S O M E E N H A N C E M E N T S

T O T H E

R E V I S E D C O M P I L E R C O M P I L E R S Y S T E M

b y

M A R T I N R . C A R T E R

Thesis submitted for the degree
of Master of Science, at the
University of St. Andrews, November 1979

ABSTRACT

Enhancements have been made to the Revised Compiler-Compiler, system [1]. In general these consist of extensions to the RCC language, improved error diagnostics and improved internal representation. The extensions to the language consist of formalising procedures local to RCC routines, adding a new FOR loop and introducing a case statement. The error diagnostics of the system have been enhanced to inform the user of the source statement causing a run-time crash. The internal representation of formal syntactic classes, has been converted from 32 bit full-word, to 16 bit half-word format, saving approximately 50% of the memory space previously used.

DECLARATION OF ORIGINALITY :

I hereby declare that the research herein has been carried out and the thesis composed by me, and that the thesis has not been accepted in partial or complete fulfilment of the requirements of any other degree or professional qualifications.

Signed :

Martin R. Carter

ACKNOWLEDGEMENTS

The author wishes to thank Dr. Richard Fisher for his supervision, help and patience throughout the year, and the SRC for their financial support.

CONTENTS

C H A P T E R O N E

1 Introduction

1.1 The background to the project :

The Revised Compiler-Compiler

The Revised Compiler-Compiler, RCC, was designed in the late 1960's as a comprehensive revision of the Compiler-Compiler, CC, of Brooker and Morris [24]. It is intended as a Systems Implementation Language suitable for general systems and non-numerical applications. It has the usual compiler-compiler features for compiler writing and general manipulation and transformation of languages : providing a formal language for defining, recognising, analysing and synthesising syntactic elements, based on an extended BNF, using the new data type phrase.

RCC is an incremental Compiler-Compiler i.e. any program written in RCC is compiled and added to the end of the existing RCC system. Thus a compiler written in RCC is automatically extensible, the user need simply write extra syntax and semantics (in RCC) which are compiled and automatically added to the existing compiler. RCC is written in itself, it is therefore itself extensible. With phrase variables and the language associated with them, the user can make powerful extensions to the syntax and semantics, without significant loss of efficiency. An exposition of extensibility in RCC is given by Napper and Fisher [1].

The fact that the system was implemented in the 1960's means that various features that appear in more modern languages are absent. The base language, for example, contains no case statement, only a simple computed switch. Another feature of the base language, the FOR statement, is not compatible with the FOR statement in Algol type languages. This can be confusing to programmers used to those other languages. In this work, the extensibility features of RCC have allowed a case statement to be implemented, using more primitive constructs already in the language. A new FOR statement has also been added, using existing semantic routines specifically designed for implementing FOR statements.

The debugging facilities in RCC are generally good, allowing the user to examine the value of all variables at the time of a crash, and producing a back-trace of the routines dynamically active at the time of the crash. But as RCC does not keep track of source information once a routine has been compiled, the position of a run-time crash is not presented to the user in terms of the source program, merely an absolute address of the machine instruction causing the crash, which leaves room for improvement. This improvement has been achieved, mainly by modifying existing routines and adding a new data structure.

The RCC system was, some time ago, transferred from an ICL 1906A with a 24 bit word, to an IBM 360 with a 32 bit word. To ease the transfer, all the information previously held in 24 bit words was directly mapped on to 32 bit words. In the case of the internal form of formal syntactic classes inparticular, this resulted in the wastage of a large part of each word. There are, therefore, considerable possibilities for compaction of this data, if the format and associated algorithms are changed. This task has been accomplished. The RCC system has also been extended, to provide new commands for handling these data items in their compacted state.

1.2 Project Specification

1.2.1 The main points

This work consists of enhancements to the RCC system and can be characterised as follows :-

1/. Extensions to the language

a/. Formalised local procedures for routines

b/. A new FOR loop statement

c/. The introduction of a case statement

2/. Improved run-time error diagnostics and object code analysis tools

3/. Conversion of formal classes from 32 bit to 16 bit representations

Background information, independent of the RCC system, on which this work is based, will be given in this chapter. An explanation of the parts of the RCC system that are relevant to this work, will be given in chapter 2.

1.2.2 Extensions to the language

a/. The advantages of using subroutines or procedures and other kinds of modular decomposition, are well known. As will be explained in Chapter 2, the RCC system is split up into routines and to extend the system a new routine can be added or an existing one amended. The routines form a one-level structure, i.e. routines are not nested. Parts of a routine, however, may be written as procedures local to that routine and usable only within that routine. In the existing system, the use of these local procedures was very informal and there were no protection mechanisms to prevent their incorrect use. Their usefulness may be increased if their use is more formalised. Many programming errors in the use of local procedures in RCC, may be detected at compile-time. Warning messages can be produced if a particular usage is unsafe or nonsensical or just bad programming practice.

b/. As Barron [2] and Aho and Ullman [3] point-out, variance in the syntax and semantics of FOR loops can be extremely confusing. Therefore, in designing a new FOR loop for RCC, the most desirable properties it should possess by default were considered, whilst wishing to retain the flexibility and efficiency of the existing RCC FOR loop. As most programmers liable to come into contact with RCC will be used to the Algol type FOR loop, it will be much easier for them to use a construct with which they are familiar. It was desirable, therefore, to produce a default syntax and semantics that a programmer used to Algol W, Pascal, BCPL, PL/1 etc would have no difficulty with. Brown [4] points-out the advantages of this approach. The FORTRAN 77 [5] DO loop has also been redesigned to be more like Algol FOR statements.

c/. The case construct was first proposed by Wirth and Hoare [6], as a replacement for the Algol 60 SWITCHed GOTO. The most primitive form of case is used in PL/360, Algol W and Algol 68 with the addition of the OUT "default" option. With the Pascal case, based on Hoare's ideas [7], and the use of case "labels", the construct is seen more as a replacement for IF ... THEN ... ELSE IF ... statements. For the reasons given by Tsichritizis [8], this type of case is far more satisfactory. BLISS, BCPL and C, all use this type of case. But in all these languages, the options must be constant expressions.

As shown by Wrandle-Barth [9], however, the case statement may be further generalised to allow any type of conditional expression allowable in an IF statement as a case option. Now replacement of IF ... THEN ... ELSE IF ... statements becomes complete. This type of extension to the case construct has been incorporated into the S-Algol language [10].

The RCC case has options for both the semantic equivalent of the SWITCHed GOTO and the IF ... THEN ... ELSE IF ... The user can clearly specify which of the types, with their attendant advantages, is required.

Consideration was also given to the form of "overall else" or "default" option, to be incorporated. Languages like BCPL, C, and Algol 68 allow the user to leave out the default option, then if no matches are found, control simply passes to the statement following the case. But Pascal has no default option, a run-time error being produced if no matches occur. The reason would seem to be, that as Pascal is quite strongly typed, the programmer should know exactly what any variable expression can evaluate to, therefore the use of a default can be seen as sloppy programming. On the other hand, S-Algol takes the diametrically opposite view and insists that a default option always be present, generating a compile-time error if it is not. The decision in the RCC case implementation, was to have an optional default option, with a strong recommendation to the user, that it is always present.

1.2.3 Improved run-time error diagnostics and deassembly of object-code

"Debugging is currently one of the largest single costs of Computing" - Horning [11]. Pool [12], says that the importance of testing and debugging are still under-estimated and that more tools to facilitate these should be made available. A major increase in the usefulness of diagnostic information can be made if all the information presented to the user, is expressed in terms of the source program. Both Horning and Pool stress that, good error messages should be source-orientated.

When a run-time error occurs in RCC, the system has been enhanced to pin-point the source line or lines at which the error occurred and if there is more than one statement per line : the statement number within this line. This has been implemented with a minimum of overhead and can be selectively applied to the parts of a program that are liable to cause run-time errors. The facility augments the existing crash-time diagnostic information available to users.

When an experienced user of RCC, interested in highly optimised code, has written a complex macro, it is sometimes difficult to gauge the efficiency of the object code generated. This can only be done properly if the sequence of instructions generated can be inspected. In the existing system, the only formats for printing memory are hexadecimal and decimal. As ploughing through a hexadecimal print is both time consuming and boring ; facilities have been added which decode hexadecimal and print the object code in terms of correctly formated assembler instructions and their mnemonic codes. As it is possible to distinguish between program and data areas in RCC, a comprehensive dump routine can be produced, that gives the correct format for the type of information stored.

1.2.4 Conversion of formal classes from 32 to 16 bit format

As was previously stated, when the RCC system was transferred from a machine with a 24 bit word to a machine with a 32 bit word, considerable wastage of bits was incurred in the internal representation of formal syntactic classes. It has been possible to compact a large amount of this data into 16 bit half-words. A sublanguage has been developed to enable half-words to be handled in RCC. The routines that manipulate formal syntactic classes have had to be altered and others affected by this change were also amended.

About 5K words are used in the RCC system to hold formal syntactic classes. The 16 bit implementation saves approximately 50% of this space, or 2.5K words.

C H A P T E R T W O

2 A description of the relevant parts of the RCC system

2.1 Introduction

2.1.1 Extensibility

An overall description of the RCC system has been given elsewhere [1], also a reference manual is available, Fisher [13]. So, only those parts of the system relevant to this work, plus some necessary general description, will be given here.

An important feature of the RCC system is its extensibility. The most basic form of extensibility available, is the "primary" routine, which can add a new statement to a language with its own syntax and semantics. This is similar to the syntax macro concept, but more powerful (see [1] for a discussion on this). Primaries are integrated into the compiler and the in-line code generated by them is as compact and efficient as if it had been part of the original language.

The extensibility of RCC is increased by the incremental nature of the system, where any user program is compiled and automatically appended to the existing system. At any stage the user may store the binary image of the current system in a separate file, to obtain a personal extended version of it. Through the use of the "master section", an RCC program and its data, are formally part of the actual RCC compiler itself. There is, therefore, no formal distinction between compiler-compile, compile and run-time.

Extensions can also be made in the form of "secondary" routines, which are conventional subroutines. As the formal parameters can be phrase variables, the user can define new syntax specifically for a particular routine.

2.1.2 Program structure

The text of an RCC program is divided into sections, all at the same level. Routines are not nested within routines and there is no block structure. Each section is called a "master section" and is headed by a "master word", which is an upper-case, underlined name on a line of its own. The commonest of these are ROUTINE, CLASS, GLOBAL and STOP. The text which may appear in a master section is dependant on the master word which heads it. The text of a ROUTINE master section consists of a routine definition; a routine heading followed by the routine body. The GLOBAL master section contains a set of global scalars or arrays. The CLASS master section contains a set of class (syntactic) definitions. A STOP must appear at the end of a program to terminate it.

Users can define their own master words using MASTER routines. An RCC program, therefore, consists of a sequence of "system" and "user" master sections. The only constraint in the ordering is that RCC is a one-pass language and all entries must be declared before they are used.

2.2 The RCC meta-language

Formal syntax can be specified following the CLASS master word in an RCC program. A meta-language is available, which at its simplest, is similar to BNF. For example, the class [INTEGER] can be defined by :-

[INTEGER] = [DIGIT] [INTEGER], [DIGIT]

The general parsing algorithm used by the RCC system is top-down, left-to-right, fast-back. The restrictions that this type of parser enforce on the rules of a grammar are that :-

1/. There must be no left-recursion

2/. In certain cases the alternatives of a rule must be ordered in a particular way

Some of the facilities available in the RCC meta-language, but not in BNF, are briefly described :-

1/. List classes. A class can be defined as a list class, which is a sequence of one or more occurrences of a given alternative. The above definition of [INTEGER] could be written :-

[INTEGER] = LIST OF [DIGIT]

A separator alternative, may also be specified :-

[EXPRESSION] = LIST OF [OPD], SEPARATED BY [OPR]

To help reduce the number of rules, if a standard class name ending in a "*" is met undeclared, it is implicitly declared to be a list class :-

e.g. [DIGIT *] is implicitly declared as
[DIGIT *] = LIST OF [DIGIT]

Also a separator may be specified by giving it as a parameter (see 5/. below).

e.g. [EXPRESSION] = [OPD */[OPR]]

2/. Restricted classes. These are used for parsing efficiency reasons only. If a user knows that each alternative must start with explicit terminal symbols of a particular type, the class can be declared as restricted. To considerably reduce the number of comparisons needed, a serial search of the alternatives is replaced by hashing. The normal top-down, fast-back parsing is overriden so that analysis can be made from the bottom-up via a hash directory, to only those alternatives with the same initial symbol(s). Top-down parsing is then resumed.

3/. Users can also override the parser, by defining their own informal CLASS routines. A routine is written describing the syntax analysis algorithm for a particular class and the information to be stored on the analysis record, if it is successfully recognised.

4/. Qualifications may be associated with a particular class, by inserting them immediately after the class name. If they are associated with the class on its definition, they are intrinsic qualifications, which apply to all uses of the class in other definitions. If they appear with a class on the right-hand side of a class definition, they only apply to those particular class instances. If a class has both intrinsic and instance qualifications, the resultant qualifications are a logical "or" of the two. Some of the more common qualifications are :-

a/. The symbol r means no record. No subtree for the class will be inserted on the analysis record. This is used when the occurrence of a class is important, but there is no point in keeping a record of it on the analysis record. e.g. a separator :-

```
[ SEPARATOR r ] = :,;
```

b/. The symbol ? appearing as a qualification means that the appearance of a class is optional.

c/. The symbol a means non advance. On successful recognition of the class, the input symbol position is not moved on. This enables the parser to peep ahead.

5/. A class word may have a parameter, which appears immediately after the class name and before any qualifications. It consists of a "/" followed by a sequence of symbols and/or class words, enclosed in round brackets (which may be omitted if the parameter contains only one symbol or class word). The use of parameters in list classes, has already been illustrated.

Parameters can also be used with informal classes. There is a system defined informal class, [Q] for example, which is used to obviate explicit definition of a bottom-level class by allowing as a parameter, strings of symbols separated by commas, which are alternatives. e.g. instead of separately defining :-

[IF] = IF, UNLESS

the following,

[Q/(IF, UNLESS)]

can be used wherever [IF] would appear in the syntax.

2.3 The RCC base language

2.3.1 Introduction

This is the language used in RCC routines, (ROUTINE master sections). It is at a moderate level, leaving out various features that are standard in newer languages, but including features that are unusual.

Variables are untyped, being effectively integers, bit patterns, or addresses, etc, as required by the user. An identifier can be declared as a constant or as a variable. If they are not explicitly declared, they are allocated automatically by the system in the automatic allocation area, AAA. The user may, however, specifically request storage of a scalar to be a specific store address or a register. Users may have complete control over storage allocation, by using machine-dependent options in declarations, if this is desired.

There is a notation for indexed addresses, e.g. (a+i) and a(i), both refer to the location whose address is given by adding i to a. Expressions have no subexpressions, no user-defined functions and no operator precedence. The operators + and -, together with logical and shift operators, are available. Evaluation is from left to right.

A scalar is referred to by a lower case name. Syntactically it must form a valid [NAME]. A [NAME] recognises an RCC name, which is a sequence of one or more letters possibly followed by digits, and puts it on the analysis record. A [NAME] may have a parameter to specify whether its format will be lower case, upper case or either of those underlined. [V] gives the possible syntactic forms for a variable, [E] for an expression, and [C] for a constant. See Appendix 6 for details of this syntax.

2.3.2 Control constructs

The organisation of control in RCC is unusual, with a formal clause and sentence structure, being used to specify implicit control. 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 ;). A sentence consists of a set of statements separated by minor terminators and finishing in a major terminator. There are 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.

1/. Imperatives.

a/. The assignment statement has the syntax :-

[V] = [E]

b/. The switch statement has the syntax :-

GO [Q/(AND, TO) r] [NAME */(, [EOL ?])] BY [E]

where [EOL], which is intrinsically no-record, recognises an end-of-line.

e.g. GO TO p,test symbol,b1,b2,b1 BY expression

All the names must be labels. A label may precede any statement in a ROUTINE master section. Syntactically it is a lower case underlined name, [NAME/a], followed by a minor terminator. The expression is evaluated to yield an integer (1 to 5, in the example), a jump is then made to the corresponding label.

c/. The PERFORM statement, enables the use of a parameterless procedure, local to a routine. Syntactically it has the form :-

PERFORM [NAME]

Where [NAME] refers to both a name and a label. Its action is to set the scalar [NAME] to the address of the next instruction and transfer control to the label [NAME]. A corresponding statement of the form :-

FINISH [NAME]

transfers control to the address held in the scalar [NAME]. This is used as a dynamic termination for the local procedure.

2/. Conditional clauses.

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, separated by minor terminators. The conditional clause qualifies all imperative statements up to the end of the sentence. The imperatives may contain a single OTHERWISE, which provides an else control branch. There is no formal type boolean in RCC. Instead, the user can define more IF clauses as required. The basic conditional clause is of the form :-

[Q/(IF, UNLESS)] [E] [RCC COMP] [E]

where [RCC COMP] = =,>=,<=,>,<,/=

e.g. IF a + b > (a + 1) + 1 : ....

Compound statement brackets can be used to enable general "boolean expressions" of conditional clauses to be written as a single clause e.g.

```
IF b = Ø ; OR IF { a = 1 ; AND b1 = Ø } :
  PRINT : YES :
OTHERWISE : PRINT : NO
```

There is also the system defined conditional clause

IF [E] :...

which is equivalent to IF [E] /= Ø :... e.g. IF a : will be true if a is non zero, but false if a is zero.

3/. The FOR loop

A FOR sentence consists of a FOR clause followed by a set of imperative statements which form the "body" of the cycle. Again, a number of FOR clauses are provided by the system and the user can define others. All FOR loops in RCC are primary and are added to the class [RCC CYCLE]. The full syntax of the basic [RCC CYCLE] is :-

[NAME] = [E] [Q/X ?] [R2 ?] [R3 ?] [Q/X ?]

where

[R2] = DOWN, ALONG CHAIN, STEP [V] = [E], STEP [E]

[R3] = TO [RCC COMP ?] [V] = [E],

TO [RCC COMP ?] [E], UNTIL [RCC COND]

The first expression specifies the start value of the control variable given by [NAME]. [R2] is optional and specifies the increment required for the control variable on each iteration. The default is an increment of constant 1. The first alternative, DOWN, is equivalent to N = N - 1 (where N is the control variable). The second, ALONG CHAIN, allows the FOR to be used on iterative operations on linked (chain) store, it is equivalent to N = (N + 1). STEP [V] = [E] produces a static increment, where the [E] is evaluated before the cycle-body is entered and placed in [V], it is equivalent to N = N + [V]. The last alternative, STEP [V], is dynamic, where [E] is evaluated on each iteration of the cycle-body, and is equivalent to N = N + [E].

[R3], which is again optional, specifies the terminating condition. TO [RCC COMP ?] [V] = [E], produces a static limit, where [E] is evaluated and placed in the [V] before the cycle-body is entered. TO [RCC COMP ?] [E] is the dynamic option. The default test for termination, is that of the control variable being equal to the limit. [RCC COMP ?] is used to specify a terminating condition other than equality. e.g. If a > is specified as the [RCC COMP ?], termination will occur when the control variable exceeds the limit. UNTIL [RCC COND] allows any defined IF clause to be used as the terminating condition ([RCC COND] is the system format class to which all IF clauses belong).

The character "X" may appear after the initialising expression and/or the limit expression. This means "exclusive", i.e. the specified initial/limit values are excluded from the rest of the values which the control variable may take.

Two imperative statements are provided, for use within the cycle body, to enable the user to explicitly terminate an iteration or the whole statement.

FINISH CURRENT [NAME]

(Where [NAME] is the previously recorded control variable) This statement causes termination of the current iteration. The second imperative :-

FINISH EACH [NAME]

causes termination of the whole cycle i.e. control is transferred to the statement following the cycle.

(FINISH CURRENT and FINISH EACH are equivalent to BCPL's "loop" and "break" respectively in for loops, or C's "continue" and "break").

The key word CYCLE may be used in place of FOR, in which case END OF CYCLE i (where i is the control variable) is used to explicitly terminate the cycle. This CYCLE option or explicit cycle, can eliminate one level of compound statement bracketing. In general, it is used for cycles with large cycle bodies, perhaps with other cycles using the FOR option and smaller bodies embedded within it.

Note from the above that :-

a/. The default is a dynamic implementation. This is less safe and less efficient than a static implementation. In order to force a static implementation, the user must resort to odd looking syntax :-

```
FOR i = 1 STEP step = a+b(q) TO term = 1Ø5 + n :
```

b/. The default test for termination is equality e.g.

```
FOR i = 1 STEP 3 TO 8 :
```

Will not terminate, as i will never be equal to 8.

c/. The cycle-body is executed by default at least once, even if the termination condition applies immediately.

The X must be used by the user to obtain the possibility of zero iterations of the cycle-body e.g.

```
FOR i = a-1 X TO > dummy = x+y X :
```

is, in fact, equivalent to the Algol W :-

```
for i := a until x+y do
```

This form now has a static termination condition, an inequality test for termination and the possibility of zero iterations of the cycle body.

d/. The syntax UNTIL [RCC COND] is confusing to programmers used to Algol type languages e.g.

```
FOR i = 1 STEP 1 UNTIL n :
```

The n is taken as a boolean and is equivalent to :-

```
FOR i = 1 STEP 1 UNTIL n /= Ø :
```

This has, therefore been avoided in the new FOR loop.

4/. The WHILE and REPEAT.

The WHILE clause has the syntax :-

```
WHILE [ RCC COND ]
```

where [RCC COND] is the set of conditional clauses e.g.

```
WHILE a < b : ....
```

The semantics of the WHILE are that the body is repeatedly executed as long as the condition remains true.

The REPEAT clause can be used as the last statement in a compound statement (where it must be followed by a }) :-

```
{ ....; ....; ....; REPEAT }
```

This unconditionally transfers control to the start of the compound statement. There is also a conditional version :-

```
{ ....; ....; ....; REPEAT UNTIL [ RCC COND ] }
```

Which is equivalent to :-

```
{ ....; ....; ....; UNLESS [ RCC COND ] : REPEAT }
```

2.3.3 The language for type chain variables

The "chain store" provided by the system, offers a linked list type of storage. A special sublanguage is provided for processing chain variables.

2.3.4 The language for phrase variables

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

```
CLASS

[ DIGIT ] = 0,1,2,3,4,5,6,7,8,9

[ INT ] = [ DIGIT ] [ INT ], [ DIGIT ]

[ MONTH ] = JAN,FEB,MAR,APR,MAY,JUN,AUG,SEPT,OCT,
            NOV,DEC,

[ DATE ] = [ INT ] [ MONTH ] [ INT ],
           [ INT ] : [ INT ] : [ INT ]
```

This declaration defines four sub-types, [DIGIT], [INT], [MONTH] and [DATE], of type phrase. A phrase variable is a variable whose type is one of the defined sub-types, e.g. m with class [MONTH] and d with class [DATE]. A phrase variable is declared as an INDEX variable and the associated class is given every time the variable is used e.g.

```
[ MONTH m ], [ DATE d ], [ MONTH m1 ]
```

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 :- JAN,FEB, ..., DEC. [DATE d] can have an infinite number of values of the forms :-

```
16 APR 79   or   16 : 4 : 79
```

There are statements provided to operate on phrase variables :-

a/. To analyse phrase variables and break down their structure.

b/. To synthesize phrase variables from integer information or from the values of existing phrase variables.

c/. To input and output phrase variables

Details of the instructions involved in processing phrase variables can be found in the RCC manual [13].

2.3.5 Routines

There are several different types of routine in RCC :-

1/. Secondary routines

2/. Primary routines

3/. Class routines

4/. Format class routines

5/. Master routines

In all cases the ROUTINE master section consists of the routine heading, on a line of its own, followed by the routine body.

1/. The secondary routine, corresponds to the conventional subroutine. i.e. During compilation of a secondary routine call, a cue is compiled so that the routine is entered at run-time. It does not have an identifier as its name, but a "format", which is a string of fixed symbols interspersed with parameters in fixed positions. The parameters can be of type phrase or integer. In the routine heading, the complete specification of the formal parameter is given in the appropriate position, enclosed in square brackets :-

```
ROUTINE

INTERCHANGE [ V x ] AND [ V y ].

dumy = x ; x = y ; y = dumy
```

[V x] specifies a formal parameter of type integer with a local name x. There are three ways in which a parameter 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 to two copies.

A routine call comprises the given set of symbols interspersed with appropriate actual parameters. A call on the above might be :-

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

Secondaries may have one of three characteristics to specify the way they are called :-

a/. BASIC - no stacking/destacking on entry/exit

b/. STANDARD (default) - the volatile environment is stacked

c/. RECURSIVE - same as STANDARD with all parameters and local scalars allocated storage on the stack

Secondaries also can have one of two characteristics to specify their control type :-

a/. IMPERATIVE - the routine is an imperative statement. FINISH is the dynamic terminator.

b/. IF - the routine is a conditional clause. The dynamic terminators being, CONDITION SATISFIED or CONDITION NOT SATISFIED.

2/. A primary routine is entered at compile-time, rather than a cue being planted to a subroutine to be obeyed at run-time. In general it describes how to compile the appropriate in-line code. Its parameters can only be of type phrase. At its simplest, the call of a primary can be expanded out into a sequence of existing language statements. To assist in this, the system provides the instruction :-

```
COMPILE { [ RCC BS * ] }
```

Where [RCC BS] consists of the set of base language statements plus any defined so far by the user. Its action is to compile code for the sequence of given statements. COMPILE also uses the control characteristics of the statements being compiled, to link together successive statements. The following is the system defined NEXT macro :-

ROUTINE

```
PRIMARY IMPERATIVE : NEXT [ Q/( *, - ) ? q ] [ NAME n ]

IF [ Q/( *, - ) ? q ] = * : COMPILE { [ NAME n ] =
                                ( [ NAME n ] + 1 ) } ; FINISH
IF [ Q/( *, - ) ? q ] = - : COMPILE { [ NAME n ] =
                                    [ NAME n ] - 1 } :
  OTHERWISE : COMPILE { [ NAME n ] = [ NAME n ] + 1 }
```

Only parameters of type phrase, with call specification E, and no explicit store specifications are allowed. The linkage types are the same as for secondaries, except that linkage type BASIC is not allowed.

The control types IF and IMPERATIVE also exist for primaries, and several others are available. One of the most important is the control type CYCLE, which means that a call of the routine is treated as a FOR clause. The routine can be called as an explicit or implicit cycle.

3/. Class routines enable the user to define a class by algorithm. Such a class has an associated routine, whose name is the class name together with any intrinsic parameter and/or qualifications. The routine operates directly on the input stream. It is called automatically by the parser, and it returns control by imperatives RECOGNISED or NOT RECOGNISED as appropriate. The information recorded by the class routine is placed on the main system stack.

4/. So far as the user is concerned, format class routines usually describe user constructs to be used in user text, and not RCC constructs to be used in RCC text. Details of these appear in the manual [13].

5/. User data is in general split into any number of master sections, each headed by one of a set of master words defined by the user. Each master word is defined by a corresponding master routine. The name of the routine is the master word it is defining.

Whenever a master word is met in the input stream, the system passes control to the corresponding master routine. This routine should then process all text following the master word, up to the next one.

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

```
ROUTINE

MASTER : SORT
         ....
         ....
         ....

SORT
487
2976
243

SORT
....
....
....

STOP
```

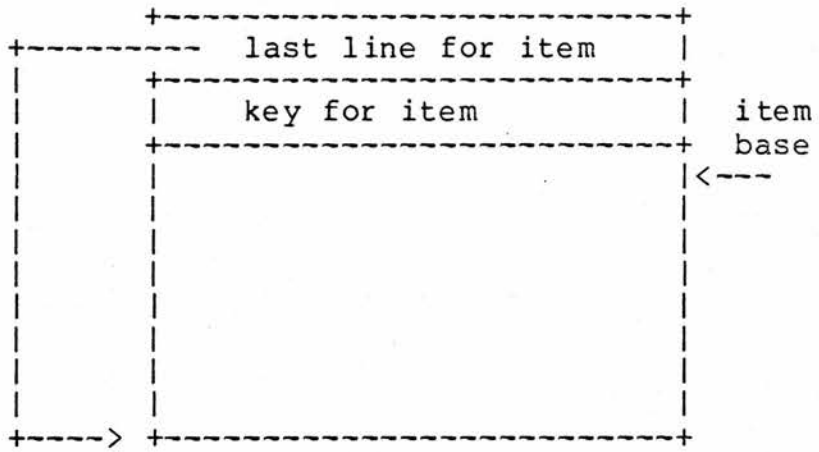
The system master sections operate in the same way. The action of the CLASS master routine is to repeatedly recognise class definitions and process them. The ROUTINE master processes the routine heading and then repeatedly recognises and compiles RCC statements.

2.4 Relevant points on the RCC implementation

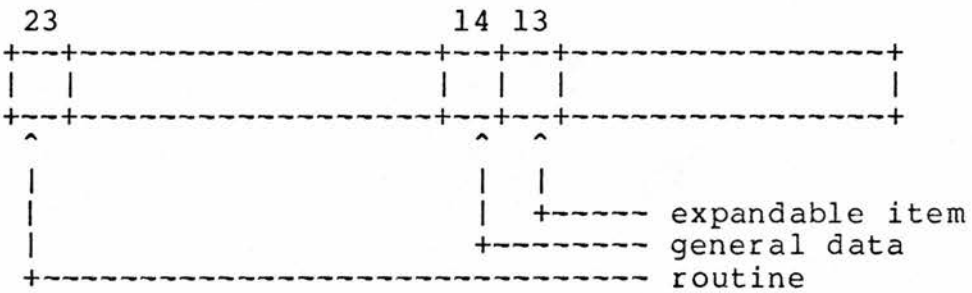
2.4.1 Items

Although is is not the purpose here to discuss the implementation of RCC, it is necessary to give some explanation of items and the RCC control structure, to enable parts of this work to be understood.

All nonscalar data and routines in the RCC system, are held in "items" on the item stack. An item is a data structure which enables all the routines and sets of data used by the system to be handled in a standard way. There is an item index giving the address of each item, and each item has general information associated with it, to identify itself to the system. Each item has the same general form :-



The index entry points to the third word in the item, which is referred to as line Ø. The first word, line -2, always contains the address of the last line of the item, relative to the item base, line Ø. The second word, line -1, always contains the "key" for the item. This identifies the item to the system and contains further distinguishing information :-



If neither bits 23 nor 14 are set, then the item is a class item. i.e. It contains the internal representation of a formal class definition.

The most common forms of items are :-

1/. ROUTINE - all routines. Routine items have an area for instructions followed by an area for data local to the routine.

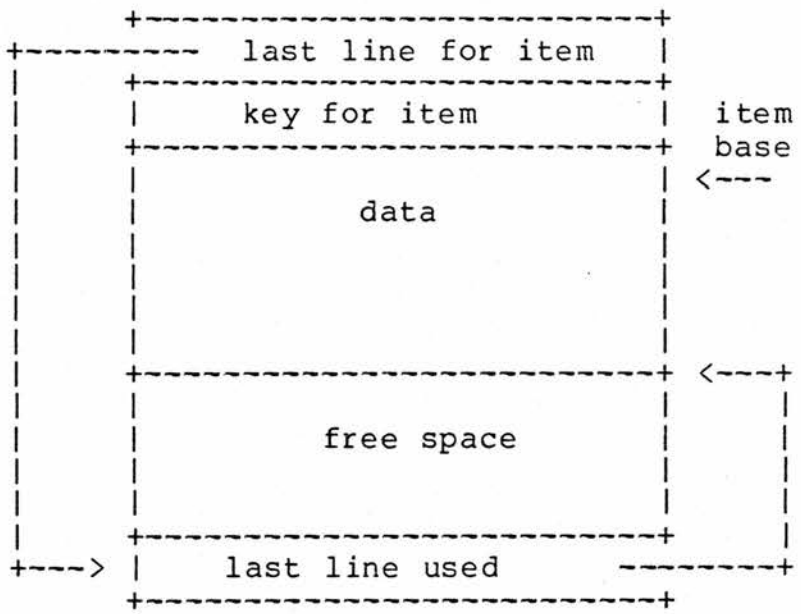
2/. CLASS - the internal form of formal syntactic classes and format classes

3/. dictionaries - class names, globals, master phrase names

4/. general data - miscellaneous sets of data

An item is deleted by deleting the index entry to it. Redefinition is achieved by redirecting this pointer to the new item. The system can compact the item stack, by placing all active items at the front. Nonactive items are garbage collected. There are several system provided routines for handling items : to delete them, compact the item stack, and relocate them in the stack.

In order to add data to an existing item, it must first be relocated to the front of the item stack. Because there are items that are added to regularly at arbitrary times, this would lead to inefficiencies. Therefore, items can be made to be expandable. An expandable item contains a number of unused locations at the end. The last line of the item contains the address (relative to the item base) of the last line used. Therefore, these items need only be relocated if they are added to when the free space area is insufficient. A system routine also exists to expand expandable items by a given amount.



2.4.2 Control

RCC provides, automatically, organisation of control for all its constructs Each new routine that is added to the system, has its control type specified, so that it can be integrated into the overall control structure. The primary control types - IMPERATIVE with a single control exit, IF with two control exits - CONDITION SATISFIED and CONDITION NOT SATISFIED, and CYCLE with two control exits - FINISH CURRENT and FINISH EACH and one control entry point - INTER CYCLE ACTION. There is also type SIMPLE JUMP, which tells the system not to update the current control situation before entry to the macro, so that all control requiring to pass to this instruction has not already been filled in to the current compiling point, as is the case for the other classifications. This is because the routine will contain a branch off to somewhere else in the code and the control can be diverted straight to that point. e.g. GO TO [NAME] and FINISH are SIMPLE JUMP primaries.

The routine that carries out the control organisation is COMPILE { [RCC BS *] }, it looks at the control type of the primary or secondary routine and updates its description of the environment in view of this new type or creates a new level if the piece is a macro.

There is also the possibility of a number of "control levels" in which and across which, this automatic control operates. A new control level is set up for each of :-

1/. A call of a primary routine, and each call within a call.

2/. The start of a compound statement (an "action" level).

3/. An explicit cycle

For each control level, current at any point during the compilation, a set of 12 words exists to define the current control status. This data set for the level, is linked in both directions to the data sets of the levels above and below, forming a nested stack. There is a global pointer to the current system level. The data set for each level contains an indicator of what type of system level this is e.g. IMPERATIVE, IF etc, and the control status e.g. cycle imperative, if status etc, plus various other information. The data set for a CYCLE control level will contain the FINISH CURRENT and FINISH EACH control pointers, plus the INTER CYCLE ACTION pointer and the control variable. An IF control level will contain the CONDITION SATISFIED and CONDITION NOT SATISFIED control pointers.

C H A P T E R T H R E E

3 Extensions to the language

3.1 Formalising the use of local procedures in routines

3.1.1 The need for a formalised local procedure

The function of the statement PERFORM [NAME] has been described in Chapter 2. The only difference between the PERFORM [NAME] and the GOTO [NAME] instruction, is that the return address i.e. the address of the statement following the PERFORM, is held in the scalar [NAME]. FINISH [NAME] is a dynamic terminator and can appear anywhere in the local procedure body. It causes a jump to the return address held in the scalar.

At present in RCC, the local procedure is not formally independent. For example there is no check on control entering the procedure other than via a PERFORM. Control can implicitly enter the procedure from the main body of the routine. At present the procedure would be allowed anywhere where a label may appear e.g. not at the end of a routine, within an action level, within a sentence, etc.

The user must, at present, explicitly terminate a local procedure, with the FINISH [NAME] dynamic terminator. Otherwise control will pass out of the bottom of the procedure, probably causing a run-time error. This is not, however, consistent with routines. During the compilation of an routine, a FINISH (or CONDITION SATISFIED or RECOGNISED) is automatically compiled if there is any implicit control when a master word is read i.e. the end of the routine is reached. The particular instruction compiled depends on whether the routine is IMPERATIVE, IF or CLASS, respectively. To be consistent, implicit termination should also be available for local procedures.

If control enters a procedure implicitly i.e. not via a PERFORM, results will be undeterministic when the FINISH [NAME] is encountered, as a return address would not have been set.

This primitive state of affairs can lead to bad programming practices and is a possible source of errors.

3.1.2 The new local procedure

A new Primary routine is introduced, the syntax of which is :-

LOCAL PROC [Q/(EDURE) ? r] [NAME /a label] :...

Its effect can be shown as follows :-

```
ROUTINE
     ....
     ....
     PERFORM a
     ....
     PERFORM a
     ....
     ....
     ....
     FINISH  <----- FINISH automatically inserted
                          by LOCAL PROCEDURE routine

LOCAL PROC a :
     ....
     ....
     PERFORM d
     ....
     PERFORM d
     ....
     ....
     FINISH a  <---- automatically inserted
                     by LOCAL PROCEDURE routine

LOCAL PROC d :
     ....
     ....
     ....
     ....
     FINISH d <---- automatically inserted
                    by MASTER : ROUTINE
```

The new routine ensures that :-

1/. The local procedure does not occur within nested bracketing. Every time a compound statement bracket {, is opened an "action level" is created. If the local procedure occurred within such a level, it would imply that it was an integral part of a larger segment of program and therefore not an independent unit suitable for use as a procedure. The logic would therefore be diffuse and errors could easily occur. (The local procedure must occur at top routine level).

2/. The local procedure does not occur within a sentence and there are not any unfinished clauses currently active. (control status must be "if imp jump major separator status" or " imperative status").

3/. If in the machine code being produced, control can pass directly into the local procedure body (machine code control is implicit) :-

a/. If the local procedure directly follows the routine, a FINISH/CONDITION SATISFIED/RECOGNISED is compiled automatically thus correctly ending the routine.

b/. If the local procedure follows another local procedure, FINISH [NAME d] is compiled, where [NAME d] is the name of the preceding local procedure.

If 1 and 2 above are not true, a recovery is attempted by terminating the current sentence or closing the open action levels and an informative query message is printed for the user to the effect that the local procedure has been wrongly positioned.

The above rules mean that a local procedure cannot be nested, although calls can be, as illustrated in the above example.

3.1.3 Implementation details

The local procedure routine was written as a Primary (i.e. Macro type) routine, which is executed at compile-time. It can thus examine the control and level status appertaining to the code being compiled and request inline code to be planted eg a FINISH, if necessary. It performs the above checks and prints informative error messages if any of them are unsatisfied. A new global was introduced, to hold the address of the name of the current local procedure in the name dictionary. This is used when compiling a FINISH [NAME d], where [NAME d] is the name of the previous local procedure.

When implicit control can pass directly into a procedure :-

1/. If the global is not set, a FINISH/CONDITION SATISFIED/RECOGNISED must be compiled as this is the first procedure local to this routine.

2/. If the global has been set, there was a previous local procedure and the global points to its name. A FINISH [NAME d], where [NAME d] is the name of the previous local procedure, must be compiled instead.

The last thing the routine does is to update this global with the address of the name of the current local procedure.

The local procedure routine calls the [NAME /a label] : routine directly. This puts the name of the local procedure on to the local name dictionary, so that it can be accessed from the routine containing the local procedure, and performs other control checks.

The routine MASTER : ROUTINE has been modified so that each time the compilation of a routine starts, the global that holds the current local procedure name is made zero. When the end of the routine is reached, if implicit control exists : the routine MASTER : ROUTINE inspects the global and takes the same action as the LOCAL PROCEDURE routine, compiling a FINISH [NAME d], if the routine ended with an unterminated local procedure. This is illustrated in the above example.

3.1.4 Possible future developments

The local procedure implementation, as it stands, does not have formal parameters nor is there any new scope for variables created. All the local declarations in a routine are in scope to all procedures local to a routine. Mechanisms could be added to facilitate formal parameters and a new scope.

Because the name of the local procedure is placed on the local name dictionary, no distinction is made between procedure names and ordinary labels. It would be best if the user was prevented from PERFORMing an ordinary label and GOing TO a local procedure name. This would require two distinct types of label (or label and procedure name) to exist, possibly by setting a marker bit in the name entry or making them syntactically different. When a PERFORM n is encountered, the system could expect n to be a procedure and conversely, when a GO TO n is encountered, an ordinary label could be expected. If this was not in either case true, an error message could be produced at the point when the procedure name or label was declared. This idea should not be too difficult to implement, but was not included for time reasons.

GOing TO a label, external to the local procedure, could be inhibited, as this is obviously bad programming practice and could lead to errors. If formal parameters for local procedures were added, assignment to external variables could be inhibited, thus cutting down on side effects. Griffiths [14], recommends that for procedures, a declaration of non-locals which change value, could be given by the compiler as this would lead to cleaner programming and better self-documentation. The nature of these improvements, however, would necessitate major surgery to the RCC system and were not seriously considered.

3.2 The new FOR loop

3.2.1 Design criteria

Consideration was given to what a FOR loop should have by default. Three main points are suggested :-

1/. A static implementation. This means that the increment and limit are evaluated once and for all before the loop and their values are assigned to "hidden" scalars, inaccessible to the user. This is as opposed to a dynamic implementation, where the increment and limit expressions are evaluated on each iteration of the cycle-body and are directly accessible to it.

2/. The possibility of zero iterations of the cycle-body.

3/. Termination on the control variable exceeding the limit.

A static implementation is safer, as the semantics are fixed and protected. Possible sources of error and bad programming practices are eliminated. It is also more efficient, as the increment and limit expressions are evaluated only once. In some RCC applications, a user may want full control over storage allocation. The user, in this case, will not want any hidden scalars allocated automatically. Therefore, a dynamic implementation may be specified. The choice of the default, however, forces the user to think about the implications. This choice relates to Griffiths notion of static and dynamic semantics [14].

If, on entry to the FOR, the conditions for termination are satisfied immediately, the body of the FOR loop should not be executed at all. This is consistent with the semantics most high-level language programmers would expect and is logically consistent with IF conditionals.

Termination on the control variable exceeding the limit, is again what most programmers would expect. Termination on equality can be confusing and can lead to infinite loops, where the control variable "hops-over" the value of the limit and carries on, see example in Chapter 2.

3.2.2 The new syntax

The new syntax is as follows :-

```
FOR [ Q/( DYNAMIC ) r ? ] [ NAME ] [ Q/( =,FROM ) r ]
[ E initial value ] [ STEP ? ] [ Q/( TO,UNTIL ) r ]
        [ E limit ] DO :...
```

where

```
[ STEP ] = [ Q/( BY,STEP ) r ] [ E increment ]
```

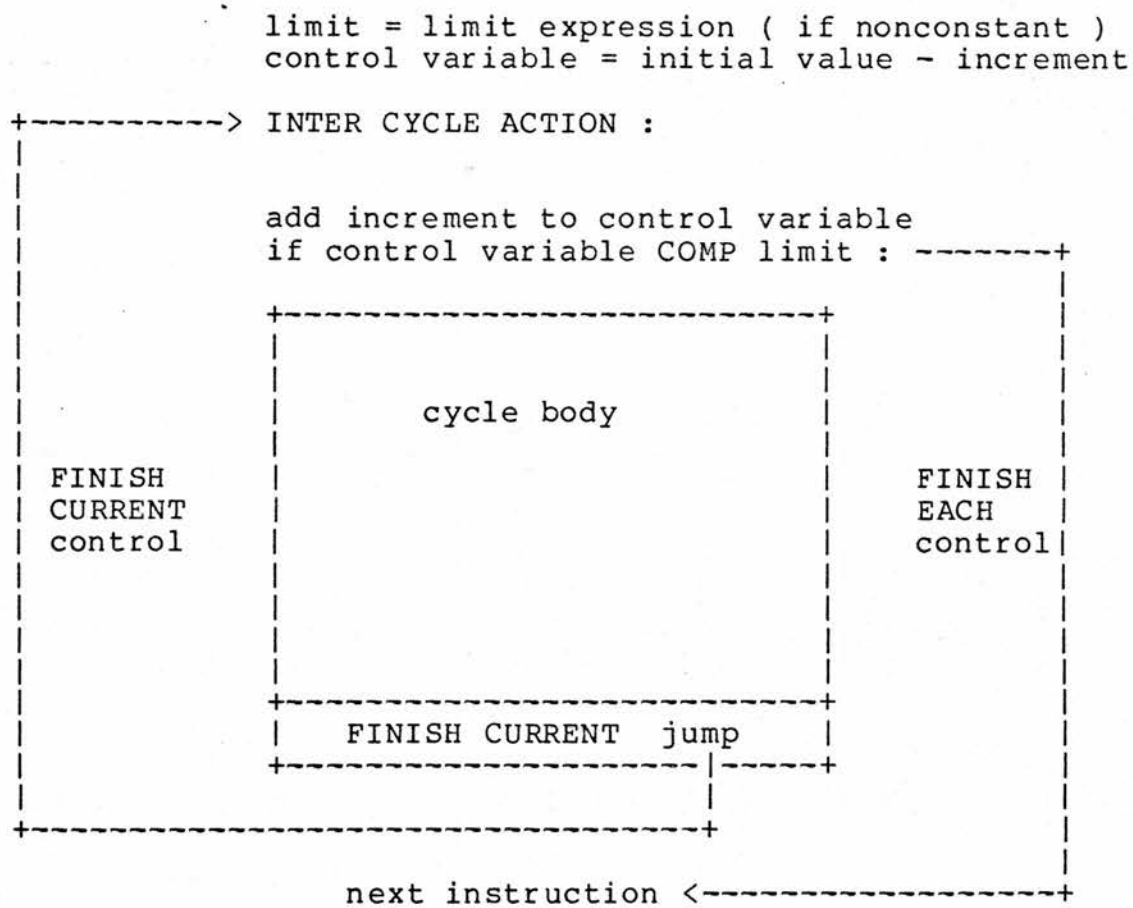
The syntax was chosen to be the same as most Algol type languages. It is also compatible with Pascal if the STEP option is omitted, when a default increment of constant +1 is used (DOWNTO, to signify an increment of constant -1, could be added very easily for full compatibility if required).

Unless DYNAMIC is specified, a static implementation is assumed by default. The limit or terminating value of the control variable, must be specified. The keyword DO appears at the end to definitely distinguish the new FOR from the existing one.

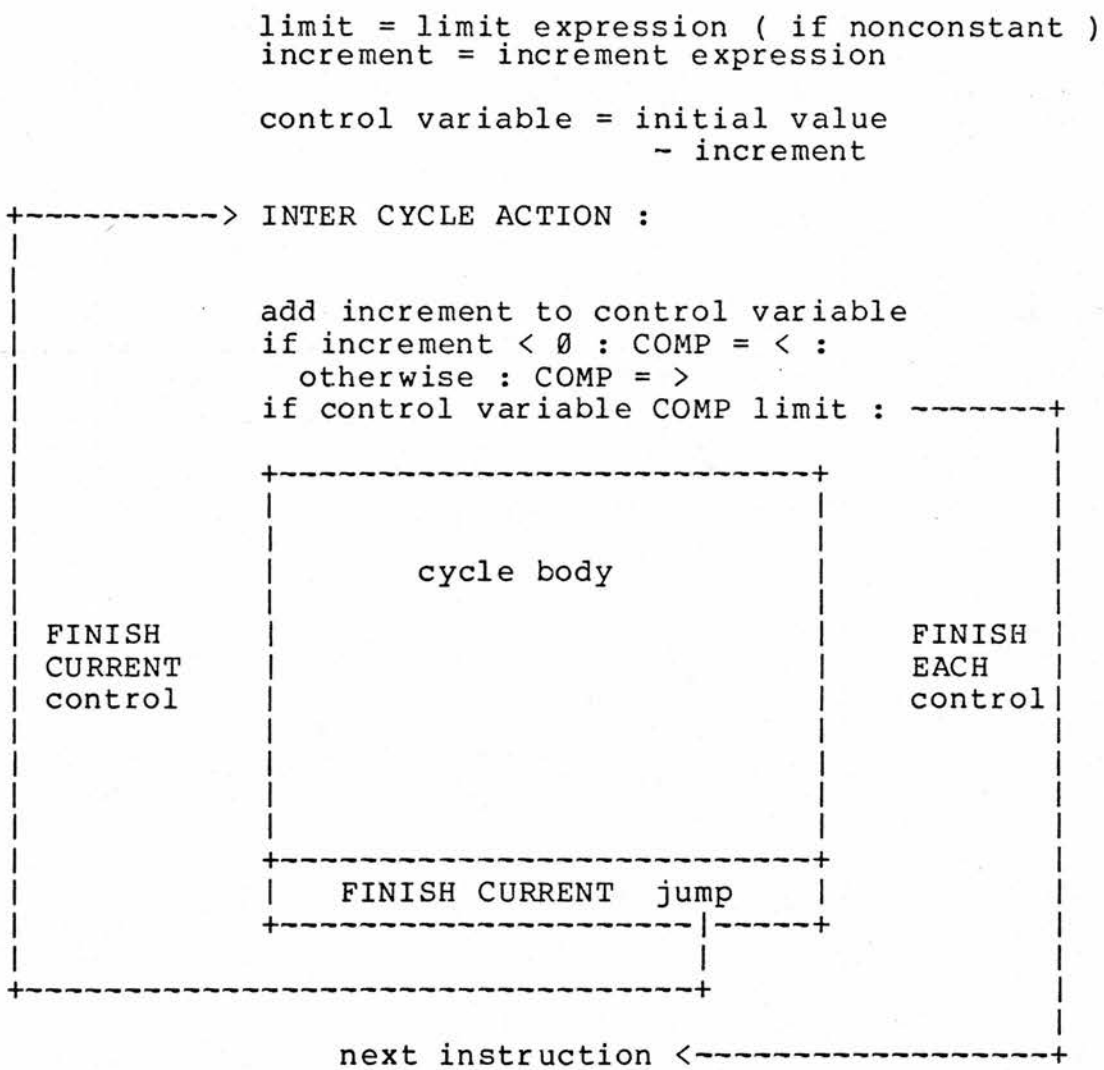
3.2.3 The Semantics of the new FOR

The semantics of the new FOR at run-time, can best be shown diagramatically. There are four diagrams, static with a constant increment, static with a nonconstant increment, dynamic with a constant increment and dynamic with a nonconstant increment.

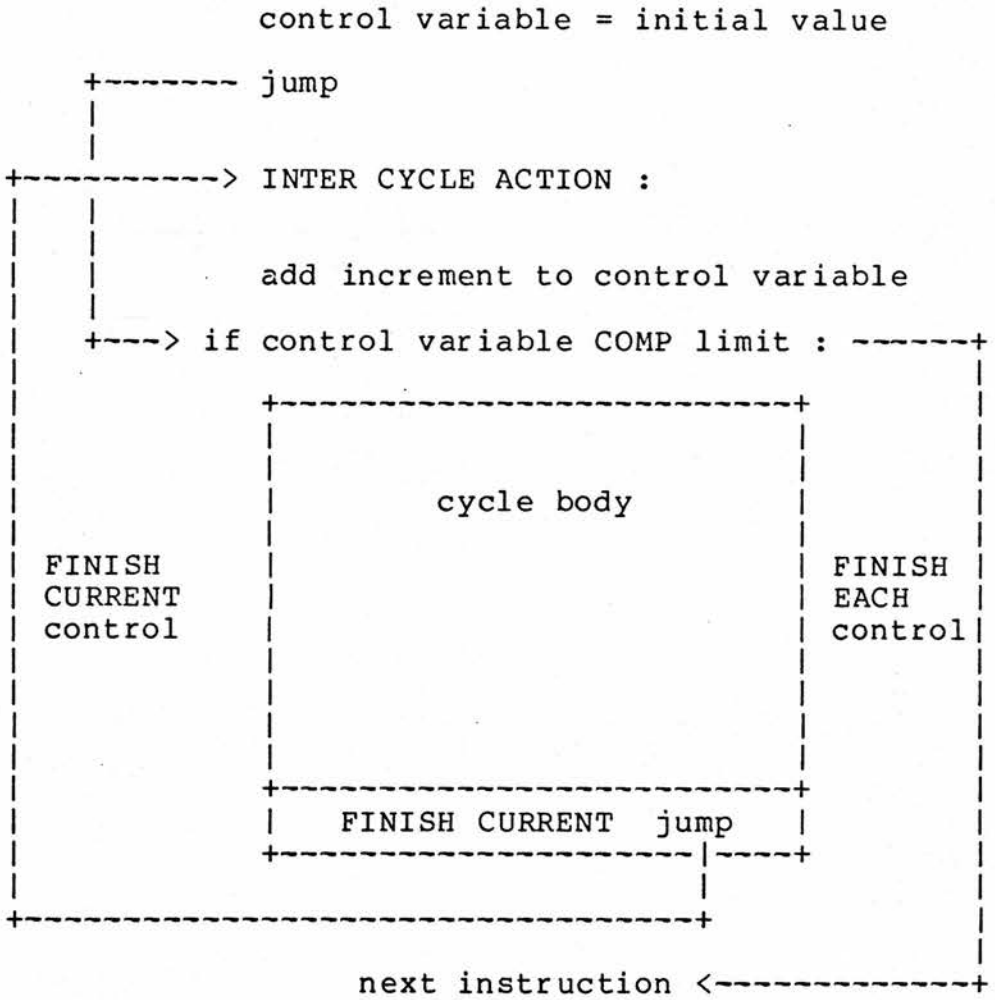
STATIC (with a constant increment)



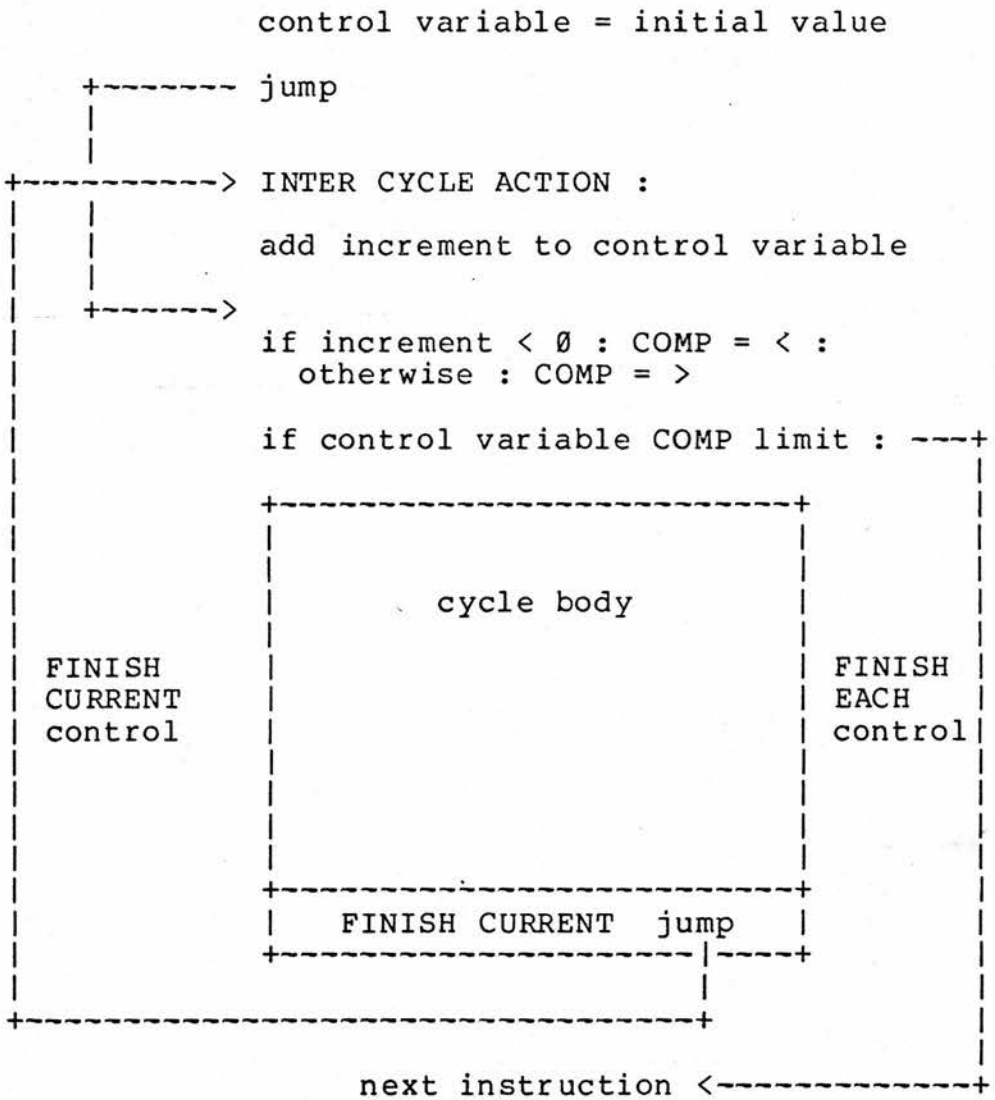
STATIC (with a nonconstant increment)



DYNAMIC (with a constant increment)



DYNAMIC (with a nonconstant increment)



The test for termination, in all cases, on a positive increment is the control variable exceeding the limit. The semantics given in the Algol 60 revised report [15], are followed for a negative increment. Thus when the increment is negative, the test for termination is not, control variable > limit, but control variable < limit. When the increment is negative and a constant, the appropriate test can be compiled, because the sign will be known at compile-time.

Static with a constant increment.

This is the simplest case. The comparator for the termination condition, COMP, is set at compile-time to >, if the increment is positive and < if it is negative.

If the limit is nonconstant, it is evaluated on entry to the FOR. A "hidden" or "anonymous" scalar is set up to hold the evaluated limit. This means that the value of the limit cannot be altered in the cycle-body.

The possibility of zero iterations of the cycle-body must be catered for. The increment is subtracted from the initial value before code is generated to assign the control variable the initial value given by the user. This allows control to pass to the increment test sequence immediately with impunity, as the first increment merely creates the proper initial value which can then be compared with the limit before the cycle-body is entered.

Static with a nonconstant increment.

The increment expression is evaluated on entry to the FOR. A "hidden" or "anonymous" scalar is set up to hold the evaluated increment. The same action is taken for the limit, if it is nonconstant.

If it is not constant, a test must be compiled to find out whether the increment evaluates to a positive or negative value at run-time and then the appropriate test of the control variable against the limit can be made. (An improvement in efficiency could be made here, where the test for the sign of the increment, could be lifted out of the iterative section of the structure and the appropriate terminating test set at run-time).

Dynamic with a constant increment.

This is in fact semantically equivalent with the static with constant increment case (except that a nonconstant limit would be evaluated on each iteration of the cycle-body), but is included for completeness.

Dynamic with a nonconstant increment.

The increment and limit expressions are evaluated on each iteration of the cycle-body. The increment may initially evaluate to a positive value, then subsequently go negative on the next iteration, or vice versa. Therefore, code has to be generated to check the sign of the increment on each iteration, and alter the terminating test accordingly.

3.2.4 Implementation details

There is a facility provided by the system to help the user build FOR loops, using the CYCLE primary mentioned above. There are three main points in the structure :-

a/. The address to which control returns on each iteration

b/. The address of the start of the cycle body

c/. The address of the instruction immediately following the cycle body

Special instructions are provided for use only in CYCLE primaries.

RECORD CYCLE [NAME]

should in general, be the first instruction obeyed in the cycle routine. It records the name being used as the cycle control variable.

INTER CYCLE ACTION :

is a system routine which acts as a label for a/. above. It is used at the appropriate point in the generated in-line code to specify the iteration or trip return address.

The other two points do not need special instructions :-

COMPILE { FINISH }

will generate code to jump to the end of the in-line macro code, which will be b/. above, the start of the cycle body.

COMPILE { FINISH EACH [NAME n] }

where [NAME n] is the cycle control variable, will generate code to jump to the instruction immediately after the cycle body, which is c/. above.

A system routine, which is also very useful is :-

```
FIND TYPE [ R t ] FOR [ Q/( NAME, VARIABLE,
        EXPRESSION ) E ] [ V pv ]
```

This is a convert-and-inspect routine, which given an [E] phrase variable, returns a coded integer specifying its complexity. This can then be interrogated using a set of system global constant identifiers. (It is typically used in the new FOR primary, to find out whether or not a given expression is simply a constant). It also converts the internal representation from syntactic (tree) form to plex form. Plex form is an intermediate representation, between the syntax and the object code, which compacts and standardises the information. As the names are going to be used more than once, it is better to convert them to plex form, as they will be handled more efficiently.

Another existing routine was used to assist in setting up the "hidden" scalars :-

```
MACRO [ Q/( LOCAL, LABEL ) r ] [ NAME R n ]
```

This routine creates a unique name, which the user cannot use, every time it is called within a particular routine. This ensures that, for the new FOR loop, "hidden" scalar names do not accidentally clash with user names, and that nested FORs are given different "hidden" scalar names. (A local procedure was written to set up the hidden scalars, as this is required to be done more than once in the routine, this illustrates a use of the new Local Procedure routine).

The following is a simplified example of the CYCLE primary to implement a static FOR loop, only, with a constant increment :-

ROUTINE

```
PRIMARY CYCLE : [ NAME cv ] = [ E init ] BY
                [ E ? increment ] TO [ E limit ] DO :...

RECORD CYCLE [ NAME cv ]
FIND TYPE t FOR EXPRESSION increment
IF t HAS NO constant data type marker :
  PRINT : INCREMENT MUST BE A CONSTANT ; FINISH
ADD ~ [ V increment ] AFTER [ E init ]
COMPILE { [ V cv ] = [ E init ] }

INTER CYCLE ACTION :

COMPILE { ADD [ E increment ] TO [ V cv ] }
IF constant for ( increment ) < Ø :
  SET [ RCC COMP c ] = < :
OTHERWISE : SET [ RCC COMP c ] = >
COMPILE { IF [ E limit ] [ RCC COMP c ] [ V cv ] :
          FINISH EACH [ NAME cv ]
        }
FINISH
```

3.3 The RCC case statement

3.3.1 Design features

The RCC case statement has two distinct types :-

1/. Choice from numeric constants, which is similar to, but more powerful than, the PL/360 or Algol W and Algol 68 case statements. This enables an efficient implementation, if the number range is small, as a table of addresses of the case options can be built up and the numeric discriminant can be used to directly index these. Hence branching straight to the appropriate code.

2/. This is semantically equivalent to the IF ... THEN ... ELSEIF ... construct. Cascades of these constructs tend to be clumsy and can require a large amount of indentation, Atkinson [16]. The larger the number of options there are, the superiority of a case syntactic format increases.

The case selectors can be any conditional expression allowable within IF statements. No discriminant is given.

There is a special case of this type, where the user can give the left-hand side of the conditional as the discriminant. In this case, the test is for equality.

Consideration was given to the type of "overall else" or default option to be incorporated. A standard approach was preferred. The decision was to have an optional default option, but with a strong recommendation that it is always put in by the programmer. The same decision seems to have been made with the Ada [17] case statement.

3.3.2 The chosen syntax

The syntax of the RCC case is as follows :-

```
SELECT [ DISCRIMINANT ? ] [ Q/( FROM CONSTANTS ) ? ]
  where
    [ DISCRIMINANT ] =  FOR [ NAME ] [ ASSIGNED ? ]
    [ ASSIGNED ] =  = [ E ]
CASE [ RCC COND selector ] :...
FINISH CASE [ NAME ? discriminant ]
ALL OTHER CASES :...
END OF SELECTION [ NAME ? discriminant ]
```

The choice of syntax reflects the need for clarity and the requirement to make logical structure explicit. The syntax is modelled on that of BCPL, BLISS and C, with an initial declaration of the discriminant expression followed by the case options, prefixed by the key word CASE. An equivalent of the BCPL ENDCASE, to dynamically terminate the case option code, is provided in the form FINISH CASE [NAME ? discriminant]. This may appear anywhere and more than once, in the case option code. If it is left out, the option is implicitly terminated by the next CASE key word (this is unlike BCPL, however, where if the explicit terminator is omitted, control passes into the code for the next case option).

The key word SWITCH, to introduce the discriminant, was thought redolent of the SWITCHed GOTO. Therefore, the key word SELECT, as in the second alternative of the BLISS case statement, was chosen. The DEFAULT prefix for the overall else option, also did not seem to suggest the semantics. Therefore, the form ALL OTHER CASES, which precisely describes the semantics, was chosen (the Ada designers have chosen the form WHEN OTHERS). An equivalent of the Algol 68 ESAC was also added for clarity, closing the initial SELECT declaration. The form END OF SELECTION was chosen.

SELECT begins the case statement. If FOR appears after a SELECT, a discriminant must follow. The discriminant, if present, must always be a simple [NAME], but it can be assigned an expression on the SELECT statement. This is presented in the usual form [NAME] = [E], where the expression [E] will be evaluated and the value placed in the [NAME]. The discriminant name will appear on any FINISH CASE [NAME ? discriminant] and the END OF SELECTION [NAME ? discriminant]. If FROM CONSTANTS appears, the implementation will be of type one, otherwise it will be of type two. FROM CONSTANTS may only appear if the [DISCRIMINANT ?] is present. ALL OTHER CASES, should appear at the end of the sequence of case options before the END OF SELECTION. This is in fact just a label, as no discriminant is given.

3.3.3 Semantics and implementation

The implementation of the RCC case illustrates the replacement of the SWITCHed GOTO and the IF ... THEN ... ELSEIF ... , by using these two more primitive constructs, at a lower level (out of sight of the user), to effect the required semantics. The implementation has thus been achieved with considerable economy.

The implementation consists of a sequence of five Primary routines, associated with the different syntactic parts of the construct. In addition there is an IF routine called by the Primaries, to make sure that these routines have been called in the correct place and in the right sequence.

The syntax of the RCC case option allows for an [RCC COND], thus giving it the power for the second classification given above. An [RCC COND] may be resolved down to an [E] (expression) or a [C] (constant), making the type one form available when efficiency is the prime consideration. The user of the RCC case statement, must explicitly state the type of case required. Once the choice is made, he/she must be consistent. If constants are specified, only constants can be used in that particular case construct. Otherwise, an informative error message is printed and the option is ignored.

The first type of case is implemented by the use of a SWITCHed GOTO. The second type by an IF ... THEN ... ELSEIF ... construct. Hence the type two statement :-

```
SELECT
  CASE a < b + 5  : a = a + 7
  CASE c = 1Ø7    : c = a
  CASE a > c      : a = c
  ALL OTHER CASES : PRINT : ERROR CONDITION
END OF SELECTION
```

Would be equivalent to the following :-

```
IF a < b + 5 : a = a + 7 :
 OTHERWISE :
  { IF c = 1Ø7 : c = a :
     OTHERWISE :
      { IF a > c : a = c :
         OTHERWISE : PRINT : ERROR CONDITION } }
```

As each case is met in the users source program, the CASE [RCC COND selector] :... routine generates :-

```
} : OTHERWISE : { IF [ RCC COND selector ] : {
```

The first CASE met after a SELECT generates only :-

```
IF [ RCC COND selector ] : {
```

And the ALL OTHER CASES :... routine generates :-

```
} : OTHERWISE : {
```

The END OF SELECTION [NAME ? discriminant] routine, generates the required number of of close actions }, to make the logical structure complete.

The user may give the left-hand side of the conditional expression as a discriminant. The CASE routine must build up the conditional statements with a test for equality between the discriminant and the selectors.

```
SELECT FOR programmer
  CASE wirth : PRINT : WIRTH
  CASE strachey : PRINT : STRACHEY
  CASE dijkstra : PRINT : DIJKSTRA
  CASE hoare : PRINT : HOARE
  CASE knuth : PRINT : KNUTH
  ALL OTHER CASES : PRINT : PROGRAMMER UNTRUSTWORTHY
END OF SELECTION
```

Would be compiled as :-

```
IF programmer = wirth : PRINT : WIRTH :
 OTHERWISE :
  { IF programmer = strachey : PRINT : STRACHEY :
     OTHERWISE :
      { IF programmer = dijkstra : PRINT : DIJKSTRA :
         OTHERWISE :
          { IF programmer = hoare : PRINT : HOARE :
             OTHERWISE :
              { IF programmer = knuth : PRINT : KNUTH :
                 OTHERWISE :
                  PRINT : PROGRAMMER UNTRUSTWORTHY } } } }
```

The FROM CONSTANTS parameter on the SELECT, is only suitable when selection is required from a limited range of contiguous (or nearly contiguous) numbers e.g. the category number of a class. Implementation is very efficient. The basic design follows that suggested by Aho and Ullman [2].

The statement :-

```
SELECT FOR n FROM CONSTANTS
  CASE 5 : b = n + f
  CASE 6 : c = n - f
  CASE 8 : d = n
  ALL OTHER CASES : PRINT : N OUT OF RANGE
END OF SELECTION n
```

Will be equivalent to the following when compiled :-

```
          code to evaluate n ( if expression )

          GO TO test
l1 :      b = n + f
          GO TO next
l2 :      c = n - f
          GO TO next
l4 :      d = n
          GO TO next
l3 :      PRINT : N OUT OF RANGE
          GO TO next
test :    IF n > 8 ;
          OR IF n < 5 : GO TO l3
          n = n - 5 + 1
          GO TO l1,l2,l3,l4 BY n
next :    ....
          ....
          ....
```

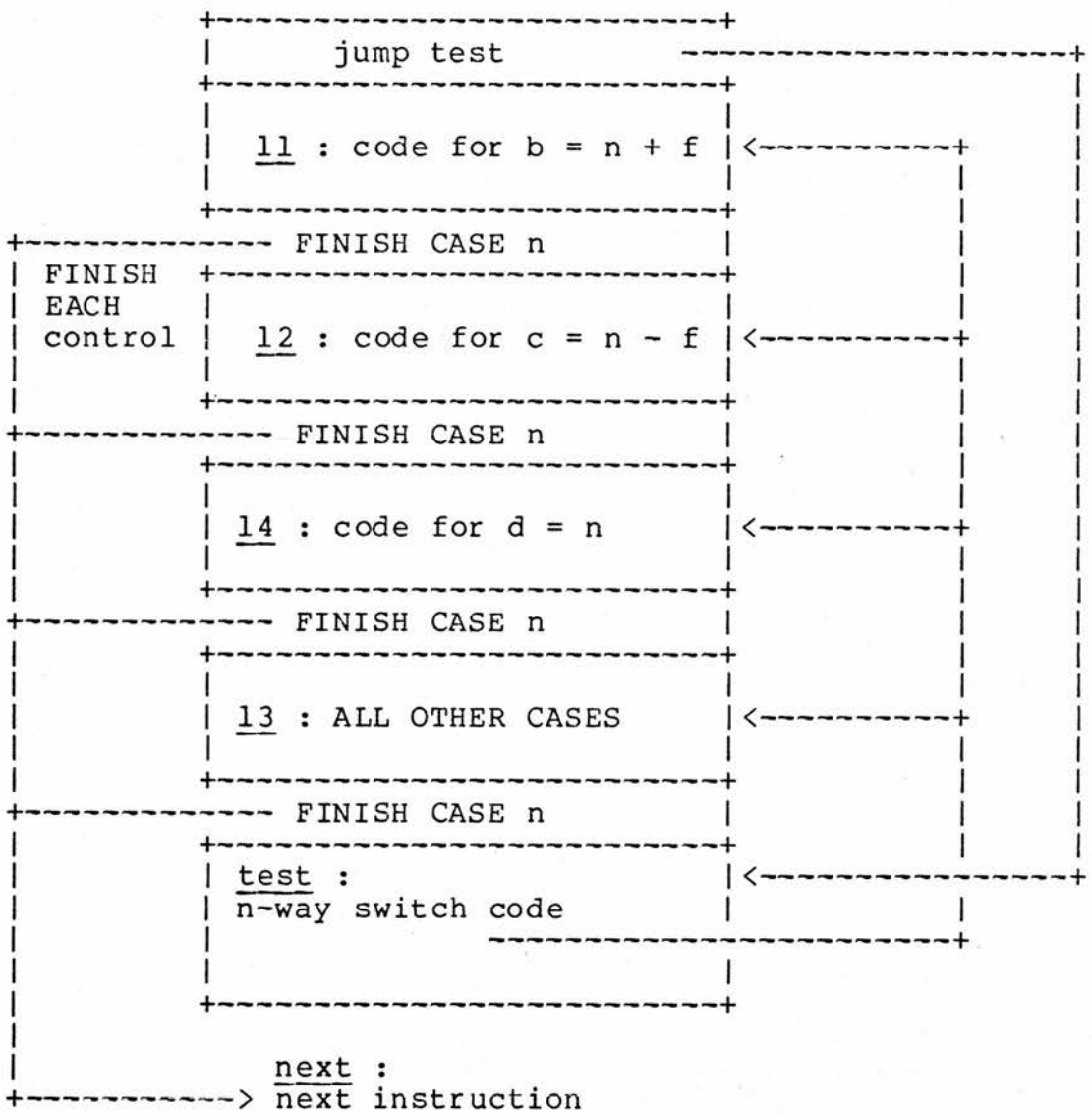
The existing RCC GO TO [NAME * label] BY [NAME n] routine is utilised to generate the n-way branch. The SELECT routine generates the labels test and next (for the next instruction after the case statement). After the discriminant has been evaluated, the jump GO TO test is generated. As each case is now encountered a new label ln is created using the MACRO LABEL routine, which generates a unique label, which the user cannot use, and puts it on the local name dictionary. The label name and the constant selector are placed on linked (chain) store. Such a pair is added for each new case option. The code for the case option is now compiled followed by a GO TO next.

When the END OF SELECTION n is encountered, thus completing the case construct, the code for the n-way branch is generated. First code is compiled to test the discriminant, to see if it lies within the range of constants given as selectors. If it does not, control is passed to the code given after the ALL OTHER CASES : label. The addresses on the chain are now placed in contiguous store in ascending order of the constant selectors. (So as to build up the form of an RCC [NAME *]). If there is a gap in the number range e.g. between 6 and 8 in the example, the label of the ALL OTHER CASES option is inserted. The discriminant minus the smallest selector plus one, is now used to index the table of labels and control is passed to the appropriate case option code.

Aho and Ullman suggest that if the range of selector constants have large gaps, it would be better to construct a hash table with the labels of the various cases as entries. But this was not thought necessary, because a type two case could be used and it is unlikely that a large number of constant selectors would be used. (In the RCC system at the moment, there are no n-way switches with as many as ten labels). Therefore selector constants should all lie in some reasonable fraction of the range selector constant (max) to selector constant (min), but as the expected most frequent use of this type of case is on class category numbers, which are small contiguous ranges, this should not present a problem.

The overall control structure for the case, is supplied by the RCC CYCLE already described. To incorporate the case statement into the existing RCC control structure, the case construct is implemented as an explicit cycle, with only one iteration of the body. The individual case options being terminated by an explicit FINISH EACH. The control structure for the above example can be expressed as follows :-

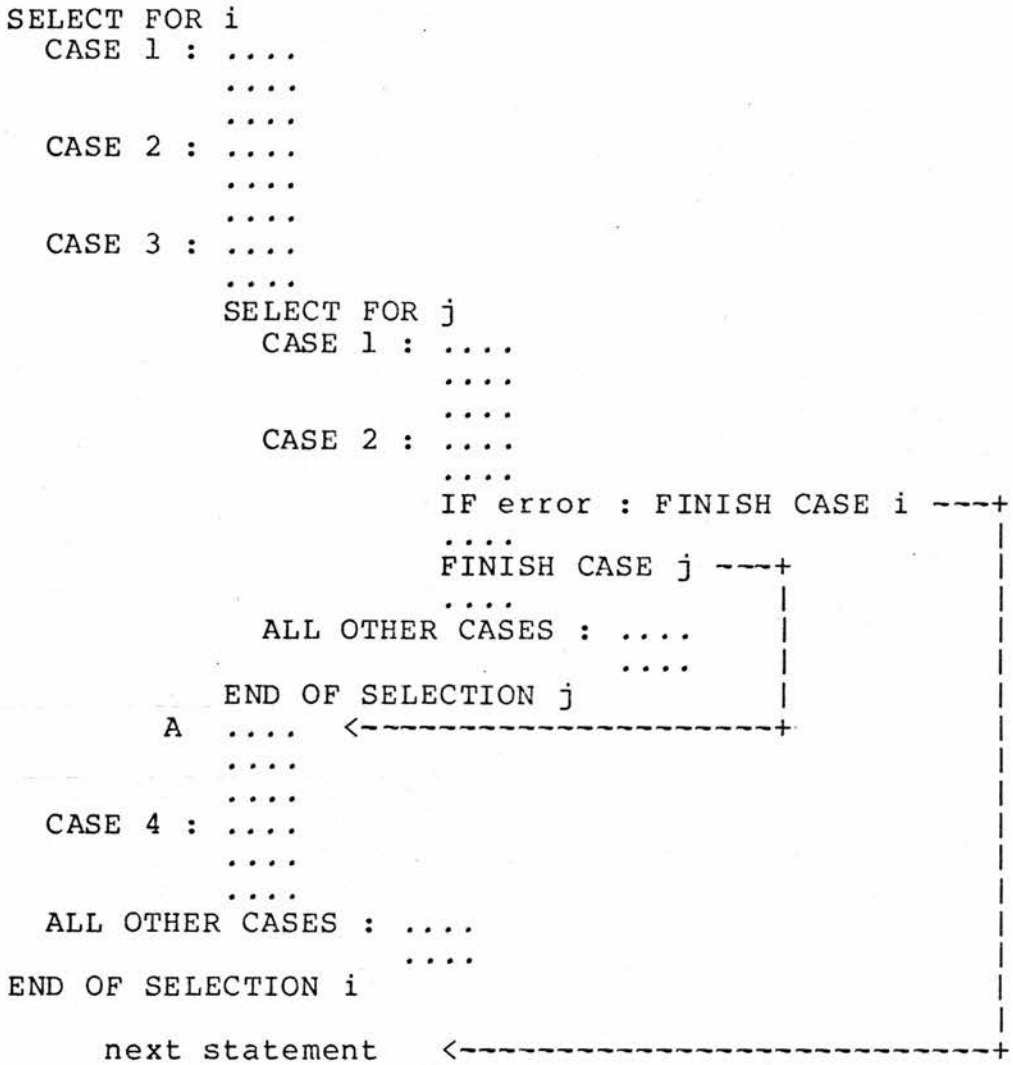
Control structure of RCC case statement



This CYCLE is hidden and is of no concern to the user. The SELECT routine compiles an explicit cycle that itself generates no code, but creates the control structure for cycles. When FINISH EACH is encountered in a cycle, as explained earlier, the cycle is terminated and control is passed to the instruction following the cycle body. This is just the effect required when a case option has finished executing , namely, control must bypass the code for the other case options and pass to the next statement after the case. FINISH CASE causes a FINISH EACH to be compiled and the END OF SELECTION routine, after generating the n-way switch, causes an END OF CYCLE to be compiled thus explicitly terminating the cycle.

When a cycle is encountered at compile-time, a new system level is created which contains the FINISH EACH pointer, the FINISH CURRENT pointer and the control variable plus other information. It is necessary for the system to tell a normal instance of a cycle from its special use embedded within the case statement. This is because, the information in the system level for a case statement is different from that in a normal CYCLE statement. The FINISH CURRENT, for example is not used in the case implementation and the pointer is redesignated to point to the chain of selector constant and label pairs. Furthermore, errors should be monitored if the user issues a FINISH EACH in a case statement, or a FINISH CASE in a cycle. A special control bit enables the system to tell whether a case cycle or a normal cycle is being set up.

This use of the cycle control structure also enables case statements to be nested. The embedded use of the cycle allows extra sophistication in nested cases. The discriminant name given on a FINISH CASE, need not refer to the inner most instance of the case statement, but an outer instance. In the example below, the issuing of a FINISH CASE i in the inner instance of a case, causes the code at point A to be omitted.



If instances of type two cases without a discriminant, are used in a sequence of nested cases (or two discriminants have the same name and an explicit FINISH CASE is issued with that name), an explicit FINISH CASE appearing on an inner instance of a case can only terminate the nearest type two case without a discriminant (or a discriminant of the same name), going outwards through the nesting.

Full details of the primaries that implement the case statement, plus the routines involved in the other enhancements to the RCC language, can be found in APPENDIX 1.

3.3.4 Possible future developments

It should be possible to extend the RCC case to enable phrase variables to be used for case selection. Routines exist to analyse phrase variables and break down their structure. The primary IF [PHRASE VARIABLE] = [MATCHING PHRASE EXPRESSION], is one of these. The [MATCHING PHRASE EXPRESSION] can be any sentential form of the associated class of the [PHRASE VARIABLE]. The conditional checks whether the phrase value is of the given form specified by the [MATCHING PHRASE EXPRESSION] and if it is, it carries out the resolution i.e. sets any phrase variables in the [MATCHING PHRASE EXPRESSION] to their appropriate subvalues ; if it is not, the condition is not satisfied and no phrase variables are set. This IF clause is itself a subclass of [RCC COND], which is the syntax for the case selectors. The options would consist of [PHRASE VARIABLE = MATCHING PHRASE EXPRESSION] and the code for the first option to that satisfied the condition, would be obeyed.

At the moment, however, this would fall foul of a system check, that makes sure that all, except certain specified primaries (like the IF clause mentioned above), are called at top level as sentences i.e. do not contain phrase variables. However, CASE [PHRASE VARIABLE] = [MATCHING PHRASE EXPRESSION], could be considered to be an addition to the instructions that are allowed to contain phrase variables. Hence extension of the type two case to act on phrase variables, can be relatively easily achieved.

Example :-

```
SELECT
  CASE [ E e ] = 1 : ....
                     ....
                     ....
  CASE [ E e ] = [ C c ] : ....
                           ....
                           ....
  CASE [ E e ] = [ V v ] : ....
                           ....
                           ....
  CASE [ E e ]  = 4 + [ V v ] - x + [ C c ] : ....
                                              ....
                                              ....
  ALL OTHER CASES : ....
                    ....
                    ....
END OF SELECTION e
```

The special case of the type two, where the user gives the left-hand side of the conditional expression, would however, be more difficult to implement and would require changing the routines that implement the case statement. This is because, no lower level phrase variable instruction exists to directly implement this semantics.

The structure of the case could be extended along the lines of Dijkstra's Guarded commands [18], as suggested by McKeeman [19]. But as this would significantly change the semantics, the resultant construct would arguably no longer be a case statement.

Wrandle-Barth [9], identifies two semantically different types of default option :-

1/. If the discriminant matches any of the selectors, perform the code for that option, else do nothing and go on to the next statement.

2/. If the discriminant does not match any of the selectors, an error condition has occurred.

The syntax IFANY and ONLY is proposed, with the former as default, to differentiate between the two. The RCC case is at the moment equivalent to Wrandle-Barth's IFANY. It could quite easily be extended to incorporate the second type of default semantics. Where IFANY or ONLY would appear as an additional parameter after the SELECT.

C H A P T E R F O U R

4 Improved run-time error diagnostics and object-code deassembly

4.1 Improved run-time error diagnostics

4.1.1 Design criteria

When a run-time error occurs in RCC at present, the user is given the following information :-

1/. All the registers, B-lines and AAA area are output in decimal and hexadecimal. This enables the user to determine the values of all the scalars at the time of the crash. The user must refer back to the compile-time map listing to ascertain the B-line or AAA address of named locations.

2/. A print of the contents of the main stack, giving the user a back-trace of routines at the time of the crash, i.e. the routine that caused the crash, the routine that called it and the one calling this one, etc, back up to top level. This is a very useful facility for debugging purposes.

3/. The contents of the input stream, at the time of the crash.

4/. The contents of the output stream.

5/. The processor status word, PSW [2Ø], which shows the address of the instruction that caused the crash and the machine level reason for the crash.

6/. A snapshot of 16 words of memory around this point. This enables a user to see the machine code context of the error.

7/. The start and finish addresses of the routine that was executing at the time of the crash. The address given in the PSW, is between these and is a very rough guide to where in the routine the erring source line is.

Whilst the above shows that RCC has comprehensive and helpful diagnostic facilities, number 7/. above is unsatisfactory, in that the user is more interested in which source instruction caused the crash. For reasons of efficiency, RCC at present, does not keep track of source information once a routine is compiled. At the moment, a user must spend some time finding out exactly where in the source program the error occurred. An inexperienced user of RCC would have some difficulty.

A user can request a source listing in RCC, by the master word LIST RCC. The source program is printed with the lines of each routine numbered separately. If, therefore, the user is given the routine serial number, and the line number within the routine plus the statement number within the line, of the statement that caused a crash : he/she can directly relate this to the source listing of the RCC program, to find out exactly where the error occurred.

In order to print the source line number or numbers at which the error occurred, (and if there was more than one statement on the line, a statement number as well), source line numbers must be related to the object code addresses that those lines have generated. A facility has been introduced to do this. In order to reduce the overhead incurred, it can be selectively applied to certain routines that are likely to cause run-time errors. Simple master word commands to switch it on and off (more than once if necessary) in the source program, are provided.

It was decided to construct, at compile-time, a table of source line numbers and the addresses at which the object code for those lines, starts. The only overhead exists, therefore, at compile-time and there is no overhead at run-time. When a run-time crash occurs, a monitor routine giving the diagnostics mentioned above, is called. This routine was basically modified to scan this table with the address at which the crash occurred, and print-out the source line number and if there is more than one, the statement number within the line.

4.1.2 The user interface and how the data is collected

Two new master words are introduced to switch on and off the source reference mechanism. ERRORLINE turns it on, and NO ERROR LINE turns it off, which is the default situation. These master words set a boolean global which is tested whenever a routine is compiled (i.e. by the routine MASTER : ROUTINE). If the master word ERROR LINE appears, every subsequent routine will be able to reference the source line causing a run-time crash, until a NO ERROR LINE is encountered. The two master words may appear any number of times in a source program.

When a routine is compiled the routine MASTER : ROUTINE is called, this basically sets up a routine item and calls the COMPILE routine to generate object code. A new routine was written,

```
PUT [ Q/( ROUTINE NUMBER, ADDRESS, LINE NUMBER,
          LINK ) ] ON SOURCE REF TABLE
```

to be called by the MASTER : ROUTINE, which collects the data necessary for referencing the source program line numbers with the object code addresses at the time of a crash. In the routine MASTER : ROUTINE there is an iterative section of code, that reads the lines of source program and calls COMPILE to generate code for them. It is basically this :-

```
{
  UPDATE INPUT/OUTPUT BUFFERS
  IF RECOGNISE [ RCCBS ] :
    COMPILE { [ RCCBS ] } ; REPEAT
}
```

The IF RECOGNISE routine causes an [RCCBS], RCC basic statement, to be read, and a line count (rcc local line number) to be incremented on each new line. The COMPILE routine, as it generates the object code, increments a pointer (current code address) to point to where the next instruction to be generated will be placed. The new routine has to be called twice in this loop.

The first time, to record the address where the object code for the next source line will begin, and again to record the line number. When the routine is called to record the line number, it checks to see whether the pointer to the object code has been incremented by the COMPILE routine. This is because some RCCBS's do not generate any code e.g. LOCAL or a comment line. If the pointer has not been incremented, the entry is not placed on the table , because, as no object code has been generated a run-time crash cannot occur at this line number. Therefore, line numbers recorded in the table will not necessarily be contiguous.

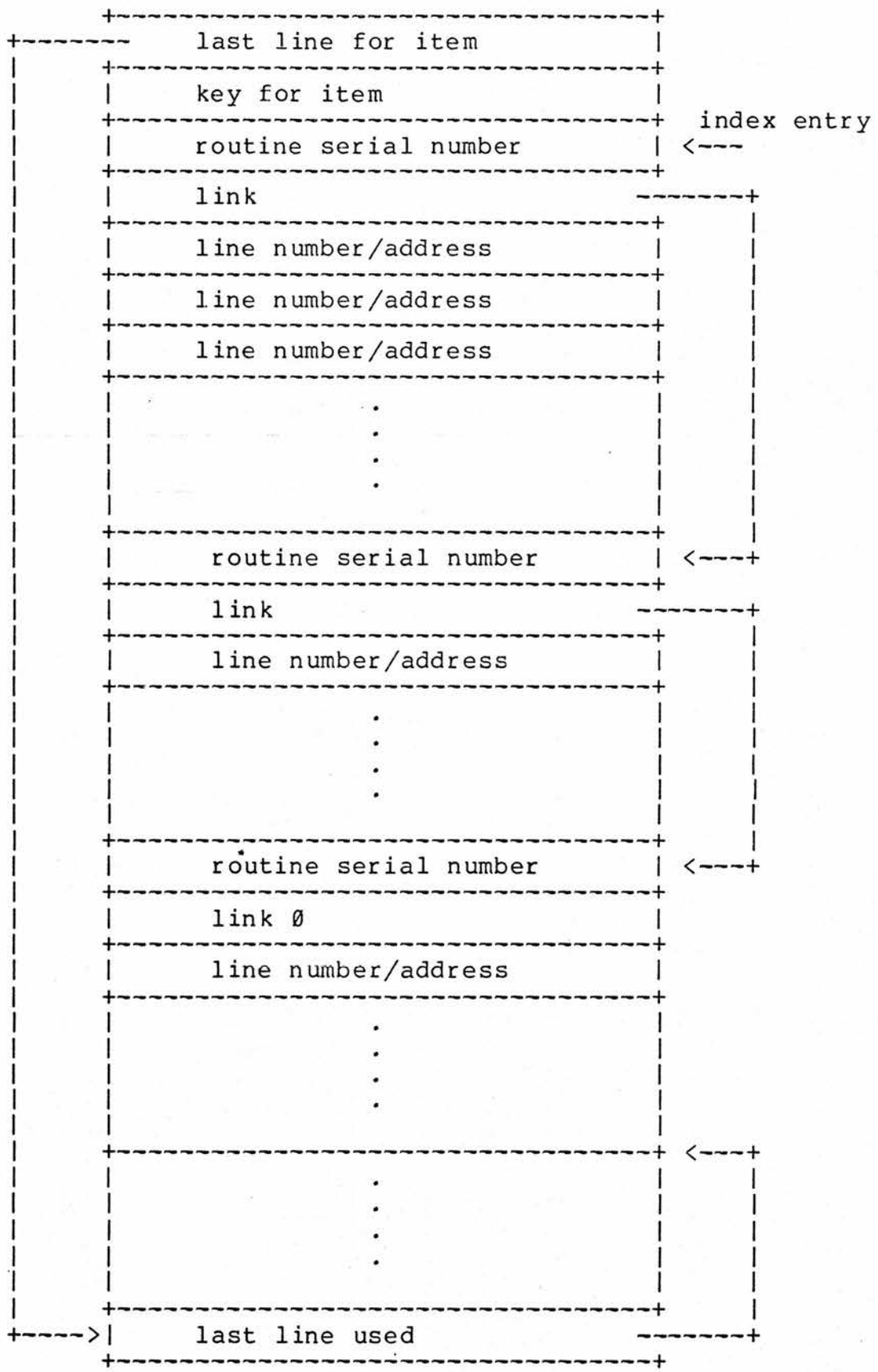
The new routine must also be called to record the serial number of the routine being compiled (a special link word is also placed on the table as will be explained in the next section). The previous example now appears :-

```
  PUT ROUTINE NUMBER ON SOURCE REF TABLE
{
  UPDATE INPUT/OUTPUT BUFFERS
  PUT ADDRESS ON SOURCE REF TABLE
  IF RECOGNISE [ RCCBS ] :
    COMPILE { [ RCCBS ] } ;
    PUT LINE NUMBER ON SOURCE REF TABLE ;
    REPEAT
}

  PUT LINK ON SOURCE REF TABLE
```

The final call to the routine also puts a last entry of the object code pointer on the table, as routine exit code is generated for each routine.

The source reference table item,
containing data for three routines



4.1.3 The structure of the reference table

There are three record types on the table, all one word long :-

1/. routine serial number

2/. link

3/. line number/relative address

As can be seen from the above diagram, when each new routine is encountered, the serial number of the routine is placed on the table, followed by a link to the next routine entry. (This link is a relative address, the last element being zero, as illustrated). This forms a linked list of routines and their line number/address entries. Then follows the line number/address pairs for each [RCCBS] in th routine. The address field is 12 bits long, as the maximum size of an RCC routine (for the 360) is 1K words. The rest of the word contains the line number.

The table is held as an expandable item. The item was created with a nominal size, so that it will take up very little space when not being used. A new global was created to point to the item. When the user switches the source referencing system on with an ERROR LINE master word, the source reference table item is expanded (by the routine MASTER : MASTER).

At the end of a run, unless the user issues a STORE CURRENT SYSTEM, the data accumulated in the table will be lost and the item will assume its nominal size once more in a subsequent run. If STORE CURRENT SYSTEM is issued, the data in the table will be saved in the system from one run to the next. This may be useful for debugging a large interacting part of the RCC system.

4.1.4 The crash-time algorithm

A run-time crash of any program running on the 360, occurs if an instruction attempts to perform an illegal operation. A program check interrupt is then caused by the operating system. RCC provides an interrupt routine, MONITOR, in the case when a program check interrupt occurs during the run-time of the RCC system. This routine has been enhanced to provide the source referencing error messages.

The IBM 360 has a controlling register called the program status word, PSW. When a crash occurs, the PSW holds the absolute address of the instruction causing the crash, plus the machine level reasons for the interrupt and other information. The serial number of the routine executing at the time of the crash, is obtained in the MONITOR routine, by testing the crash address against the address range for each routine in turn.

The source reference table is scanned as a linked list. The routine serial number previously obtained is compared with the serial number entries in the table, following the link address if there is no match.

When a match is found, the absolute crash address is converted to a relative address by subtracting the base address of the routine. The table is now searched serially, from this point on, comparing the relative crash address with the address entries on the table. An address on the table signifies where the object code for a source statement starts, the next entry in the table holds the address of the start of the code for the next source statement. Therefore, the statement causing the error has been found when the relative crash address is greater than or equal to an address on the table and less than the address held in the next entry on the table.

Some statements in RCC can take more than one line, so that the statement would have more than one line number. In this case, there would be a discontinuity in the line numbers between the start of the line address and the end of line address. If this statement caused a crash, the line number range that it occupies can be output.

Also, more than one statement can exist on one line, separated by minor terminators. In this case, more than one entry would occur in the table with the same line number. When the line numbers in subsequent entries are the same, the algorithm counts these entries and if the crash address matches a range in the table, the line and statement numbers can be given.

Combinations of these are possible e.g. if a statement that causes a crash is the third statement on a line and then continues over several more. This can be illustrated with an excerpt from a hypothetical source listing and the output the user would receive if a crash occurred :-

```
19      ....
20      ....
21      ....
22      SET NO a
23      NEWLINE; NEWLINE; STACK 0,
24      0,
25      0,
26      0,
27      0,
28      0,
29      0,
30      0 AT a
31      ....
32      ....
33      ....
```

This would cause a run-time crash, as address 0 is in protected memory belonging to the operating system. Attempting to stack data at this address is illegal. When this code is obeyed, a crash will occur and the following will be printed, assuming that the user requested the source referencing facility.

```
ERROR IN ROUTINE WITH SN 573, AT LINE NUMBERS 23,
                STATEMENT 3, TO 30
```

4.2 Object code deassembly

4.2.1 Design and implementation

The code for a routine in RCC is stored in an item. There is an existing routine PRINT ITEM [E routine item] (which is invoked by the DUMP master word), to produce a printed version of the item. It does this by printing the item in hexadecimal and decimal, regardless of whether it is object code or data.

If say, an experienced user wishes to inspect the object code produced by a complex macro for optimisation, the hexadecimal must be deassembled by hand. This is complicated by the fact that the machine code format of an instruction is different from the format used in assembler programs e.g. the various fields are in different positions. Also, the output in decimal is redundant in the portion of routine items devoted to object code. There is, however, a marker bit in the key for the item, which denotes that the item is a routine item. The object code and data areas of a routine item are separated by a zeroised word. The PRINT ITEM routine has been enhanced to automatically deassemble the object code part of routine items and then print the data area in hexadecimal and decimal. The printed format of data items is unchanged.

Instruction formats on the 360 are multiples of 16 bit half-words : There are 16, 32 and 48 bit instructions. Therefore, the item must be read a half-word at a time, building up the instructions that are more than 16 bits long. The register, base and displacement fields are rearranged to look like assembler format and the correct mnemonic codes are inserted in the op-code position. As RCC uses only a small subset of the available instruction set, only a small amount of space has to be devoted to storing the mnemonic op-codes to be output. (This has been implemented with the new case statement, there is a case option for each op-code). If an unusual instruction is encountered (that is not on the list of case options), the hexadecimal op-code is printed (by the ALL OTHER CASES option). The opportunity has also been taken to improve the general print format of items. The full details of the new PRINT ITEM routine can be found in Appendix 2, which also illustrates a use of the new case statement.

C H A P T E R F I V E

5 Converting the internal representation of formal classes from full-word to half-word format

5.1 Design criteria

As mentioned in Chapter 1, the motivation for this work is a 50% reduction in the space occupied by the internal representation of formal classes. This has been done by storing the necessary data in 16 bit half-words.

The RCC system is entirely word orientated, it assumes a model of memory with an indefinite number of locations of type INDEX, addressed serially. INDEX is defined as a binary number of n bits, used either as a store address or a fixed-point constant under modulo 2**n arithmetic (n = 24, for the ICL 1906A and 32, for the IBM 360). The constant is scaled, so that unit 1 counts one store (word) address. On the 360, which is byte addressable, an INDEX location is a full word or 4 bytes. Therefore, the binary representation of an INDEX constant is left shifted by 2 bits. e.g. the integer 5 is represented by the following bit pattern, which is "true" hexadecimal 20, because 5 words = 20 bytes.

00000000 00000000 00000000 00010100

This is what is required for indexing full-words in memory, as adding constant 1 will increment an address to the next full-word boundary. Because it is now necessary to address half-words, a true hexadecimal constant of 2 (rather than 4), is required in order to increment an address to the next half-word boundary. Also required, is the ability to load a half-word from memory, and to store a half-word to memory.

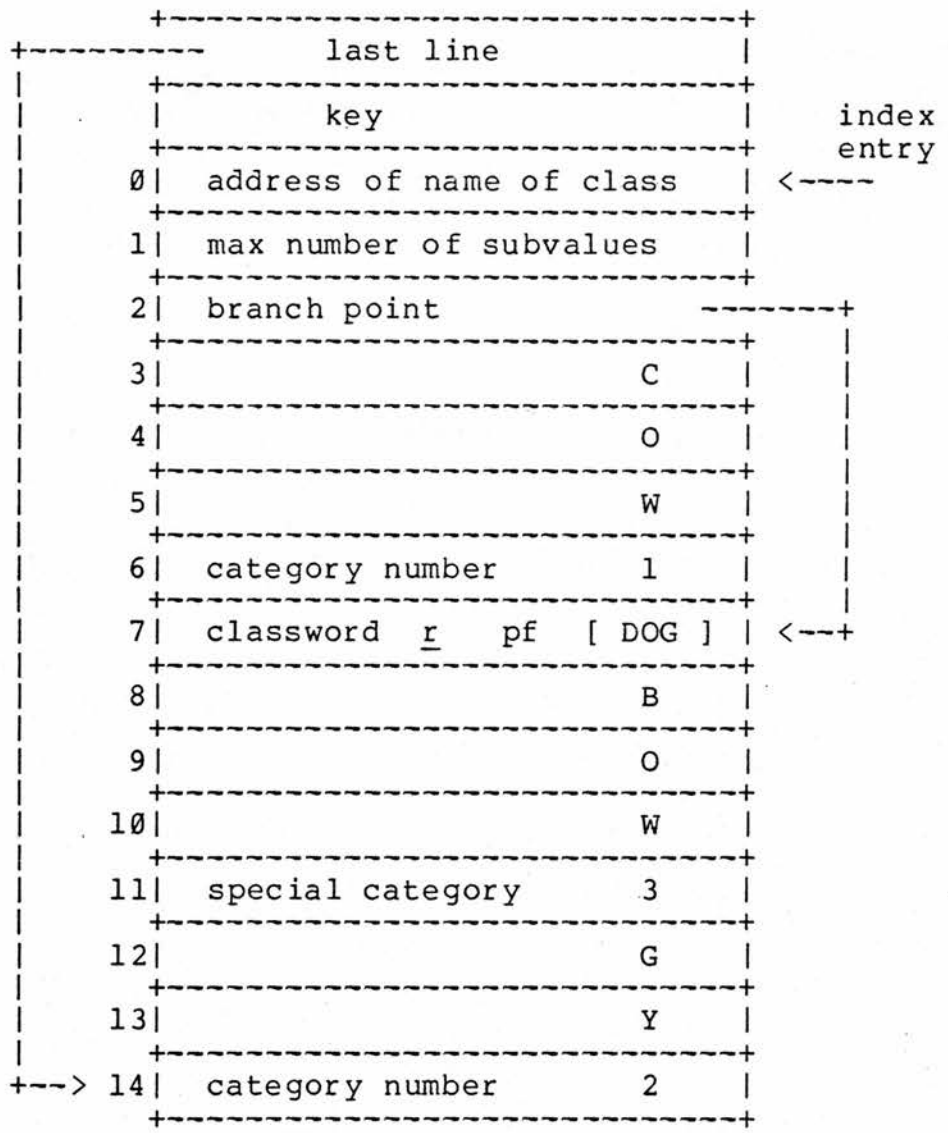
For these reasons, a special sublanguage to manipulate half-words had to be developed. The routines which create and use formal class items, have been modified to use these new commands. The format of the four types of data words, in the dictionary form of class items, and various marker bits have had to be altered because of the reduction to 16 bits. These changes have resulted in some minor complication of the algorithms associated with formal classes, but due to the space savings achieved, this was deemed worthwhile.

5.2 The structure of a full-word class item

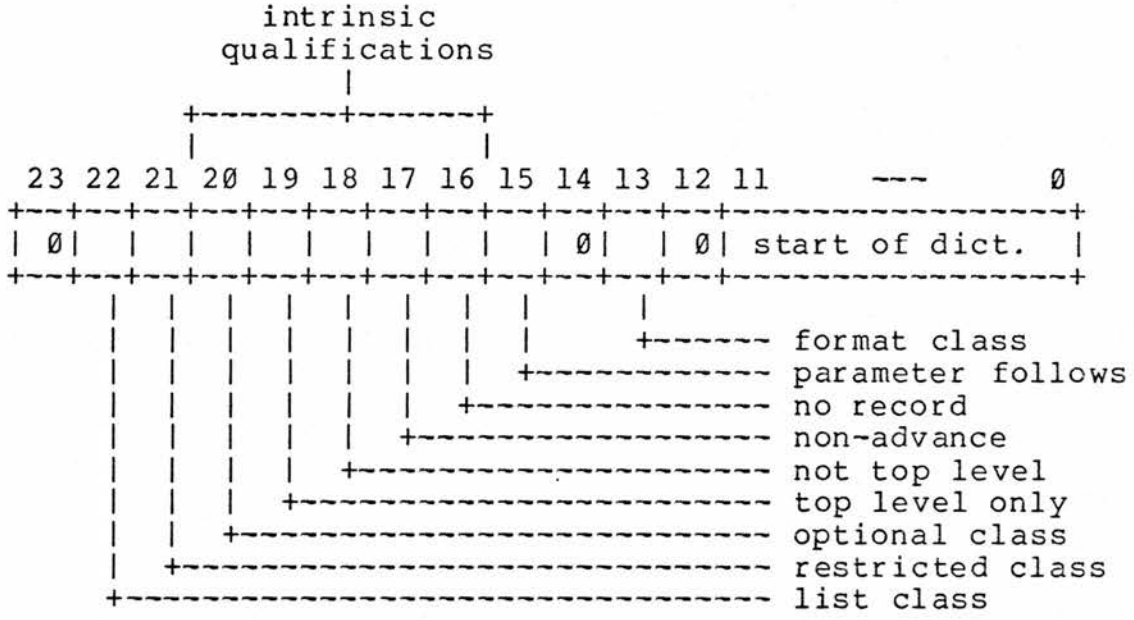
Associated with a class is an RCC item with a serial number. The item stores the information associated with the class. The following diagram shows the class item that would be set up for the class :-

[CLASS] = COW, [DOG/(BOW) r] GY

This class has two alternatives, the second contains another class, DOG, which has a no record qualification, r, and has a parameter, BOW.



The key for the item, holds information to distinguish this as a class item plus the intrinsic qualifications, which apply to all subsequent uses of the class in other definitions. The key for the class item is as follows :-



A bit set denotes the appropriate qualification. The start of dictionary field gives the address (relative to the base of the item) of the first alternative of the class. This is because any intrinsic parameters are stored after the fixed information words (line 2 above). As the above class does not have any intrinsic parameters, the start of dictionary field will be set to 2.

The next entry, line number 0 above, holds the address of the class name on the class name dictionary. Line number 1, holds the maximum number of subvalues that can occur for any alternative of the class. Then follows the stored version of the class definition, lines 2 to 14.

There are four types of data word in a class item :-

1/. A terminal RCC ISO symbol e.g. lines 3, 4, and 5.

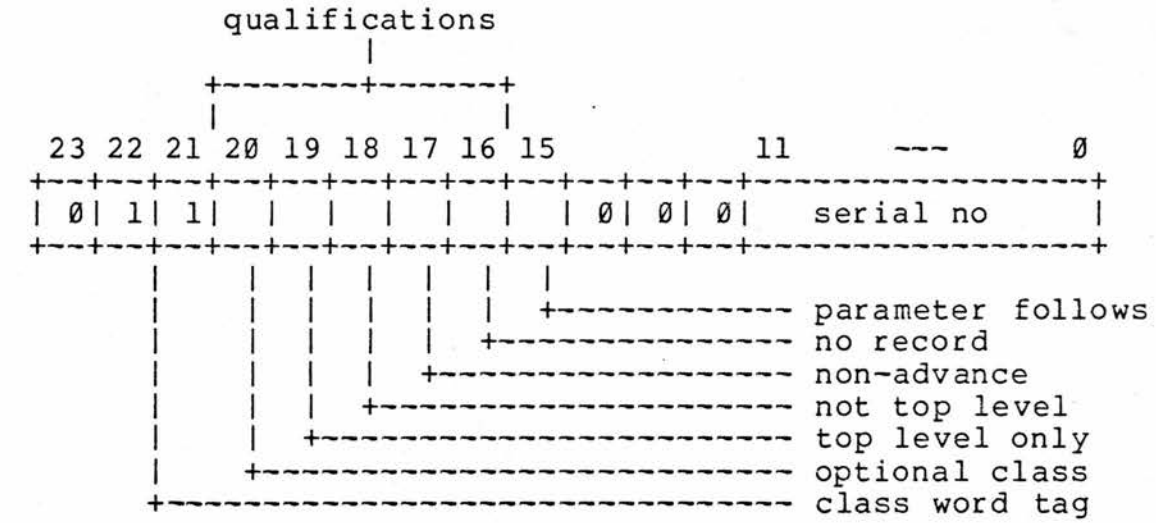
2/. A branch point. This gives the self relative address of an alternative within the class item e.g. line 2.

3/. A category number. This is a number which terminates an alternative and gives which category (alternative) number it is e.g. lines 6 and 14.

4/. A class word. This represents a class in the alternative. This comprises, the serial number of the class plus its qualifications e.g. line 7.

If a classword has a parameter, like (BOW) above, the parameter follows marker is set and the parameter follows the classword as symbols or classwords (without parameters themselves) e.g. lines 8, 9 and 10, finishing in a special category word e.g. line 11.

The break-down of the bits in the full-word class word is as follows :-



A bit set denotes the appropriate qualification.

In the full-word implementation, the four types are distinguished by two tag bits : bits 21 and 22 of the full-word.

22	21	
0	0	symbol
1	1	class word
0	1	branch point
1	0	category number

To save time on analysis and storage in the item, alternatives with the same stem are merged together. A branch is inserted where two alternatives differ and the rest of the second alternative is placed at the end. This represents only a partial merging, as only the left-hand side of alternatives may be merged.

As mentioned above, if the class has an intrinsic parameter, this is stored after the fixed information words and the start of dictionary in the key, points to the first alternative. The parameter follows marker is also set. If the class is a list class, the separator alternative follows at this point terminated by a special category word, and the list class marker is set.

The stored form of format classes is the same as for simple classes, except that the format class marker is set and the item is expandable. This is because a format class will be added to every time a new format is read.

A restricted class is again the same as a simple class, except that a type of restriction word (e.g. lower case only, uppercase underlined etc), upper and lower limit of restriction words and then the hash table, follow the fixed information words.

5.3 The half-word sublanguage

For the reasons already mentioned, the normal use of an integer constant to index word addresses cannot be used on half-words. A new primary was written :-

[Q/(INCREMENT, DECREMENT) d] [V ptr]

To increment or decrement a given pointer on to the next half-word boundary. It does this by compiling an addition or subtraction of true hexadecimal 2.

Another primary is introduced for manipulating the half-word data :-

[EXT OR ASS] [E ptr] [Q/(AND INCREMENT) ? r]
[Q/(BEFORE, AFTER) q]

where [EXT OR ASS] = EXTRACT [V] FROM,
ASSIGN [E] TO

This is used for the loading and storing of half-words from and to memory. It also allows a half-word increment or decrement of the given pointer, either :-

1/. not at all

2/. before the load or store

3/. after the load or store

Examples :-

EXTRACT hwd FROM hwptr

This will load the half-word addressed by hwptr into hwd. hwptr is left unchanged.

ASSIGN a + b TO hwptr AND INCREMENT AFTER

The value of a + b will be stored at the half-word addressed by hwptr, which will then be incremented to point to the next half-word.

Another routine is provided to handle branch points within half-word class items :-

TAKE BRANCH [V ptr]

This loads the branch point, which is a half-word relative address, pointed to by the ptr parameter. It then adds the relative address to ptr. Now the parameter given as ptr will point to the destination of this branch point.

The IBM 360 instruction set has several instructions which specifically deal with half-words [21]. Two of these, load half-word and store half-word, are generated by the new routines. This enables the new routines to be implemented efficiently. See Appendix 3 for details of these routines.

5.4 The changes to the data words

Most of the globals associated with the routines that handle formal classes, have needed to be altered. This is due to the change in position of the various marker bits, because of the compaction into 16 bits of class words, category numbers and branch points, and the reduction in size by 50%, of the value of the various offsets used in traversing class items.

The item key remains the same, to preserve compatibility with other items. The address of the name of the class, on the class name dictionary, can easily be contained in a half-word, as can the maximum number of subvalues. The four types of data word that make up the internal representation of formal classes are :-

1/. terminal symbols

2/. class words

3/. branch points

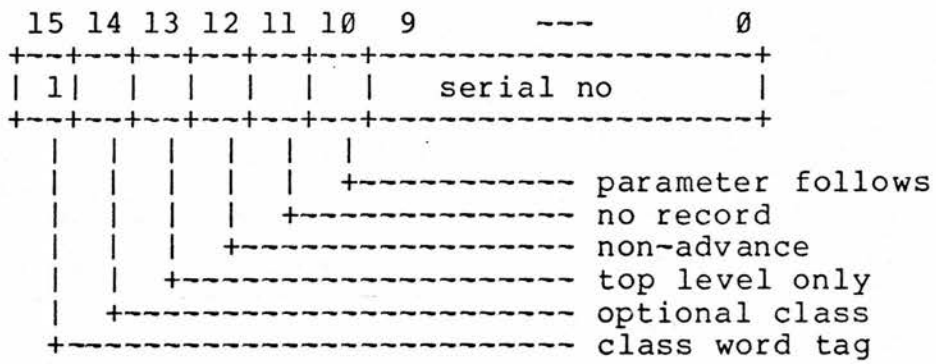
4/. category numbers

There is no problem with converting symbols as these only occupy 8 out of the 16 bits. Branch points and category numbers, also, can be sufficiently represented by 16 bits. Class words, however, require modification in order to fit them into 16 bits. The serial number field was reduced to 10 bits (0 - 9), which is in fact sufficient. The class word tag, was reduced to one bit (bit 15) set to a 1.

Because of this forced reduction in size of the class word tag to one bit, the top three bits of each half-word are used to distinguish the other three types of data word, as in these words those bits are unused. The tag bits, therefore, in the half-word implementation are :-

15	14	13	
0	0	0	symbol
1			class word
0	0	1	branch point
0	1	0	category number

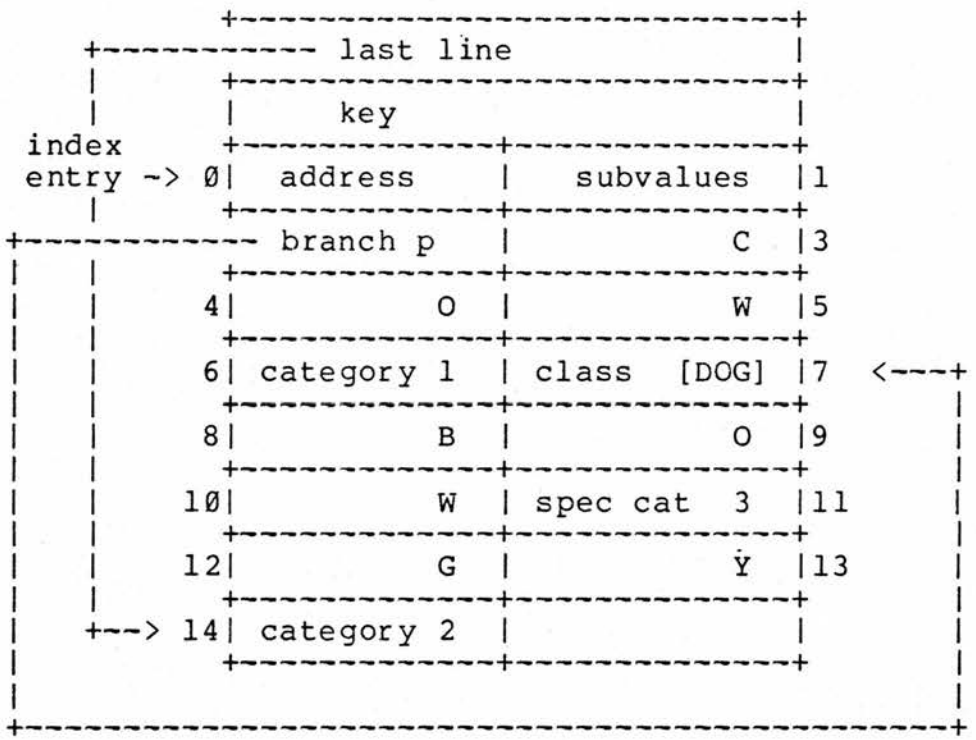
This reduced the size of the class word to 17 bits. It was therefore necessary to sacrifice one of the qualification bits. The not top level and top level only markers are used by the system, but very rarely by users. As it was mandatory to map the class word onto 16 bits, it was found that not top level could most easily be sacrificed. For the details of the use of these system markers, see Napper [25]. Therefore, the format of the new half-word class-word is :-



5.5 The structure of a half-word class item

The following diagram shows the half-word version of the class item for the same class that was used to illustrate the full-word class item.

[CLASS] = COW, [DOG /(BOW) r] GY



This diagram illustrates a typical problem, in that the last half-word is unused. As the new half-word items must be compatible with other (full-word) items in the RCC system, this unused half-word is left blank so that the item will always end on a full-word boundary.

5.6 The routines that required modification

The routines that needed altering most fall into two groups :-

a/. Those that set up the formal class items from the class definitions.

b/. The routines that refer to them

These routines were extensively modified to use the new half-word sublanguage.

a/. The routines that set up formal class items are (see Appendix 4 for details of syntax) :-

SET CLASS ITEM FROM [CLASS DEFINITION E cd]

SET EMPTY ENTRY ENTRY FOR [Q/(FORMAT, LIST, ROUTINE) ? E flr] [CLASSWORD E cw] [P6 ? E r] (SN [R serial no])

CONVERT [CLASSWORD E cw] TO DICTIONARY FORM

MERGE PHRASE AT [E start of phrase] TO ITEM [E serial no]

SET CLASS ITEM is directly called in the CLASS master routine to create an item for a new class that has been declared by the user. This routine calls the lower level routines in turn. These create an empty item for the new class, then repeatedly build up and merge in its alternatives. The main changes involved building up symbol strings at main stack front in half-words instead of full-words, using the new half-word commands. Great care has to be taken, generally, in all modified routines, not to leave the main stack front on a half-word boundary after having used it for half-word operations. This is because, if a full-word instruction is used to access main stack front in this state, a run-time crash will occur.

SET EMPTY ENTRY FOR creates the empty class item for the different types of classes. After setting up the key for the new class item, which is unchanged, the routine creates the first 2 or 5 (for restricted classes) information lines. These are now in half-word format. The address of the class name in the class dictionary and the maximum number of subvalues (and for restricted classes, the restriction type and the upper and lower limit of restriction), are all successive half-words. The hash table, for restricted classes, is also converted to be held in half-words.

CONVERT is used to convert a classword from syntactic form into the form in which it occurs in a class item. It was modified to convert the classword into the half-word format given above.

MERGE merges an alternative of a class into the class item dictionary. When dealing with restricted items, the calculation of the hash address is different (the standard hash field must be divided by 2), because the hash table and the symbols are all now half-words.

b/. The routines that use formal classes also had to be altered :-

IF RECOGNISE [R9 * phrase element list]

IF PHRASE [E dictionary line] RECOGNISED (STORE AT [E subvalue position])

REGENERATE CLASS [E d1] FROM [V syntax ptr]

([R9] is defined in Appendix 6)

IF RECOGNISE is a primary called to recognise an instance of a phrase. It builds up the phrase at main stack front as a class word alternative and has been modified to do this in half-words. The routine plants a call to IF PHRASE, which does the syntax analysis at run-time. IF PHRASE, again, was modified to use the half-word sublanguage in traversing the half-word class items created by the routines previously mentioned. As in the MERGE routine, the calculation of hash addresses for restricted classes has been modified.

The regeneration package [22] basically allows the value of a phrase variable to be printed. The analysis record for a phrase variable consists of only category numbers and pointers; it does not contain the terminal symbols comprising the phrase value. To regenerate the value, therefore, it is necessary to scan the analysis record in conjunction with an examination of the associated class items (which contain the terminal symbols). So the regeneration routine that traverses formal class items, REGENERATE CLASS, also had to be modified for use with the new half-word class items. This routine is initially called by the PRINT routine, when a user has requested a phrase variable to be printed. The PRINT routine also had to be modified to place, at main stack front, the class word and parameters (if any) plus the category number, in half-word format, of the phrase to be regenerated.

Another routine affected, was the class routine [V2]. This is used, as part of the input machinery, to recognise the converted form of a class word in the input-stream. The class word and any parameters that may follow it, are now stored in half-word format at main stack front. Again, the routine must ensure that this is not left on a half-word boundary.

5.7 Conclusion

What remains to be done, is to fully convert the RCC system to using half-word class items. (The half-word class processing machinery was tested by altering the names of the routines and globals, to distinguish them from their full-word equivalents). As RCC is written in itself, all the system syntax classes will need altering. This will require the RCC system to be rebootstrapped using half-word formal class items.

The half-word class routines have been thoroughly tested and on test no appreciable loss of efficiency in processing class items has been observed. As the RCC system uses a large memory allocation, the memory savings achieved are valuable.

C H A P T E R S I X

6 Conclusion

6.1 A description of this work in terms of extensibility

Extensibility in RCC can be classified by relating the level of extension to the amount of interaction there is between a new extension and the basic RCC system, plus any extensions already made. The classifications also specify the implementational details the user needs to know in order to make that type of extension. These classifications are Formal Syntactic Extension, Compiler Extension and Compiler Modification. These roughly correspond to the classifications of "paraphrase", "orthophrase" and "metaphrase" made by Standish [23].

1/. Formal Syntactic Extensions are extensions which can be made by a user with no knowledge of the implementation of RCC. In the case of primary routines, they rely on COMPILE { [RCCBS *] } to generate the required in-line code and organise control. They tend to use existing, lower level, constructs to build up more sophisticated ones.

2/. Compiler Extensions are written as additions to the RCC system routines. They use system data structures and routines, e.g. control levels and system dictionaries and the routines that process them. Formal Syntactic Extension is often used in conjunction with Compiler Extension.

3/. Compiler Modification involves alteration of existing routines of the RCC system. To do this requires access to the source of the routines of the implementation and their full documentation.

This work can be expressed in terms of this classification scheme as follows. Of the extensions to the RCC language, given in Chapter 3, the new FOR loop can be seen as Formal Syntactic Extension, using the system facility for implementing new RCC CYCLEs.

The work contained in Chapters 4 and 5, can be seen as Compiler Modification as in both cases, existing routines are altered to introduce new facilities. The new half-word sublanguage, described in Chapter 5, is an example of Compiler Extension, as new (to the RCC system) machine level instructions are generated by the new routines that implement it.

The local procedure and case statement implementations, however, incorporate all three types of extension. The LOCAL PROCEDURE routine can be seen as Formal Syntactic Extension, in that it directly calls a lower level routine, i.e. the routine to process a label. But it is also Compiler Extension, as it refers to the system control level structure. The implementation also involved Compiler Modification to the MASTER : ROUTINE routine.

The RCC case statement can also be seen as Formal Syntactic Extension, because its implementation is based on lower level constructs (the switched GOTO and the conditional). It also uses the CYCLE control level to create the required semantics, which is Compiler Extension. Existing routines also had to be modified, i.e. FINISH EACH which must now check that the cycle it terminates is not a case type cycle. This is Compiler Modification.

6.2 The size and efficiency of the new routines

The LOCAL PROCEDURE has a routine item length of about 200 words (program and data). The modifications to MASTER : ROUTINE resulted in a slight increase in size of that routine. The overhead at compile-time is negligible and the object code produced will possibly be more efficient due to better error detection.

The new FOR primary is about one third larger than the existing FOR (with a routine item length of about 350 words). The object code produced, by the default option of the new FOR loop, will be more efficient than the existing one, due to the static implementation, (although, of course, it will use two extra words of storage).

The total size of all the items for the routines that implement the RCC case statement is about 1K words. The compile-time overhead is very slight. If the user chooses wisely between the type one and type two case, the object code produced will be as efficient as if he/she had used the lower level instructions (GOTO ... BY ... ; IF ... : OTHERWISE : { ...).

The modifications to the MONITOR routine (to facilitate printing the source statement that caused a run-time crash), have increased the size of its routine item by some 400 words to about 1.4K words. The necessary modifications to MASTER : MASTER (to extend the table item) and MASTER : ROUTINE (to collect the data), have resulted in a slight increase in size of these routines. MASTER : ROUTINE is now very close to its maximum allowable size. The new routine PUT called by MASTER : ROUTINE has an item length of about 100 words. The overhead at compile-time, to build up the table with the line number/address pairs, is again very slight. The only overhead at run-time is the space in memory taken up by the table. This will depend on the number and size of the routines, for which this facility has been requested (by the new MASTER words). The time taken to scan the table at crash-time is again negligible.

The size of the routines that create and use the new half-word class items have marginally increased, and there has been some minor complication of the algorithms concerned. The total space used by the routines that implement the new half-word sublanguage is about 270 words. There has not been any appreciable loss of efficiency, however, and the considerable savings in the space occupied by formal class items is of obvious benefit.

6.3 Closing remarks

The overall aims of this project, it is suggested, have been achieved. The RCC language has been enhanced to provide more up-to-date constructs. The debugging facilities have been enhanced to provide the printing of the erring statement in the case of a run-time crash and the deassembly of routine items. The implementation of the internal representation of formal syntactic classes has been changed from 32 bit full-word to 16 bit half-word format, saving 50% of the data space previously needed. All of these enhancements to the RCC system have been implemented and tested successfully.

This work could be continued along the lines already suggested, more particularly :-

1/. The local procedure could be given formal parameters and a syntactic distinction made between labels and procedure names.

2/. The code produced by the new FOR loop could be made more efficient in some cases.

3/. To enable the RCC case statement to use phrase variables for case selection, and incorporate two types of "default" option.

-----oOo-----

Thence issuing we again beheld the stars.
- Dante

REFERENCES

1/. Napper, R.B.E. and Fisher, R.N. RCC - a user extensible systems implementation language Comptr. J. to be published

2/. Barron, D.W. An introduction to the study of programming languages
Cambridge Computer Science Texts, CUP, 1977

3/. Aho, A.V. and Ullman, J.D. Principles of compiler design Addison-Wesley, 1978

4/. Brown, P.J. Writing software in ALGOL
Software - Practice and Experience, Vol. 4, 1974

5/. Brainerd, W. (ed) FORTRAN 77
CACM, October 1978

6/. Wirth, N. and Hoare, C.A.R. A contribution to the development of ALGOL
CACM, June 1966

7/. Hoare, C.A.R. Notes on data structuring - in - Structured programming
Academic press, 1972

8/. Tsichritzis, D. Reliability - Design and construction of reliable software - in - Software Engineering, Springer Verlag, 1975

9/. Wrandle Barth, C. Notes on the CASE statement
Software - Practice and Experience, Vol. 4, 1974

10/. Morrison, R. S-ALGOL Reference manual
St. Andrews University Computer Science dept., CS/79/1

11/. Horning, J.J. What the compiler should tell the user - in - Compiler Construction, Springer Verlag, 1976

12/. Pool, P.C. Debugging and testing - in -
Software Engineering, Springer Verlag, 1975

13/. Fisher, R.N. RCC reference manual
St. Andrews University Computer Science dept., CS/78/2

14/. Griffiths, M. Relationship between definition and implementation of a language - in - Software Engineering, Springer Verlag, 1975

15/. Naur, P. (ed) Revised report on the algorithmic language ALGOL 60
CACM, Vol. 6, 1963

16/. Atkinson, L.V. Should if ... then ... else ... follow the Dodo ?
Software - Practice and Experience, Vol. 9, Sept 1979

17/. Ichbiah, J.D. (ed) Rational for the design of the Ada programming language Sigplan Notices, June 1979

18/. Dijkstra, E.W. Guarded commands, nondeterminacy and formal derivation of programs CACM, Vol. 18, August 1975

19/. McKeeman, W.M. Programming language design - in -
Compiler Construction, Springer Verlag, 1976

20/. IBM system/360 Guide to system use
IBM Publication form C28-6812-2

21/. Struble, G. Assembler language programming : The IBM system/360
Addison-Wesley, 1969

22/. Lindsay, H.E. The design, implementation and use of the regeneration machinery of RCC M.Sc. Thesis, University of Manchester, 1975

23/. Standish, T.A. Extensibility in programming language design
Proc. AFIPS, Vol. 44, 1975

24/. Brooker, R.A., MacCallum, I.R., Morris, D., & Rohl, D.S. The Compiler-Compiler ARAP, Vol. 3, 1963

25/. Napper, R.B.E. RCC system documentation
not published

ADDITIONAL BIBLIOGRAPHY

Gries, D. Compiler construction for digital computers
Wiley, 1971

Higman, B. A comparative study of programming languages
(2nd ed), Macdonald, 1977

Hopgood, F.R.A. Compiling techniques
Macdonald/Elsevier, 1969

Cole, A.J. Macro processors
Cambridge Computer Science Texts, CUP, 1976

APPENDIX 1

The new LOCAL PROCEDURE routine

```
ROUTINE

PRIMARY INFORMAL :

LOCAL PROC[P/(EDURE) ? ] [ NAME/a label ] :...


IF current system level /= top routine level ;
OR IF control status ( current system level ) > imperative status :
   {
      TERMINATE SENTENCE
      MAP ? : POSITIONING ERROR OF LOCAL PROC
   }
IF machine code control + control from previous imperative c
   ( current system level ) :
  {
   IF NO current local proc name :
     {
      IF rcc compiling mode HAS compiling class routine marker;
      AND IF NO regeneration entry point for class routine :
        {
         MAP ? : IMPLICIT CONTROL FROM RECOGNITION
         COMPILE { RECOGNISED }
        } :
      OTHERWISE :
        {
         IF rcc compiling mode & system type field = c
            if system type ;
         AND IF rcc compiling mode HAS secondary routine marker :
           {
            MAP ? : IF ROUTINE WITH IMPLICIT CONTROL (YES) AT END
            COMPILE { CONDITION SATISFIED }
           } :
         OTHERWISE : COMPILE { FINISH }
        }
     } :
   OTHERWISE :
     {
      SET UP LABEL d FOR current local proc name
      COMPILE { FINISH [ NAME d ] }
     }
  }
CALL { [ NAME/a label ] : }
current local proc name = address of looked up name
```

The new FOR loop routine

CLASS

```
[ STEP ] = [ Q/( BY,STEP ) ] [ E step size ]
```

ROUTINE

```
PRIMARY CYCLE :

[ Q/(DYNAMIC) ? ] [ NAME  ] [ Q/( =,FROM ) ] [ E initial value ]  c
          [ STEP ? step ] [ Q/( TO,UNTIL) ] [ E  ] DO :...


LOCAL control var = A2,name = A3,a1 = A3,
      set up hidden scalar = A4,constant one = A7,
      exp = A5,explicit = A6,t1 = A4,t2 = A7,step size = A8,
      init test = A9,limit = A10,
      constant or simple var marker = % 14 -> 15

PRESET ARRAY a(0) = -3,C(register marker|1),0
PRESET ARRAY b(0) = -3,C(b line marker|11),0
PRESET ARRAY constant one(0) = -3,0,1

SET NO explicit
SET [ RCC COMP c ] = <
SET [ E limit ]= [ E ]
SET [ NAME control var ] = [ NAME ]
RECORD CYCLE [ NAME control var ]
FIND TYPE not required FOR NAME control var
FIND TYPE t1 FOR EXPRESSION limit
IF NO CAT [ STEP ? step ]  :  step size  =  constant one  :
OTHERWISE : LET [ STEP ? step ] = BY [ E step size ]
FIND TYPE t2 FOR EXPRESSION step size
IF NO [ Q/(DYNAMIC) ? ] :
   {
      IF t1 HAS NO constant data type marker :
        {
           exp = limit
           PERFORM set up hidden scalar
           limit = name
        }
      IF t2 HAS NO constant data type marker :
        {
           exp = step size
           PERFORM set up hidden scalar
           step size = name
           FIND TYPE t2 FOR NAME step size
        }
      ADD - [ V step size ] AFTER [ E initial value ]
   }
```

```
COMPILE { [ V control var ] = [ E initial value ]  }
IF [ Q/(DYNAMIC) ? ] :
   {
      MACRO LABEL [ NAME init test ]
      COMPILE { GO TO [ NAME init test ] }
   }

INTER CYCLE ACTION :

COMPILE { ADD [ E step size ] TO [ V control var ] }
IF [ Q/(DYNAMIC) ? ] :
     COMPILE { [ NAME/a init test ]: }
ADD - [ V control var ] AFTER [ E limit ]
IF [ Q/(DYNAMIC) ? ] ;
   AND IF t2 HAS NO constant or simple var marker :
   {
       explicit = b
       exp = step size
       PERFORM set up hidden scalar
       step size = name
       FIND TYPE t2 FOR NAME step size
   }
IF t2 HAS NO constant data type marker :
   {
      explicit = a
      exp = limit
      PERFORM set up hidden scalar
      limit = name
      COMPILE { IF [ E step size ] < Ø :
                  [ V limit ] = - [ V limit ] }
   } :
OTHERWISE:
   {
      IF constant for ( step size ) < Ø : SET [ RCC COMP c ] = >
   }
COMPILE { IF [ E limit ] [ RCC COMP c ] Ø :
            FINISH EACH [ NAME ] }
FINISH

LOCAL PROCEDURE set up hidden scalar:

IF NO explicit :
   {
      MACRO LOCAL [ NAME name ]
      FIND TYPE not required FOR NAME name
   } :
OTHERWISE : name = explicit
COMPILE { [ V name ] = [ E exp ] }
FINISH set up hidden scalar
```

The routines that implement the RCC case statement

```
ROUTINE

STANDARD IF :

CAN ORGANISE CASE ENVIRONMENT AT [ RT level ] :...

LOCAL n = A2,name = A3,
      case control = finish current item control,
      case cycle marker = % 27,first case marker = % 28

level = current system level
{
  IF type of system level( level ) = top routine level :
    {
      MAP ? : CASE DOES NOT APPEAR WITHIN SELECT
      CONDITION NOT SATISFIED
    }
  IF control status( level ) /= cycle imp status :
    SET level = system level above( level ) ; REPEAT
}
IF case control( level ) HAS NO case cycle marker :
  {
    MAP * : UNTERMINATED CYCLE WITHIN CASE
    CONDITION NOT SATISFIED
  }
IF control status( current system level ) /= imperative status :
  {
   MAP ? : CASE IN SENTENCE
   TERMINATE SENTENCE
   SET control status( current system level ) = imperative status
  }
IF case control( level ) HAS NO first case marker :
  {
   IF machine code control ;
     OR IF control from previous imperative( current system level ) :
      {
       n = cycle control variable name( level )
       SET UP NAME name FOR n
       COMPILE { FINISH CASE [ NAME name ] }
      }
  } : OTHERWISE : FORM case control( level )  & % 29 -> 27
CONDITION SATISFIED
```

```
ROUTINE

PRIMARY INFORMAL :

SELECT [ CV ? ] [ Q/( FROM CONSTANTS ) ? ] [ E1 ]

LOCAL case control = finish current item control ,
      non const with control var = % 26,c = A2,dumy name = A3,
      d = A4,exp = A5,n = A6,
      constant case bit = % 25,case cycle marker = % 27,
      first case marker = % 28


SET NO case statement
IF NO [ CV ? ] :
  {
    MACRO LOCAL [ NAME n ]
    COMPILE { LOCAL [ NAME n ] = BØ }
    SET [ E exp ] = [ NAME n ]
  } :
OTHERWISE :
  {
    LET [ CV ? ] = FOR [ NAME n ] [ ASSIGNED ? d ]
    IF NO [ ASSIGNED ? d ] : SET [ E exp ] = [ NAME n ] :
    OTHERWISE : LET [ ASSIGNED ? d ] = = [ E exp ]
  }
COMPILE { CYCLE [ NAME n] = [ E exp ] X STEP Ø [ E2 ar ] c
          [ E2 ar ] [ E1 ] }
level = current system level
SET level = system level above( level )
IF [ Q/( FROM CONSTANTS ) ? ] :
  {
    DIVERT CONTROL c
    ADD CHAIN c AFTER CHAIN case control( level )
    FORM case control( level ) | constant case bit
  } :
OTHERWISE :
  {
    IF [ CV ? ] : FORM case control( level )   c
                  | non const with control var
  }
FORM case control( level ) | case cycle marker
FORM case control( level ) | first case marker
FINISH
```

```
ROUTINE

PRIMARY INFORMAL :

CASE [ RCC COND selector ] :...

LOCAL case chain = address of inter cycle action ,
      case control = finish current item control,
      level = A2,name = A3,const = A4,exp = A5,
      addr = address of looked up name,n = A6,
      const val = A7,case code addr = A8,
      constant case bit = % 25,
      non const with control var = % 26,
      case cycle = % 27,first case = % 28


UNLESS CAN ORGANISE CASE ENVIRONMENT AT level : FINISH
IF case control( level ) HAS constant case bit :
  {
    IF CAT [ RCC COND selector ] = SN { [ RCC COND ] c
       = [ Q/(NO) ? ] [ C ] { :
      {
        LET [ RCC COND selector ] = [ C const ]
        SET const val TO CONSTANT [ C const ]
        MACRO LABEL [ NAME case code addr ]
        COMPILE { [ NAME/a case code addr ]: }
        ADD addr AFTER CHAIN case chain( level )
        ADD const val AFTER CHAIN case chain( level )
      } :
    OTHERWISE : MAP * : NOT A CONSTANT
  } :
OTHERWISE :
  {
    IF case control( level ) HAS non const with control var :
      {
        IF CAT [ RCC COND selector ] = SN { [ RCC COND ] c
          = [ Q/(NO) ? ] [ E ] { :
          {
            LET [ RCC COND selector ] = [ E exp ]
            n = cycle control variable name( level )
            SET UP NAME name FOR n
            SET [ RCC COND selector ] = [ NAME name ] = [ E exp ]
          } :
        OTHERWISE : MAP * : NOT AN EXPRESSION
      }
    IF type of system level( current system level ) /= action level :
      COMPILE { IF [ RCC COND selector ] : { } :
    OTHERWISE :
      COMPILE { } : OTHERWISE : { IF [ RCC COND selector ] : { }
  }
FINISH
```

```
ROUTINE

PRIMARY SIMPLE JUMP : FINISH CASE [ NAME ? cv ] .

LOCAL case control = finish current item control,
      constant case bit = % 25,n = A3,
      non const with control var = % 26,level = A2,
      control var present = % 25 -> 26

IF NO [ NAME ? cv ] :
level = current system level ;
{
  IF type of system level( level ) = top routine level :
    {
      MAP ? : SELECT WITHOUT CONTROL VAR NOT FOUND
      FINISH
    }
  IF control status( level ) = cycle imp status ;
    AND IF case control( level ) HAS NO control var present :
     {
      n = cycle control variable name( level )
      SET UP NAME cv FOR n
     } :
  OTHERWISE : SET level = system level above( level ) ; REPEAT
}
CALL { FINISH EACH [ NAME cv ] }


ROUTINE

PRIMARY INFORMAL :

ALL OTHER CASES :...

LOCAL case chain = address of inter cycle action,
      case control = finish current item control,
      all other cases addr = A2,level = A3,
      constant case bit = % 25

UNLESS CAN ORGANISE CASE ENVIRONMENT AT level : FINISH
IF case control( level ) HAS constant case bit :
  {
    MACRO LABEL [ NAME all other cases addr ]
    COMPILE { [ NAME/a all other cases addr ]: }
    ADD address of looked up name AFTER CHAIN case chain c
              ( level )
  } :
OTHERWISE : COMPILE { } : OTHERWISE : { }
```

```
ROUTINE

PRIMARY INFORMAL : END OF SELECTION [ NAME ? cv ].


LOCAL case control = finish current item control,
      lowest const = A6,position = A7,
      case chain = address of inter cycle action,level = A5,
      case labels = A3,n = A4,label of case = A2,
      all other cases addr = A9,constant case bit = % 25,
      non const with control var = % 26

PRESET ARRAY lowest const plex(Ø) = -3,Ø,Ø
PRESET ARRAY highest const plex(Ø) = -3,Ø,Ø

UNLESS CAN ORGANISE CASE ENVIRONMENT AT level : FINISH
IF NO [ NAME ? cv ] :
  {
    n = cycle control variable name( level )
    SET UP NAME cv FOR n
  }
IF case control( level ) HAS NO constant case bit :
  {
    IF type of system level( current system level ) = action level :
        COMPILE { } } ; REPEAT
  } :
OTHERWISE :
  {
    FORM case control( level ) & % ( Ø -> 22,3Ø,31 )
    FILL IN CONTROL case control( level )  c
              TO current code address
    DELETE CHAIN case control( level )
    WITHDRAW all other cases addr FROM CHAIN case chain( level )
    SET UP LABEL other cases label FOR all other cases addr
    SET NO highest const
    SET NO lowest const
    SET NO number of labels
    FOR EACH position ON CHAIN case chain( level ) :
      {
        number of labels = number of labels + 1
        NEXT * position
        n = ( position )
        IF n < lowest const : lowest const = n :
        OTHERWISE : { IF n > highest const : highest const = n }
      }
    highest const plex(2) = highest const
    lowest const plex(2) = lowest const
    COMPILE {
      IF [ NAME cv ] > [ NAME highest const plex ] ;
        OR IF [ NAME cv ] < [ NAME lowest const plex ] :
          GO TO [ NAME other cases label ]
            }
    FOR i = 1 TO number of labels :
      case labels( i ) = other cases label
```

```
    FOR EACH position ON CHAIN case chain( level ) :
      {
        label of case = ( position )
        NEXT * position
        n = ( position )
        n = n - lowest const + 1
        SET UP LABEL l FOR label of case
        case labels( n ) = l
      }
    DELETE CHAIN case chain( level )
    STACK number of labels AT case labels
    COMPILE { GO TO [ NAME* case labels ] BY [ NAME cv ] }
  }
SET case statement
CALL { END OF CYCLE [ NAME cv ] }
SET NO case statement
FINISH
```

APPENDIX 2

The new PRINT ITEM routine (illustrating use of case statement)

ROUTINE

```
STANDARD : PRINT ITEM [ E routine serial no = B16 ].

LOCAL bottom three bits = #7,i = A2,j = A3,sn = A2
LOCAL no of instrs on line = A3,reg = A7,regtwo = A2
LOCAL op code = A8,index = A2,b = A9,d = A10
LOCAL base = A4,limit = A5,hword = A6
LOCAL rx bits = #tc000
LOCAL op code mask = #tff00
LOCAL reg mask = #tf0
LOCAL b mask = #tf000
LOCAL d mask = #tfff
LOCAL index mask = #tf
LOCAL regtwo mask = index mask
LOCAL half word boundary = #t2
LOCAL half word inc = half word boundary
LOCAL fn mask = #ff
LOCAL load halfword fn = #4000048

sn = routine serial no
PRINT : [ N 4 ] [ S 45 ] PRINTING ITEM SERIAL NO [ E sn ]
base = index entry( sn )
limit = ( base - 2 ) + base
NEWLINE ; NEWLINE
PRINT : [ S 35 ] LAST LINE FOR ITEM = [ E ( base - 2 ) ]
PRINT : [HW (base - 2) T ],    KEY FOR ITEM = [HW (base - 1) T ]
NEWLINE ; NEWLINE
PRINT : [ S 45 ] ITEM [ E sn ]
IF key for item( base ) HAS routine item marker :
  {
   PRINT : DEASSEMBLED [ N 2 ] ; NEWLINE
   SET NO no of instrs on line
   PRINT : [ E base ]  [ HW ( SL 6 ) base T ] [ S 8 ]
   OBEY load halfword fn,6,0,4,0
   WHILE hword :
     {
      IF hword HAS rx bits :
       {
        op code = hword & op code mask LD 6
        reg = hword & reg mask LD 2
        index = hword & index mask LU 2
        FORM base + half word inc
        OBEY load halfword fn,6,0,4,0
        b = hword & b mask LD 10
        d = hword & d mask LU 2
        SELECT FOR op code :
          CASE load fn & fn mask : PRINT : L
          CASE store fn & fn mask : PRINT : ST
          CASE add fn & fn mask : PRINT : A
          CASE sub fn & fn mask : PRINT : S
          CASE load address fn & fn mask : PRINT : LA
          CASE uncond branch fn & fn mask : PRINT : BC
```

```
          CASE bal fn & fn mask : PRINT : BAL
          CASE lm fn & fn mask : PRINT : LM
          CASE stm fn & fn mask : PRINT : STM
          CASE and fn & fn mask : PRINT : N
          CASE load halfword fn & fn mask : PRINT : LH
          ALL OTHER CASES : PRINT : [ HW ( SL 2 ) op code ]
        END OF SELECTION
        PRINT :  [ E ( SL 2 ) reg ]   [ HW ( SL 3 ) d ] c
            ( [ E ( SL 2 ) index ],[ E ( SL 2 ) b ] ) [ S 13 ]
      } :
    OTHERWISE :
      {
       op code = hword & op code mask LD 6
       reg = hword & reg mask LD 2
       regtwo = hword & regtwo mask LU 2
       SELECT FOR op code :
         CASE load reg fn & fn mask : PRINT : LTR
         CASE load neg reg fn & fn mask : PRINT : LCR
         CASE add reg fn & fn mask : PRINT : AR
         CASE sub reg fn & fn mask : PRINT : SR
         CASE uncond branch on reg fn & fn mask : PRINT : BCR
         CASE balr fn & fn mask : PRINT : BALR
         CASE svc fn & fn mask : PRINT : SVC
         ALL OTHER CASES : PRINT : [ HW ( SL 2 ) op code ]
       END OF SELECTION
       PRINT :  [ E ( SL 2 ) reg ]   [ E ( SL 2 ) regtwo ] c
           [ S 23 ]
      }
    FORM base + half word inc
    OBEY load halfword fn,6,0,4,0
    FORM no of instrs on line + 1
    IF no of instrs on line = 3 :
      {
       SET NO no of instrs on line
       NEWLINE
       PRINT : [ E base ]  [ HW ( SL 6 ) base T ] [ S 8 ]
      }
  }
  IF base HAS half word boundary :
    FORM base + half word inc
  PRINT : [ N 2 ] [ S 32 ] DATA AREA OF ROUTINE,
}
PRINT : CONTENTS IN TRUE HEX AND DECIMAL RESPECTIVELY [ N 2 ]
CYCLE j = 1 TO 2 :
  PRINT : [ E base ]  [ HW ( SL 6 ) base T ]
  FOR i = base TO limit :
    {
     IF i HAS NO bottom three bits :
       {
        NEWLINE
        PRINT : [ E i ]  [ HW ( SL 6 ) i T ]
       }
     IF j = 1 : PRINT : [ HW ( SL 12 ) ( i ) T ]  :
     OTHERWISE : PRINT : [ E ( SL 11 ) ( i ) ]
    }
  NEWLINE ; NEWLINE ; NEWLINE
END OF CYCLE j
```

APPENDIX 3

The routines that implement the half-word sublanguage

```
CLASS

[ EXT OR ASS ] = EXTRACT [ V ] FROM, ASSIGN [ E ] TO

ROUTINE

PRIMARY : [ Q/(INCREMENT, DECREMENT) d ][ V ptr ].
COMPILE { FORM [ V ptr ][ RCC OPR d ] #t2 }


ROUTINE

PRIMARY: [ EXT OR ASS e ][ E ptr ][ P/(AND INCREMENT) ? ] c
           [ Q/(BEFORE, AFTER) ? q ].

LOCAL fn=A2, a0=A3, a1=A4, t=A5, x=A6

PRESET ARRAY a0(0) = already processed block,C(register marker),0
PRESET ARRAY a1(0) = already processed block,C(nonzero   c
                accumulator working reg | register marker),0

FIND TYPE t FOR EXPRESSION ptr
IF { t = general expression data type ; OR t= hybrid data type ;
  OR t HAS constant data type marker } ;
    AND IF  [Q/(BEFORE,AFTER)? q ] :
      MAP * : EXPRESSION PARAMETER NOT ALLOWED ; FINISH
IF CAT [EXT OR ASS e] = 1 : fn = load halfword fn :
OTHERWISE : fn = store halfword fn
IF [Q/(BEFORE, AFTER)? q] = BEFORE : PERFORM increment
IF [EXT OR ASS e] = ASSIGN [E x] TO : COMPILE {[Va0]=[Ex]}
COMPILE { [V a1 ] = [E ptr ] }
COMPILE fn,accumulator working reg,0,nonzero accumulator working reg,0
SET NO lds in a0
IF [EXT OR ASS e] = EXTRACT [V x] FROM :
  COMPILE { [V x] = [V a0 ] & #t0000ffff }
IF [ Q/(BEFORE, AFTER) ? q ]  = AFTER : PERFORM increment
FINISH

LOCAL PROCEDURE increment : COMPILE {INCREMENT [V ptr ] }
FINISH increment
```

APPENDIX 4

The syntax of the RCC meta-language (in the RCC meta-language)

This appendix basically gives the formal syntactic descriptions of CLASS master sections.

```
[ CLASS MASTER SECTION ] = [ CLASS DEFINITION * ]

[ CLASS DEFINITION ] = [ CLASS WORD ] [ P6 ? retriction ] = [ EOL ]
                       [ P5 body of definition ] [ EOL ]

[ CLASS WORD ] = [ [ NAME/B ] [ P1 ? ] [ P2 ? ] [ NAME/ar ? ]
                   [ P3 r ],
                   [ [ NS NAME/ ( ] ? [ e t a r / | ] ) ]
                   [ P1 ? ] [ P2 * ? ] [ P/ | ? ]
                   [ NAME/ar ? ] [ P3 r ]

[ P1 parameter ] = /( [ P4 * ? ] ), /[ CLASS WORD ], /[ SYMBOL ]

[ P4 ] = [ CLASS WORD ], [ SYMBOL/) excepted ]

[ P2 qualification ] = ?, e, t, a, r

[ P3 r ] = ],]

[ P6 restriction ] = UC, LC, UC, LC, RD, R

[ P5 body of restriction ] = [ P/( LIST OF ) ] [ PHRASE ],
          [ EOL ? ] SEP [ P/( ARATED BY) ? ] [ P4 * ],
          [ P/( LIST OF) ] [ PHRASE ],
          NOT [ PHRASE */(, [ P7 r ] ) ],
          [ EOL ? ] BUT [ PHRASE */(, [ EOL ? ] ) ],
          [ PHRASE */(, [ EOL ? ] ) ]        ? NB,=,

[ PHRASE ] = LIST OF [ P8 ]

[ P8 ] = [ CLASS WORD ], [ SYMBOL/, excepted ]

[ P7 r ] = NOT [ EOL ? ] BUT, BUT [ EOL ? ]
```

APPENDIX 5

RCC base language syntax

This is the overall syntax of a ROUTINE master section.

```
[ ROUTINE MASTER SECTION ] = [ ROUTINE HEADING ] [ EOL ]
                             [ ROUTINE BODY ]

[ ROUTINE BODY ] = [ SENTENCE * ]

[ SENTENCE ] = [ LABEL * ? ] [ COND SENTENCE ], [ LABEL * ? ]
       [ FOR SENTENCE ],
       [ STATEMENT */ [ T ] [ MAJOR T ]

[ COND SENTENCE ] = [ COND CLAUSE ] [ STATEMENT */ [ T ]
       [ OTHERWISE CLAUSE ? ] [ MAJOR T ],
       [ COND CLAUSE ] :... [ EOL ? ]

[ OTHERWISE CLAUSE ] = [ T ] OTHERWISE [ T ] [ STATEMENT */ [ T ] ]

[ COND CLAUSE ] = [ Q/( IF, UNLESS) ] [ RCC COND ] [ T ]
       [ AND OR SEQUENCE ? ]

[ AND OR SEQUENCE ] = [ AND SENTENCE ], [ OR SENTENCE ]

[ AND SENTENCE ] = LIST OF AND [ Q/( IF, UNLESS) ] [ RCC COND ] [ T ]

[ OR SENTENCE ] = LIST OF OR [ Q/( IF, UNLESS) ] [ RCC COND ] [ T ]

[ FOR SENTENCE ] = FOR [ RCC CYCLE ] [ T ] [ STATEMENT */ [ T ] ]
    [ MAJOR T ],
    CYCLE [ RCC CYCLE ] [ T ] [ SENTENCE * ] END OF CYCLE
    [ NAME ] [ MAJOR T ],
    WHILE [ RCC COND ] [ T ] [ STATEMENT */ [ T ] ] [ MAJOR T ]

[ STATEMENT ] = [ LABEL * ? ] [ BASIC STATEMENT ],
    [ LABEL * ? ] { [ SENTENCE * ] }

[ LABEL ] = [ NAME /a ] [ T ]

[ T ] = : [ EOL ? ], ; [ EOL ? ]      ? minor terminator ?

[ MAJOR T ] = [ P/} a ], [ EOL ]      ? major terminator ?
```

Notes

[RCC COND] is the system format class to which a new alternative is added whenever the user defines a routine with the IF control characteristic. [RCC CYCLE] is the format class for CYCLE control characteristic routines.

APPENDIX 6

Subsidary syntax

```
[ E ] = [ +- ? ] [ RCC OPD */ [ RCC OPR ] ]

[ +- ] = +, -

[ RCC OPR ] = +, -, &, |, XOR, AU, LU, LD

[ RCC OPD ] = [ V ], [ C ], CAT [ P/( EGORY OF ? ] [ PV ],
    VAL [ P/( UE OF) ? ] [ PV ], NUM [ P/( BER OF) ? ] [ PV ]

[ V ] = ( [ NAME ] ), ( [ NAME ] [ +- ] [ C ] ),
    ( [ NAME ] [ +- ] [ NAME ] ),
    ( [ NAME ] [ +- ] [ NAME ] [ +- ] [ C ] ),
    ( [ C ] ), [ NAME ] ( [ C ] ),
    [ NAME ] ( [ NAME ] ), [ NAME ] ( [ NAME ] [ +- ] [ C ] ),
    [ NAME ]

[ C ] = [ N ], -[ N ], #[ HW ], "[ SYMBOL ]", %[ E3 ],
    C ( [ +- ? ] [ E4 */ [ RCC OPR ] ] ),
    SN { [ PV ] = [ MP ] },
    SN [ PV ]

[ E3 ] = [ N ] -> [ N ], [ N ], ( [ E3 */, ] )

[ E4 ] = [ NAME ], [ C ]

[ R5 ] = LIST OF [ RCC OPR ] [ RCC OPD ]

[ E7 ] = [ E4 ], [ E4 ], [ E4 ], [ E4 ]    ? NB, =,

[ R7 ] = [ E ( S [ Q/( LP, L, RP, R) ] [ ED ] ) [ E6 ] ],
    [ E [ E6 ] ],
    [ HW ( S [ Q/( L, R) ] [ ED ] ) [ E6 ] [ Q/T ? ] ],
    [ HW [ E6 ] [ Q/T ? ] ],
    [ SYM [ E6 ] ],
    [ [ Q/( N, S) ] [ ED ? ] ],
    [ PV ],
    [ NSNAME/( C;:.}})]    ? any symbol string excluding ;:.}} ?
[ E6 ] = [ CHAIN [ VED ], [ ED ] ( [ ED ] TO [ ED ] ),
    [ ED */ [ EOL ? ] )

[ E1 ] = :..., ;, :, [ EOL ], [ [ P/( } } ) a ]

[ Q8 ] = LIST OF [ NAME ] [ Q7 ? ], SEP, [ EOL ? ]

[ Q7 ] = = G [ NAME ], = AAA [ HW ], = [ Q/( A, B ) ] [ N ],
    = ( [ Q/( A, B) ] [ N ] + [ N ] ), = ( [ Q/( A, B) ] [ N ],
    = [ Q/C  ? ] [ E ]
```

```
[ RCC COMP ] = =, >=, <=, >, <, /=

[ R1 ] = LIST OF [ Q/( NOT ) ? ] [ RCC COMP ], SEP ; [ EOL ? ]
     [ AND, OR ]

[ AND, OR ] = AND, OR

[ R2 ] = DOWN, ALONG CHAIN, STEP [ V ] = [ E ], STEP [ E ]

[ R3 ] = TO [ RCC COMP ? ] [ V ] = [ E ],
     TO [ RCC COMP ? ] [ E ], UNTIL [ RCC COND ]

[ R9 ] = [ PV ], [ SYMBOL/( ;:.}}) ]
```