is treated as an imperative statement. Inside the routine dynamic return is specified by the imperative FINISH, and is implicit at the end of the routine.

IF means that a call on the routine is treated as a conditional clause. The format does not include the IF itself and the routine can be called in any context that a basic conditional statement can be used in. Dynamic return is specified (always explicitly) by either imperative CONDITION SATISFIED or CONDITION NOT SATISFIED with obvious meaning.

e.g. **ROUTINE**
BASIC IF: [E a] IS BETWEEN [E b] AND [E c]: . . .
 IF $a > b$: {IF $a < c$: CONDITION SATISFIED;
 OTHERWISE: CONDITION NOT
 SATISFIED}
 IF $a < b$; AND IF $a > c$: CONDITION SATISFIED
 CONDITION NOT SATISFIED

Program structure

RCC is a one pass language; all entities must be declared before they are used.

The text of an RCC program is divided into sections all at the same level. There are only two 'scopes' in RCC (except for global data declarations), viz. local to a section and global. For sections specifying global information the scope is all the following sections; for statements specifying information local to a section, the scope is the rest of the section. Routines are not nested within routines and there are no blocks.

Each section is called a 'master section' and is headed by an identifier comprising underlined capital letters on a line of its own, a 'master heading'. The commonest type of master heading is **ROUTINE**, which gives a routine definition. The heading is followed by the specification of the routine on the next line, and then the routine body.

The heading **GLOBAL** introduces declarations of identifiers with global scope. These can include STATIC or PRESET arrays.

The heading **FORMAT** introduces a set of routine specifications (for secondary, primary, or format class routines). Each specification gives in effect the first line of the routine definition, except that names of formal parameters can be omitted. Such a specification in a **FORMAT** section is optional, and it allows a routine call to occur in a subsequent routine even if the specified routine has not yet been defined.

User master sections

The running of a user program is also controlled by master sections. The user data is in general split into any number of

master sections each headed by one of a set of underlined capital names defined by the user. For each of these 'master names' a corresponding 'master routine' must be defined by the user. The name of the master routine is the master name it is defining.

Whenever a master heading is met in the input stream the system passes control automatically to the corresponding master routine. This routine should then process all the source text up to the next master heading. The system automatically replaces a master heading by a special symbol code which acts as a terminator which can be detected but not read beyond.

In a simple program the user defines just one master name, e.g. **ROUTINE**. His program consists of a **ROUTINE** master routine together with any routines called by it. Its action is to recognise, sort and output a set of numbers. His program data consists of one or more sections headed **ROUTINE**, and finishes with the system master heading **STOP**.

e.g. **ROUTINE**
MASTER : **ROUTINE**
 :
 :
ROUTINE
 487
 2976
 243
ROUTINE
 :
 :
STOP

The language processing machinery

The formal language for processing syntactic elements is based on the data type *phrase*. The user can define any number of 'classes', corresponding to non-terminal symbols of BNF. These are contained in master sections with heading **CLASS**.

e.g. **CLASS**
[DIGIT] = 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
[N] = **[DIGIT][N]**, **[DIGIT]**
[MONTH] = JAN, FEB, MAR, APR, MAY, JUN,
 JUL, AUG, SEP, OCT, NOV, DEC
[DATE] = **[N][MONTH][N]**, **[N] :[N] :[N]**

A 'phrase variable' is a variable with an associated class, for example '*m*' with class **[MONTH]** and '*d*' with class **[DATE]**. However, scalar identifiers are *not* declared to be of type *phrase* with associated class given e.g.

PHRASE *m*, *ml* **[MONTH]**; PHRASE *d* **[DATE]**

Instead they are declared as ordinary (*integer*) scalars and where they are used as phrase variables the associated class is given every time the identifier is referred to, e.g.

LOCAL *m*, *ml*, *d*

with all references as **[MONTH *m*]**, **[MONTH *ml*]**, **[DATE *d*]**. (This is to avoid problems of syntactic ambiguity in the non-terminal <matching phrase expression>—see later).

The value of a phrase variable is any valid symbol string according to the definition of the associated class. Thus a phrase variable **[MONTH *m*]** can have only 12 possible values, viz 'JAN', 'FEB', . . . , 'DEC'. A phrase variable **[DATE *d*]** can have an indefinite number of values of one of the two given forms, e.g.

'12 MAR 78' or '12:3:78'

The statements provided to operate on phrase variables are as follows:

input IF RECOGNISE <explicit phrase>

Here an <explicit phrase> is a string of symbols maybe inter-

spersed with phrase variables, e.g.

IF RECOGNISE COME TO ROOM [N *r*] AT [N *hrs*] HRS
ON [DATE *d*]:

This checks the current head of the input stream to see if it is of the given form, i.e. if it starts with the given string of symbols but with any valid string of the associated class occurring in the corresponding position for each phrase variable. If there is a match, the input pointer is advanced over the matched portion and the phrase variables are set to the corresponding sub-strings, e.g. given the input string starting with

COME TO ROOM 15 AT 1600 HRS ON 1:4:84 PLEASE
it would be left starting at PLEASE with [N *r*] set to '15', [N *hrs*] to '1600', and [DATE *d*] TO '1:4:84'.

If a full match is not achieved the input pointer is not advanced and no phrase variables are reset.

output PRINT : <explicit output phrase>

Here <explicit output phrase> is a string of symbols maybe interspersed with phrase variables and/or specifications of formatted output of integer data, e.g.

PRINT : GO TO ROOM [N *r*] AT [E (SL 6) *t*] ON [DATE *d*]

The current value of any phrase variable is output at the corresponding points, and any *integer* data generated appropriately, e.g. for [E (SL 6) *t*] the current value of '*t*' as an integer in a field width of 6 columns with Spaces to the Left.

analysis Given a phrase value, possibly from an IF RECOGNISE statement, it is often necessary to break down its structure and, if required, convert it to integer form. Three facilities are provided to do this.

- (a) The basic function CATEGORY OF <phrase variable> yields the serial number 1, 2, 3, . . . of the alternative in the associated class definition to which the current value of the phrase variable belongs, e.g. for [MONTH *m*] set as 'MAR', CATEGORY OF [MONTH *m*] is 3, and for [DATE *d*] set as '12 MAR 78', CATEGORY OF [DATE *d*] is 1.

Where the associated class is a simple one, without subclasses, e.g. [MONTH], this obviously provides a convenient and unique integer for every possible phrase value. If however the class has subclasses, e.g. [DATE], then the category only specifies to which alternative subset of possible strings a phrase value belongs.

- (b) RESOLVE <phrase variable> INTO <matching phrase expression>

This is an imperative statement which enables a phrase variable whose associated class contains subclasses to be resolved into its constituent phrase variables, e.g. given [DATE *d*] currently as '12 MAR 78'

RESOLVE [DATE *d*] INTO [N *day*][MONTH *m*]
[N *year*]

will reset [N *day*] to '12', [MONTH *m*] to 'MAR', and [N *year*] to '78'.

The statement is undefined if the current value is not of the given form, e.g. for [DATE *d*] as '12:3:78'.

The <matching phrase expression> can be any sentential form of the associated class of the <phrase variable>. Formally, it must either be a phrase variable of the associated class, or one of the alternatives in the definition of the class but with each subclass in the alternative replaced by a <matching phrase expression> for the appropriate subclass. Thus the following are also valid statements at compile time:

RESOLVE [DATE *d*] INTO [DATE *p*]
RESOLVE [DATE *d*] INTO [N *day*][MONTH *m*]

[DIGIT *d1*][DIGIT *d2*]

RESOLVE [DATE *d*] INTO 1[DIGIT *dm*] MAR 78

For the value '12 MAR 78' they will also be defined at run-time, but for value '12:3:78' only the first statement is defined. Note that the first statement is equivalent to setting scalar *p* equal to *d*.

- (c) IF <phrase variable> \equiv <matching phrase expression>:

To avoid obeying undefined RESOLVE statements at run time it is usually necessary to check that a given phrase variable is of the given form beforehand, e.g. by testing the category. This conditional statement does this automatically, i.e. it first checks if the phrase value is of the given form specified by the <matching phrase expression> and if it is, it carries out the resolution as in (b); if it is not, the condition is not satisfied and no phrase variables are reset.

synthesis Two statements are provided to carry out the reverse operations to analysis, e.g. to create phrase values from integer information, or from existing values of component phrase variables.

CATEGORY OF <phrase variable> = <expression>

resets the value of the <phrase variable> to the alternative string in the associated class definition specified by the integer <expression>, e.g.

CATEGORY OF [MONTH *m*] = 7

will set [MONTH *m*] to 'JUL'. The statement is undefined if the specified alternative does not exist or contains subclasses.

SET <phrase variable> = <matching phrase expression>

sets a new value for the <phrase variable> to the string specified by the <matching phrase expression>, with the current value of each component phrase variable inserted at the corresponding point,

e.g. given [N *dm*] as '12' and [N *m*] as '3'

SET [DATE *d*] = [N *dm*] : [N *m*] : 78

will reset [DATE *d*] to '12:3:78'.

Storage of phrase variables

A phrase value is stored, not as a symbol string, but as a tree (or 'analysis record') which reflects the relationship of the string to the definition of its associated class. The phrase variable is set to the address of this tree; the first node in the tree gives the category number of the alternative followed by the addresses of the trees for each constituent subclass in the alternative. How efficient this is in storage compared with symbol string storage depends on the class definition, e.g. [N] will give less efficient storage, but [MONTH] more efficient.

A particular point to note however is that the RESOLVE and IF analysis instructions do not require any extra storage, and are efficient provided that the resolution is carried out at the level of an alternative of the associated class definition, e.g.

IF [DATE *d*] \equiv [N *day*][MONTH *m*][N *year*] : . . .

is implemented as

IF $d(0) = 1$: $day = d(1)$; $m = d(2)$; $year = d(3)$; . . .

Special class definition facilities

There are a large number of enhancements to the basic BNF-based class definition mechanism, which enable programs to be written more conveniently and/or efficiently. Many of them did not occur in the original Compiler Compiler, or occurred in less satisfactory forms.

The formal parsing algorithm is top-down, left-to-right, and fast-back (which imposes the usual constraints on the way classes are defined). It can be modified in a number of ways.

A class used in an alternative of the definition of another class can be modified by adding 'qualifications'—'?' to specify an optional occurrence (category 0), 'r' to specify that the analysis record of the subclass is to be suppressed, and/or 'a' to specify that the input position is not to be advanced over the substring recognised for the class.

A class in which all alternatives start with one or more symbols (i.e. not a subclass) can be specified as 'restricted' in a number of ways. In this case initial recognition is bottom up using a hash algorithm.

The case of a class which defines a list of strings, possibly with separators, is acknowledged explicitly in the definition machinery, instead of relying on recursive definition. Similarly the analysis record is stored in a straightforward manner, with the first node giving the number of strings in the list followed by the set of values of the subclasses in order as they occur, e.g. CLASS

```
[N] = LIST OF [DIGIT]
[BOUNDS] = LIST OF [N]:[N], SEPARATED BY,
[ARRAY LIST] = LIST OF [IDENTIFIER] ([BOUNDS]),
SEP; [END OF LINE?]
```

Thus a value for [ARRAY LIST *a*] might be 'x(3:15); y(1:8, 1:8); z(0:1023)'

There are also a number of facilities to help automatic transformation of language without having formally to program the transformation using the analysis and synthesis statements, as demonstrated by Lindsay (1975), i.e. where the controlling program is of the form

```
IF RECOGNISE [CLASS s]: PRINT: [RELATED CLASS s]
```

Class routines

A particularly useful facility is to be able to define a class by algorithm. Such a class has an associated routine, whose name is the class name, and this routine operates directly on the input stream according to given conventions. It is called automatically by the parser, and it returns control by imperatives RECOGNISED or NOT RECOGNISED as appropriate,

```
e.g. ROUTINE
CLASS : [SMALL LETTER]
LOCAL symbol
symbol = (input symbol position)
IF symbol < 'a'; OR IF symbol > 'z':
NOT RECOGNISED
ADVANCE input symbol position
STACK symbol - 'a' + 1 AT main stack front
RECOGNISED
```

The information recorded by the class routine can be of any size, and is simply put on the main system stack. The information is recovered on subsequent processing by treating the scalar of the associated phrase variable as an array name to access the information.

In fact the language contains a further basic function and statement analogous to CATEGORY OF, i.e. VALUE OF, to deal with the common case where the information associated with a class routine is a single word. Thus VALUE OF [SMALL LETTER *s*] will return the serial number 1 to 26 planted by the routine, and VALUE OF [SMALL LETTER *s*] = *p* will set up a phrase value for [SMALL LETTER *s*] equivalent to a class routine having planted the number *p*.

Class routines are typically used where a set of alternatives start with related system symbol codes (as for [SMALL LETTER]) and/or where the syntactic element being recognised is an indivisible unit of the language being processed, e.g. a constant or identifier. Here it is often more convenient and efficient to carry out the conversion and/or packing that will eventually be required directly, as part of the recognition process, rather than at a later time by processing a formal

analysis record using the formal analysis instructions.

Another enhancement of the class definition machinery, which has particular relevance to class routines, is that class words occurring in definitions of other classes can have 'parameters', i.e. a string of symbols and/or class words. These can be used by class routines to modify their behaviour. In particular there is a system class [Q], which always requires a parameter, in the form of a simple class definition, i.e. one or more symbol strings separated by commas. This is useful for avoiding formal class definitions, e.g. [Q/(AND, OR)], and for specifying abbreviations in syntax, e.g. IMP[Q/(ERATIVE)?r] to allow IMP or IMPERATIVE at some given point in a syntax.

Phrase parameters

Phrase variables are permitted as formal parameters to routines. They are specified in the same way as integer parameters, except that the associated class is given between the '[' and the E, V or R specification. In the case of an E specification, the actual parameter can be any <matching phrase expression> for the associated class. The effect is to

```
SET <formal parameter> = <matching phrase expression>
```

on entry to the routine. For V and R parameters, the actual parameter must be a phrase variable with associated class the same as the formal parameter, and the effect is simply to copy the underlying integer variable. If the parameter call type for a phrase parameter is E, the E can be omitted.

An obvious context for requiring phrase variables as parameters is in processing text that has been read in by IF RECOGNISE. Particularly if the input text is in the form of a 'language', i.e. a set of strings each belonging to one of a significant number of alternative formats, the natural structure of the program is to process the input in a set of routines, one for each alternative. The parameters of these routines would be the component syntactic elements of the alternatives (i.e. the subclasses).

Of course, where different alternatives contain the same syntactic elements, which require similar processing, further sub-routines are likely to be created which have a phrase variable of the appropriate class as a formal parameter.

Format classes

RCC provides a facility specifically for the situation where each alternative of a given class is always processed in the same way, preferably by a separate routine for each alternative. In this situation, the parameters of a routine will tend to be the subclasses of the corresponding alternative, and indeed the format of the routine will tend to be the same as the alternative, possibly preceded by a general verb, e.g. PROCESS or TRANSLATE. A class defining the set of possible statements in a 'language' is typically of this nature, e.g. [FORTRAN].

Such a class can be introduced as a 'format class' using a special master section,

```
e.g. FORMAT CLASS
[FORTRAN]
```

Then subsequently any number of 'format class routines' can be defined as belonging to this format class,

```
e.g. ROUTINE
[FORTRAN] : [VAR v] = [EXPR e]
.....
ROUTINE
[FORTRAN] : DIMENSION [DIM LIST ds]
.....
```

The effect of such a routine heading, or of the corresponding specification in a FORMAT section, is two fold. First, it introduces a new routine in the program, with characteristic 'format class routine'. Second, it adds the format as a new alternative to the definition of the associated format class.

A format class can be treated in most respects just like an ordinary class, e.g. as if it had been defined as

```
[FORTRAN] = [VAR] = [EXPR], DIMENSION
          [DIMLIST], . . . . .
```

Thus the conditional statement IF RECOGNISE [FORTRAN *f*] [EOL *r*] (where [EOL] recognises 'end of line') will attempt to recognise one of the alternatives of [FORTRAN] and if successful it will set phrase variable [FORTRAN *f*].

However in addition the imperative statement PROCESS <phrase variable> is available for format classes. This automatically passes control to the format class routine corresponding to the current alternative of the phrase variable, and initialises its formal parameters to the appropriate subvalues in the alternative. Note that parameters can only be of type *phrase* with characteristic *E*. Thus the sentence

```
IF RECOGNISE [FORTRAN f] [EOL r] : PROCESS
          [FORTRAN f]
```

does the complete job of trying to recognise a FORTRAN statement and if successful passing control to the corresponding processing routine. This would otherwise be done by a large switch on the category of [FORTRAN *f*] and the programmer would have to invent a format for each different alternative, which did not clash with the current language, and call the appropriate routine at each point in the switch.

Note that a format class routine, e.g. [VAR] = [EXPR] cannot be called directly in the program, i.e. it does not become part of the current RCC 'language' (i.e. the basic language plus user secondary routine formats). The only way a particular format class routine can be called is by the statement

```
PROCESS (<classname>) <matching phrase expression>
```

e.g. PROCESS ([FORTRAN]) [VAR *v*] = 1

This is equivalent to

```
SET [FORTRAN dummy] = [VAR v] = 1
PROCESS [FORTRAN] dummy
```

RCC as a compiler compiler

RCC is designed to provide the user with a powerful compiler compiler, provided he wishes to write a compiler for a language which consists of suitably sized 'pieces', e.g. statements, each of which has a context free syntax. Note that ALGOL 60 and PASCAL can be described in 'pieces' relatively easily, using straightforward semantic checks that they form a consistent program, but ALGOL 68 cannot.

The class definition machinery permits straightforward definition of syntax, and IF RECOGNISE . . . provides automatic recognition of pieces of source text, together with an analysis record. The analysis instructions provide an efficient and readable mechanism for extracting semantic information. The format class mirrors the characteristic top level structure for a language, with a controlling routine continuously recognising a piece of source text and passing control automatically to the appropriate processing routine. Ancillary routines are written as secondary routines, which can have both *integer* parameters, and, where syntactic elements are involved, *phrase* parameters.

Note however that RCC does not provide an automatic itemisation/lexical analysis phase.

Fig. 1 shows part of a FORTRAN compiler (somewhat simplified), giving a sample set of formats for both secondary and format class routines, the controlling routine, and the processing routine for the computed GO TO switch.

The context of the undefined routines illustrated should be essentially self explanatory. LOAD [TYPE][EXPR] INTO REGISTER [E] is a key code generation routine of the compiler, and SKIP R [E] INSTRUCTIONS is a specialist routine to plant the appropriate machine code for organising a

FORMAT CLASS [FORTRAN]

FORMAT

STANDARD : REJECT REST OF LINE.

[FORTRAN] : [VAR] = [EXPR]

RECURSIVE : LOAD [TYPE][EXPR] INTO REGISTER [E].

STANDARD : SKIP R [E] INSTRUCTIONS.

[FORTRAN] : DIMENSION [DIM LIST]

[FORTRAN] : GO TO [LABEL]

[FORTRAN] : GO TO ([LABEL LIST]), [VAR]

STANDARD : INITIALISE.

STANDARD : TIDY UP.

ROUTINE

MASTER : FORTRAN

LOCAL *f*

INITIALISE

{IF RECOGNISE [FORTRAN *f*] [EOL *r*] :

PROCESS [FORTRAN *f*]:

OTHERWISE : REJECT REST OF LINE

REPEAT UNTIL RECOGNISE ### #}

TIDY UP

ROUTINE

[FORTRAN] : GO TO ([LABEL LIST *ll*]),
[VAR *switch no*]

LOCAL *i*

LOAD INTEGER [VAR *switch no*] - 1 INTO REGISTER 6

SKIP R6 INSTRUCTIONS

FOR *i* = 1 TO CATEGORY OF [LABEL LIST *ll*]:

PROCESS ([FORTRAN]) GO TO [LABEL *ll*(*i*)]

Fig. 1

switch on the given machine. The switch processing routine operates by setting up the appropriate machine code switch and then calling the processing routine for GO TO [LABEL] repeatedly to plant the following jumps.

An important effect of the format class machinery is that it makes it easy to extend the language and its compiler. If a new statement is to be added to the language, it is sufficient to add, at the end of the compiler, a new format class routine together with any new class definitions required. This both extends the syntax of the language being compiled and specifies how to process the new statement. There is no need in general to alter any other part of the compiler to accommodate the new statement.

Further, it is easy to write the processing routine itself if the new statement is easily describable in terms of the existing language. Using PROCESS (as in the computed GO TO processing routine), the processing routines of the existing language can be called directly in a readable manner, passing the syntactic elements through from the new statement to calls on existing statements using phrase variables.

This method of extensibility was investigated and formalised in the design and implementation of the language ALEC (Napper and Fisher, 1976).

Extensibility in RCC

The design of RCC facilitates the writing of compilers for extensible languages. RCC is implemented in RCC. So it is not surprising that RCC is itself extensible.

The most obvious method of extensibility is the analogue of the format class mechanism, illustrated in Fig. 1 for FORTRAN. The user can write a 'primary' routine which will be entered automatically by the compiler on each subsequent

recognition of a source statement matching the format. This contrasts to the secondary routine, where a cue is compiled to the corresponding routine, which is obeyed at run time. Thus the primary routine must describe how to compile the code required to effect the statement at run time, whereas the secondary routine gives the code directly.

In general the addition of a primary routine does not require any alteration to the RCC implementation, or any detailed coding in the routine to link it into the implementation.

At its simplest, the primary routine behaves like a conventional syntax macro, although its implementation is by compiler extension and not by a preprocess. However there are a number of levels of sophistication beyond the straightforward syntax macro at which primary routines can be used to extend the language.

In addition there are a number of other mechanisms of extensibility in RCC, notably syntactic extensibility in both primary and secondary routines, and the overall unity of the stages of compiler-compile, compile, and run time. This unity means that even a piece of run time data is formally an extension to the implementation of RCC.

Simple primary routines

The description of routines in the basic language left out a further characteristic of routines, that is whether a routine is SECONDARY (the default) or PRIMARY. For a PRIMARY routine the parameters are restricted to type *phrase* with characteristic E; option BASIC is not available, but a number of addition control characteristics are available beyond IMPERATIVE and IF, notably CYCLE.

A primary routine is entered whenever a statement matching its format is met in subsequent routines. Therefore in general it describes how to compile the appropriate in-line code. The language it is written in is, as for all types of routine, full RCC.

Where the user wishes to use another RCC statement to cause an appropriate piece of code to be compiled, he uses the routine COMPILE {[RCCBL]}, where [RCCBL] is the set of formats of the RCC language plus the user's primary and secondary routine formats. [RCCBL] is analogous to the format class, e.g. [FORTRAN], and COMPILE to PROCESS, in the typical use of RCC as a compiler compiler. They are different from the standard format class machinery to cater for the special (and known) requirements of the system language. In particular COMPILE also carries out the automatic processing of control to link together successive statements being compiled, using their control characteristics, and arranges to set up a new control level for each macro call.

The actual parameter corresponding to [RCCBL] in a call of COMPILE can be any sentential form of [RCCBL] since it is a type E phrase parameter. An extended version allows a string of [RCCBL]s with major or minor terminators separating them as usual.

Example 1

ROUTINE

PRIMARY IMPERATIVE : INTERCHANGE [V a] AND [V b].

MACRO LOCAL *dummy*

COMPILE {[NAME *dummy*] = [V a]; [V a] = [V b];
[V b] = [NAME *dummy*]}

Note that MACRO LOCAL reserves a storage location which is used always and only by the in-line sequences generated by the routine; it can be referred to inside the COMPILE statements as e.g. [NAME *dummy*]

Example 2

ROUTINE

PRIMARY IF : [E a] IS BETWEEN [E b] AND [E c]:...

COMPILE {IF [E a] > [E b] : {IF [E a] < [E c]:...}
IF [E a] < [E b]; AND IF [E a] > [E c]:...}.

Note here ':...' standing for ':CONDITION SATISFIED: OTHERWISE : CONDITION NOT SATISFIED'. This abbreviation can be used in primary or secondary routines, but in primaries, when used at the end of the in-line sequence, it ensures complete efficiency in linking up with the control environment of each call on the macro.

Syntactic extensibility using primary routines

The format of a primary routine is any symbol string with any *phrase* parameters embedded. The string matching a *phrase* parameter can be any string definable by the RCC class definition machinery. Therefore a new primary statement in RCC can have any syntax the user cares to define (provided ambiguity with the existing language is avoided).

Note that classes can be defined by the user for the purpose of extending the RCC language in routine formats just as much as they can be used to define language to be recognised in master sections.

Of course where new syntax is created the problem of specifying the code to be compiled becomes less straightforward. However the type *phrase* has been designed specifically to facilitate the extraction of information from syntactic elements.

If the user can break down the new syntax into classes of the RCC language and then express the required in-line code as a set of COMPILE statements, then he can define a primary routine without any knowledge of compiler implementation in general and the RCC implementation in particular. The routine then reads as a simple language transformation.

The key rule to follow is that the set of COMPILE statements generated by any call in the primary routine, when strung together, should form a coherent routine.

As a comparatively simple example of syntactic extension in primaries, consider an instruction that, given a list of integers and an explicit sequence of integers, checks if the sequence occurs anywhere in the list. The list may be specified (say) either as two store addresses, pointing to the first and last integers on the list, or by a single address pointing to a vector, e.g. *v*, where by convention the element *v*(0) gives the number of integers in the list, e.g. *n*, and they follow on from *v*(1) to *v*(*n*).

The required extra syntax is:

CLASS

[LIST] = STORE FROM [E] TO [E], VECTOR [NAME]
[ELEMENTS] = LIST OF [E], SEPARATED BY,

An appropriate routine is given in Fig. 2. Examples of calls might be

- (a) IF SEQUENCE 0, 1, 0 IN STORE FROM *f* TO *g* + 1:
- (b) IF SEQUENCE *a* - *x*, *a* + *x* IN VECTOR *p*:
- (c) IF {*a* = 0; OR {SEQUENCE *q*, *r*, *s*, *t* IN VECTOR *b*;
AND *r* IS BETWEEN *q* AND *s*}}:
- (d) REPEAT UNTIL SEQUENCE 'E', 'N', 'D' IN STORE
FROM *st*(*i*) TO *fin*(*i*)

The in-line sequence compiled for case (a) would be in effect
finish = *g* + 1

FOR *i* = *f* TO *finish* - 2:

{IF (*i*) = 0; AND IF (*i* + 1) = 1; AND IF (*i* + 2) = 0;
CONDITION SATISFIED}

CONDITION NOT SATISFIED

The sequence compiled for case (b) would be in effect

finish = *p* + (*p*)

FOR *i* = *p* + 1 TO *finish* - 1:

{IF (*i*) = *a* - *x*; AND IF (*i* + 1) = *a* + *x*;
CONDITION SATISFIED}

CONDITION NOT SATISFIED

CONDITION SATISFIED and CONDITION NOT SATIS-

ROUTINE

```
PRIMARY IF : SEQUENCE [ELEMENTS els] IN  
[LIST l] : . . .  
LOCAL start, f, v, n, n1, j, j1  
MACRO LOCAL finish, i  
IF [LIST l]  $\equiv$  STORE FROM [E start] TO [E f]:  
  COMPILE {[NAME finish] = [E f]}:  
  OTHERWISE :  
  {RESOLVE [LIST l] INTO VECTOR [NAME v]  
  SET [E start] = [NAME v] + 1  
  COMPILE {[NAME finish] = [NAME v] +  
    ([NAME v])}}  
n = CATEGORY OF [ELEMENTS els] (number of  
  elements)  
  VALUE OF [N n1] = n - 1  
  COMPILE {FOR [NAME i] = [E start] TO  
    [NAME finish] - [N n1]:  
    {IF ([NAME i]) = [E els (1)];}  
  UNLESS n = 1: {FOR j = 2 TO n :  
    VALUE OF [N j1] = j - 1;  
    COMPILE {AND IF ([NAME i] + [N j1]) =  
      [E els(j)]};}  
  COMPILE {CONDITION SATISFIED}  
  CONDITION NOT SATISFIED}
```

Fig. 2

FIED would link into the control surrounding the call without loss of efficiency. In particular the NO exit from the sequence will automatically be a jump direct from the test on *i* terminating the cycle to the required destination implied by the surrounding control, e.g. in (d) to the beginning of the compound statement enclosing the REPEAT UNTIL statement.

Syntactic extensibility in secondary routines

It is equally true of secondary routines that a secondary statement in RCC, i.e. a conventional routine call, can have any syntax the user cares to define. Secondary routines can have *phrase* parameters, and again class definitions can be made solely to provide syntactic extension.

Of course where *phrase* parameters are used for this purpose it will in general be necessary to use analysis instructions to 'decode' the actual parameters before giving the algorithm required to effect the statement. However provided that the new syntax is defined and decoded sensibly, the loss of efficiency will be very small compared with decoding the same information given in purely numeric form.

A particular feature is that integer parameters can be embedded in phrase definitions and passed as *E*, *V* or *R* parameters as required. This is of course in addition to the standard use of integer parameters declared in their own right. Such embedded parameters are specified in class definitions by the classes [E'], [V'], and [R'], with effective syntax [E], [V] and [V] respectively. The decoding mechanism is to use analysis instructions to uncover any such phrase values, e.g. [E' *a*], and then the underlying integers can be accessed (for [E'] and [V']) or reset (for [V'] and [R']) by using the accessing function 'value of', e.g. 'value of (*a*)'.

Thus it is possible to define a routine such that a call on it may have an indefinite number of *integer* parameters. Again the loss of efficiency in using this mechanism is small.

As an example, the primary routine defined in Fig. 2 could equally have been defined as a secondary which had the same range of possible calls.

The associated syntax might now be defined as

```
CLASS  
[LIST] = STORE FROM [E'] TO [E'], VECTOR [E']  
[ELEMENTS] = LISTOF [E'], SEPARATED BY,
```

Note that in fact VECTOR [E'] allows a general expression in a call instead of just VECTOR [NAME] as in the primary version.

An appropriate definition for the secondary routine is given in Fig. 3. In fact it is inefficient in that the extraction of an embedded integer in the [ELEMENTS] sequence is repeated for each comparison. However further types of phrase exist, i.e. [E'*], [V'*] and [R'*] to cover lists of integers directly without this inefficiency and the special definition [ELEMENTS] is not necessary.

As an example of an embedded output parameter, consider modifying the statement so that it will optionally return the address of the start of the matched string.

In the primary version this could be achieved by adding the class

[ADDRESS] = AT [V]

and adding [ADDRESS?] at the end of the format, thus permitting a call like

IF SEQUENCE 0, 1, 0 IN STORE FROM *f* TO *g* + 1 AT *p*:

The required modification to Fig. 2 is to insert before

COMPILE {CONDITION SATISFIED}

the sequence

IF [ADDRESS? *at*] \equiv AT [V *ad*] : COMPILE {[V *ad*] =
[NAME *i*];}

For the secondary version the extra class would instead be

[ADDRESS] = AT [R']

and the modification to Fig. 3 is to insert before CONDITION SATISFIED

IF [ADDRESS? *at*] \equiv AT [R' *ad*]: value of (*ad*) = *i*

Unity of compiler-compile, compile and run time

In RCC, in its most general form, the following stages are continuous and indistinguishable:

1. Compiling the RCC compiler (in its finally bootstrapped form).
2. Compiling a user program.
3. Running a user program.

The RCC compiler has the property that its description (in RCC) can be fed through itself without altering itself. Classes and routines can be redefined at any time, and a redefinition takes place immediately the class or routine has been processed. An exception is that *FORMAT* redefinitions are treated as errors. The recompilation is a continuous renewal of the existing system, and not a compilation of a separate core image which behaves identically.

ROUTINE

```
SECONDARY IF : SEQUENCE [ELEMENTS els] IN  
[LIST l] : . . .  
LOCAL a, b, start, finish, base, n, i, j, el  
IF [LIST l]  $\equiv$  FROM [E' a] TO [E' b]:  
  start = value of (a); finish = value of (b):  
  OTHERWISE : {RESOLVE [LIST l] INTO VECTOR [E' a]  
  base = value of (a); start = base + 1; finish = base + (base)}  
  n = CATEGORY OF [ELEMENTS els]  
  CYCLE i = start TO finish - n + 1:  
  FOR j = 1 TO n:  
    {SET [E' el] = [E' els(j)]  
    IF value of (el)  $\neq$  (i + j - 1): FINISH CURRENT i}  
  CONDITION SATISFIED  
  END OF CYCLE i  
  CONDITION NOT SATISFIED
```

Fig. 3

The job that any RCC program can do is defined by the set of jobs the individual master routines can do. Different master routines may do independent tasks, or all or some of them may co-operate. The language for programming the set of jobs is that used within ROUTINE master sections, i.e. [RCCBL]. [RCCBL] consists of the set of formats of the basic RCC language plus the primary and secondary formats defined by the user. Auxiliary language definitions are contained in CLASS definitions, to define the classes contained in the routine formats, or to define syntax used in the master sections. Primary routines are characteristically obeyed at compile time, and secondary routines at run time when called by the master routines processing the program data.

RCC as written in RCC also has this structure. The set of primary formats is the basic RCC language, and the secondary formats are the subroutines of the implementation. The master routines are of course FORMAT, ROUTINE, CLASS, GLOBAL, etc, and the 'run time' data of the implementation is the user program, which consists of a collection of such master sections.

However in stage (2) the user's program is compiled as an extension of the RCC compiler. New primary and secondary formats are added to those of the system. Note that the secondary formats of the RCC implementation refer both to routines that are generally useful to the user, e.g. input/output routines, and those that are concerned only with the detailed implementation of the compiler. As with the formats, user class definitions are added to the system class definitions, and user master routines to the system master routines.

As soon as a new primary routine is processed, the language is extended, and a statement invoking it can be used immediately in the next ROUTINE master section. As soon as a new master routine has been defined, a corresponding master section can occur immediately, even as the following master section. In either case any secondary (or primary) routine used in the primary or master routine must have been defined before the routine is invoked.

There is no restriction on the order in which master sections occur, be they system or user, except that their master routines are fully defined. There is no restriction on what order primary or secondary routines of the system or the user can be defined or redefined, except that they must be fully defined before they are used. The RCC system can be halted, and if required suspended as a binary image, after any master section.

Therefore the three stages of compiler-compile, compile, and run time, in terms of the source text, or program execution, do not exist as separate units. The only unit which exists anywhere throughout all three stages is the master section. Classifications can only be made on individual master sections relative to the other parts of the system. Without knowing the full story of a particular instance of an RCC section there is no certain way of classifying a particular section of program. A master section headed ROUTINE is likely to be in effect in stage (2). But it could be a redefinition of part of the RCC implementation (stage (1)) or it could be the source data for an RCC translation program or cross-compiler (stage (3)). A master section headed FORTTRAN is likely to be in effect in stage (3). But it could be due to an extension made to RCC to allow routines of RCC programs to be written in FORTTRAN (stage (2)), or a rewrite of part of the RCC implementation using this extension (stage (1)).

In practice the system is not often used as flexibly as implied above. The user's program will start with any primary routines he is using plus the list of secondary formats and any class definitions required, either for his formats or to recognise his source data. The program will then continue as a set of routines, or GLOBAL definitions. Then will follow his data as a set of user master sections.

Facilities also exist for removing parts of the RCC system at appropriate points. Thus at the end of stage (1) most of the names and formats internal to the RCC implementation may be removed—but not the routines themselves. At the end of stage (2) those routines of the RCC system not required by the set of user master routines can be removed. In particular, for a production program where the user wishes to dispense with master sections in his data, the last routine of his program is likely to be a redefinition of the RCC master controlling routine as the user's main program.

Further extensibility in RCC

The unified structure of the RCC implementation means that there is a wide spectrum of extensibility beyond the straightforward syntactic extensions illustrated in Fig. 2.

The first level of sophistication is that a primary routine may contain a non-trivial algorithm in RCC to work out what code to compile (e.g. what string matching algorithm to use in Fig. 2). It may need to acquire storage and it may need to communicate between different instances of the same statement or with instances of other statements within a package of related statements.

The next level of sophistication is to write primary routines which require to use routines of the RCC implementation not usually used by an ordinary user, for example a routine which compiles an expression into a given central register. This may happen simply to improve the efficiency of in-line code produced by a statement which can be expressed in terms of the existing statements. However it is particularly likely when statements are being defined which cannot be so expressed, for example when adding a floating point package to RCC. It is easy for both secondary and primary routines to obey special machine codes, by using OBEY [MCI] or COMPILE {OBEY [MCI]} respectively, where [MCI] is a machine code instruction on the host machine. However to integrate such new code into the language satisfactorily usually requires direct use of specialised routines of the implementation. Therefore a package of such routines, i.e. their formats and a specification of their action, is made available to the user.

At the next level of extension, it may be necessary to make small alterations to routines of the RCC implementation itself, e.g. in adding a new control type like a *case* statement which, in addition to the primary routines for the new statements, will require small alterations to be made to the RCC control machinery to integrate it into the language. It is possible to redefine a system routine at any time, and RCC is implemented in RCC at a high level of language and generality. Therefore given the documentation of the implementation and its source text, it is made as easy as possible to carry out such an extension.

The next level is to make significant changes in the implementation of RCC, involving the rewriting of one or more routines. The system will accept a rewritten routine without any fuss, but it is effective immediately and must therefore be compatible with the existing system. It is in fact possible to hold over the replacement of an old routine by a new one until a set of routines has been defined and this is sometimes highly desirable. Allison (1976) has shown a major example of such a revision by producing an experimental version of RCC to permit user extensibility of data types. This required straightforward rewriting of some of the routines concerned with identifier dictionaries, and a less straightforward rewrite of the routine cue compilation machinery. However this was accomplished using documentation with minimal direct help from the authors. As Allison showed, there are many features of the RCC system which facilitate such major surgery, and the process is significantly different from just getting a copy of someone else's compiler and playing around with it—hardly a definition of an

extensible language!

In parallel with extensions to the RCC language through primary routines, there are the possibilities of extensions to the system as a whole, or redefinition of parts of the system, in terms of master routines, for example as hinted at in the previous section (for *ROUTINE* and *FORTTRAN*).

Note that RCC does not provide directly for user extension at levels below a statement of the language. This is possible, but has not been done on efficiency grounds. For example extensions to the available set of constant (literal) forms, or ways of printing information in the general print routine, can be done straightforwardly by altering appropriate class definitions and the routines that process them. It would be easy to turn these classes into format classes, rewrite the appropriate routines once and for all, and then extensions could thereafter be made without redefining routines of the compiler, by adding a new format and a processing routine for it which satisfies a certain set of conventions.

This is a small example of an extension made at a higher level of sophistication enabling subsequent extensions to be made at a lower level. This is likely to be a common occurrence throughout the tree of extensions and compilers that grows from a highly extensible system. The basic pattern of such a tree is a different extension for each application or research group, within each group a further extension for each member, and finally for each member a further extension for each program. The higher an extension is in the tree, the more sophisticated is the extension likely to be and the more experienced the person designing and implementing it.

Comparison with other systems

Because there are a number of different aspects, which are contained in a unified system in RCC, but which exist separately in most other systems, it is not possible here to include a detailed comparison with other related work. It is hoped that subsequent papers on different aspects of RCC will be written, and these will contain detailed comparisons. Meanwhile some comments are given below under the different main headings.

Compiler compiler

The RCC compiler compiler facilities are rather too general for writing efficient compilers for the common languages. They are designed to enable compilers to be written quickly and in an understandable, easily modifiable way, and hence are useful for experimental or extensible compilers.

Language transformation

RCC provides facilities for synthesising language as well as analysing it, and therefore it can also be used as a general language transformation system. Compared with string processing languages, it is limited by being BNF-based, but within its limitations it is quite powerful yet efficient.

Systems implementation language

The basic language of RCC, considered as a systems implementation language, has some novel features, but in general does not compare favourably with the more sophisticated and powerful SILs. In particular it does not provide structured data.

However so far as efficiency of the compiled object code is concerned the comparison is much more favourable. One of the key aims of RCC is to provide the highest level of language (i.e. readability) consistent with producing efficient code. The result is that the compiled code is almost as efficient as hand coding.

Extensibility

Under the categorisation of Solntseff and Yezerski (1974), RCC comes under type E, implying that macro expansion

takes place as late as the code generation phase of the compiler. Only MAD (Arden *et al.*, 1969), LACE (Newey, 1968), and ECT (Solntseff and Yezerski, 1972) are given under this category. Under the definitions of Standish (1975), RCC provides all the forms of extension, 'paraphrase', 'orthophrase' and 'metaphrase'.

The nearest systems to RCC outside the family of the Brooker-Morris Compiler Compiler (BMCC) appear to be ECT and ECL (Wegbreit, 1971), although they are still very different. In particular no other systems appear to have the equivalent of format classes or master routines.

Within the BMCC family, SPG (Morris *et al.*, 1970) was developed in a complementary manner, applying the ideas of BMCC at a low and practical level using the smallest possible system. Only BMCC itself has either format classes or master sections. Format class routines are the only aspect of BMCC where both the general idea and the detail of the implementation are recognisably similar to RCC. Master phrases in practice only existed as part of the implementation of BMCC and were not defined for user programs.

Experience with RCC

RCC has been fully implemented on an ICL 1906A and IBM 360/44. In addition there was a version on Atlas which was nearly finished but was turned into a cross-compiler for the 1906A instead, and work is progressing on a CDC 7600 version. Also a subset of RCC, RCCT, (Eissa and Napper, 1976) has been completed on all four machines to assist in bootstrapping RCC. This essentially provides the RCC basic language for *integer* data, without the RCC language processing machinery, but with a limited compiler extension facility.

The size of the standard RCC compiler on the 1906A is about $37K \times 24$ -bit words, of which $19K$ is code. The equivalent 360 figures are $35K$ and $19K \times 32$ -bit words. A further minimum of $4K$ is required to run a small program, and the store requirements of code and data for compiling a large program have to be added to this. The system is therefore a large system, requiring a medium to large computer, particularly if used with full flexibility.

The code generated on all four machines is close to hand coding, approximately of the order of 10% worse. The code required to implement analysis and synthesis statements, and routine calls using phrase variables, including embedded integer parameters, can be kept efficient so long as certain standard conventions are followed.

The compiler itself is relatively slow compared with a compiler for a fixed language. This is particularly because it must be prepared to meet any syntax, and therefore many conventional input and analysis techniques are not applicable (e.g. lexical analysis). Also the generalised structure of the compiler tends to slow it down. User extensions are implemented as efficiently as the basic language, although of course it is the provision of extensions that slows down the whole compiler in the first place.

In the use of the language there is a mixed reaction to 'minor terminators' and considerable support for the introduction of a case statement to replace the simple switch. The authors would agree that there are several improvements which could (and should) be made to bring the base language 'up to date'. New users tend to find most difficulty initially with the language processing facilities. The reason would seem to be that the concepts are completely new to them, since they appear in few other languages. However, the readability of the language and its simple extensibility have been a great success.

Considering other systems, apart from RCC, which have been written in RCC, experience has been gained in writing an extensible compiler, ALEC (Napper and Fisher, 1976), an experimental compiler, MPL1700 (Fisher and McQuarrie,

1977), and a TTL logic network simulator (Kahn and Kinniment, 1976).

In all these systems it has proved entirely successful. In addition experimental work has confirmed the potential of RCC as an extensible language, in particular in providing powerful and efficient application packages (Allison, 1976), and as a language transformation language (Lindsay, 1975).

The best overall test of RCC has been in the logic simulator, where users can describe a network, specify a set of tests to exercise it over a period of time, and then get as output monitoring and performance information. This is an independent project, with a number of people making large or small contributions to it over a period of years. The extensibility of RCC has proved useful in allowing the project leader to make an extension for general use by the group, with individuals making further extensions. The language processing facilities have made easy the task of designing and implementing the user oriented languages for network description and test specifications. The high level language has proved very useful in documentation and maintenance. A particular feature is that some of the logic components, which have to be modelled in

great detail, require large routines to describe them. The modelling is done using a small set of highly descriptive primary and secondary statements tailor-made to the specific problem.

There are a number of further applications of RCC currently in progress, or projected in the near future. Research is being carried out into a general purpose language that provides user definition of primitive data types using in-line code. A package is being written to generate tailor made COBOL updating programs given a (non-procedural) specification of the required operation by the user. It is hoped to write a cross compiler for a microprocessor, and to investigate the automatic transformation of working programs in various languages to and from related forms, e.g. a very compact form (for archiving) or a very descriptive form (for documentation). Of course if the language is RCC, a working program should already be highly descriptive; here an interesting possibility is to transform it into a similarly descriptive program in another natural language, e.g. French, by providing the corresponding vocabulary for identifiers and the corresponding sentence structure for primary and secondary formats.

References

- ALLISON, L. (1976). Extensibility in a Systems Implementation Language, Ph.D. Thesis, University of Manchester, 1976.
- ARDEN, B. W., GALLER, B. A., and GRAHAM, R. M. (1969). The MAD Definition Facility, *CACM*, Vol. 12 No. 8, pp. 432-439.
- BROOKER, R. A., MACCALLUM, I. R., MORRIS, D., and ROHL, J. S. (1963). The Compiler Compiler, *Annual Review in Automatic Programming*, Vol. 3, pp. 229-275.
- EISSA, I. F. and NAPPER, R. B. E. (1976). RCCT—A Simple Extensible Systems Implementation Language, Proceedings of the International Conference of Statistics and Computer Science and Social Science, Cairo, April 1976.
- FISHER, R. N. and MCQUARRIE, G. W. (1977). MPL1700—A High(er)-Level Microprogramming Language, *Software, Practice and Experience*, Vol. 7 No. 6, pp. 747-757.
- KAHN, H. J. and KINNIMENT, D. J. (1976). A Design Automation System for the Teaching of Computer System Design, Proceedings of the International Conference of Statistics and Computer Science and Social Science, Cairo, April 1976.
- LINDSAY, H. E. (1975). The Design, Implementation and Use of the Regeneration Machinery of RCC, M.Sc. Thesis, University of Manchester, 1975.
- MORRIS, D., WILSON, I. R., and CAPON, P. C. (1970). A System Program Generator, *The Computer Journal*, Vol. 13 No. 3, pp. 248-254.
- NAPPER, R. B. E. (1968). The need to Revise the Compiler Compiler, *IFIP Conference Proceedings*, pp. B23-B27.
- NAPPER, R. B. E. and FISHER, R. N. (1976). ALEC—A User-Extensible Scientific Programming Language, *The Computer Journal*, Vol. 19 No. 1, pp. 25-31.
- NEWBY, M. C. (1968). An Efficient System for User Extensible Languages, *AFIPS Proceedings (FJCC)*, Vol. 33 No. 2, pp. 1339-1347.
- SOLNTSEFF, N. and YEZERSKI, A. (1972). ECT—An Extensible Contractible Translator System, *Information Processing Letters*, Vol. 1, pp. 97-99.
- SOLNTSEFF, N. and YEZERSKI, A. (1974). A Survey of Extensible Programming Languages, *Annual Review in Automatic Programming*, Vol. 7 pp. 267-307.
- STANDISH, T. A. (1975). Extensibility in Programming Language Design, *AFIPS Proceedings*, Vol. 44, pp. 287-290.
- WEGBREIT, B. (1971). The ECL Programming System, *AFIPS Proceedings (FJCC)*, Vol. 39, pp. 253-262.

Book review

Research Directions in Software Technology edited by P. Wegner, 1979; 869 pages. (The MIT Press, £15.00)

This is a long and ambitious book aimed at covering the complete field of software technology. It has its origins in papers presented at meetings in 1976 and 1977. The stated purpose of the book is to stimulate a dialogue between research workers and practitioners in the field of software technology. The book is divided into four parts each consisting of a set of papers by experts in the field followed by commentaries on these papers by others. This style of presentation leads to a significant amount of repetition with each author seeing the need to define terms and provide a framework for his own contribution.

The Software Engineering section includes papers by Boehm and Mills which review the tools available for managing software production introducing the software life cycle, specification languages, top down design, error days, etc. Anybody familiar with the field would find little new here and yet the level of detail is such that it provides little more than a pointer to other sources for a newcomer. The most interesting paper in this section is a frank discussion of the management of the MULTICS project describing both the successes and failures.

The Software Methodology section includes a good paper by

McGowan and McHenry suggesting that the software life cycle model breaks down when a system is under continuous evolution and proposes a continuum model in its place. This same criticism is touched upon by a number of authors. This section also includes papers by Liskov and London on formal methods of program specification and verification.

The Computer Systems Methodology section has an excellent paper by Wegner on concepts and research directions in programming languages. Other papers discuss the current state of operating systems research and architectural design including software and hardware problems associated with distributed computing systems.

The final section on Applications Methodology is the shortest and perhaps least successful. It covers too wide an area—from CAD packages and data base management systems to natural language and artificial intelligence systems. It does no more than indicate the current state in a superficial manner. Even then, the passage of time has dated some of the presentations.

I found the book disappointing. The presentation is shallower than should have been possible in a book of this size. It also spends more time cataloguing the past than providing clear indications of research directions. Even so, the book has good parts. It is just a pity that over 800 pages have to be read to find them.

F. R. A. HOPGOOD (Didcot)