The Design and Implementation of RCCT,

A Bootstrapping Language for the Revised Compiler-Compiler

BY

Peter M. Dewar

Thesis submitted for the degree of Master of Science at the University of St. Andrews, June 1975

ABSTRACT

A compiler has been designed and written to execute programs written in the RCCT Language, a simple subset of the Revised Compiler-Compiler (RCC) Language [3], on an IBM 360/44 computer. The primary objective of the compiler is to bootstrap RCC onto the /360. In general, an RCCT program will describe a compiler, called the object compiler. In generating an object compiler, the RCCT compiler has adopted routine linking and register conventions which are non-standard to the /360 Operating System. This has meant that the problems of interfacing the object compiler with the /360 System have been more easily isolated from the process of generating an object compiler.

ACKNOWLEDGEMENTS

I would like to thank Dr. R. N. Fisher for his zeal in supervising the work described in this thesis. I also own a debt of gratitude to the Science Research Council for their financial support during the last year. Finally I would like to thank Mrs. Maureen Sanders and Mrs. Catherine Evans-Smith for their patience and endurance in typing this work.

CONTENTS

CHAPTER 1

INTRODUCTION

1.1 Project Specification

The original problem giving rise to the work described in this thesis was the desire to implement the Revised Compiler Compiler (RCC) [3] on the IBM 360/44 machine installed at the St. Andrews University Computing Department.

RCC is a high-level (word-oriented) machine independent language which is written in its own language. It is designed as a systems programming language but is particularly useful as a compiler-compiler because of its syntactic and semantic language extension facilities. It is at present running on an ICL 1906A machine and work is currently being undertaken to implement it on a CDC 7600 machine at Manchester. However, it has not yet been implemented on an IBM machine.

The problem of portability of RCC is non-trivial. It is a large system which, of necessity, incorporates a number of machine-dependent features, such as I/O. However, the portability problem has already been investigated when RCC was transported on to the 1906A from the old Atlas machine. It was found that a large subset of RCC, called Atlas-RCCT, was sufficient to describe the RCC compiler. (Note, however, that the RCCT implementation described for the /360 is a different RCCT than the Atlas RCCT. The /360 RCCT has been designed almost from scratch.) Thus, given an RCCT compiler, this could be

used to compile a version of RCC written in RCCT. Since an RCCT compiler is not concerned with efficiency of compiled code, the result would be an inefficiently written RCC compiler. By definition, an RCC compiler generates efficient code. Thus, the version of RCC written in RCC could then be compiled by the inefficiently written RCC compiler to give an efficiently written RCC compiler which generates efficient code (see fig. 1.1). Therefore the bootstrapping problem involves first designing a compiler for the RCCT language.

The RCCT language is, in a sense, as powerful a descriptive tool as RCC, since it can be used to describe RCC. However, the primary difference is that RCCT is not as high-level as RCC and does not have the richness of facilities.

This thesis describes the RCCT language and the implementation of an RCCT compiler, subsequently referred to as "the compiler", on the 360/44.

1.2 The Compiler Design

An informal specification of the syntax and semantics of RCCT was provided in the form of notes. To get the flavour of the language, it was decided to write as much of the compiler as possible in RCCT before translating it into the implementation language. This method of program design was found to be ideal since it formalised the informal language specification into a set of routines very similar in structure to the corresponding routines coded in the

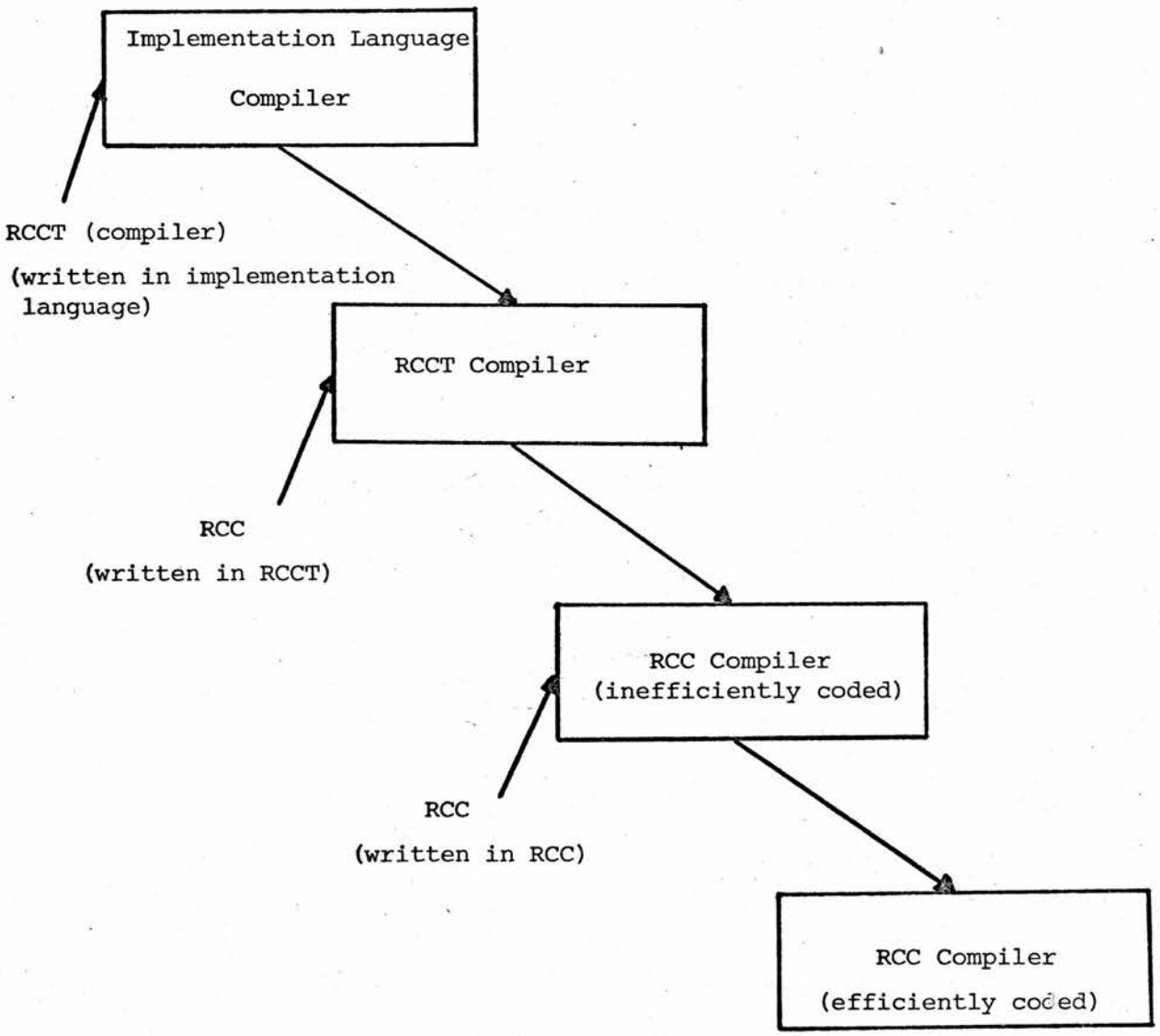


Fig. 1.1 : The Bootstrapping Mechanism

implementation language. The only disadvantage was that it slowed down the initial development of the compiler. However, this was outweighed by the reduced time required to debug the final versions of the various routines since the initial design language was reasonably high-level.

Where a routine describes a machine-dependent function, such as I/O, it was designed in flow-chart or algorithmic form prior to translation into the implementation language. This was because an RCCT routine performing the same function is less efficient since it cannot make use of the byte-addressability of the /360 machine. Hence the translation process from RCCT into the implementation language is less obvious.

1.3 The Implementation Language

For efficiency, the ideal choice of language to write a compiler in is Assembler. However, Assembly language programming is tedious and the advantages gained by writing efficient code are outweighed by the disadvantages of reduced throughput. An excellent alternative, however, is the PL360 language [7]. This is a high-level format assembly language combining many of the advantages of high-level programming with those of low-level programming. Block-structuring, procedures and looping control features make PL360 very similar in format to ALGOL. This is combined with the fact that the programmer is always aware of exactly what machine instructions are being generated and of the exact configuration of his data. Hence, PL360 was chosen as the implementation language.

Each RCCT routine (or algorithm) is described by a PL360 procedure. Where possible, the PL360 procedure has the same name as the corresponding RCCT routine. However, PL360 does not provide any parameter passing mechanism between procedures. So the convention was adopted that an array of ten fullwords, called "myplist", would be used for passing parameters. Thus, for example, the RCCT routine:

"CHECK BLOCK SYNTAX GIVING RESULT a"

is coded as the PL360 procedure:

"CHECKBSYNT"

and passes its return parameter "a" back in "myplist". This convention implies that if a procedure calls another procedure the calling procedure must save any entries in "myplist" which it requires, locally. Similarly, the return address used to transfer control on exit from a routine is always passed in a register, and the calling procedure must first save the old return address, locally, before calling another procedure. For further details on PL360 programming, see [7].

To test the compiler, it has been initially designed as a load-and-go system. It is this version of the compiler that is described in this thesis. For details of how to convert it into a stand-alone compiler, see § 6.4

1.4 Notes

The descriptive part of the thesis occupies four chapters. The first of these (Chapter Two) describes the RCCT language, its syntax

and semantics, and the remaining three chapters summarise the implementation of the RCCT compiler. The implementation description consists of:

i) Chapter Three, describing the conventions adopted for coding an RCCT program on cards using an IBM 2903 card punch, together with a synopsis of the processes involved in the Lexical Analysis of a program.

ii) Chapter Four, describing the implementation of Syntax Analysis.

iii) Chapter Five, describing the implementation of the Semantics associated with a program (code generation).

A number of Appendices are included at the end of the thesis:

i) Appendix A describes the formal syntax of RCCT, and may be referenced in conjunction with Chapter 2.

ii) Appendix B lists all the diagnostic messages generated by the compiler, together with an explanation of the cause, and hence source of each diagnostic.

iii) Appendix C lists, formally, the Syntax definition of RCCT used by the Syntax Analyser.

iv) Appendix D summarises the use of general purpose registers at compile-time and run-time of an RCCT program.

v) Appendix E contains some simple test program results.

vi) Appendix F contains a listing of those routines contained in the compiler which are not explained in the text, with the omission of the main program and data declarations. These routines are the RCCT versions of the actual PL360 procedures incorporated in the compiler i.e. they summarise the design of the compiler. (One difference in notation in the RCCT listing of routines is that the use of "...." implies that code is planted to call the Message Interpreter to print the appropriate diagnostic (see § 3.2))

vii) Appendix G summarises the difference between the RCC syntax notation used in the thesis text and Appendix A and C, and Backus Naur Form (B.N.F.).

In the text, the RCCT routine is referred to by its name (in upper case letters with any embedded parameters and commas). The use of "...." notation after a routine name or partial name implies that the routine has already been mentioned in the text. e.g. the routine "CHECK BLOCK SYNTAX GIVING RESULT a" would be subsequently referred to (in the text) by, say, "CHECK BLOCK....".

CHAPTER 2

THE RCCT LANGUAGE

2.1 The Program

An RCCT program consists of a string of basic RCCT symbols, taken from the symbol set shown in fig. 2.1, and conforming to the Formal Syntax summarised in Appendix A. This Formal Syntax is complete apart from a description of how blanks and comments are used. A comment is a string of EBCDIC symbols enclosed in the comment brackets, "(" and ")". Comments and blanks may be inserted anywhere in a program and both are ignored by the compiler.

In the current card-input environment, the first 72 columns of each card, only, are assumed significant. (The first 72 columns of one card image are assumed to follow on from the first 72 columns of the previous card image.) Blank cards, or cards containing comments, only, are ignored. Unless the RCCT continuation symbol, "c", appears as the last significant symbol in a card image, the implicity defined [EOL] symbol is assumed to appear at the end of the card. This class, [EOL], stands for the End Of Line terminator, and is used in defining the Syntax of the language. If a card image contains the continuation symbol as its last significant character, then the continuation symbol is ignored and the next card image is assumed to follow on immediately from the previous card image (without an intervening [EOL] symbol). For further

0, 1, 2, 3, 4, 5, 6, 7, 8, 9,

0̲, 1̲, 2̲, 3̲, 4̲, 5̲, 6̲, 7̲, 8̲, 9̲,

a, b, c, ..., z,

a̲, b̲, c̲, ..., z̲,

A, B, C, ..., Z,

A̲, B̲, C̲, ..., Z̲,

=, ≠, <, ≤, >, ≥,

+, -, &, |, ≡, ≠,

*, #, ", %, →,

(,), [,], (̲,)̲, {, }, ,,

Fig. 2.1 : RCCT basic symbols

details of card input see §3.3.

A program is defined syntactically (see Appendix G) by:-

[PROGRAM]=[RCCT BLOCK*/[EOL]] [EOL] END OF PROGRAM [EOL]

[RCCT BLOCK] = GLOBAL [EOL][DECLARATION */[,]],

ROUTINE [EOL][RTYPE SPEC] : c

[ROUTINE HEADING PART*][EOL STATEMENTS?]

[EOL STATEMENTS]=[EOL][STATEMENTS]

An[RCCT BLOCK]is thus a Global or a Routine block. Routine blocks define a section of program called a "routine", and a name which may be used to refer to the routine. Global blocks define global data on which a program may operate. The END OF PROGRAM statement defines the end of an RCCT program.

e.g.

```
ROUTINE
   .
   .
   .
GLOBAL
   .
   .
   .
GLOBAL
   .
   .
   .
ROUTINE
   .
   .
   .
END OF PROGRAM
```

Each routine defines a set of operations to be performed on data. Hence a program consists of a set of operations, defined in terms of "routines", and a set of data defined either in Global blocks, or locally inside a routine (see §2.2).

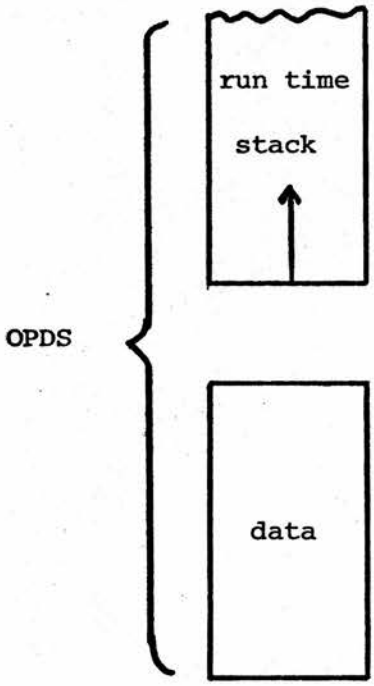
The operating environment of an RCCT program consists of the set of routines defined by the program compiled into machine code, collectively referred to as the Object Program, together with a set of data, collectively referred to as the Object Program Data Space (OPDS). OPDS contains all the data items defined by a program, together with a run-time stack, which consists of an area of store. (See fig. 2.2) General purpose register 7 is used to hold the base address of the run-time stack. This stack is used during the execution of a program to perform certain routine linkage functions (see §2.6), but may also be used by the program as a work-space.

In addition to this special purpose assignment to register 7, the compiler also assigns permanent run-time functions to registers 0 and 8 - 15. However, a program is free to use registers 1 - 6 as it wishes.

The set of operations defined in a routine constitutes the class [STATEMENTS] as defined in Appendix A. Thus, a routine body is either empty or consists of a list of [CONTROL STATEMENT]'s and [SIMPLE STATEMENTS]'s.

The use of terminators in a routine is primarily to act as statement separators, but in addition:-

(i) The [EOL] terminator is called a "major terminator" since its main function is to define the end of a [CONTROL STATEMENT]. When used to define the end of a list of [IMPERATIVE STATEMENTS]'s, its only function is as a statement separator.



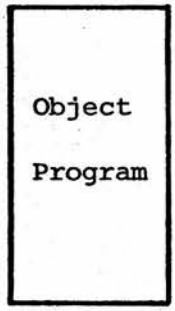


Fig. 2.2 : Operating Environment of an RCCT Program

(ii) The semicolon terminator is called a "minor terminator" and its main function is to bind together a set of [IMPERATIVE STATEMENT]'s into a [CONTROL STATEMENT] body. Another use is to bind together the various conditions of a multiple condition [IF CLAUSE] (see §2.6)

(iii) The colon terminator is also a "minor terminator". It has been included in the definition of RCCT to aid legibility of programs. (Logically, RCCT only requires one major and one minor terminator). Colons appear after [LABEL]'s, and at the end of [IF CLAUSE]'s and for-clauses, as well as before and after an OTHERWISE statement (see §2.6)

Note: The distinction between major and minor terminators is different in RCC although the end result is much the same.

Compound statements are used to avoid ambiguities when [CONTROL STATEMENT]'s are nested.

e.g. IF a=b : IF c=d : x=y : OTHERWISE : x=z

is ambiguous, but, using a [COMPOUND STATEMENT] resolves the ambiguity:-

IF a=b :{IF c=d ; x=y}: OTHERWISE : x=z

or

IF a=b :{IF c=d : x=y : OTHERWISE : x=z}

Another use of compound statement bracketing is their significance to a "REPEAT" instruction (see §2.6).

During execution of a routine, control is assumed to be sequential i.e. no facilities are included in the language to allow for parallel execution of statements.

2.2 Data Definition

The data which an RCCT program operates on consists of constants, which are items whose value is fixed, and scalars, which are items whose value may vary. RCCT is a word-oriented language, having one type only, type "index" i.e. all data items occupy a 32-bit IBM 360/44 computer word. However, the /360 is a byte-addressable machine. Hence, to map word-addressing onto a byte-addressing machine requires that all computed addresses must be a multiple of 4, since there are 4 bytes per word. This mapping may be performed at either run-time or compile-time of a program. If performed at run-time, then each time an address is computed, the result must be multiplied by 4 to refer to the correct word. However, this involves executing extra instructions at run-time each time an address is required. The other solution, which has been adopted by the compiler, is to multiply all constants by 4 during syntactic recognition. This ensures that any computed address involving constants will always be a multiple of 4. A facility is available for avoiding this automatic multiplication, for example when a constant is used to mask out parts of the actual 32-bit word. However, this facility should be used with great care, since, if a constant is used to address a data item and this constant has not been automatically multiplied, an attempt may be made to access half-words etc. This would cause a run-time addressing error.

A constant may be defined in six different ways:-

(i) integer
(ii) octal number
(iii) hexadecimal number
(iv) system symbol
(v) field specifier
(vi) hexadecimal bit pattern

An integer is a string of decimal digits:

e.g. 249, 0, 12, 123459674

An octal number is a string of octal digits preceded by a "*" symbol:

e.g. *3, *777, *01235

A hexadecimal number is a string of hexadecimal digits preceded by a single "#" symbol:

e.g. #3, #49fb2, #fff0, #00ba

These three constants may be thought of as different ways of writing a positive integral value:

e.g. 249, *371, #f9 all define the positive integral value corresponding to decimal 249.

The system symbol constant consists of any [SYSTEM SYMBOL] enclosed in a pair of double-quote symbols. A [SYSTEM SYMBOL] may be any one of the EBCDIC characters shown in [1], except a single quote, ', which has a special significance to the compiler (see Chapter Three). This form of constant is used to denote the internal EBCDIC representation of the particular symbol in quotes.

e.g. "," "B", "#", "?", " ", "-", """

Note that the EBCDIC blank symbol can be represented. An alternative way to denote the blank symbol is by juxtaposing two double-quote symbols:

e.g. ""

The fifth form of constant is the field specifier constant consisting of a "%" symbol, followed by either a single bit-sequence specification, or a list of bit-sequence specifications separated by commas and enclosed in round brackets. A bit-sequence specification consists of a single integer, or a pair of integers separated by the symbol "→".

e.g. %12, %0→5, %(2), %(1→3), %(2→6, 9, 18→24)

The bit-sequence specification signifies either a single bit position to be set, or an ascending sequence of bits i.e. 2→6 is an abbreviation for the list 2, 3, 4, 5, 6. Therefore 6→2 is undefined since the sequence is not an ascending one. (Note that % 2→2 is another way of writing %2.) The bit positions referred to by a field specifier are the bit positions occupied by the four previously mentioned constants i.e. positions 0→29, with the addition that bit positions 30 and 31 are the two least significant bits in the 32-bit word which are not used by the other constants (see figure 2.3).

The hexadecimal bit pattern constant consists of a string of hexadecimal digits preceded by two "#" symbols:

e.g. ##ff, ##129fa, ##00f04

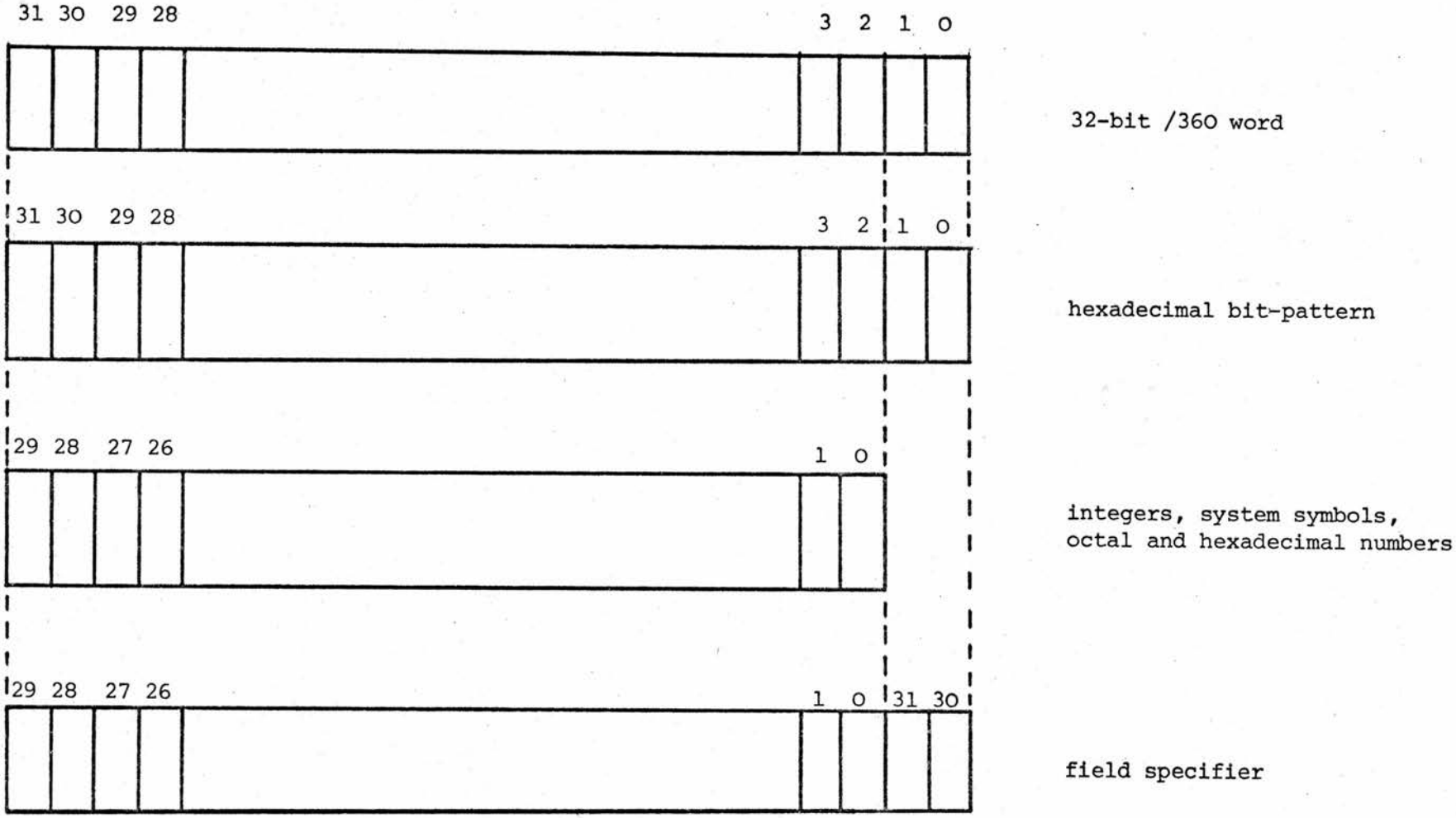


Fig. 2.3 : Internal representation of constants

This form of constant is used to avoid the automatic multiplication of a constant on syntactic recognition, and hence has the effect of specifying the bit sequence corresponding to the given hexadecimal number to appear in the full width of the 32-bit word. Since 8 hexadecimal digits can fit into a 32-bit word, the given hexadecimal number cannot exceed 8 characters, and, if less, has zeros appended at the left. Note that the necessity for including this type of constant is implementation dependent.

Figure 2.3 illustrates the mapping of the various constants into a 32-bit word. Note that integers have a maximum permitted value of 536 870 911 (2^{29}-1), which, as an octal number is *377 777 777 7, and as a hexadecimal number is #1ff fff ff. Numbers less than these values have zeros appended at the left i.e. #fff4f is an abbreviation for #000 fff 4f etc.

A program uses identifiers to denote scalars. An identifier has the syntax : [IDENTIFIER] = [SMALL LETTER*] ≤ [DIGIT*?]

e.g. pl2 , a , stack top

Another use of an identifier is as a synonym for a constant, in which case the identifier is then referred to as a "semantic constant".

All identifiers have associated scope, depending on the context in which the identifier is first declared, and there are four ways to declare an identifier:

1. By its appearance in a GLOBAL declaration statement

2. By its appearance in a LOCAL declaration statement

3. By its appearance in a PRESET ARRAY statement

4. By appearing within any remaining statement form without a prior declaration

Any identifier declared within a GLOBAL block is defined to be in scope from its point of definition to the END OF PROGRAM statement. Any other form of identifier declaration may only occur within the body of a routine, and the identifier is then defined to have local scope, only, within the remainder of that particular routine i.e. it is defined to be in scope from its point of definition up to and including the last statement in that routine.

A declaration statement is defined by the syntax:

[DECLARATION] = [IDENTIFIER] ≡ [DECL RT PART], [IDENTIFIER]

[DECL RT PART] = [IDENTIFIER], [CONSTANT], A[INTEGER]

e.g. a ≡ 12, error condition, register 1≡A1, b ≡ a

The declaration statement allows a program to associate certain attributes with an [IDENTIFIER].

(i)	[IDENTIFIER] ≡ [CONSTANT]	defines the identifier to be a semantic constant having the same value as [CONSTANT].
(ii)	[IDENTIFIER]≡ A[INTEGER]	defines the identifier to be a scalar whose storage location is the general purpose register given by the [INTEGER]. Under this implementation, the [INTEGER] must be in the range 0-15.

(iii)	[IDENTIFIER]	defines the identifier to be a scalar and requests that storage be automatically allocated to hold its value.
(iv)	[IDENTIFIER] =[IDENTIFIER]	defines the identifier on the left hand side to share the same storage location, register, or constant value as the identifier on the right. If the identifier on the right has not been previously declared, then this form of declaration defines both identifiers as scalars which share the same automatically allocated storage location.

A PRESET ARRAY statement is the [IMPERATIVE STATEMENT] having the syntax:-

PRESET ARRAY [IDENTIFIER]([INTEGER]) = [CONSTANT*/ξ,϶]

e.g. PRESET ARRAY n (1) = 12, #ff, #ff00

This statement form is used to declare a one-dimensional array with the name, [IDENTIFIER], whose length is "k", where there are "k" constants listed on the right hand side, and to initialise its elements to the list of defined constants. The initialisation is static i.e. it takes place at compile-time. The [IDENTIFIER] is implicitly defined as a semantic constant. The [INTEGER] specifies how to access the first element of the array. Thus [IDENTIFIER] ([INTEGER]) appearing subsequently in the routine refers to the first element of the array; [IDENTIFIER] ([INTEGER] + 1) to the second element, etc. Note that it is possible to subsequently assign new values to preset array elements if so desired (see §5.9).

The appearance of an undeclared [IDENTIFIER] in any remaining statement form declares it to be a local scalar and storage for it is automatically allocated.

Note that:

(i) If an identifier is currently in scope and is then explicitly declared using a declaration statement or PRESET ARRAY statement, a double-declaration fault is monitored.

(ii) Because of the storage allocation philosophy adopted by the compiler, all local scalars requiring automatic storage allocation are, in effect, "own".

2.3 Variables and Expressions

An RCCT variable is defined by the syntax in Appendix A.

e.g. n, n(4), x(y), start of (stack-12),

(#2f), (p), (p-q), (fortytwo + *37), (x-y+16)

If an identifier precedes a bracketed sub-expression, then an equivalent form for the variable can be obtained by including the preceding identifier in the bracketed sub-expression and inserting an addition sign between it and the remainder of the sub-expression.

e.g. n(4) is equivalent to (n+4)

The bracketing notation may be thought of as meaning "contents of location".

e.g. (n+4) means the "contents of location n+4".

Thus, a variable is either a simple identifier, in which case its value is the value of the identifier; or it is in subscripted form, in which case its value will be contained in a run-time storage allocation. The subscripted form of a variable having an identifier preceding a bracketed subexpression is convenient since it looks like, and has the effect of, an array access.

An RCCT expression is defined by the syntax:-

[EXPRESSION] = [Q/(+,-)?][OPERAND][OPTR OPND *?]

[OPTR OPND] = [Q/(+,-,&,|,≠)] [OPERAND],

[Q/(AU, AD, LU, LD)][CONSTANT]

[OPERAND] = [VARIABLE], [CONSTANT]

e.g. -b + 12 & v(2) AU 16 | (x+5), 2, +c AU 2, a

If the first operand in an expression is not preceded by a unary operator (+ or -) then unary + is assumed. The rule for evaluation of an expression is that all operators have equal precedence, and that evaluation is strictly from left to right.

Apart from the unary operators, there are three types of binary operator, and these correspond to the set of machine operations available under the /360:-

(i) addition "+" and subtraction "-".

(ii) logical-and "&", or "|" and exclusive-or "≠".

(iii) the shift operators: "AU" (arithmetic up)
"AD" (arithmetic down),
"LU" (logical up), and
"LD" (logical down).

Note that only a constant may follow a shift operator and, in this context, the constant is assumed to be a positive integral value.

2.4 I/O Facilities

Two basic I/O operations can be specified by the [IMPERATIVE STATEMENT]'s:-

[Q/(READ CARD TO, WRITE LINE FROM)][EXPRESSION]

The READ CARD .... instruction causes the 80 EBCDIC coded characters, which in their packed format on cards occupy 20 words, on the next card image (if there is one) to be copied into a 20-word data area whose address is defined by the value of the [EXPRESSION]. If there is no card image or a Job Control Language Statement is read, execution of the program will terminate, and the message

JOB CANCELLED DUE TO READ ERROR

will be output.

The instruction WRITE LINE.... expects the value of the [EXPRESSION] to be a pointer to a 34-word data area containing a line-printer image, and copies the contents of this area to the line-printer. The 34-word data area must be in the format described in figure 2.4.

Note that the I/O operations are primitive in that all characters are in their EBCDIC codes [1], including the "listing

d

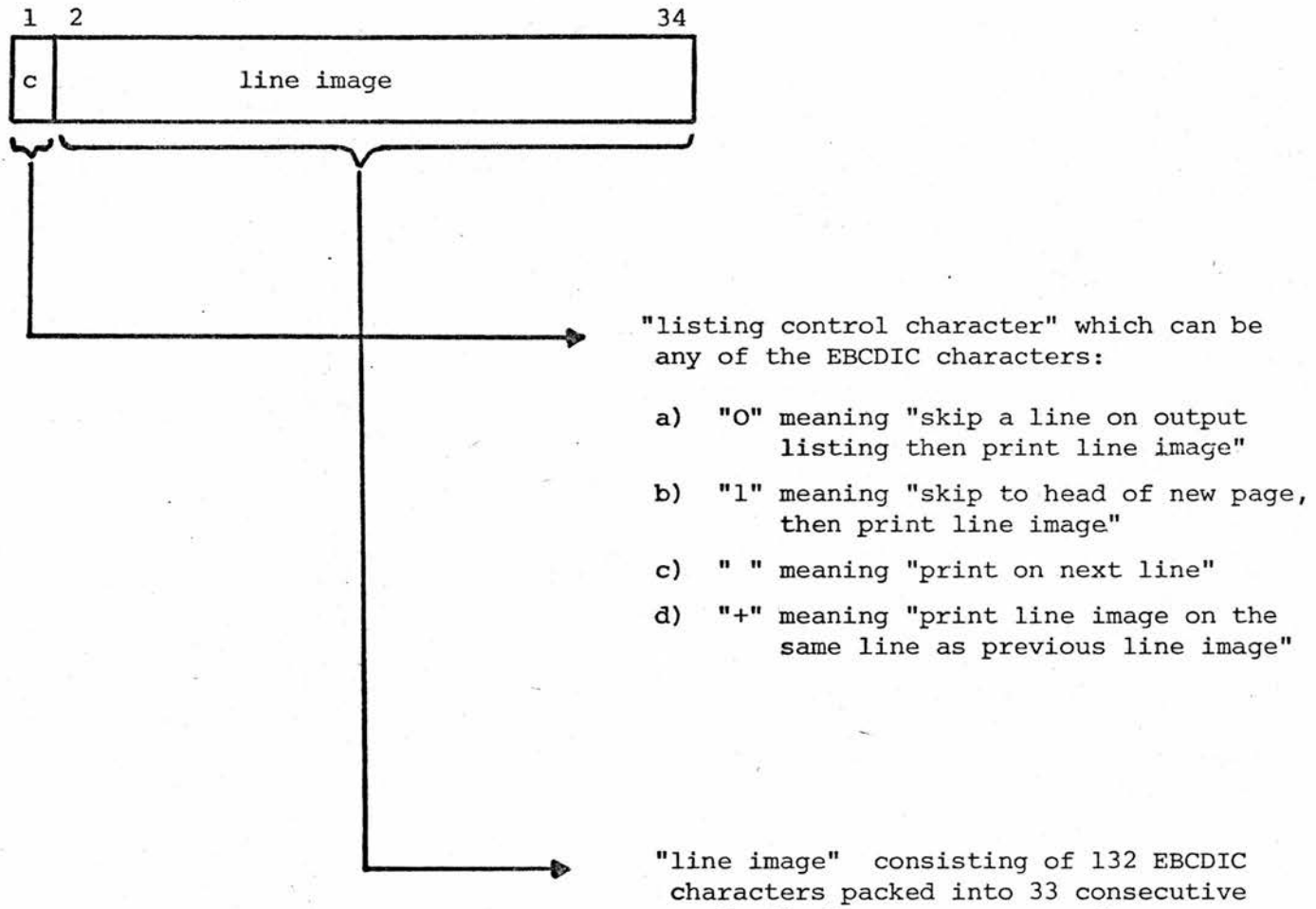


Fig. 2.4 : Data area format expected by a WRITE.... operation

control character" referred to in figure 2.4, and also that READ operations expect input in the form of card images from SYSIPT (the card reader), and WRITE operations copy line images to SYSOPT (the line printer). For further details, see [2].

2.5 Assignment

There is one basic Assignment statement defined by the syntax as the [IMPERATIVE STATEMENT]:-

[P/(SET)?] [VARIABLE]=[EXPRESSION]

e.g. SET x = 12, a(22) = b AU 4 | c AU 4 | (v-5)

The "SET" prefix is syntactic sugar and may be omitted. This instruction causes the value of the left hand side variable to be set to the value of the right hand side expression. Assignments to a semantic constant are not allowed.

A further five instructions are provided for performing special assignments:-

[NEXT] [Q/(*,-)?][IDENTIFIER]

and [SET] [Q/(NO)?] [VARIABLE]

These forms are alternative forms for the following special assignments:-

(i) NEXT [IDENTIFIER]
is equivalent to [IDENTIFIER] = [IDENTIFIER] + 1

(ii) NEXT - [IDENTIFIER]
is equivalent to [IDENTIFIER] = [IDENTIFIER] - 1

(iii) NEXT * [IDENTIFIER]
is equivalent to [IDENTIFIER] = [IDENTIFIER] (1)

(iv) SET [VARIABLE]
is equivalent to [VARIABLE] = 1

(v) SET NO [VARIABLE]
is equivalent to [VARIABLE] = 0

Multiple assignments to be performed in stacking operations are allowed.

(i) STACK [EXPRESSION */ξ,ϑ] AT [IDENTIFIER]
assumes that [IDENTIFIER] is a pointer to the top of a stack, and stores the list of evaluated [EXPRESSION]'s (evaluated in the order in which they appear) in consecutive locations, starting at the top of the stack. The value of [IDENTIFIER] is then updated to point to the next free location at the top of the stack.

(ii) DESTACK [VARIABLE */ξ,ϑ] AT [IDENTIFIER]
specifies the reverse of a STACK .... instruction. If there are "k" variables listed, then this instruction causes the value of [IDENTIFIER] to be decremented by "k", then the set of "k" values stored at the new location pointed to by [IDENTIFIER], are stored into the "k" [VARIABLE]'s in left-to-right sequence.

Note that items are stacked in left-to-right sequence, and destacked in the same order. Hence, the STACK and DESTACK instructions are compatible.

2.6 Control Facilities

Automatic looping control is achieved using the for-statement defined by:-

```
[FOR STATEMENT] = [L*?] FOR [IDENTIFIER] = [EXPRESSION]      ⊆
                  [STEP CLAUSE ?] TO [EXPRESSION]:           ⊆
                  [EOL?][IMPERATIVE STATEMENT */(;[EOL?])]
[STEP CLAUSE]   =  STEP [EXPRESSION]
e.g.  FOR i = 1 TO n : [IMPERATIVE STATEMENT */(;[EOL?])]
      FOR name = -n+4 STEP -12 TO p: [IMPERATIVE STATEMENT    ⊂
                                      */(; EOL?])]
```

This statement defines a sequence of operations, viz the list of [IMPERATIVE STATMENT]'s, which are to be executed repeatedly under the control of the control variable, [IDENTIFIER], whose initial value is the value of the [EXPRESSION] immediately following the "=" symbol. Prior to each execution of the cycle body i.e. list of [IMPERATIVE STATEMENT]'s, the value of the [IDENTIFIER] is compared with the limit [EXPRESSION] to determine whether or not to execute the cycle body. This comparison depends on the sign of the increment (or STEP) [EXPRESSION]. Note that if a [STEP CLAUSE] is not present, the increment is implicitly defined as the expression, +1. If the increment is positive (negative) then control will pass to the next statement succeeding the for-statement when the control variable exceeds (is less than) the limit [EXPRESSION]; otherwise the cycle body is executed, then the increment is added to the control variable, and the comparison made with the limit, as defined above.

The increment and limit expressions are evaluated once only, on entry to the for-statement. Thus the for-statement defines a static cycle statement. Also, note that if the jump-out-of-cycle test succeeds prior to the first execution of the cycle body, it will never be executed.

Two [IMPERATIVE STATEMENT]'s are associated with the for-statement:-

FINISH [Q/(CURRENT, EACH)][IDENTIFIER]

These statements are only defined if the [IDENTIFIER] is a control variable, and the statement(s) appear within the cycle body of a for-statement having the same control variable. The FINISH EACH .... statement causes control to pass to the statement following the specified for-statement, and the FINISH CURRENT ... statement causes control to jump to the end of the current execution of the cycle body.

The if-statement is used to specify alternative actions to be taken, dependent upon a set of conditions. For its syntactic structure see Appendix A.

e.g.
```
IF a=b : SET x
IF a<b ; OR IF c=d;
        OR IF x≠y: x=q:
OTHERWISE: {IF x=0: fred: SET NO q}
```

The list of [IMPERATIVE STATEMENT] immediately following an [IF CLAUSE] is called the "true branch body" of the if-statement, and, if an [OTHERWISE CLAUSE] is present, the list of [IMPERATIVE STATEMENT]'s following it is called the "false branch body" of the if-statement.

A [CONDITION] is interpreted as having a boolean value, either "true" or "false". There are three forms of condition:

(i) the [PHRASE VARIABLE] special condition (see §2.7)

(ii) [EXPRESSION][Q/(=,≠,<,≤,>,≥)][EXPRESSION] which has the value "true" when the relation between the values of the two expressions holds, and "false" otherwise.

(iii) [ROUTINE CALL PART*] which is a call of an IF-routine having the value "true" or "false" dependent upon its execution (see below).

The other three forms of [CONDITION] are alternatives for the following special comparisons:-

(i) [VARIABLE] HAS [IDENTIFIER]
is equivalent to [VARIABLE] & [IDENTIFIER] ≠ 0.

(ii) NO [EXPRESSION]
is equivalent to [EXPRESSION] = 0

(iii) [EXPRESSION]
is equivalent to [EXPRESSION] ≠ 0

The use of "UNLESS" as an alternative to "IF" specifies that the true branch body should be executed when the given condition is false, and the false branch body (if any) should be executed when the condition is true.

e.g. IF a < b is the same as UNLESS a ≥ b

The boolean result of a condition is used to determine whether to execute the "true branch body" or "false branch body" of the if-statement, according to figure 2.5. Thus, the "true branch body" of an if-statement is executed when either a single conditional clause is "true", or any of a sequence of OR-conditions is "true", or when all of a sequence of AND-conditions are "true". Otherwise, the "false branch body" (if there is one) is executed.

The REPEAT instruction is only defined when it is the last instruction inside a compound statement. In this context, it implies that control is to jump to the start of that compound statement.

e.g.
```
{.....
 .....
 .....REPEAT}
```

The REPEAT UNTIL [SIMPLE CONDITION] instruction may also appear as the last instruction within a compound statement and is an alternative for the if-statement:-

UNLESS [SIMPLE CONDITION] : REPEAT

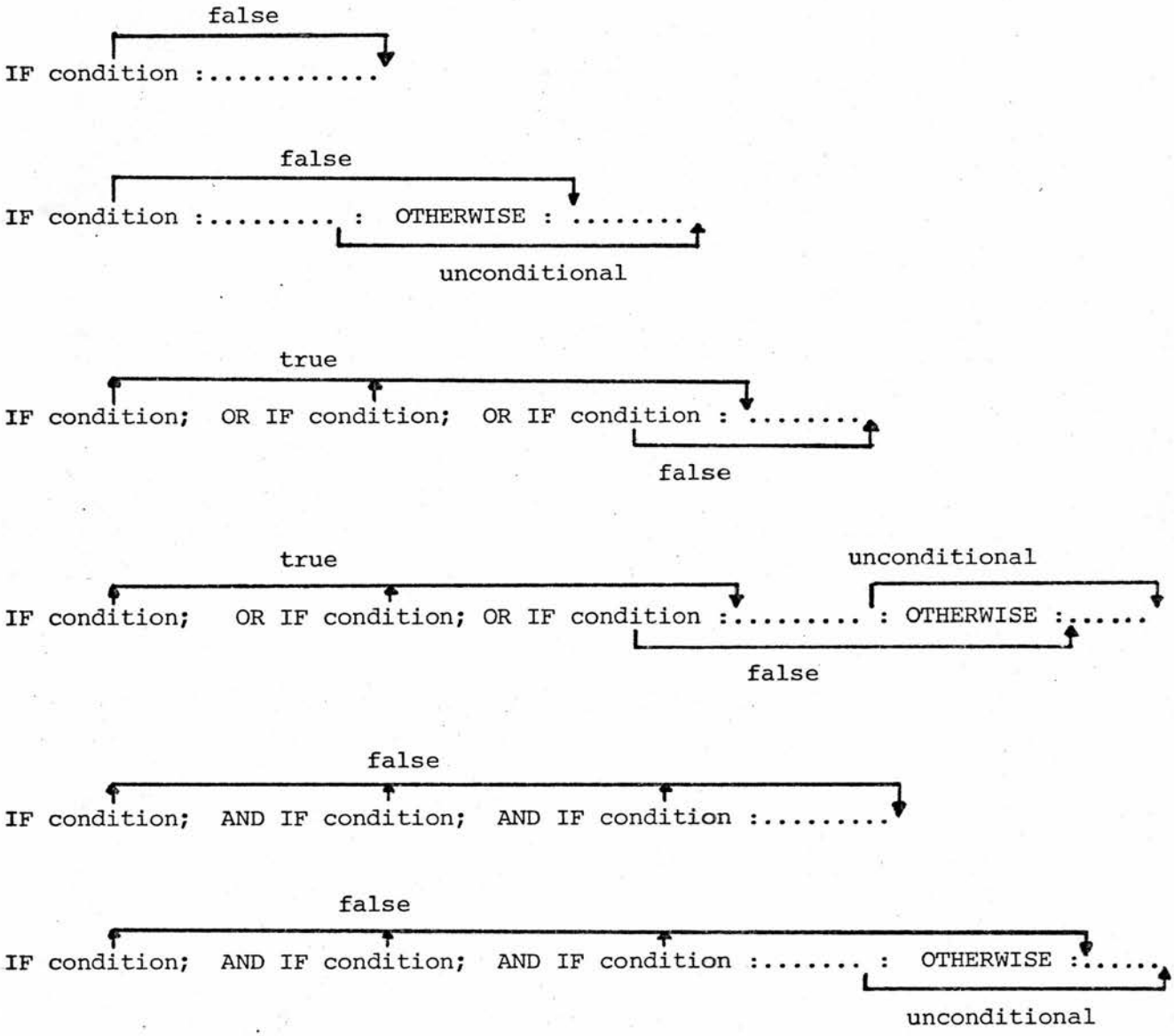


Fig. 2.5 : Control flow through if-statements

e.g. {.....

.....

REPEAT UNTIL x = 0}

Simple go-to control is achieved using either of the equivalent instructions:-

GO [P/(AND,TO)][LABEL IDENTIFIER]

Execution of one of these instructions causes control to be transferred to the instruction immediately following the specified label. Note that a label consists of the same letter/digit sequence as a label identifier, but is underlined.

e.g. loop: .....

..... IF x<y : GO AND loop

i.e. when referring to a label, the label name need not be underlined.

Labels and label identifiers have local scope within a routine. Thus, goto control cannot be used to jump out of a routine.

A label is used to define the address of an instruction. This instruction may be a [CONTROL STATEMENT] or an [IMPERATIVE STATEMENT].

Switching control is achieved using the instruction:-

GOTO [LABEL IDENTIFIER */∈,∃] BY [EXPRESSION]

e.g. GOTO a, b, c BY x+y-z

The [EXPRESSION] must evaluate, at run-time, to a positive integral value in the range 1→k where there are "k" [LABEL IDENTIFIER]'s in the list. Control is then transferred to the corresponding labelled point. The switch statement is undefined when the [EXPRESSION] value is outside the specified range.

The instruction, PERFORM [IDENTIFIER] causes the following sequence of actions to be taken:-

(i) the run-time address of the next sequential instruction after the PERFORM statement is stored into the scalar, [IDENTIFIER].

(ii) control is transferred to the label, [IDENTIFIER].

The instruction, FINISH [IDENTIFIER] assumes that the scalar, [IDENTIFIER], contains the address of an instruction, and transfers control to this address.

The PERFORM/FINISH [IDENTIFIER] instructions are typically used when a sub-routine is required within the body of a routine. RCCT does not provide facilities for defining routines within routines (and all routines are assigned global scope), but a PERFORM sequence may be used for this purpose.

```
e.g. test:    IF x≤y-4&p : SET switch:
                  OTHERWISE : SET NO switch
              FINISH test
       .
       .
       .
       .
     PERFORM test
       .
       .
       .
       .
     PERFORM test
       .
       .
       .
```

Note that this use of labels in no way restricts their use for switching or goto control, but any ambiguities which may arise are left in the hands of the programmer to resolve. Hence, care must be taken that when a FINISH [IDENTIFIER] instruction is executed, the scalar contains a meaningful value.

2.7 Routines

All routines appearing in an RCCT program have a type specification associated with them:-

[RTYPE SPEC] = MAIN, BASIC [Q/(IF, IMP)?], [STANDARD SPEC]

[STANDARD SPEC] = STANDARD [Q/(IF, IMP)?], [Q(IF, IMP)]

e.g.

```
          ROUTINE
IF : [ROUTINE HEADING PART *]
     .
     .
     .
      ROUTINE
STANDARD IMP : [ROUTINE HEADING PART *]
               .
               .
               .
```

A MAIN-routine defines the point at which control will commence when the program is executed i.e. it defines the "main program", and declares, implicitly, that the routine will not be called at any point in the program. The name of a MAIN-routine is at present ignored, although a syntactic check is made that it constitutes a valid [ROUTINE HEADING PART *] (see below).

If a routine is not defined as MAIN then it is defined as

$$\left\{\begin{matrix}\text{BASIC}\\ \underline{\text{STANDARD}}\end{matrix}\right\} \qquad \left\{\begin{matrix}\text{IF}\\ \underline{\text{IMP}}\end{matrix}\right\}$$

where underlining indicates the default values. The basic difference between an IF-routine and an IMP-routine simply describes the manner in which it will be called (an IF-routine being, in effect, a boolean function). The main difference between a BASIC and a STANDARD specification is that at run-time a call of a STANDARD-routine causes a dump of all the general purpose registers to be made onto the run-time stack, registers being unstacked at the end of execution of that routine; a call of a BASIC-routine does not involve any stacking of registers. Thus, at any point in the execution of an RCCT program, the run-time stack contains a trace-back of register contents for each level of STANDARD-routine currently "active".

The scope rule associated with routine names is that each routine has global scope from its first declaration. A routine is declared by the first appearance of a call of it, or its definition, depending which comes first. Any routine which is declared by being called before it is defined is assumed to be a STANDARD-routine with IF/IMP status dependent upon the context of the call. A routine may be redefined, in

which case the former definition is ignored. However, if a routine is defined with a type specification differing from the type specification associated with it when it was declared, a fault is monitored. Thus, in particular, all BASIC-routines must be defined before they are called.

The call of an IF-routine is interpretted as having a boolean result. Apart from this difference, IF-routines and IMP-routines have similar format. A routine heading is defined as a list of [ROUTINE HEADING PART], and a routine call as a list of [ROUTINE CALL PART] (see Appendix A).

e.g. the routine having the heading

```
                ROUTINE
IMP : START i CHAIN, GIVING switch RESULT
                .
                .
                .
```

might be called by, say:

```
                .
                .
                .
          START x+y CHAIN, GIVING @ P RESULT
                .
                .
                .
```

One break with normal conventions in language description is that parameter positions are considered to be part of a routine name, as opposed to the convention of having parameters listed separately. A call of a routine is thus identified by comparing the list of capital letter sequences (delimiter words), commas and parameter positions with any previously defined routines to ascertain whether or not the routine has been previously declared.

e.g. the routine call, above, is identified as the

symbol string:

START ℘ CHAIN, GIVING ℘ RESULT

where "℘" represents the parameter position.

If the routine has already been declared then this symbol string, called the "routine identifier" is known to the compiler, otherwise it is added to the set of routine identifiers currently in scope.

The parameter passing mechanism is "call by value" i.e. at the call, all actual parameters are evaluated and copied into the corresponding formal parameter locations prior to the transfer of control to the called routine. All actual parameters are assumed to be input-only parameters unless they are preceded by an "@" symbol. This symbol is used to specify that an actual parameter is to be treated as a "call by value-result" parameter i.e. on returning from the execution of the routine, all "value-result" parameters, which have input/output status, are set to the values of the corresponding formal parameters. (Note that I/O marked actual parameters must be variables or phrase variables.)

Since the responsibility for passing parameters is undertaken at the call, and the parameter passing mechanism is the same for all parameters, a routine need not know whether parameters are "call by value" or "call by value-result".

To specify dynamic exit from a routine there are three instructions available. An IMP or MAIN routine may contain the instruction:

FINISH

anywhere in its body, which specifies dynamic exit from the routine. Dynamic exit from a MAIN routine defines the end of execution of the "main program". If the FINISH statement is the last executable instruction in an IMP or MAIN routine, it may be omitted.

To denote dynamic exit from an IF-routine, two actions must be taken:

(i) The boolean result of the call of the IF-routine must be specified, and

(ii) Dynamic exit must be specified.

In RCCT, one instruction only is required to perform both actions. To denote the result "true", the instruction

CONDITION SATISFIED

is used, and to denote the result "false", the instruction

CONDITION NOT SATISFIED

is used. If either of these instructions appear within an IMP or MAIN routine, or if a FINISH instruction appears within an IF-routine, then the dynamic exit instructions are undefined, and an error condition will be monitored.

2.8 Phrase Variables

A phrase variable is defined syntactically by:-

[PHRASE VARIABLE] = [[CAPITAL LETTER][PVSYMBOL *?][IDENTIFIER]]

[PVSYMBOL] = NOT [EOL], ;, :,], [SMALL LETTER], BUT [SYSTEM SYMBOL]

e.g. [EXPRESSION e2], [A ?, C12 - A p], [VAR r]

Its inclusion in the language was for compatibility with the RCC phrase variable [3]. Phrase variables are used mainly in conjunction with matching phrase expressions, which are defined by:-

[MATCHING PHRASE EXPRESSION] = [MPE PART *]

[MPE PART] $\neq$ NOT [EOL], ;, :, BUT [PHRASE $\subseteq$ VARIABLE], [SYSTEM SYMBOL]

e.g. [VARIABLE r] = [EXPRESSION e1] * [EXPRESSION e2]

DIVIDE [VARIABLE r1] BY [VARIABLE u2]

XYZ ab/?-A+B

There are three instructions available in the RCCT language for manipulation of phrase variables and matching phrase expressions:-

(i) LET [PHRASE VARIABLE] $\equiv$ [MATCHING PHRASE EXPRESSION]

(ii) SET [PHRASE VARIABLE] ([INTEGER]) = [MATCHING PHRASE EXPRESSION]

(iii) IF [PHRASE VARIABLE] ([INTEGER]) $\equiv$ [MATCHING PHRASE EXPRESSION]

Note that the instruction, RESOLVE [PHRASE VARIABLE] INTO [MATCHING PHRASE EXPRESSION] is simply an alternative way of writing LET [PHRASE VARIABLE] $\equiv$ ....

The sequence of operations performed by each of these instructions is best shown by example:-

(i) LET [VARIABLE v1] ≡ [IDENTIFIER i] ([CONSTANT c])

is exactly equivalent, in code generated, to the sequence of instructions:

```
i = v1 (1); c = v1 (2)
```

(ii) SET [VARIABLE v1] (4) = [IDENTIFIER i] ([CONSTANT c])

is exactly equivalent, in code generated, to the sequence of instructions:

```
v1 = main stack front; STACK 4, i, c AT main stack front
```

(iii) IF [VARIABLE v1] (4) ≡ [IDENTIFIER i] ([CONSTANT c]):...

is exactly equivalent, in code generated, to the sequence of instructions:

```
IF (v1) = 4: LET [VARIABLE v1] ≡ [IDENTIFIER i] ([CONSTANT c]);....
```

in other words:

```
IF (v1) = 4: i = v1 (1); c = v1 (2);....
```

Hence, in RCCT, everything except the embedded identifier in a phrase variable is disregarded. However, in RCC the upper case letter and subsequent characters have a special significance. Note that the SET [PHRASE VARIABLE] = ... instruction assumes that the program has defined a global variable, "main stack front", which points to the top of a stack defined by the program. For further information on the function of phrase variables and matching phrase expressions see [3].

In RCCT, the identifier embedded in a phrase variable may be either a local or a global scalar, depending on where it was first declared. These identifiers are declared in exactly the same way as ordinary identifiers i.e. in explicit declaration statements, or by their appearance without a previous declaration (see §2.2) Thus, the identifier "c" embedded in the phrase variable [CONSTANT c], is the same as the identifier "c" in any other imperative.

e.g.

```
        ROUTINE

IMP :   INITIALISE

        LOCAL: c = A2
        .
        .
        .
        IF [CONSTANT c] = .....
        .
        .
        .
        .
        c = c(1)
        ,
        .
        .
```

CHAPTER 3

LEXICAL ANALYSIS AND COMPILER I/O

3.1 Basic I/O facilities

The primitive I/O routines incorporated into the compiler are the PL360 "READ" and "WRITE" routines. The "READ" routine is passed a single parameter which is a pointer to an 80-byte array called the "card buffer", which is used to hold the image of the card which has just been read in. The "WRITE" routine is passed a single parameter which is the address of a 132-byte array called the "output buffer", and copies the contents of this buffer to the line printer. Each call of the "WRITE" routine causes printing to commence on a new line. The parameter passing mechanism for these routines is explained in [4] and involves special register assignments.

A further set of routines are used to construct line images in the "output buffer", and this set of routines, in conjunction with the "WRITE" routine are referred to as the output routines.

i) The "NEWLINE" routine is passed a single parameter which is a positive integer. The action taken is to call the "WRITE" routine to copy the line image contained in the "output buffer" to the line printer. Then the "output buffer" is filled with EBCDIC blanks and the "output buffer symbol position" pointer, which is used to indicate the next free printing position in the "output buffer" relative to its start, is set to zero. Now, if the input parameter value is greater than 1, a sequence of calls of the "WRITE" routine are made to print the required number of blank lines.

The remaining three routines are used to construct line images:

ii) The "SPACE" routine is passed a single parameter which specifies the number of blanks to be inserted in the "output buffer", starting at the current printing position. The action taken is simply to update "output buffer symbol position" by the specified amount.

iii) The "PRINT" routine is passed a single parameter which is assumed to be a positive or negative integer. This routine inspects the global variable "field width", initialised by the compiler to 10, which defines the number of print positions to be filled. The calling routine can alter the "field width" at any time. The action taken is to copy the positive or negative decimal value of the input parameter, right-justified, into a field whose starting position is defined by the value of "output buffer symbol position", and whose width is defined by the value of "field width". If the value of the parameter is negative, a minus sign is inserted to prefix the value; and if the "field width" is too small to hold the value (possibly with sign), a sequence of asterisks are copied into the field. Finally, "output buffer symbol position" is updated by the value of "field width".

iv) The "PRINTCOLON" routine inspects "myplist" byte-by-byte from left-to-right. The action taken is to copy all bytes (characters), up to but excluding a special non-printable character delimiting the end of the printcolon sequence, into the "output buffer", updating "output buffer symbol position" by 1 for each byte copied.

3.2 The Message Interpreter

The compiler uses the line printer for two purposes:

i) To list the source program

ii) To print diagnostic messages

The output routines described in the previous section are used directly to copy the "card buffer" into the "output buffer" when listing the source program during Lexical Analysis (see § 3.4). However, since

diagnostic messages are generated from a variety of sources during all three phases of compilation, Lexical Analysis, Syntax Analysis and Semantic processing, and a number of similar operations are required to output each diagnostic message, a message interpretation scheme was designed. The basis of this scheme is the "MESSAGE INTERPRETER" routine.

There are three types of diagnostic message at present generated by the compiler:

i) Warning messages, which indicate anamolous conditions, but do not prohibit attempted execution of the program.

ii) Error messages, which indicate that errors have been found which are severe enough to prohibit execution of the program.

iii) End of compilation messages, which summarise diagnostics at the end of compilation.

Any routine in the compiler that wishes to print a diagnostic message, simply passes a code to the "MESSAGE INTERPRETER" before continuing. The function of the "MESSAGE INTERPRETER" is to print the required message, and if the diagnostic is an error message, to set the global variable "compile time fault detector", which is a switch initialised at the start of compilation to zero. The setting of the "compile time fault detector" is checked at the end of compilation to decide whether or not a program is executable.

The set of all diagnostic messages, each of which is stored in a separate array, is held in a data area called "messages", which can only be accessed by the "MESSAGE INTERPRETER". Each message has an associated serial number (see Appendix B), and it is this value which is passed to the "MESSAGE INTERPRETER". In addition, the type of message is specified by:

i) setting the most significant bit of the coded serial

number, to indicate a warning message.

ii) setting the second most significant bit of the coded serial number, to indicate an end of compilation message.

iii) not setting any additional bits, to indicate an error message.

Thus, the Lexical Analyser, part of whose function is to read in cards and copy their contents to the line printer, together with the "MESSAGE INTERPRETER" are the only two routines in the compiler which have access to the output routines. All other routines use the "MESSAGE INTERPRETER" as a communications interface between them and the line printer.

3.3 Card Input Conventions

The RCCT symbol set is a subset of the RCC symbol set and is summarised in figure 2.1. Immediate problems arose as to how to represent some of these symbols, since they are not printable or punchable as single symbols on cards. Figure 3.1 outlines the symbol concatenation conventions adopted to obviate this problem.

There remained two problems:

i) Small letters are not directly punchable on cards

ii) Underlining of RCCT master words and of labels is not directly punchable on cards.

To distinguish between upper and lower case letters, the single quote symbol (') has been reserved. When it appears on a card it is ignored, since it is not defined as an RCCT symbol, but its appearance alters the case of all subsequent letters, only, appearing between it

RCCT symbol	IBM card representation	
{	(<	
}	>)	
[	<_	
]	>_	
≠	/=	
$\underline{=}$	=_	
$\underline{\neq}$	'XOR'	
≤	<=	
≥	>=	
→	->	
$\underline{c}$	c_	no blanks allowed between these special symbol concatenations
$\underline{(}$	(_	
$\underline{)}$	)_	

Fig. 3.1 : Symbol conventions for card image input

and the next (if any) single quote symbol. By default, when a card is being read in, all letters are assumed to be initially in lower case. Note that the quote symbol only affects the case of letters, and hence all other symbols are not affected.

e.g. the RCCT lines:

```
IF stacktop-12≤p: GO AND terminate loop
x=y; STACK x-4 AT stacktop
```

would be punched on cards as:

```
'IF' STACKTOP-12<=P: 'GO AND' TERMINATE LOOP
X=Y; 'STACK' X-4 'AT' STACKTOP
```

Note that:

i) In figure 3.1, the exclusive-or logical operator is represented by the uppercase string of letters, XOR, which is why it has been enclosed in quotes.

ii) The RCCT continuation symbol is defined as the symbol sequence "C_" without any intervening blanks. When used in this context, the letter "C" is unaffected by the use of the single quote. The appearance of this symbol sequence at the end of a card signifies that the line continues onto the next card, and that the case of the lettering at the beginning of the continuation card is the same as at the end of the first card.

e.g

```
LOCAL: a = b, c,      c
       x = 12, y = A2
```

would be coded on cards as:

```
'LOCAL': A=_B, C,          C_
         X=_12, Y =_'A'12
```

iii) Since the quote symbol only affects the case of lettering and is otherwise ignored then 'X5' is equivalent to 'X'5.

Hence, the quote symbol is needed to define and delimit uppercase letter strings.

The convention adopted for underlining is only used in conjunction with masterwords and labels, and consists of using an underline symbol (_) as a prefix and suffix to the desired symbol string.

e.g. i) fred:.....

would be coded on cards as:-

FRED:

ii) GLOBAL would be coded on cards as: '_GLOBAL_' or

'GLOBAL' etc.

In addition to the above conventions adopted for coding RCCT programs on cards, the compiler also has facilities for recognising compiler directives. (At present there is only one.) A compiler directive is used to control the listing output generated by the compiler. Its format consists of a "£" symbol, in the first column of any card, followed by a directive occupying the next six columns. (The only directive at present consists of a sequence of blanks.) The remainder of the card is assumed to be non-significant and is ignored. The single directive available at present causes a dump of all the object code generated for the program at the end of compilation.

Finally, only the first 72 columns of a card are assumed significant. The remaining 8 columns may be used for sequencing numbers etc.

3.4 Lexical Analysis

The function of the Lexical Analysis routine is to read in a single line of RCCT program, consisting of either a single card image, or sequence of card images when continuation symbols are present, and to

convert the line image into a form suitable for processing by the Syntax Analyser.

Card images are copied into the "card buffer" using the "READ" routine, and the processed text is copied into a 256-byte array called the "input buffer", which is the input source for the Syntax Analyser. The global variable "input symbol position" is used locally by the Lexical Analyser to indicate the next free position within the "input buffer" relative to its start. On entry to the Lexical Analyser, "input symbol position" is set to zero, and on exit it is set to point to the base of the "input buffer". The value of "input symbol position" is then used by the Syntax Analyser to scan the "input buffer". The textual processing involved in the transmission of data (illustrated in figure 3.2) is concerned with intercepting compiler directives; eliminating blanks and comments, which are assumed non-significant; interpretting which case letters should be; intercepting master words; and intercepting continuation symbols.

The Lexical Analysis routine obeys Algorithm 3.1.

Algorithm 3.1 (Lexical Analysis)

1. Initialise a) "input symbol position" = 0

   b) local variable "quote mark" = 0 to signify letter transmission is in lower case

   c) local variable "comment count" = 0 to signify that the Analyser is in "transmission mode" rather than "comment mode".

   d) local variable "quote error" = 0.

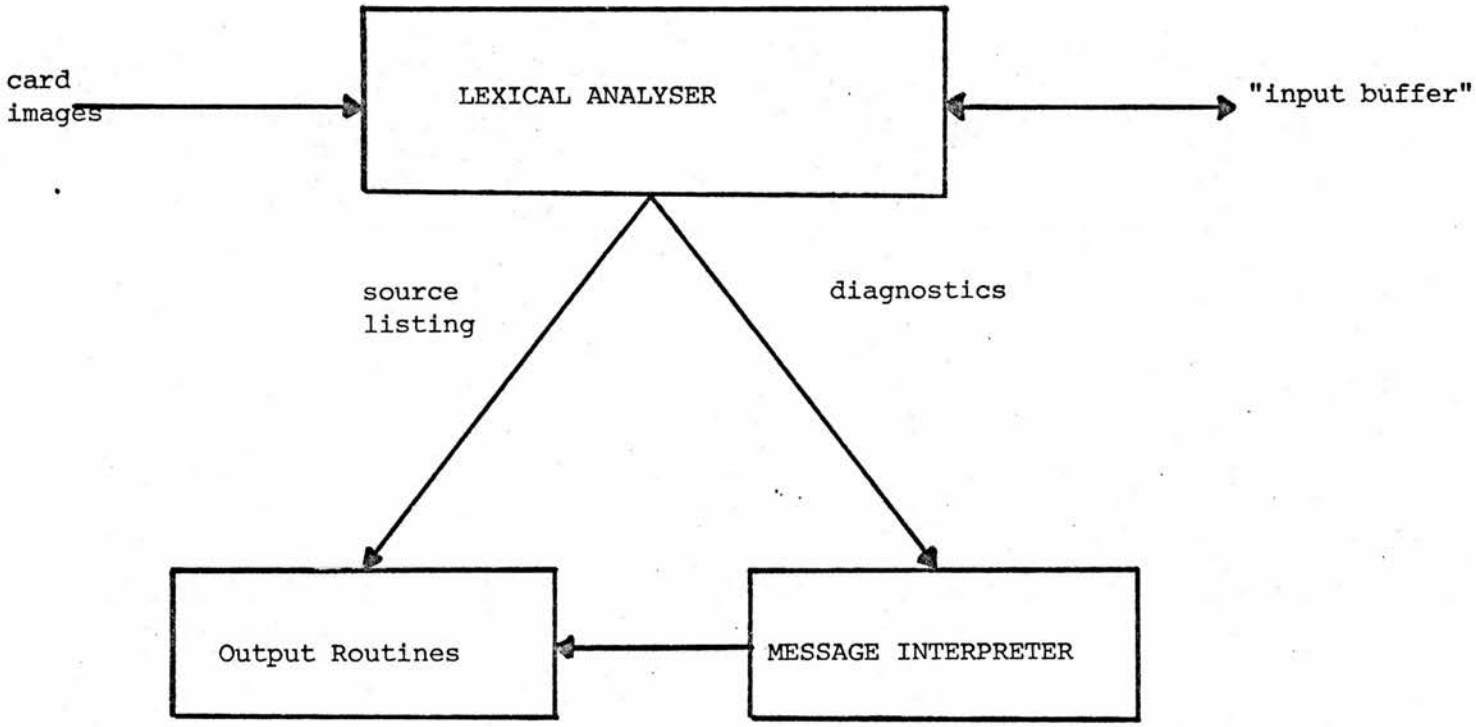


Fig. 3.2 : Information flow through Lexical Analyser

2. Read a card image into the "card buffer" and test for an erroneous read-operation. If an error occurred then set global variable "endmet" = 1 and finish, otherwise continue. ("endmet" is initialised at the start of compilation to zero).

3. If the card contains a compiler directive, then take the appropriate action and goto 2.

4. Scan the symbols in the "card buffer" as follows: ignoring the last 8 characters on the card image:-

   a) If the symbol sequence "C_" appears then inspect the remainder of the card image. If further symbols remain on the card, goto 4d; otherwise

      i) Copy the "card buffer" to the line printer

      ii) Go to 2. to read in the next card

   b) If the symbol sequence "(_" appears, then add 1 to "comment count" to signify that the Analyser is in "comment mode", skip the two symbols and continue scanning the "card buffer" i.e. go back to 4.

   c) If the symbol sequence ")_" appears, then subtract 1 from "current count" and test whether "comment count" < 0. If so, then:

      i) Copy the "card buffer" to the line printer

      ii) Output the diagnostic error message: COMMENT BRACKETING

      iii) Go to 1. to restart Lexical Analysis.

      otherwise, ignore the two symbols and go to 4. to continue scanning the "card buffer".

   d) If in "comment mode" ("comment count" ≠ 0) then ignore the symbol currently being scanned and goto 4.

      If in "transmission mode" ("comment count" = 0) then:

      i) If the symbol being scanned is a single quote, invert "quote mode" by exclusive-or'ing it with 1, ignore the quote, and continue scanning at 4.

ii) If the symbol being scanned is a blank, then ignore it and go to 4.

iii) If the symbol being scanned is a letter, then use the value of "quote mode" to decide whether or not to convert it to lower case.

iv) Copy the symbol into the "input buffer", incrementing "input symbol position" by 1. If the "input buffer" has overflowed ("input symbol position" = 256) then copy the "card buffer" to the line printer, print the error diagnostic message STATEMENT LENGTH, and goto 1 to restart Lexical Analysis; otherwise goto 4. to continue scanning card.

5. When a complete line of input has been processed, copy the [EOL] character into the "input buffer" and test for anomalies:-

a) if "quote mode" $\neq$ 0 then set "quote error" = 1

b) if "comment count" $\neq$ 0 then:

i) Copy the "card buffer" to the line printer

ii) Print the diagnostic error message COMMENT BRACKETING

iii) Goto 1. to restart Lexical Analysis

c) if "input symbol position" = 0 then a blank card, or card full of comments has been read, so copy "card buffer" and go to 1. to restart Lexical Analysis.

6. Inspect "card buffer" for master word. Master words must "stand alone" on a line, and hence, if one has been read, it will be the only item in the "input buffer":

a) if GLOBAL or ROUTINE is present then:

i) Substitute an internal code for the master word as the first character in the "input buffer" followed by the [EOL] character, and update "input symbol position" accordingly.

ii) Call the "FINISH OFF PREVIOUS ROUTINE" (see § 5.5)

iii) Print two blank lines

iv) Copy the "card buffer" to the line printer

v) If "quote error" = 1, print diagnostic warning message ODD QUOTECOUNT and set "quote error" = 0.

vi) Go to 2.

b) If END OF PROGRAM is present then:

i) Call the "FINISH OFF PREVIOUS ROUTINE"

ii) Print 2 blank lines

iii) Copy the "card buffer" to the line printer

iv) Test "quote error" as in 6 a)v)

v) Call the "TRANSFER CONTROL" routine (see § 5.16)

7. Copy the "card buffer" to the line printer, and test "quote error" as in 6a)v).

8. Set "input symbol position" to the base address of the "input buffer", and finish.

From Algorithm 3.1 it can be seen that the Lexical Analyser assigns priorities to its input:

i) Compiler directives (which are not printed)

ii) Continuation symbols

iii) Comment brackets

iv) Other symbols

Listing control causes 2 blank lines to appear between consecutive RCCT blocks. In addition, the global variable "line number", initialised at the start of compilation to zero, is updated by 1 each time a card image is copied to the line printer, and the value of "line number" is printed as a prefix to the line.

Note that the EBCDIC character set includes the set of lower case letters, although they are not directly punchable on cards [1]. When upper case letters are converted to lower case, it is the EBCDIC lower case character set that is used.

CHAPTER 4

SYNTAX ANALYSIS

4.1 Introduction

The Syntax Analyser of the RCCT compiler is concerned with the flow of information described in figure 4.1. Its constituents may be broken down into four parts:

(i) the parsing routine
(ii) the Format (or Syntax) Table
(iii) the set of Recognition routines
(iv) the Recognition Table

Parsing is achieved using the "PERFORM SYNTAX ANALYSIS GIVING RESULT r" routine, which follows a top-down, fast back parsing algorithm, in attempting to match the symbols in the "input buffer", pointed to by "input symbol position", against any valid [RCCT PIECE] as described in Appendix C. The result of a successful parse is a one-level parse tree (simple enough to be also referred to as a parse chain) stored on the "Analysis Stack", together with an integer code "r" giving an indication of which [RCCT PIECE] was recognised. If this integer code is negative then this indicates that parsing was unsuccessful.

The Format Table holds the set of valid [RCCT PIECE]'s that may appear in an RCCT program (in a suitable internal form).

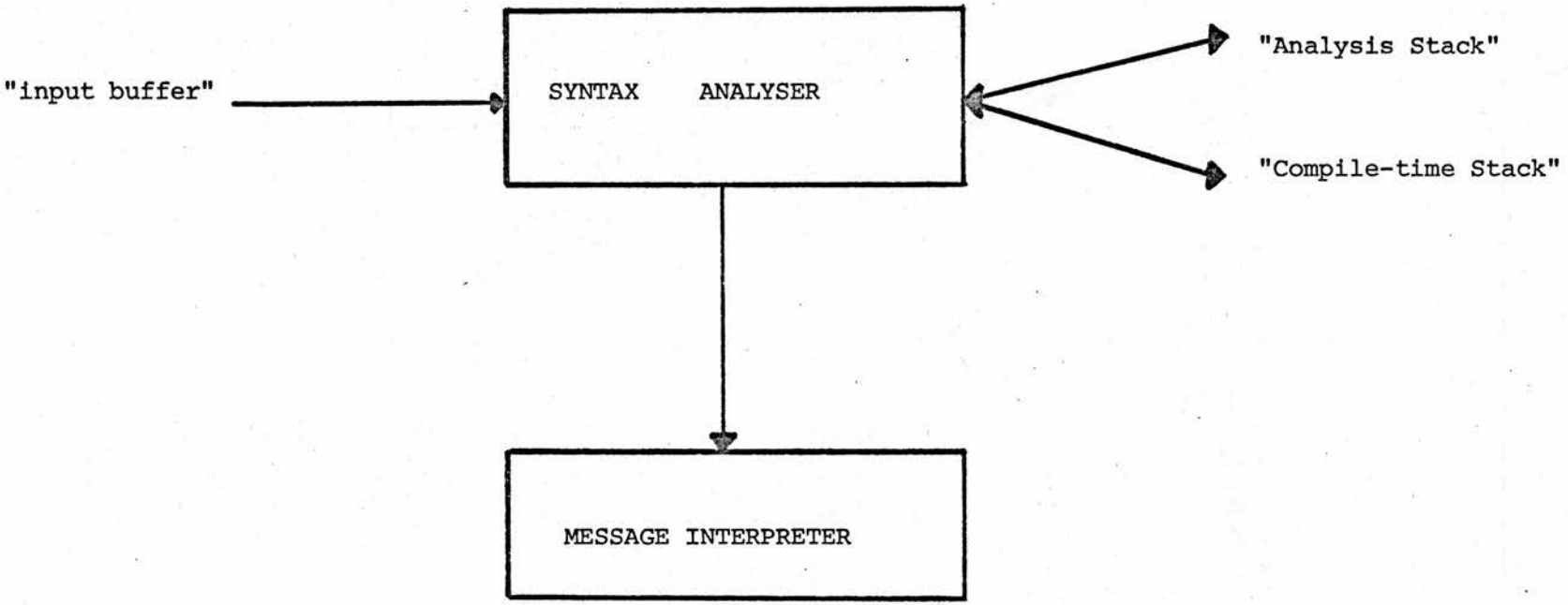


Fig. 4.1 : Information flow through Syntax Analyser

To each syntactic class in the syntax of an [RCCT PIECE] there corresponds a Recognition routine. This routine scans the "input buffer", starting from "input symbol position", and attempts to recognise a sentence of the syntactic class associated with it. If successful, it also constructs the nodes of a parse tree on the "Analysis Stack". Non-recognition of the class sets a global switch, "recognised", to zero, and leaves "input symbol position" unchanged: recognition sets "recognised" to one and updates "input symbol position" to point to the next symbol on the "input buffer" after the recognised class.

The Recognition Table holds the (relative) base addresses of the Recognition routines, and entry to them is achieved via this table during Syntax Analysis.

When Recognition routines scan the various names that can appear in a program i.e. identifiers, label identifiers and routine identifiers, where, in this context, a routine identifier consists of the set of delimiter words, commas and parameter positions in the order in which they appear in the routine heading or call, the symbols constituting the name are packed up onto the "Analysis Stack" in the packed format unique to that particular name. Figure 4.2 illustrates the structure of a packed name format and shows how the first byte of the first word of the packed

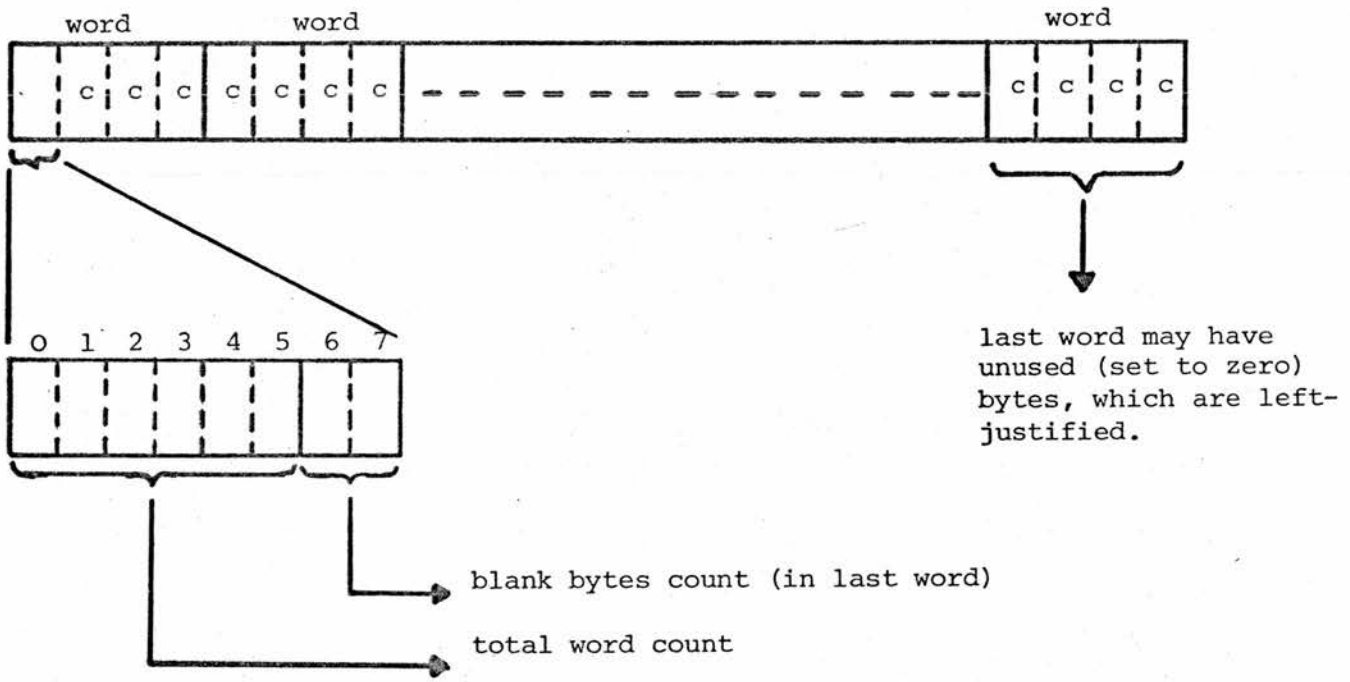


Fig. 4.2 : Packed name format

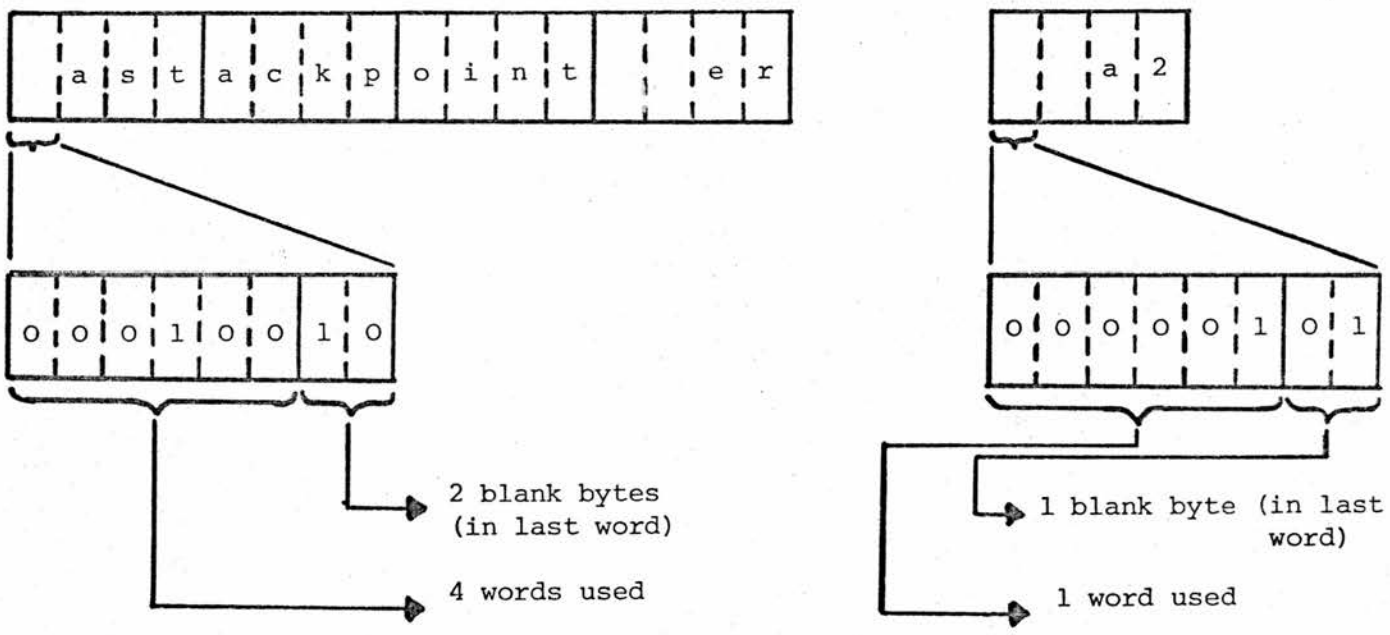


Fig. 4.3 : Two examples of packed identifiers

name format holds a code consisting of:

(i) total number of words used for packed name
(ii) total number of unused (blank) bytes in the last word

The total word count is required when copying a packed name or comparing it with another, and the number of blank bytes indicates how many bytes are unused in the last word of the packed name (where unused bytes are left-justified in that word and contain zeros) and is used to speed up non-recognition when comparing two packed names. Figure 4.3 gives two examples of packed name formats.

Partial semantic processing of names is also performed by the name Recognition routines. This processing has the function of declaring (undeclared) names to the compiler by making additions to a set of compile-time dictionaries maintained on the "Compile Time Stack". There are three dictionaries referenced during Syntax Analysis:

(i) the identifier dictionary, which holds the set of all identifiers currently in scope (both global and local)

(ii) the label dictionary, which holds the set of all label identifiers currently in scope

(iii) the routine dictionary, which holds the set of all routine identifiers currently in scope

In addition, each element in these dictionaries holds the synonymous plexform of the particular identifier. A nameplex is the processed form of a name which may be substituted for it to facilitate further processing. Each identifier, label identifier

and routine identifier has an associated plex form which holds all the information required during the semantic (code generation) phase of the compilation. Identifier and label identifier plexes are two words in length, and routine identifier plexes are three words in length.

4.2 Plexes

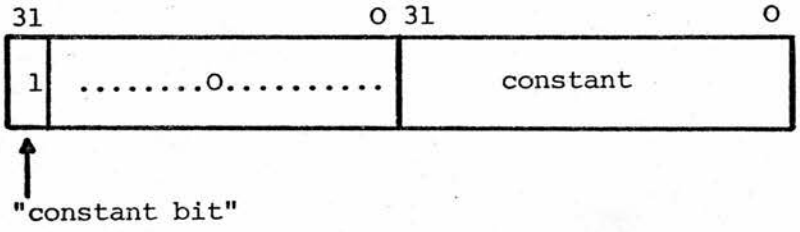
The two basic forms of plex are that of a constant and an identifier. Their formats are summarised in figure 4.4a and 4.4b. A constant plex consists of two consecutive words in which the most significant bit of the first word is set, with the remainder of that word containing zeros, and the second word contains the value of the constant. An identifier plex associates an attribute with a particular identifier. There are four such attributes:

(i) a scalar requiring a location to be automatically allocated to hold its value, in which case no tag bits are set in the plex, the "constant part" = 0 and the "store part" gives the displacement from the start of the run-time "scalar area" where the scalar will be held during execution of the program. (see §5.1).

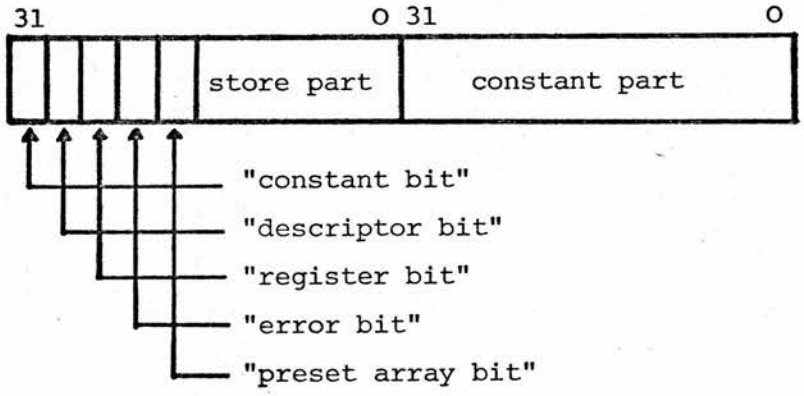
(ii) a scalar requiring a specific register to be allocated to hold its value, in which case the "register bit" of the plex is set, the "store part" indicates with which register the identifier is synonymous, and the "constant part" = 0.

(iii) a constant, in which case the plex form is identical to the plex form of the constant.

(iv) preset array constant, in which case the "constant" and "preset array" bits are set to indicate that the identifier is a special semantic constant to be initialised at run time. The "store part" is set to zero, and the "constant part" gives the displacement within the run-time "constant table" where the semantic constant value will be stored at run time. (see §5.1).



a) constant plex



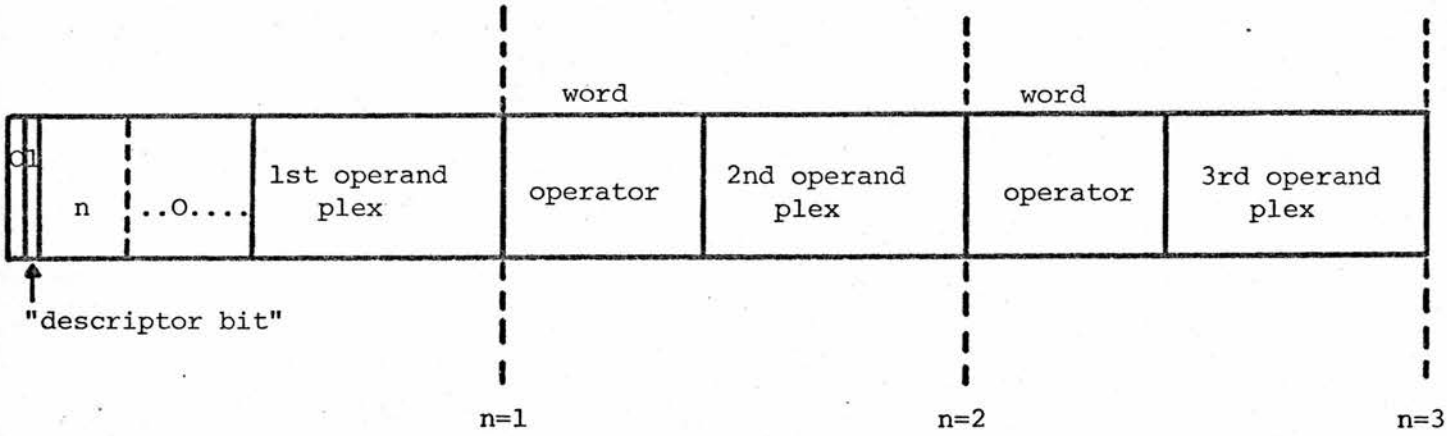
b) identifier plex

Fig. 4.4 : Basic plex forms

Identifier and constant plexes are used to construct the more complicated plexes of a variable and an expression. A variable is either an identifier, in which case its plex form will be that of the corresponding identifier, or it has a subscripted format (see §2.3). The plex of a subscripted format variable consists of a dummy identifier plex which acts as a descriptor for the variable length plex form following. This dummy identifier plex is identified by having the "descriptor bit" set, the "constant part" = 0 and the "store part" = number of identifiers and/or constants involved in the subscripted format variable. This is followed by the set of identifier/constant plexes involved in the subscripted form, separated by single words containing the binary operators (plus or minus) which are simply copied straight from the "input buffer".

Note that the variable, a(20), is equivalent to the variable, (a + 20), so that variables written in array element format must be converted to their equivalent fully enclosed sub-expression (subscripted) format. Figure 4.5a illustrates the format of a subscripted format variable.

An expression consists of a list of duples, where each duple consists of an operator, followed by an operand. The first operand in an expression is associated with a unary operator (plus or minus), which may have optionally been omitted. In this case unary plus is assumed.

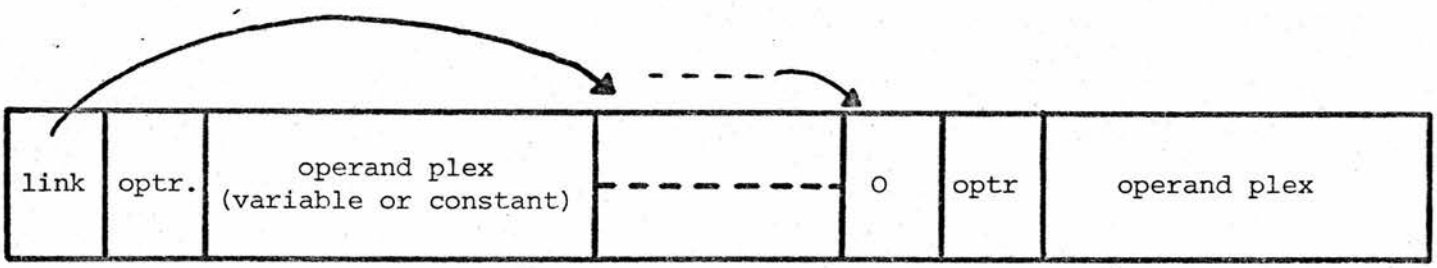


end of plex for three possible values of n

a) subscripted format variable plex

	operator	code
arithmetic (binary or unary)	+	1
	-	2
logical	&	3
	|	4
	XOR	5
shift	AU	6
	AD	7
	LU	8
	LD	9

b) operator coding in expression plex



c) expression plex

Fig. 4.5 : Variable and expression plexes

e.g. (i) the expression: x

consists of the list of duples: (+, x)

(ii) the expression: + a + b LD 2

consists of the list of duples: (+, a), (+, b), (LD, 2)

(iii) the expression: -16 AU 12 + 4 ≠ n

consists of the list of duples: (-, 16), (AU, 12), (+, 4), (≠, n)

The plex of an expression consists of a forward chained list of elements, where each element consists of:

(i) a forward link to the next element

(ii) a duple, consisting of an operator code followed by the plex of the corresponding operand (operator coding conventions are illustrated in figure 4.5b).

The end of the list is defined by an element whose link value is zero. Figure 4.5c illustrates the expression plex format.

The RCCT phrase variable and matching phrase expression were introduced for compatibility with the RCC structures of the same name but are handled in a different manner. A phrase variable, in RCCT, has exactly the same plex form as the corresponding identifier embedded in it.

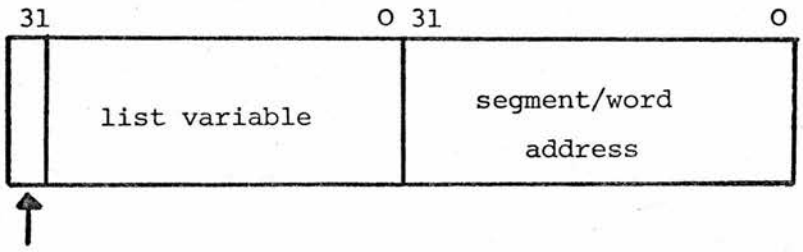
e.g. [EXPRESSION a12] and a12

have the same plex form. Thus, in RCCT, when a phrase variable is reduced to a plex form used for generating code, the phrase and brackets appearing in its structure are discarded.

Similarly, when the plex form of a matching phrase expression is constructed the terminal symbols, and the phrases within constituent phrase variables, are discarded. The resultant plex form consists of a full word containing a count of the number of phrase variables included in the matching phrase expression, followed by the set of phrase variable plexes used.

Labels that appear in a program may be reduced to a two-word plex form containing all the information required during compilation of a particular routine. The label plex format is illustrated in figure 4.6. The "defined bit" is set when the label has been found, and its "address" is then inserted in the plex in the format of a segment number and word-within-segment. The "list variable" gives access to a chain of instructions referencing the label. (The function and use of segment numbers and list variables is discussed further in §5.2 and §5.3).

Figure 4.7 illustrates the format of a routine plex. The "index number" of a routine (also referred to as the "serial number") is the displacement within the run-time "index table" where the base address of the routine will be stored at run-time (see §5.1). The "currently being defined bit" is only set during compilation of that particular routine, and the "defined bit" remains unset until compilation of that particular routine has finished. Three bits are used to code the type specification of a routine and will be set when a routine heading or call is

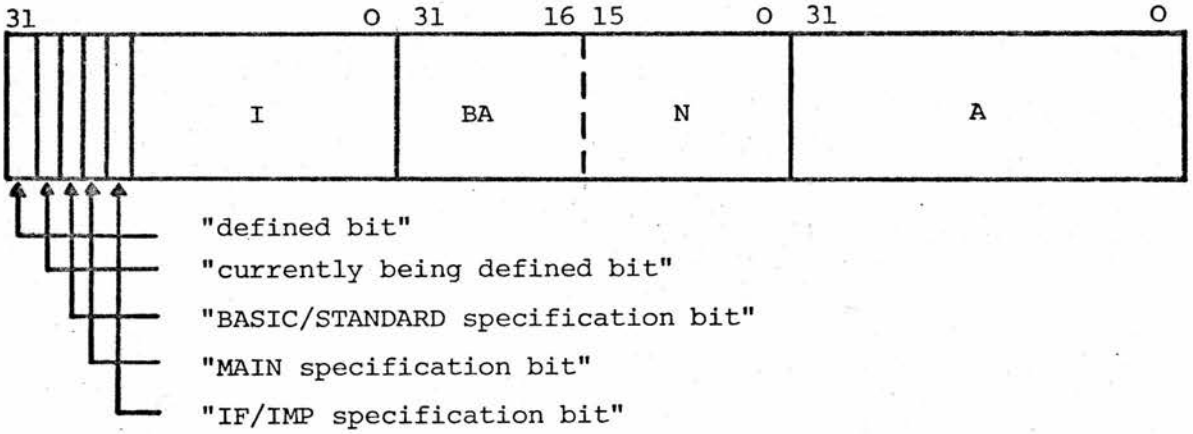


"defined bit"

list variable - points to last label reference

address - split into 16-bit segment and 16-bit word form.

Fig. 4.6 : Label plex format



I - "index number" of routine (also called "serial number")

BA - If routine is BASIC, gives displacement from start of "scalar area" where three-word save area associated with routine is located; otherwise, zero.

N - number of parameters

A - If N ≠ 0 then, displacement from start of "scalar area" where first formal parameter is located; otherwise, zero.

Fig. 4.7 : Routine plex format

first recognised. A routine that is called prior to its definition is assumed to be STANDARD, and its IF/IMP specification is determined by the context of the call. Hence, any BASIC routine must be defined before it is called. Redefinition of routines is allowed, but will generate a fault condition if the definition or redefinition has a different type specification from its previously assumed, or prior specification.

Formal parameters consist of either phrase variables or identifiers, and the associated plex form is thus the plex of a phrase variable or identifier. Actual parameters consist of variables or phrase variables (possibly prefixed by an "@" symbol) or expressions, and thus the plex form of an actual parameter is that of a phrase variable, variable or expression (but see §4.5).

4.3 The Compile-time Dictionaries

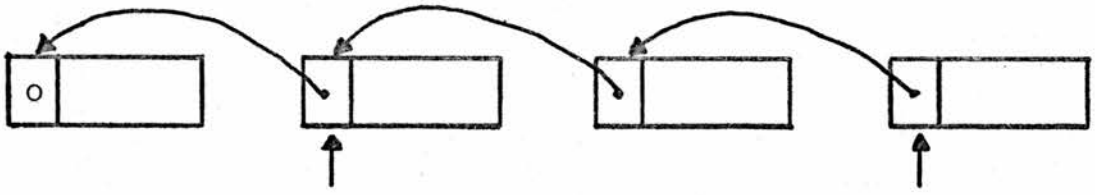
The RCCT programmer is allowed the facility of associating names with storage locations, registers and constants (identifiers); or with specific points within the body of a routine (labels); or with a complete procedure (routine identifiers). One of the main functions of the compiler is to map these names accordingly, and to implement the scope rules associated with them. In addition, any PRESET ARRAY identifiers cannot be mapped completely until the run-time of a particular program, since a PRESET ARRAY declaration is an implicit declaration of a special

semantic constant whose value is not known till run-time. As has been shown, the method of mapping names is to substitute a partially processed plex form for their occurrence, and three dictionaries are maintained on the "Compile Time Stack" to hold the various names, together with their corresponding plexes, during compilation. An additional dictionary is maintained, to hold relocation information associated with the PRESET ARRAY identifiers.

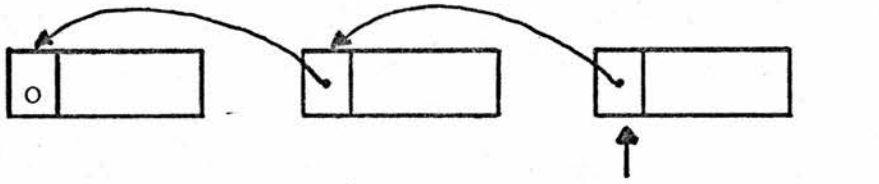
(i) The identifier dictionary, illustrated in figure 4.8a, is sub-divided into the global identifier dictionary, and local identifier dictionary. It consists of a backward chained linked list whose elements contain a link, an identifier plex and a packed name form in consecutive words. The last element in the list has a zero link. Two global variables, "global dictionary head" and "dictionary head", point to the head of the global identifier dictionary and local identifier dictionary, respectively. Any additions to the identifier dictionary involve updating "dictionary head", and, if the additions are made as a result of a global declaration, "global dictionary head" is then altered to point to "dictionary head". The implementation of the scope rules associated with identifiers is as follows. At any point in the compilation of an RCCT program, the compiler will have accumulated a set of identifiers in the identifier dictionary, and this set of identifiers defines those currently in scope. At the end of compilation of a routine, "dictionary head" is retracted to point to "global dictionary head", thus deleting all the identifiers having local scope within the body of the completed routine.

Initially, "global dictionary head" and "dictionary head" are set to zero, indicating that the identifier dictionary is empty.

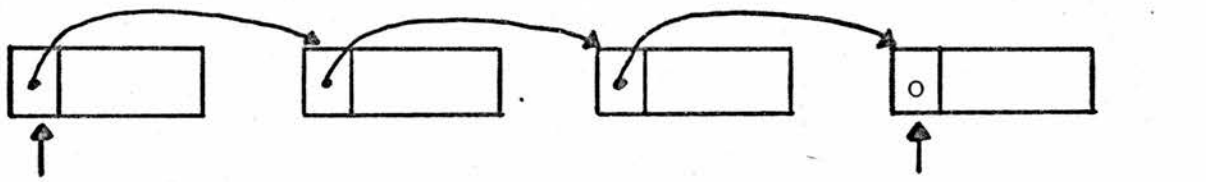
(ii) The label dictionary, illustrated in figure 4.8b, is also a backward chained linked list whose elements have the format of a link, label plex and packed label name in consecutive words, where a zero link defines the last entry. The global variable "label dictionary head", initialised to zero at the start of compilation of each routine, points



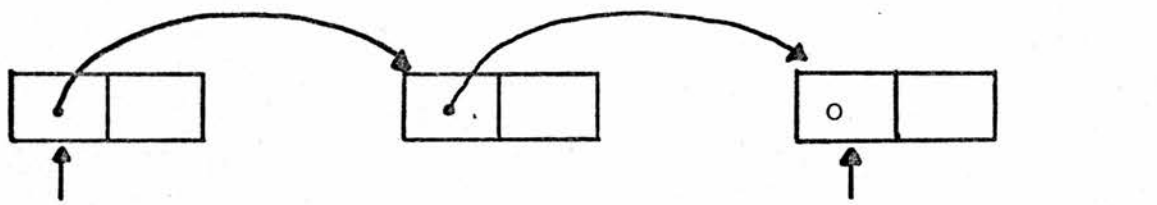
a) Identifier Dictionary



b) Label Dictionary



c) Routine Dictionary



d) Preset Array Dictionary

Fig. 4.8 : Compile-time Dictionary formats

to the most recent addition to the list. Implementation of the scope rules attached to labels is achieved by resetting "label dictionary head" to zero at the end of compilation of each routine, implying that labels only have scope within the routine in which they appear.

(iii) The routine dictionary, illustrated in figure 4.8c, is organized as a forward chained linked list whose elements are again in the format of a link, a routine plex and a packed routine name, where the last element in the list is defined by having a zero link. The two global variables, "routine dictionary head" and "routine dictionary tail", point to the first and last (most recently added) elements in the list, respectively, and are initialised at the start of compilation to zero.

Routines have global scope in an RCCT program, so the routine dictionary is a strictly expanding item. The use of forward chaining is to facilitate garbage collection on the "Compile Time Stack" that may be required at the end of compilation of each routine. The pointer, "routine dictionary tail" is necessary to speed up the operation of adding a new element to the routine dictionary. (see §5.5).

(iv) The special purpose Preset array dictionary, illustrated in figure 4.8d, has the same organisation as the routine dictionary but its elements have a different format, namely, a link and a fullword containing relocation information required to evaluate the absolute address constant corresponding to the preset array identifier and to store it at the correct run-time location prior to execution.

The two global variables used to implement the preset array dictionary are "preset array dictionary head" and "preset array dictionary tail", and are initialised at the start of compilation to zero. This dictionary is maintained in the same manner as the routine dictionary.

The relocation information held in the preset array dictionary will be required immediately prior to execution of the program. Hence, this dictionary is also a strictly expanding item at compile time of an RCCT program.

(For further details on preset array handling, see §5.6.)

The compiler provides four routines to perform elementary operations on the three name dictionaries. These are:

(i) "SEARCH i DICTIONARY FOR x GIVING PLEX ADDRESS a" whose function is to search the identifier (i = 1), label (i = 2) or routine (i = 3) dictionary for a given packed name (pointed to by "x"). The result "a" is either a pointer to the corresponding plex form, or, if zero, implies that the given name has not yet been declared to the compiler, i.e. does not appear on the specified dictionary.

(ii) "ADD x TO i DICTIONARY WITH PLEX ADDRESS a" which is passed a pointer to a packed name "x" and to a plex "a" and specifies that the given name and plex should be added to the identifier (i = 1), label (i = 2) or routine (i = 3) dictionary, respectively, thus declaring it to the compiler.

(iii) "SUBSTITUTE IDENTIFIER PLEX AT p" which specifies that the packed (identifier) name pointed to by "p" should be replaced by its corresponding identifier plex. One of the important side effects of this routine is the automatic allocation of run-time storage locations to hold scalar values. The run-time "scalar area" consists of 1024 full-word locations. At the start of compilation, the compiler initialises a global variable "next free runtime location" to refer to the first free run-time location (i.e. it acts as a relative pointer into the "scalar area"). When this routine is called, the "SEARCH ...." routine is called. If an identifier cannot be found on the identifier dictionary, then a plex is constructed for it, using the value of "next free run time location" and the "ADD ...." routine is called. This global variable is then updated to refer to the next (consecutive) free run-time location. (see § 5.1)

(iv) "SUBSTITUTE RPLEX AT p WITH n PARAMETERS" which specifies that the packed routine name pointed to by "p", having "n" parameters, should be replaced by its corresponding routine plex. (The number of parameters is specified in case a new routine plex has to be constructed).

One of the main functions of this routine is the construction of a new routine plex when the given routine identifier is not in the routine dictionary, i.e. the routine is being declared. The plex

is constructed according to figure 4.7 and involves reserving a location in the "index table" to hold the base address of this routine, and reserving space in the "scalar area" for any formal parameters (see §5.1).

Thus, the two "SUBSTITUTE..." routines may have the side effect of creating a new entry in the identifier or routine dictionaries. There is no equivalent routine to substitute a label plex for a packed label name.

4.4 The Recognition Table

The Recognition Table is organized as an array of elements corresponding to the range of EBCDIC codes for the letters A - Z. Each syntactic class that can appear in the syntax of an [RCCT PIECE] is associated with an alphabetic letter according to figure 4.9. Each letter has an associated position in the Recognition Table determined by its EBCDIC code, and this position gives the location (relative to the start of the code area of the compiler) of the entry point of the particular class (Recognition) routine. Unused entries in the Recognition Table contain zeros. The non-zero elements in the Table are given in figure 4.10.

Access to the Recognition Table is thus by a digit in the range 0 - 40 corresponding to the EBCDIC coded letter, and this then enables access to be made to the associated class (Recognition) routine.

Syntactic Class	Associated letter
[CONSTANT]	c
[DECLARATION]	d
[EXPRESSION]	e
[COMPARATOR]	f
[ROUTINE HEADING]	h
[INTEGER]	i
[LABEL]	l
[MATCHING PHRASE EXPRESSION]	m
[NAME]	n
[PHRASE VARIABLE]	p
[ROUTINE CALL]	r
[CONDITIONAL]	u
[VARIABLE]	v
[TERMINATOR EX NL]	x
[BRACKETS TERMINATOR]	y

Fig. 4.9 : Syntactic class abbreviations

word	Recognition Routine for which entry point is given	letter
0	0	A
1	0	B
2	CONSTANT	C
3	DECLARATION	D
4	EXPRESSION	E
5	COMPARATOR	F
6	0	G
7	ROUTINE HEADING	H
8	INTEGER	I
	⋮	
16	0	J
17	0	K
18	LABEL	L
19	MATCHING PHRASE EXPRESSION	M
20	NAME	N
21	0	O
22	PHRASE VARIABLE	P
23	0	Q
24	ROUTINE CALL	R
	⋮	
33	0	S
34	0	T
35	CONDITIONAL	U
36	VARIABLE	V
37	0	W
38	TERMINATOR EX NL	X
39	BRACKETS TERMINATOR	Y
40	0	Z

Fig. 4.10 : Recognition Table

4.5: The Recognition Routines

There are nineteen Recognition routines in all, and these are:

(i) "NAME"
(ii) "LABEL"
(iii) "CONSTANT"
(iv) "INTEGER"
(v) "DECLARATION"
(vi) "VARIABLE"
(vii) "EXPRESSION"
(viii) "PHRASE VARIABLE"
(ix) "MATCHING PHRASE EXPRESSION"
(x) "COMPARATOR"
(xi) "CONDITIONAL"
(xii) "TERM EX NL"
(xiii) "BRACKETS TERMINATOR"
(xiv) "ROUTINE HEADING"
(xv) "ROUTINE CALL"
(xvi) "SPECIAL NAME"
(xvii) "SHIFT OPERATOR"
(xviii) "OPERATOR"
(xix) "TERMINATOR"

They are all written as conventional routines and have three common functions:

(i) To scan the symbols in the "input buffer" in an attempt to match them against a particular syntactic class. This involves manipulating "input symbol position" in such a way that if the class is not recognised "input symbol position" has the same value on routine exit as it did on entry, and, if the class is recognised, "input symbol position" is set on exit to point to the next symbol position after the recognised syntactic sentence.

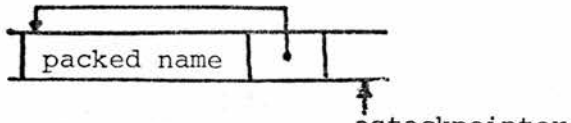
(ii) To leave a result on the "Analysis Stack". A global pointer, "astackpointer" is used for this purpose. In general, this points to the next free location at the top of the "Analysis Stack". If a given class is not recognised, then "astackpointer" must be reset on routine exit to its value on entry. If the class is recognised, then it must be updated to point to the next free location at the top of the stack.

(iii) To set the global "recognised" switch, just prior to exit from the routine to 0, if the class was not recognised, or to 1 if the class was recognised.

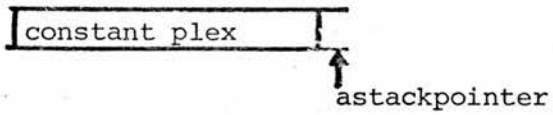
RCCT Recognition routines may call one another (but not recursively). The results, left on the "Analysis Stack" after a successful call of each Recognition routine are illustrated diagramatically in figure 4.11.

Two Recognition routines are provided for scanning a syntactic [NAME], viz. the "NAME" and "SPECIAL NAME" routines. [Note that identifiers and label identifiers have the same syntactic definition (see Formal Syntax in Appendix A), so in the implementation these have been combined into a single syntactic class, [NAME] (see Appendix C).] Both of these Recognition routines scan for a syntactic [NAME], but, whereas the "NAME" routine simply puts the packed (identifier or label identifier) name form on the "Analysis Stack", the "SPECIAL NAME" routine performs some partial semantic processing by substituting the corresponding identifier plex on the "Analysis Stack" (see figure 4.11). This, in effect, declares the syntactic [NAME] as an identifier to the compiler. Hence, the "NAME" routine is invoked during Syntax Analysis to scan the "input buffer" for either a label identifier or an identifier, and simply to place a packed name form on the "Analysis Stack", without declaring it to the compiler. The "SPECIAL NAME" routine is invoked when it is desired to scan for an identifier, and to declare the identifier to the compiler.

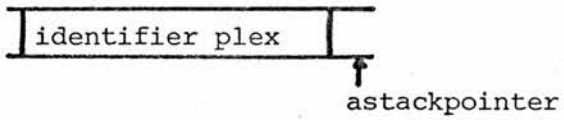
NAME
LABEL



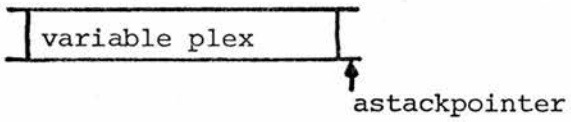
CONSTANT
INTEGER



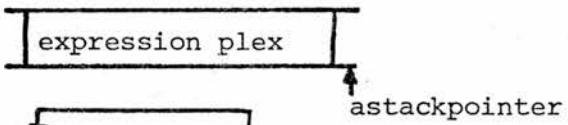
DECLARATION
SPECIAL NAME



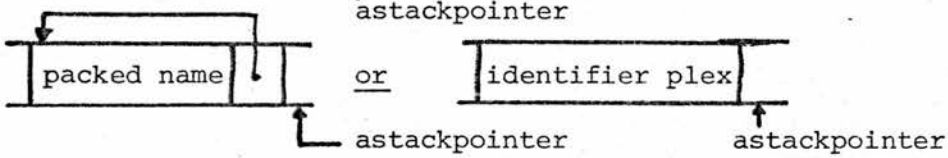
VARIABLE



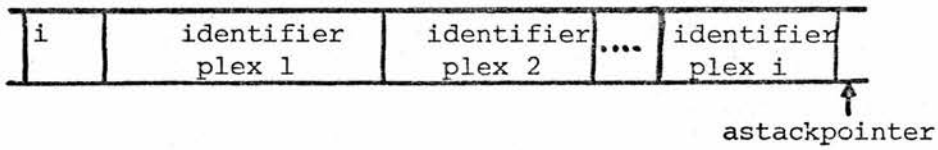
EXPRESSION



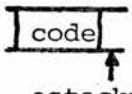
PHRASE VARIABLE



MATCHING PHRASE EXPRESSION



COMPARATOR
CONDITIONAL
SHIFT OPERATOR
OPERATOR



where "code" is given by the following table:

		code
COMPARATOR	=	1
	/=	2
	<	3
	<=	4
	>	5
	>=	6
CONDITIONAL	IF	0
	UNLESS	1
	OR IF	2
	OR UNLESS	3
	AND IF	4
	AND UNLESS	5
OPERATOR	+	1
	-	2
	&	3
	|	4
	XOR	5
SHIFTOP.	AU	6
	AD	7
	LU	8
	LD	9

Fig. 4.11 : Results left on Analysis Stack by Recognition Routines

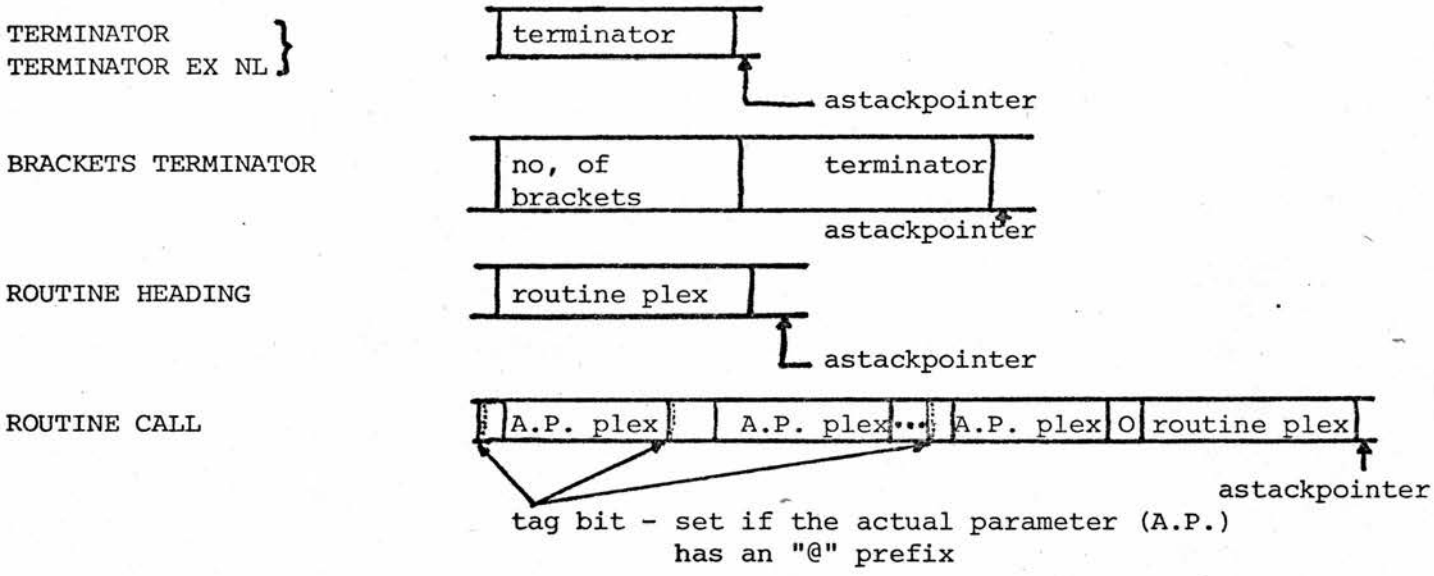


Fig. 4.11 : Results left on Analysis Stack by Recognition Routines (continued)

The recognition of a [LABEL] i.e. the "LABEL" routine, scans the "input buffer" for a syntactic [NAME] parenthesised by underline symbols. The underline symbols are then discarded, and the result left on the "Analysis Stack" is the result of a call of the "NAME" routine viz. a packed (label) name form. Label plexes are not used during Syntax Analysis.

In most cases, it is desirable to perform automatic storage allocation during Syntax Analysis, e.g. wherever an expression appears, any undeclared identifiers need to have space reserved for them. However, in the case of a [ROUTINE HEADING] or [ROUTINE CALL], two scans of the syntactic sentence are required:

(i) to check the syntax of the call or heading and construct the plexes of any embedded parameters, and

(ii) to construct the packed routine name form

Two scans are needed because of the admissibility of embedded parameters inside a routine identifier and because of the difficulty of simultaneously constructing two items of varying length (parameter plexes and the packed routine name) on the same stack. Note that a further function of the "ROUTINE HEADING" routine is to check for the presence of a valid routine type specification followed by a colon terminator prefixing the formal routine name.

When scanning a routine call, identifiers and phrase variables are recognised using the "SPECIAL NAME" and "PHRASE VARIABLE" routines (in both scans), which cause the corresponding identifier plexes to be placed on the "Analysis Stack". In the first scan, the actual parameter plexes are linked together (see figure 4.11), and in the second scan they are discarded (whilst the packed routine name is being constructed).

When scanning a routine heading, identifiers and phrase variables are recognised using the "NAME" routine and the "PHRASE VARIABLE" routine in conjunction with a global variable, "phrase variable switch". When this switch is set, scanning of a phrase variable involves a call of the "NAME" routine to scan the embedded identifier; otherwise, if the switch is not set (default) the "SPECIAL NAME" routine is called. So the "phrase variable switch" is set when scanning for a routine heading. (Note: the implementation is not the same as the "NAME" and "SPECIAL NAME" routines since the decision to admit phrase variables as formal parameters was taken at a late stage in the project). Thus, the first scan of a routine heading collects a set of packed (identifier) name forms in a linked list on the "Analysis Stack". (delimiter words and commas being ignored). During the second scan, the "NAME" and "PHRASE VARIABLE" routines are used in the same way, but this time the results (of scanning the identifiers and phrase variables) are discarded.

The reason why, when scanning a routine heading, identifiers are not declared to the compiler (by looking them up on the identifier dictionary) is that the routine may already have been declared, in which case, locations would already have been reserved to hold formal parameter values (when the routine plex was constructed). Thus, after the packed routine name has been constructed, its plex form is substituted for it on the "Analysis Stack", and the address of the first formal parameter (found in the routine plex) is used to construct identifier plexes for the formal parameter identifiers. It is not until this has been done that the identifiers (formal parameter names) are added to the (local) identifier dictionary. Finally, note that formal parameter identifiers have local scope, only, within a routine and that they are always scalars having storage automatically allocated to hold their values.

A detailed synopsis of the main Recognition routines is included in their definition (see Appendix F), however, the handling of routine headings and calls requires some further explanation.

A routine call consists of a string of delimiter words, commas and actual parameters, which may or may not be marked with an "@" symbol, and a routine heading is of the same form except that formal parameters are used in place of actual parameters. The algorithms used by the "ROUTINE CALL" and "ROUTINE HEADING" routines are as follows:

Algorithm 4.1: ("ROUTINE CALL")

1. Scan the symbols on the "input buffer" checking the syntax and forming a linked list of actual parameter plexes on the "Analysis Stack". Delimiter words and commas contained in the call are otherwise ignored during this scan.

2. If syntax is invalid then exit NOT RECOGNISED.

3. Pack routine name format on "Analysis Stack".

4. Substitute the routine plex for its packed name form and exit RECOGNISED.

Algorithm 4.2 ("ROUTINE HEADING")

1. Scan the symbols on the "input buffer" for valid routine type specification, constructing a 3-bit tag code if one is recognised, otherwise exit NOT RECOGNISED.

2. Check for ":". If not there, exit NOT RECOGNISED.

3. Check the syntax, forming a linked list of formal parameter plexes i.e. packed name forms, on "Analysis Stack". If syntax check fails, exit NOT RECOGNISED.

4. Pack routine name format on "Analysis Stack".

5. Search routine dictionary for given routine identifier:

   (a) If routine identifier is on dictionary, then:

      (i) If "defined bit" is already set for the routine print the warning message REDEFINITION.
      (ii) Compare the two type specifications (3-bit tag codes), and if different, print the error message ROUTINE TYPE.
      (iii) Set "defined bit" and "currently being defined bit" of routine plex.
      (iv) If there are any formal parameters then construct plexes for the formal parameter identifiers such that they refer to the previously reserved storage locations, then enter the identifiers and plexes in the (local) identifier dictionary.

   (b) If routine identifier is <u>not</u> on dictionary, then:

      (i) Construct routine plex on "Analysis Stack" (reserving space for a 3-word save area if the routine is BASIC) and add routine identifier and plex to routine dictionary.
      (ii) If there are any formal parameters, then, for each one, call the "SUBSTITUTE IDENTIFIER PLEX AT p" routine to reserve storage and enter the identifiers in the (local) identifier dictionary.

6. Retract "Analysis Stack" to its position at entry to this routine, then stack the routine plex at the top of the stack. Exit RECOGNISED.

The Syntax checking of a routine call or heading is performed by the routine "CHECK RSYNTAX FOR s PARAMETERS GIVING parmcount", which is handed one input parameter, S = 0 implying to search for actual parameters, or S = 1 implying to search for formal parameters, and which returns a value in "S" to determine whether recognition was successful (S = 1) or unsuccessful (S = 0), together with the number of parameters involved in the call/heading, "parmcount". When checking for formal parameters, this routine uses the "NAME" routine and the "PHRASE VARIABLE" routine with "phrase variable switch" = 1. When checking for actual parameters, the "EXPRESSION", "VARIABLE" and "PHRASE VARIABLE" routines are used to construct the actual parameter plexes on the "Analysis Stack". In addition, the most significant bit of each link word in the linked list is used as a marker bit to reflect whether or not an actual parameter was preceded by an "@" symbol. Any link word having its most significant bit set is thus followed by a variable plex form (an input/output marked parameter), and any other actual parameter plexes are expression plexes. Note that it is possible to have [PHRASE VARIABLE] as an input-only actual parameter. This causes a problem since the plex of an input-only actual parameter must be in expression plex form. The compiler overcomes this by taking advantage of the fact that a phrase variable plex is an

identifier plex and can thus be used to construct an expression plex even though a phrase variable cannot be part of a syntactic [EXPRESSION].

The packing up of a routine identifier is performed by the routine, "PACK ROUTINE NAME ON ANALYSIS STACK, s", which is given an input parameter "S", having the same input function as for the above routine. Again, depending on the type of parameter being scanned, the "NAME" or "SPECIAL NAME" routine is used to skip over identifiers contained in parameters. In the packed routine name format a parameter position is represented by an unprintable character defined by a global constant, "internal code for parameter", declared by the compiler.

4.6: The Format Table

This table forms the basis of the Syntax Analyser. It contains a complete set of the syntax of all valid [RCCT PIECE]'s that may appear in an RCCT program. An entry in this table has the form illustrated in figure 4.12. The coded [RCCT PIECE] consists of a sequence of fullwords containing either terminal symbols or a non-terminal entry.

A non-terminal entry has its most significant bit set, followed by a digit in the range 0 - 40 used to access an entry in the Recognition Table corresponding to the particular syntactic class.

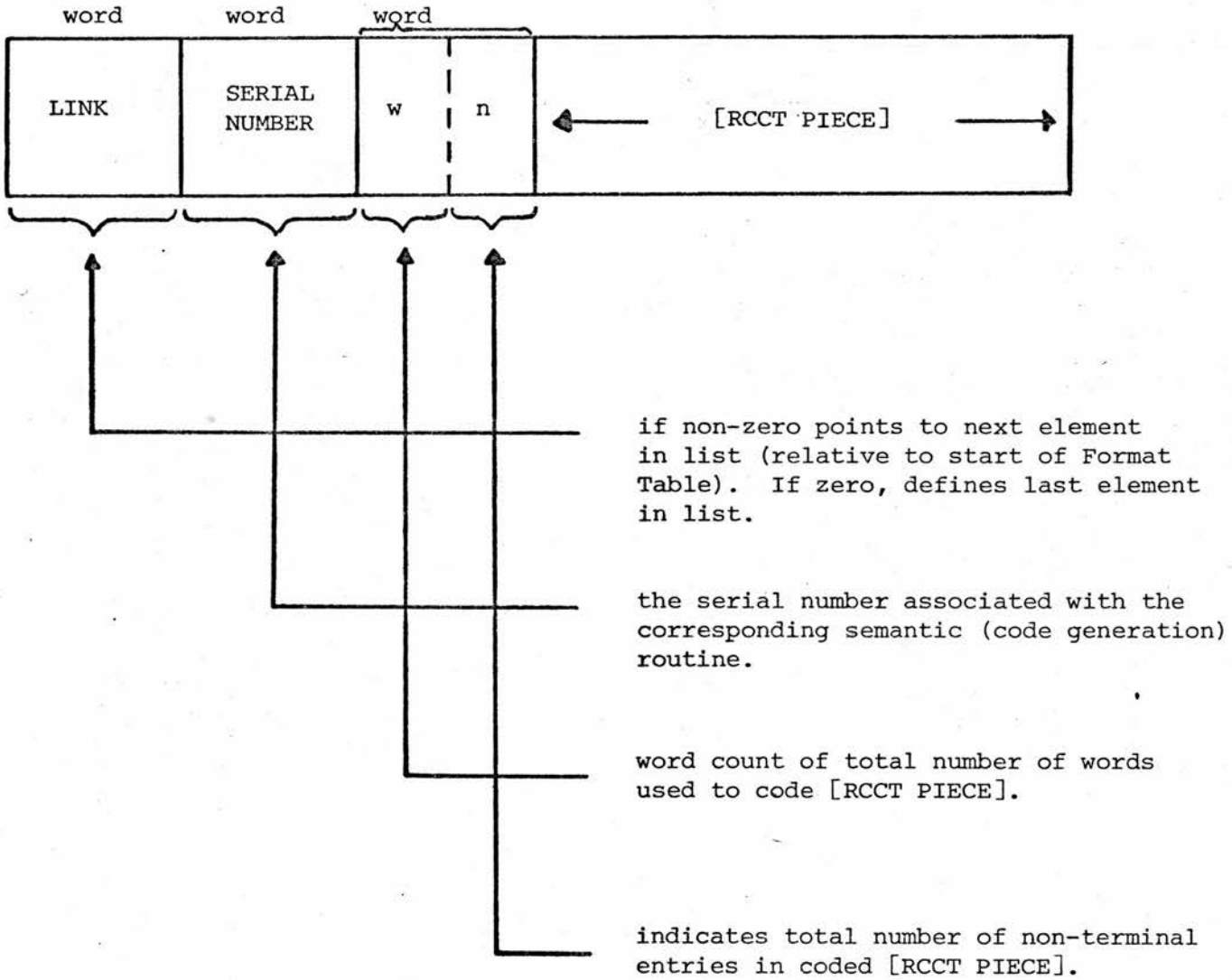


Fig. 4.12 : Format Table entry

If a non-terminal entry is followed by the terminal symbol, "*", then this has a special significance. It implies that the particular syntactic class being referenced may be repeated an indefinite number of times, where each occurrence is separated from its predecessor by a comma.

The initial layout for the Format Table was as a forward chained linked list of entries, where the [ROUTINE CALL] [BRACKETS TERMINATOR] entry was the last in the list. However, since there are forty-two entries in this table it was foreseen that such an organisation might make recognition or non-recognition of [RCCT PIECE]'s a lengthy process, so a simple hashing technique was introduced [8]. The table was reorganised into a set of forward chained linked lists arranged so that each member of a given set began with the same capital letter. One of these sets contained all the [RCCT PIECES]'s not beginning with a capital letter.

With such an organisation, the first character on the "input buffer" could be used to decide which sub-list should be accessed, by referencing into the hash table within the Format Table.

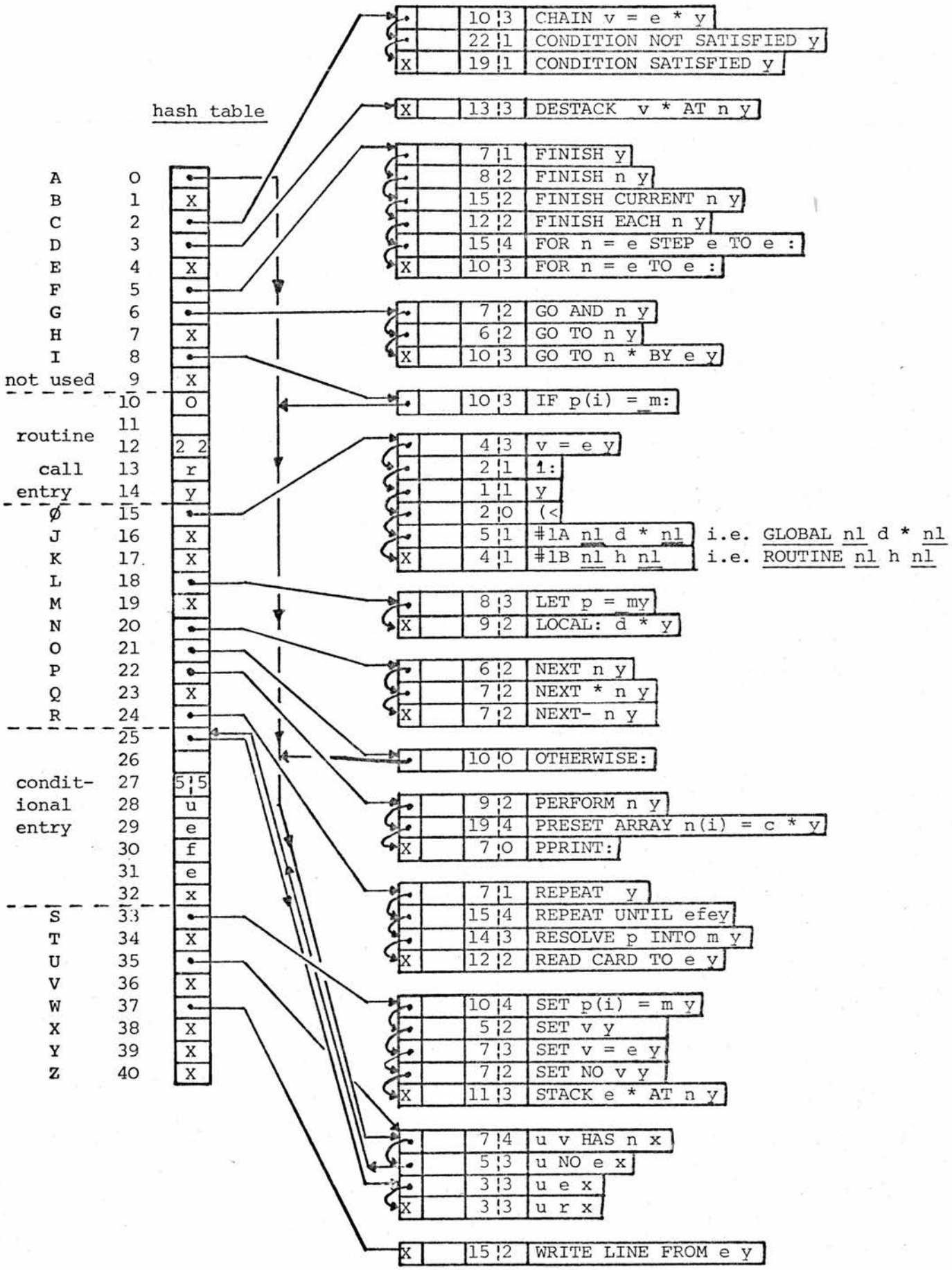
The hash table consists of a set of twenty-seven entries, where each entry points to a sub-list, and the position of the entry in the hash table is associated with a capital letter (one entry being reserved to point to the non-alphabetically indexed sub-list). Because of the problem of EBCDIC coding of the alphabetic characters, the hash table in reality has a length of 41 words c.f. Recognition Table.

The set of [RCCT PIECE]'s beginning with the syntactic class, [CONDITIONAL] are grouped into a separate sub-list of their own since they are a special case of a sub-list whose elements may begin with any of the letters A(ND), I(F), O(R), U(NLESS). This sub-list will be searched after searching the sub-lists associated with "I" or "O", or will be indexed directly from the hash table entries for "A" or "U". This is because there are no sub-lists corresponding to the alphabetic characters "A" and "U", but there are for "I" and "O".

As with the straightforward linked list approach, the last [RCCT PIECE] to be searched for must be the [ROUTINE CALL] [BRACKETS TERMINATOR] entry. This is catered for by ensuring that the last entry in each sub-list points to the routine call entry, and that any unused hash table entries point directly to the routine call entry. The link part of the routine call entry has the value zero to indicate that it is the last entry to be searched in a parse.

Advantage was taken of the redundant words in the hash table (due to EBCDIC conventions) to actually place the routine call entry in one of the "gaps" in the hash table. Another gap is used to hold one of the conditional [RCCT PIECE] entries. Another word in one of these "gaps" is used for the non-alphabetic hash table entry used to access the non-alphabetic sub-list.

Figure 4.13 shows the layout of the Format Table with some notational differences:



Note: "Ø" stands for the non-alphabetic hash table entry point

Fig. 4.13 : The Format Table

(i) The "serial number" part of each table entry has been left blank.

(ii) Links are indicated with an arrow, and an "X" symbol indicates a link to word 10 containing the routine call entry. All links are pointers relative to the base of the Format Table i.e. to the start of the hash table.

(iii) Entries for non-terminals are shown as small letters and the tag bit associated with the entry is not shown. These small letters may be interpretted as abbreviations for the Recognition routines being indirectly referred to via the Recognition Table. The relationship between letters and Recognition routines is as defined in figure 4.9.

4.7 The Parsing Routine

This is the routine, "PERFORM SYNTAX ANALYSIS GIVING RESULT r", and obeys Algorithm 4.3 defined by:-

Algorithm 4.3

1. Inspect character at "input symbol position" and use it to reference into the hash table contained in the Format Table to select a sub-list of entries to be searched.

2. Attempt a parse according to the entry selected, and

   (a) If parsing is successful, then set output parameter "r" to the value of the serial number associated with the Format Table entry. Then finish.

   (b) If parsing is unsuccessful, then either follow the Format Table entry's link to the next entry and go to 2, or, if the link is zero,

      (i) Monitor a SYNTAX error message.

      (ii) Ship characters on "input buffer" past the next terminator and set output parameter "r" = -1 (if the "input buffer" requires refilling) or "r" = -2 (if there are further symbols on the "input buffer" to be parsed). Then finish.

When the parsing routine scans a Format Table entry (see figure 4.12) to attempt a parse, the non-terminal wordcount "n" is used to reserve a set of "n" pointers on the "Analysis Stack" corresponding to the top level set of pointers on a parse chain. Then the wordcount "w" is used to control a cycle consisting of scanning the individual words of the coded [RCCT PIECE] in the Format Table entry. The algorithm used in this scanning cycle is defined by Algorithm 4.4.

Algorithm 4.4

1. If a word contains a terminal symbol, then compare it with the symbol at "input symbol position". If the symbols differ, then the parse of that particular [RCCT PIECE] fails, otherwise, update "input symbol position" and continue the scanning cycle defined above.
2. If a word contains a non-terminal symbol, the contents of the location reserved to hold a forward pointer to this syntactic class entry (see above) is set to point to the current free "Analysis Stack" position, then an inspection is made of the next Format Table entry to see if it contains the specially significant EBCDIC symbol, "*".

   (a) If it does, then the top word on the "Analysis Stack" is reserved for an integer count of the number of occurrences of the particular syntactic class, and a cycle is entered to repeatedly look for the class followed by a comma (see note below). If no occurrences of the class are found, the parse fails, otherwise the count of class occurrences is entered in the above mentioned word and the scanning cycle, defined above, is continued (although the "*" symbol is now skipped).

(b) If it doesn't then the class is looked for once, only (see note below). If the class is recognised then the scanning cycle continues, otherwise the parse fails.

Note: To search for a class, the digit entry in the non-terminal word position is used to access into the Recognition Table to obtain the relative base address of the Recognition routine required. This value is used to transfer control to the corresponding Recognition routine, and on returning from this the global variable "recognised" is tested.

If all words in a Format Table entry are processed successfully, the parse succeeds. Note that the resultant parse chain only contains entries corresponding to the syntactic classes contained in an [RCCT PIECE], and in some cases it may thus be empty.

A global variable "plexbase", usually initialised to point to the base of the empty "Analysis Stack" prior to a call of the parsing routine, defines the head of a parse tree (or chain).

An example of a parse tree is given in figure 4.14 which illustrates

(a) the logical parse tree structure, and

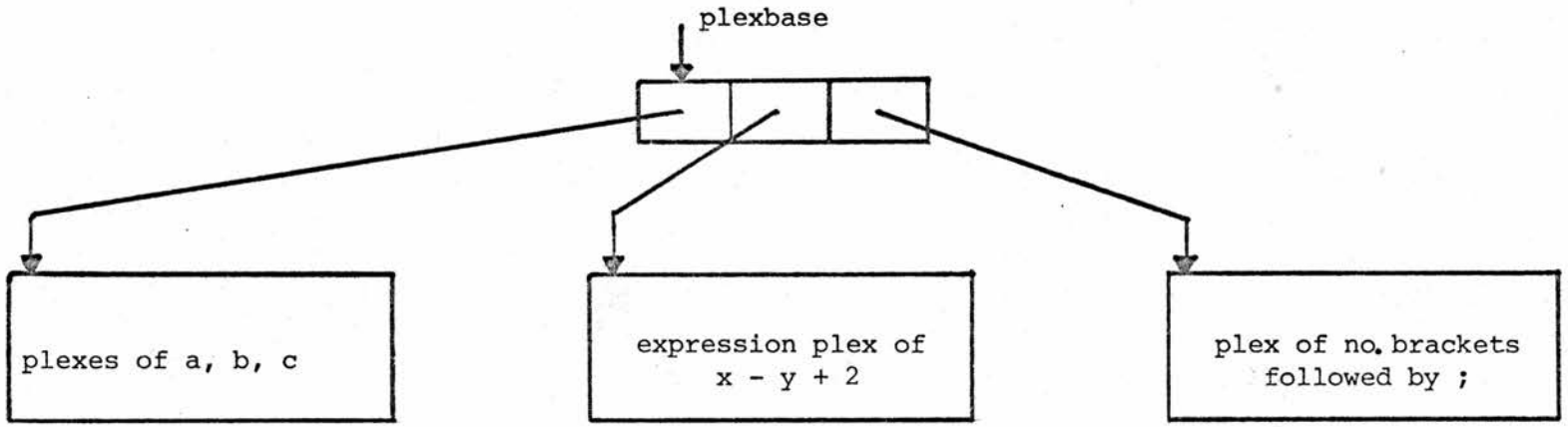
(b) its actual "Analysis Stack" coding

obtained by parsing the input statement:

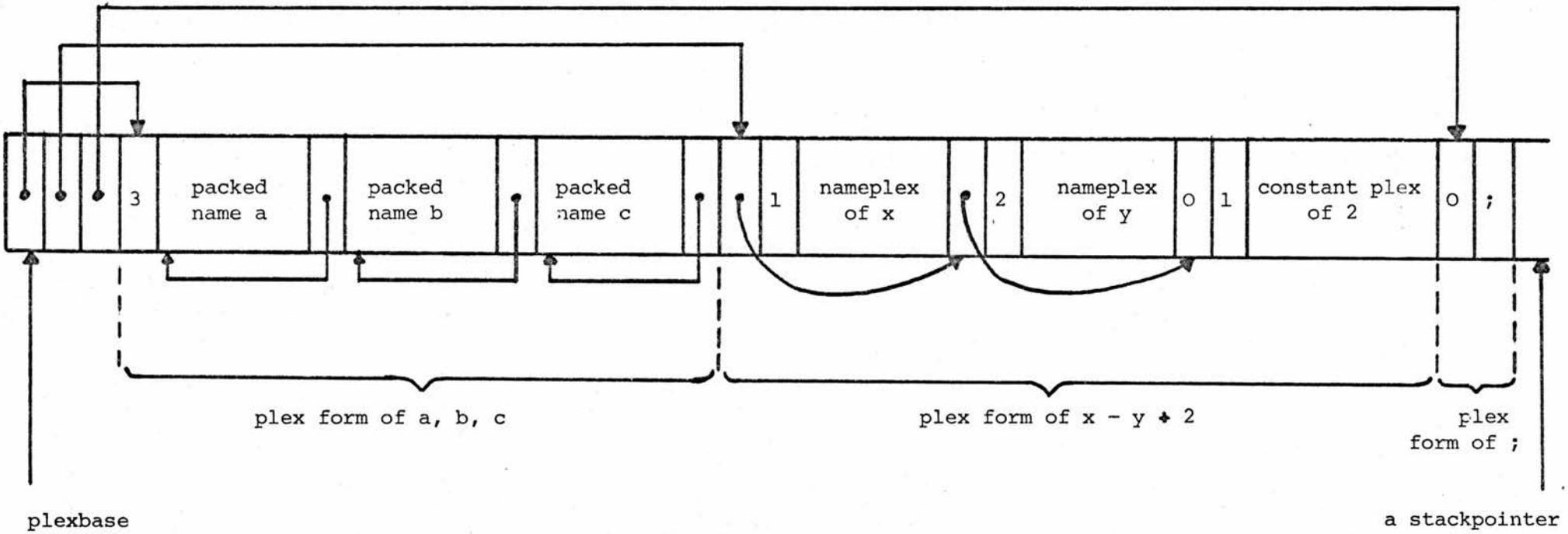
GO TO a, b, c BY x - y + 2;

which is a valid sentence of the [RCCT PIECE]:

GO TO [NAME*/&,3] BY [EXPRESSION][BRACKETS TERMINATOR]



a) Logical parse tree structure



b) Actual configuration on Analysis Stack

Fig. 4.14 : Parse tree of GOTO a,b,c BY x-y+2;

CHAPTER 5

SEMANTICS

5.1 Object Program Data Organization

The object program data space generated by the compiler is called "Program Data Area" (PDA) and is a sequence of 2½K words organized according to figure 5.1.

At run-time, two permanently assigned registers are used to point to the base of the scalar area and the auxiliary data area. They are referred to at Compile-time as the "base register" and "constant base register", respectively. The auxiliary data area, containing the index table and constant table, together with the preset array table is required during the compilation of an RCCT program; the index table to hold base addresses of compiled routines, the constant table to hold the set of constants referred to by the compiled routines, and the preset array table to hold all the elements associated with preset arrays.

Under the present load-and-go testing environment, PDA is initialised as an array at compile-time. To load elements into the index, constant and preset array tables at compile-time, the global variables "index table", "constant table" and "preset array table" are initialised to point to the start of each respective table,

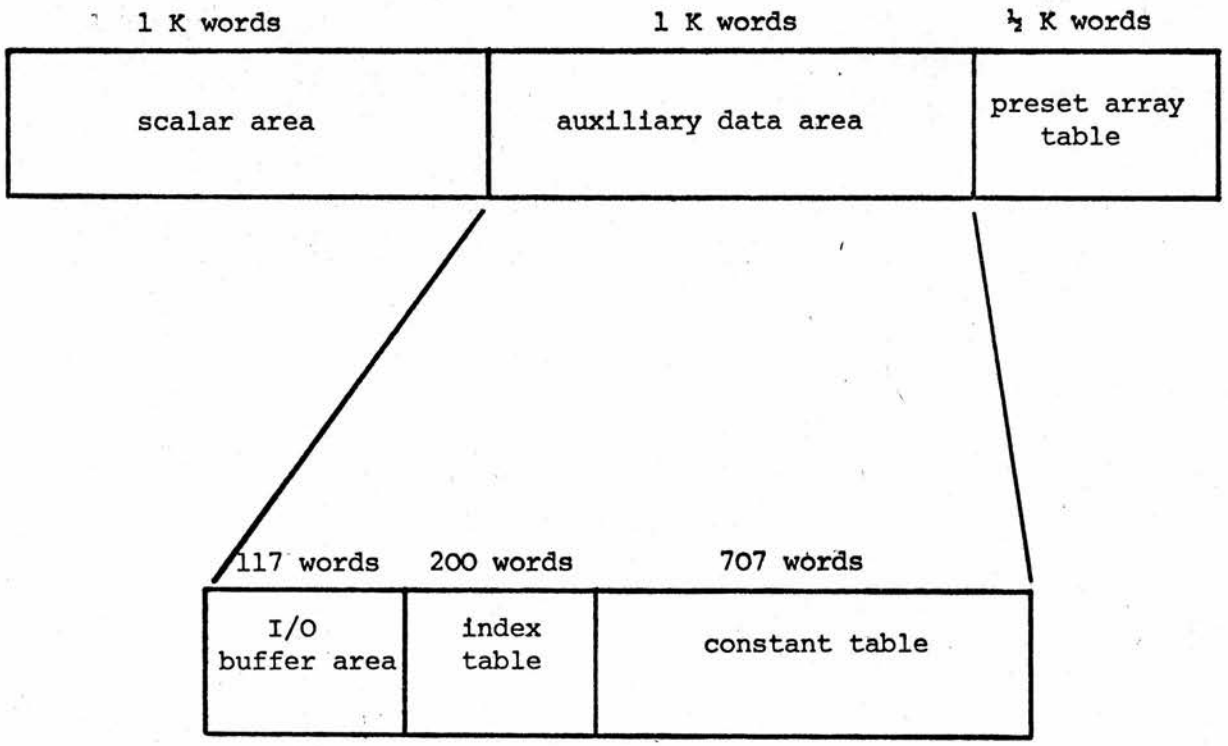


Fig. 5.1 : Program Data Area Configuration

and the globals "current serial number", "current free constant position" and "current free pa constant position" are used as relative pointers into each respective table to indicate the next free location in it.

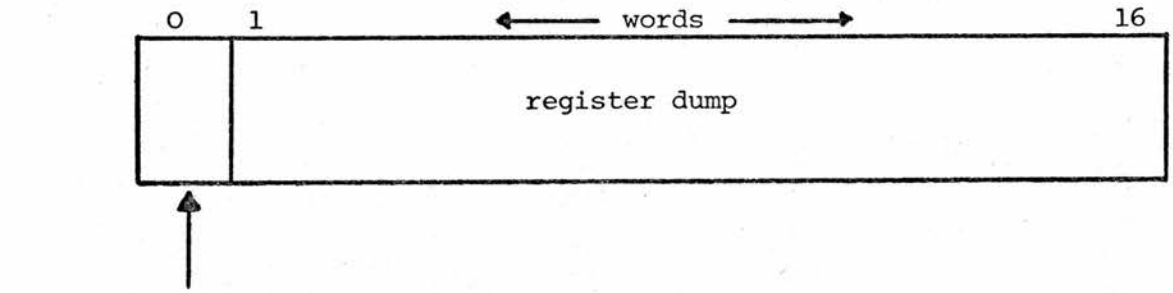
All user-defined identifiers which require run-time storage allocation are implemented as "own" variables by having a unique storage location reserved for them in the run-time scalar area. The global "next free runtime location" is used as a relative pointer to the next (as yet not allocated) free run-time scalar location, and when this value is used to reserve a location for a scalar by inserting it into the corresponding identifier plex, "next free runtime location" must be updated accordingly.

Elements in the preset array table are not accessed (at run-time) by direct addressing (using a base register and displacement), however, the elements in the scalar area and constant and index tables are. In the case of run-time scalars, these will be addressed using the "base register",defined by the compiler, and a displacement defined in the identifier plex. Index table elements will be addressed by the "constant base register" and a displacement defined as the sum of the appropriate current serial number and the global constant "index area" (the displacement of the start of the index table from the start of the auxiliary data area). Constant table elements will be addressed using the "constant base register" and a displacement defined by the sum of the relative position of the constant in the constant table together with the global constant "constant area" (the displacement of the start of the constant table from the start of the auxiliary data area).

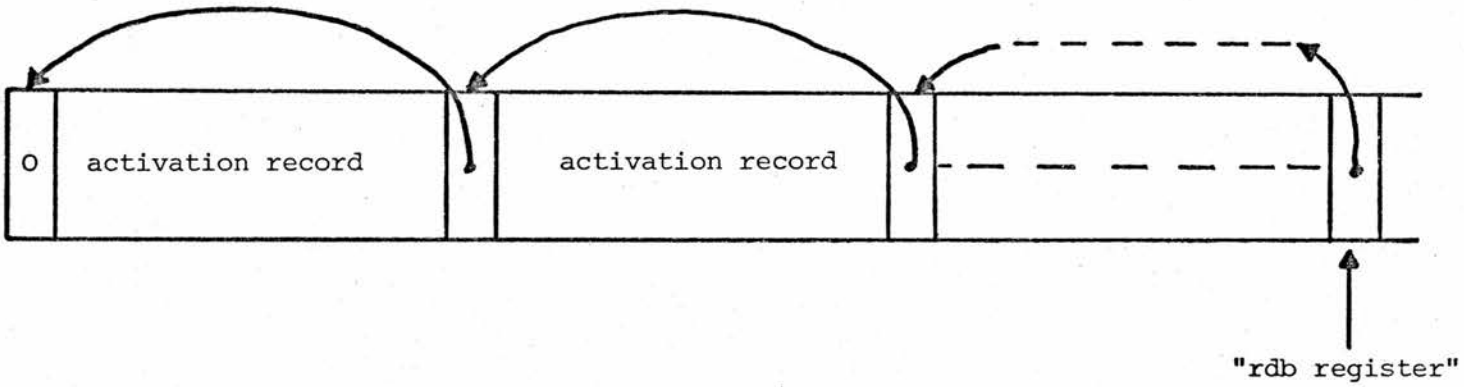
The I/O buffer area at the base of the auxiliary data area contains all the data required to perform run-time I/O operations. (see §5.8).

In addition to PDA, the compiler organises a run-time stack to be used for routine linking. In the load-and-go environment this is implemented by arranging for the "rdb register" (general purpose register 7) to point to a location in main memory which may be assumed to be the base of the run-time stack, prior to the execution of the object program. Since the prime function of the run-time stack is to provide a dynamic "DISPLAY" of runtime STANDARD routine activation, the object program must use the run-time stack with care, once a STANDARD routine has been activated. However, the MAIN routine of the object program does not place any activation record on the stack. Hence, the MAIN routine can use the base of the run-time stack as a run-time data area by initialising the associated run-time variables to positive values relative to register 7 (the "rdb register"), and adjusting the value of register 7.

The activation "DISPLAY" record left on the run-time stack during the execution of the object program consists of a linked-list of "activation records". The format of an activation record is illustrated in figure 5.2a, and a typical run-time stack configuration, during execution of an object program is shown in figure 5.2b.



a) "activation record" format



b) typical run-time stack configuration

Fig. 5.2 : Run-time Stack

Each time a STANDARD routine is entered at run-time, an activation record consisting of its serial number (the relative position in the index table where its base address is stored) and a dump of all the general purpose register contents, is put on the top of the run-time stack, then the "rdb register" is updated to point to the next free location at the top of the stack, and a back pointer is placed at this location to point to the previous "level" of the stack. On exit from a STANDARD routine, the backpointer is used to retract the run-time stack to its previous "level", and all registers (except register 0 which may contain the result of an IF-routine call) are restored to their values on entry to the routine.

With this run-time configuration, an RCCT program may have upto 1024 scalars, upto 707 constants, and up to 200 routines. In addition, up to 512 preset array elements may be declared. These storage characteristics should be sufficient for most RCCT programs, but the compiler has been designed to allow extensions in the size of the scalar area, constant table and preset array table.

5.2 Object Program Code Organization

At present, all the code generated by the compiler is simply planted in main memory in the compile-time "codestack". The global "code address" is used as a pointer to the current free location at the top of the stack, which is organised as a 10 K-word array.

The compiled code has been designed to be relocatable. This relocatability was achieved by assigning special functions to three run-time registers, referred to at compile-time as the "routine index register", "return register" and "routine base register", and arranging that all absolute code addresses (base addresses of routines) are stored in the index table rather than being embedded in the code. The three named registers are referred to as the routine linkage registers and the subset obtained by disregarding the "return register" is known as the set of routine addressing registers.

Three other permanent run-time register allocations are made: the "expression accumulator" is reserved for accumulating the intermediate and final results of expression evaluation, the "variable address accumulator" is reserved for evaluating sub-expressions of a subscripted variable and the "comparison register" is reserved as a general purpose register.

Special conventions have been adopted to perform routine linking:

(a) Responsibilities of the calling routine:

(i) To evaluate and copy all actual parameters into formal parameter locations.

(ii) To load the "comparison register" with the base address of the called routine (stored in the index table).

(iii) To load the "return register" with the return address in the calling routine, then to pass control to the called routine (whose address is in the "comparison register").

(iv) To copy back any output parameters from their formal parameter locations.

(b) Responsibilities of the called routine:

(i) To store away the routine linkage registers (for a BASIC routine) or to dump an activation record on the run-time stack (for a STANDARD routine).

(ii) To initialise "routine base register" = "comparison register", "routine index register" = 0, and at the end of the routine:

(iii) To restore old registers and jump to address in "return register".

A MAIN routine cannot, by definition, be called within a program, hence, no code is generated at its entry point to perform routine linking; and the code generated at the end of a MAIN routine is a Supervisor Call to specify the end of a job-step.

One of the main problems associated with code generation is that of branching within routines. The linking conventions outlined above take care of external routine referencing. However, the IBM/360 architecture is designed so that code referencing may only be relative to some address by a positive displacement of 0 - 4095 bytes. The specified address may be defined by the contents of one register (giving a base/displacement format address), or the sum of the contents of two registers (giving a base/index/displacement format address) [5].

The function of the "routine base register" is to point to the base of the currently-active routine and thereby act as the base register required for code addressing. This arrangement allows any routine to be upto 4096 bytes in length and control to be transferred internally, using the base/displacement form of address, to and from any point in the routine. However, the amount of

generated code corresponding to a routine cannot be related directly to the number of source statements that it contains, in a satisfactory manner, so it became necessary to allow routines to generate more than 4096 bytes of code without errors being caused. The solution, to the addressing problem, adopted by the compiler, is to generate "code segments" corresponding to user-defined routines, and to use the base/index/displacement form of branching address. The function of the "routine index register" is to act as the index register used for internal referencing within routines.

A code segment is defined to be a block of up to 4000 bytes of object code. Each routine consists of either a single (partially full) code segment, or a contiguous sequence of full code segments (4000 bytes in length) followed by a partially full code segment. The "routine index register" contains, at run-time, the displacement of the currently-active code segment from the start of the currently-active routine. Thus, on entry to a routine, control starts at the beginning of its first (or only) code segment, so the "routine index register" is initialised to zero.

All branch instructions may now be thought of as references to points within the same segment.

During the compilation of a routine, all references generate a single branch instruction whose address part is incomplete. The base and index parts of the address are known, but the displacement part will, in general, be unknown when the instruction is generated.

All implicit (control) references are filled in as soon as their displacements (from the beginning of the currently-being-compiled code segment) become known. Explicit references (defined by GOTO's, FINISH statements etc.) generate incomplete branch instructions that are chained together within a code segment until the segment is completed. A segment will be completed either because a routine has been fully defined, or because the current code segment exceeds 4000 bytes. In either case, the compiler must plant "segmentation code", whose function is

(i) To fill in any branch instruction whose referenced address is known.

(ii) To bring down any incomplete forward (label or control) references so that the respective branch instructions will cause control to pass to a section of code (part of the "segmentation code") which updates the "routine index register" to refer to the current code segment, and plants a new branch instruction.

Note Incomplete references found at the end of a routine cause error condition messages to be output.

Figure 5.3 illustrates how code segments "overlap" to allow communication between consecutive code segments via the "segmentation code".

The effect of this segmentation technique is that internal code segment references involve the execution of a single branch instruction at run-time, but external code segment references will cause a sequence of branch instructions, or "hops", to be executed.

The overlapping of code segments implies that the "segmentation code" planted after a code segment forms the start of the next code segment. The only exception to this rule is where the "segmentation code" is being planted at the end of a routine, in which case it forms part of the last code segment of the routine.

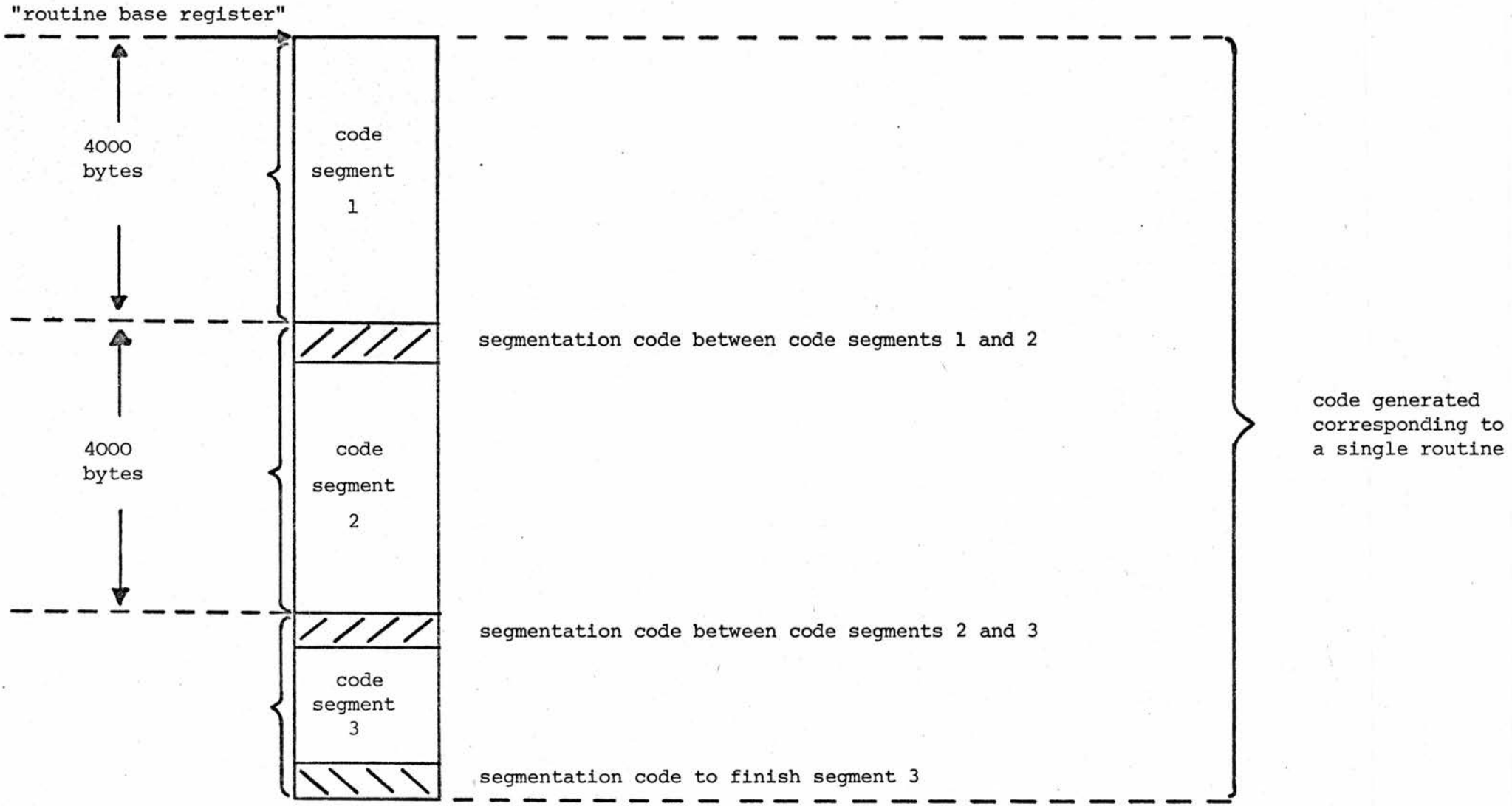


Fig. 5.3 : Overlapping of code segments

The routine "COMPLETE BRANCHING BY s" is responsible for generating "segmentation code". The input parameter "s" is used to signify whether "segmentation code" is required to complete the last segment in a routine (s = 0) or to perform automatic segmentation (s = 1). (See §5.12)

The choice of 4000 as the code segment size implies that there is a maximum size of 96 bytes allowed for "segmentation code". This was an arbitrary choice which will allow upto about 40 external code segment references to be made within any code segment. If the "segmentation code" between two segments exceeds 96 bytes, the error message LABEL REFERENCES is output, since this condition can only arise where a large number of labels are used in a routine. The user is thus encouraged to use labels sparingly.

Segmentation of code is achieved by manipulation of the global variables "routine byte count", which indicates the total number of bytes of code which have currently been compiled within a code segment, and "index value", which indicates the value that the "routine index register" must contain when control is in the current code segment.

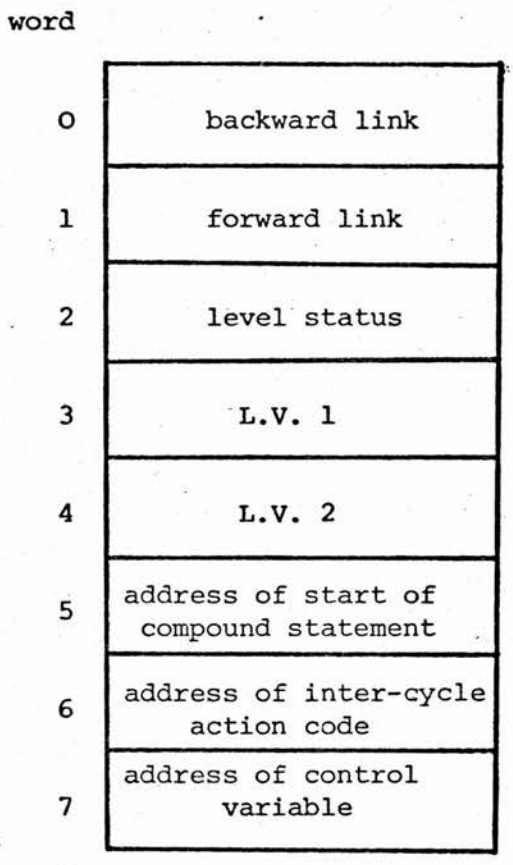
An additional global variable may be used by the semantic (code generation) routines to delay automatic segmentation of compiled code for limited periods if the contiguity of a specific code sequence is desired. This is the "segmentation delay switch", which is initialised by the compiler to zero, to allow automatic segmentation. Any routine which sets the "segmentation delay switch" has the responsibility of switching it off again as soon as possible.

Having introduced the concept of code segmentation, it is now possible to refer to the address of any machine instruction within a routine in terms of an index value (defining the code segment number) and a displacement (defining the position of the instruction within that code segment). This format, referred to as the segment/word address, is used to specify code addresses at compile-time, and is the format used within the label plex (see figure 4.6) to describe the address of a point within the currently-being-compiled routine relative to its starting point.

5.3 Control

Control is implemented in "control lists" which access incomplete instructions requiring control to be filled in at some point. This is the function of a list variable, which gives the address in the compiled code of the last instruction requiring completion at some later point. The address part of that instruction contains the (self) relative address of the previous instruction making the same reference, and so on upto the last instruction in the control list chain whose address part is zero. A zero valued list variable defines an empty control list.

A set of information is reserved for each level of compound statement bracketing. This information is known as a set of "control data", or a "control data block" (CD block). The format of a CD block is illustrated in figure 5.4.



where word 0 = backward link to previous CD block

1 = forward link to succeeding CD block

2 = level status, coded as follows:

0 : IMP	4 : IF IMP
1 : IF	5 : OTHERWISE IMP
2 : AND IF	6 : CYCLE IMP
3 : OR IF	7 : EXPECTING OTHERWISE

3 = List Variable giving access to Control List 1, used for chaining together forward references in an if-statment, or FINISH EACH references in a for-statement.

4 = List Variable giving access to Control List 2, used for chaining together FINISH CURRENT references in a for-statement.

5 = Address of start of compound statement, or, for the top level CD block, gives address of start of routine (address stored in segment/word format).

6 = Address (in segment/word format) of the "inter-cycle action" code generated by a for-clause (see § 5.9).

7 = Address of control variable identifier - consisting of the first word of the identifiers name plex.

Fig. 5.4 : CD block format

CD blocks reside on the "Compile Time Stack", and are implemented using the global variable "current control data" to point to the base of the CD block associated with the current compound statement level.

Whenever an opening compound statement bracket is met, another CD block is created on the "Compile Time Stack", pointed to by the forward link in word 1 of the previous CD block. The backward link of the newly created CD block points to the previous CD block (corresponding to an outer level of compound statement bracketing). Backward linking is used to release a CD block when a closing compound statement bracket is met: forward linking is used to save having to set up a CD block which may have only recently been released. (When a CD block is released, on meeting a closing compound statement bracket, the space it occupied on the "Compile Time Stack" is not recovered until the end of compilation of a routine - see §5.5).

The top level of control data, which resides permanently on the "Compile Time Stack" has a slightly different format in that word 5 contains the address of the start of the routine, and an additional word precedes this CD block, which holds the list variable used to chain FINISH instructions together.

Each [RCCT PIECE] except "[LABEL]:" may be classified in terms of being either a for-clause, a conditional clause, an imperative statement, or a local declaration. The colon terminator is associated with a for-clause; the colon or semi-colon terminators may be associated with a conditional clause; and any number of closing compound

statement brackets, followed by any terminator may be associated with an imperative statement, or, if standing alone, may be assumed to follow the null imperative statement. The concept of level status was introduced to enable syntax checking of valid if and for-statements to be performed during the semantic processing phase of compilation, and to aid in the construction of automatic control referencing.

The top (routine) level of control data has its level status initialised to zero (IMP) at the start of compilation of each routine. Similarly, when an opening compound statement bracket is met, it causes a new level of control data to be set up whose level status is initialised to zero, and the global variable "brackets count" to be incremented by one. Syntax and control checking is performed by analysing the context of the currently-being-scanned [RCCT PIECE], given the old level status and the terminator immediately succeeding the [RCCT PIECE] (Every closing compound statement bracket has an implicit newline terminator preceding it). This analysis may result in a change of level status, and is referred to as "processing control".

Processing control after a for-clause is performed inside the semantics routine associated with the for-clause. A check is simply made that the level status is zero (IMP). If it is not, then no code is generated for the for-clause and the message UNEXPECTED FOR-STATEMENT indicates an error condition; otherwise code generation continues. In either case, the appearance of the for-clause causes the level status to be set to six (CYCLE IMP).

Processing control after a conditional clause (any [RCCT PIECE] beginning with the class [CONDITIONAL]) is performed in the code generation routine associated with conditionals. The action taken is to check the syntax of an if-clause by considering the current conditional [RCCT PIECE] in the context of the old level status and the terminator appearing at the end of the conditional [RCCT PIECE]. The valid syntactic combinations are:

(i) If the conditional clause has the IF/UNLESS form, then the previous level status must have been "IMP". It is altered to "IF IMP" when the associated terminator is a colon, or to "IF" for a semi-colon.

(ii) If the conditional clause has the OR IF/UNLESS form, then the previous level status can only be "IF" or "OR IF". It is altered to "IF IMP" for a colon terminator, or to "OR IF" for a semi-colon.

(iii) If the conditional clause has the AND IF/UNLESS form, then the previous level status can only be "IF" or "AND IF". It is altered to "IF IMP" or "AND IF" corresponding to a colon or semi-colon terminator, respectively.

If any of the above rules are violated, an error condition occurs either because of misplacing the if-statement, mixing OR and AND conditions, or mis-use of punctuation in an if-clause, and the error message ILLEGAL CONTROL is output.

The control processing involved for an "OTHERWISE" statement is to check that the level status is "EXPECTING OTHERWISE", in which case it is altered to "OTHERWISE IMP". Any other level status causes the error message UNEXPECTED OTHERWISE to be output.

Processing control after an imperative statement is performed by the "PROCESS CONTROL AFTER IMPERATIVE p" routine, which is handed a pointer, "p", to the "Analysis Stack" entry created by the class [BRACKETS TERMINATOR], namely, a count of the number of closing compound statement brackets followed by the appropriate terminator. Algorithm 5.1 lists the action taken by the PROCESS CONTROL.... routine.

Algorithm 5.1

1. For each closing compound statement bracket:

   (a) take control action appropriate for the (implicit) appearance of a newline terminator

   (b) check that global variable "brackets count" is non-zero. If it is, output the error message TOO MANY CLOSING BRACKETS; otherwise, subtract one from "brackets count" and remove a layer of control data by setting "current control data" = backward link in current CD block.

2. Take control action appropriate to the final terminator.

The table in figure 5.5 indicates the valid combinations of (old) level status and terminator, and the corresponding level status alterations. Any blank entry in this table indicates an error condition and causes the error message PUNCTUATION to be output, since these invalid conditions are usually the result of mis-using terminators.

Old Level Status		terminator :	terminator ;	terminator newline
IMP	(0)		IMP	IMP
IF	(1)			
AND IF	(2)			
OR IF	(3)			
IF IMP	(4)	EXPECTING OTHERWISE	IF IMP	IMP
OTHERWISE IMP	(5)		OTHERWISE IMP	IMP
CYCLE IMP	(6)		CYCLE IMP	IMP
EXPECTING OTHERWISE	(7)			

Fig. 5.5 : Level status alterations caused by PROCESS CONTROL.... routine

The compiler provides two routines to perform operations on the control lists associated with the list variables than can appear in a CD block or label plex. These routines allow additional branch instructions to be added to a particular control list, or specify that a particular control list can have all its elements completed. The routine "ADD a TO l" assumes that at the code address pointed to by "a" there is an incomplete branch instruction whose address part contains all zeros, and the routine adds this instruction to the control list of similar instructions accessed by the list variable pointed to by "l". (The control list may be empty, in which case the list variable at "l" will be zero). The routine "FILL IN l TO a" assumes that "l" points to a list variable giving access to a (possibly empty) control list, and that "a" indicates the displacement from the start of the current code segment to which the branch instructions in the control list should refer. The action taken is to construct a base/index/displacement form address using the "routine base register", "routine index register" and "a", respectively, then to chain down the control list filling in all the incomplete branch instructions with the constructed address. Finally, the list variable at "l" is set to zero.

5.4 Further Syntax

As well as checking for valid syntactic constructs such as the for-and if-statement, the compiler must check for other structures.

(i) The concept of an [RCCT BLOCK] is implemented by manipulation of the global variable "instruction count" through the routine

"CHECK BLOCK SYNTAX GIVING RESULT switch"

The global "instruction count" is initialised to the smallest negative number (-2^{32}) and is incremented by one after processing of each recognised [RCCT PIECE]. The semantics routine associated with a global declaration resets "instruction count" to -2^{32}, whereas the semantic routine associated with a routine heading sets "instruction count" to -1. All other [RCCT PIECE] can only appear in the body of a routine, and to check this condition, each semantic routine starts by calling the CHECK BLOCK.... routine. This routine inspects "instruction count". If it is non-negative then the [RCCT PIECE] appears in the body of a routine and the output parameter "switch" is set to 1, otherwise the error message SYNTAX is output and the "switch" set to zero. If this syntax error occurs, then the calling semantic routine generates no code.

Another use of "instruction count" is to check whether a local declaration appears at the head of a routine, in which case its value will be zero. The semantic routine associated with a local declaration checks this condition and outputs the warning message DECLARATION POSITIONING if it is not fulfilled. It then resets "instruction count" in case any more local declarations appear at the head of a routine.

(ii) A check must be made that all routines are systematically accessed in the correct way defined by its type. As well as the possibility of redefining routines, there is the problem that when a routine is called before it is defined, it is assumed to be a STANDARD routine. This means that all BASIC routines must be defined before they are called. The routine

"TEST ROUTINE p TYPE BY switch"

is passed a pointer to the plex form of a routine call, "p", and a "switch" value determined by the context of the call. The routine inspects the routine plex contained in the parse of the routine call and compares its IF/IMP tag bit with "switch". If they are the same then the context of the call

is valid, otherwise it is not, and the error message INVALID ROUTINE CALL is output. This routine is called by the code generation routines associated with a routine call or an if-condition routine call.

(iii) The semantic routine associated with the REPEAT statement checks that at least one closing compound statement bracket followed the REPEAT statement, and if not generates the error message NO BRACKET AFTER "REPEAT".

(iv) When a routine is complete, a check must be made for

(a) undefined labels
(b) unfinished for- or if-statements
(c) unclosed compound statements

This is one of the functions of the routine, "FINISH OFF PREVIOUS ROUTINE".

5.5 "FINISH OFF PREVIOUS ROUTINE"

This routine is called whenever a master word is recognised. Its first action is to test "instruction count". If it is negative, then the master word was preceded by a global declaration or is the first part of an RCCT program, so no action is taken: otherwise, a routine preceded the master word, and code must be compiled to complete the routine. The action taken in this case is as follows:

(i) The "COMPLETE BRANCHING BY s" routine is called to plant automatic segmentation code (see §5.2).

(ii) Syntax checks are made for undefined labels etc. and any error conditions are monitored.

(iii) The control list associated with FINISH references, whose list variable is stored at the base of the top level of control data, is filled in to the current code position.

(iv) Code is planted to complete the routine according to its type specification (see §5.11).

(v) The corresponding routine plex is accessed and its "defined bit" tag is set, and its "currently-being-defined bit" tag is switched off.

(vi) The label dictionary is deleted together with the "local" part of the identifier dictionary.

(vii) A garbage collection is performed on the "Compile Time Stack" to pack up elements in the routine dictionary and preset array dictionary, eliminating "holes" left by labels, (local) identifier dictionary elements and CD blocks (apart from the top level of control data).

(viii) The top level of control data is re-initialised.

Three global variables are associated with garbage collection: "old ct stack front", "old routine dictionary tail" and "old pa dictionary tail". Initially, the "Compile Time Stack" only contains the initialised top level of control data, and "old ct stack front" points to the top of the "Compile Time Stack". Hence the stack is in a compacted form. "old routine dictionary tail" and "old pa dictionary tail" are set to zero. When a global declaration is recognised, and identifiers are added to the identifier dictionary, the semantic routine associated with global declarations simply updates "global dictionary head" to reflect the additional items in the identifier dictionary, then it updates "old ct stack front" to point to the new top of the "Compile Time Stack". As no other items will have been added to the stack

since the last (if any) garbage collection, then the stack is still in a compacted form. However, as soon as a routine has been completed, the routine and preset array dictionaries must be checked to see if any new entries have been made which may need shifting down the stack to remove any "holes". "Old routine dictionary tail" is used to point to the last compacted routine dictionary element after the previous garbage collection. Similarly, "old pa dictionary tail" is used to point to the last compacted preset array dictionary element. By comparing the two "old" configurations of these two dictionaries with the current configurations, the FINISH OFF .... routine ascertains whether any compaction is required, and what order to perform the compaction. As soon as the garbage collection is complete, the two "old" dictionary pointers are updated, and "old ct stack front" is adjusted (as well as "compile time stack front") to point to the new top of the "Compile Time Stack".

5.6 Preset Arrays

The problem associated with the inclusion of the PRESET ARRAY statement in the RCCT language is that of initialising run-time data at compile-time. Space in the run-time "scalar area" is scarce, so the decision was taken to store all preset array elements in a separate area, the preset array table. Provisions had to be made for the array identifiers to point to the associated array bases.

However, the run-time address of the preset array table will not, in general, be known at compile-time, so the preset array dictionary was introduced to keep track of preset array identifiers during compilation in order that the corresponding run-time address constants could be stored in the correct locations prior to execution of the program. Thus, a preset array identifier is an address constant whose value is not known till run-time, and the preset array dictionary contains the relocation information required to initialise address constants correctly, prior to execution of the program.

When a PRESET ARRAY statement is recognised, the associated semantic routine takes the following action:

(i) A location is reserved in the constant table to hold the address constant corresponding to the array identifier.

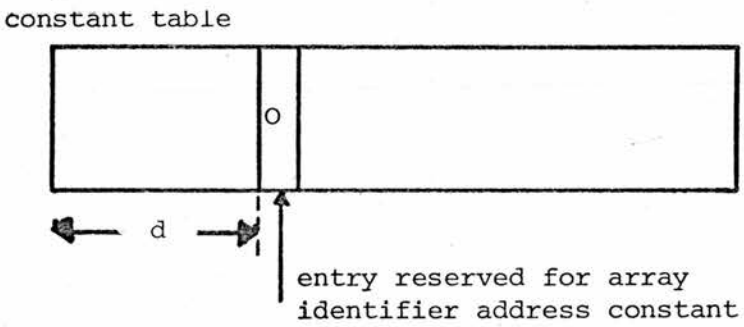
(ii) A corresponding identifier plex is constructed, having, in its first word the "constant bit" and "preset array bit" set, and, in its second word, the displacement from the start of the constant table of the reserved location to hold the array identifiers address constant value. The identifier and plex are then added to the identifier dictionary (a fault being monitored if the identifier is already present on the dictionary).

(iii) An entry is made in the preset array dictionary.

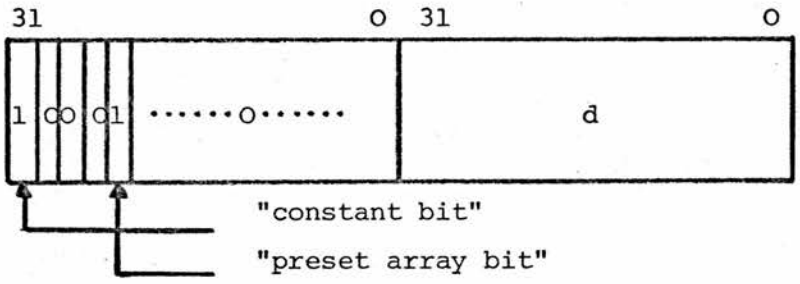
(iv) The list of constants contained in the PRESET ARRAY statement are stored in the preset array table (in contiguous locations) and "current free pa constant position" updated accordingly.

Figure 5.6 illustrates:

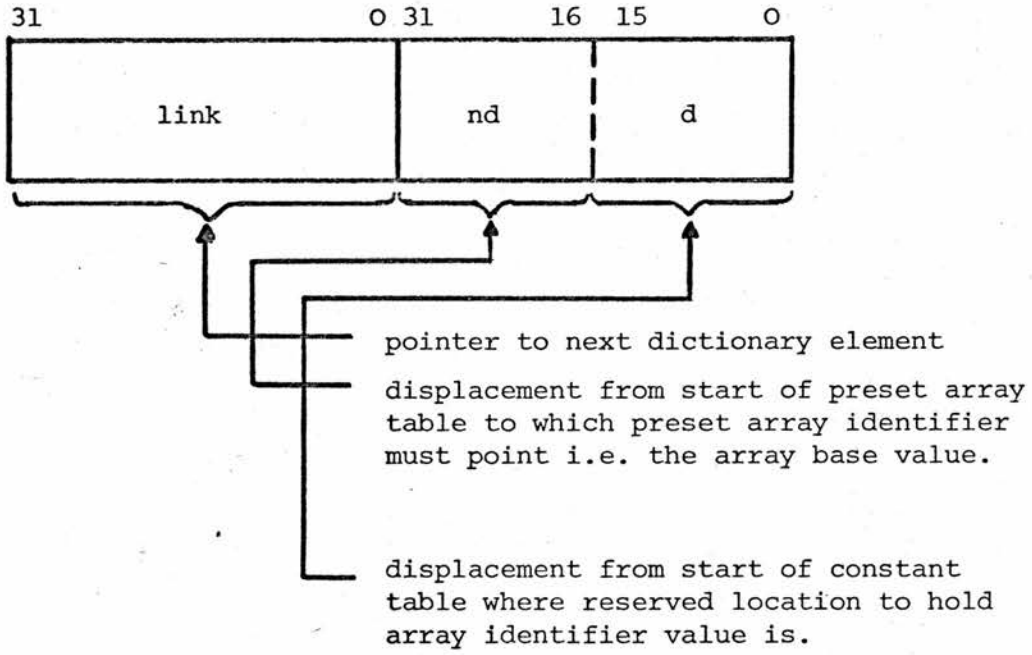
(a) the reserved location in the constant table
(b) the corresponding identifier plex
(c) the corresponding entry in the preset array dictionary



a) constant table



b) (array) identifier plex



c) preset array dictionary element

Fig. 5.6 : Preset Array Handling

5.7 The Constant Table

The constant table is used at compile time to hold all the constants referred to by the object code generated for a given RCCT program. If different routines require access to the same constant (at run-time) then both routines will refer to the same storage location in the constant table. Thus only one copy of each required constant is kept.

At compile-time, only two routines refer to the constant table. One is the semantic routine associated with the PRESET ARRAY statement, and the other is the "ACCESS CONSTANT c GIVING POSITION p" routine.

When a semantic routine requires code to be compiled to access an element, it tests the tag bits in the plex of that element to ascertain whether or not it is a constant ("constant bit" set). If it is, then the ACCESS CONSTANT.... routine is called. (Note that a semantic routine makes no distinction between an ordinary constant plex and a preset array identifier plex). When this routine is called it is passed one input parameter "c" which is a pointer to a constant plex. The result parameter "p" passed back to the calling routine is the displacement of the required constant or address constant from the start of the auxiliary data area. To compute "p", the ACCESS CONSTANT .... routine takes the following action, after checking to see whether the "preset array bit" of the plex is set:

(i) If it is, then the second word of the plex contains the displacement of the required address constant from the start of the constant table. So the global constant "constant area", initialised by the compiler and giving the displacement of the base of the constant table from the start of the auxiliary data area, is added to the contents of the second word of the plex to give "p".

(ii) If it isn't, then the second word of the plex contains an ordinary constant value. A search is made of the constant table, starting at its base, and ending at the position indicated by "current free constant position", to see if the given constant is already present in the table. If the constant is found, then its position in the table is used to construct "p", otherwise, the given constant is added to the table (at the position indicated by "current free constant position", which is then updated) and this position is used to construct "p".

One problem that arises with the technique adopted for reserving space in the constant table for preset array identifiers (address constants) is that the reserved location must contain a value, and care must be taken that this value (which will be altered before the object program executes) is not accessed as if it were an ordinary constant (by the ACCESS CONSTANT ... routine). The solution to this problem was to initialise the constant table at the beginning of compilation such that it already contains one entry, the constant zero. Then whenever a location is reserved for a preset array identifier address constant, this location is set to zero. Now any attempt by the ACCESS CONSTANT .... routine to find a zero in the constant table will find the one initialised by the compiler, and hence no ambiguities will arise.

5.8 I/O Organisation

At present the compiler allows an RCCT program to read card images and to write line images (under format control). Reading is from SYSIPT (the card reader), and writing to SYSOPT (the line printer) [2]. Since I/O operations are machine dependent, and the decision was made to provide only very primitive I/O operations in the language, there was the problem of achieving compatibility between code compiled to perform I/O operations, and other generated code. The IBM conventions for I/O [2], require that the I/O request is executed using the appropriate I/O Supervisor-Call instruction; that certain registers are used for special functions; and that certain data items are available. All the data required is initialised by the compiler and stored in the I/O buffer area residing at the base of the auxiliary data area.

To implement the basic I/O operations, two macros were written in /360 Assembler code to perform the appropriate operations. This was then converted into its hexadecimal machine-code format and stored in two arrays pointed to by the global variables "read" and "write". Thus the two arrays are included in the data declarations associated with the compiler.

The arrays are, in effect, treated as parameterless macros where each macro assumes that:

(i) General purpose register 0 contains the address of a buffer to be used in the transmission of data.

(ii) All other registers are available for use.

The code contained in a macro body is relocatable. This means that either of the "read" or "write" arrays may be copied directly into the object code without any alterations.

The code generated by the two RCCT I/O statements READ CARD.... and WRITE LINE .... follows the sequence:

(i) Compile code to compute the [EXPRESSION] in the RCCT statement and store the resultant value in register 0.

(ii) Compile code to save all the registers in a save area (at the start of the I/O buffer area).

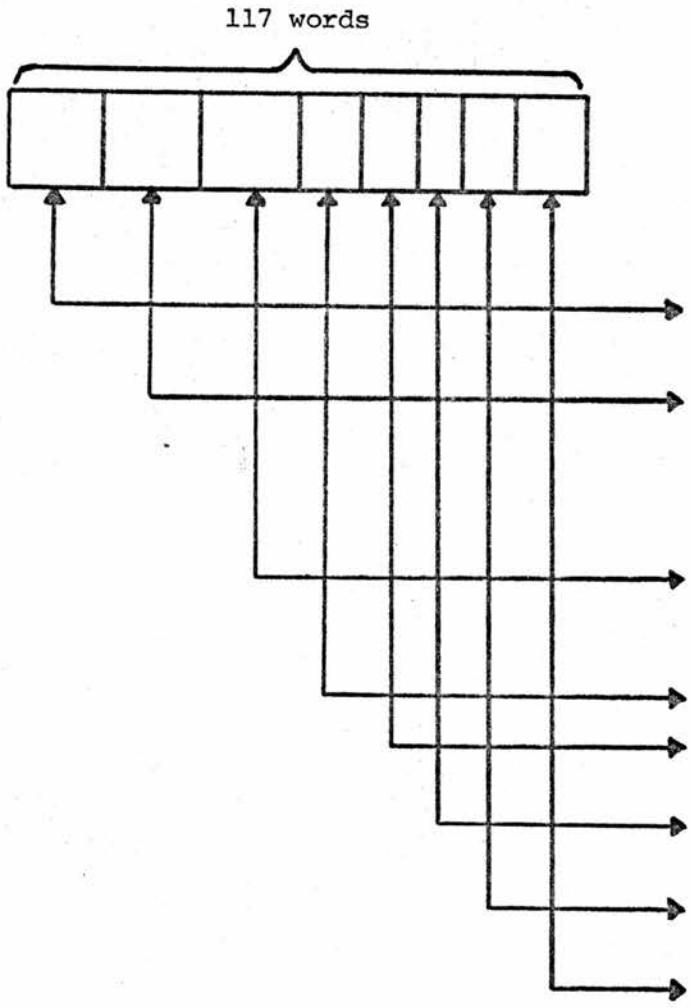
(iii) Copy the corresponding macro into the code stack, updating "code address" and "routine byte count" accordingly.

(iv) Compile code to restore the registers from the save area.

The read macro obeys Algorithm 5.2, and the write macro obeys Algorithm 5.3. The configuration of the I/O buffer area is outlined in figure 5.7.

Algorithm 5.2 (Performs a read operation)

1. Obtain the absolute address of the next instruction and store into register 15. (All branching will be relative to this address).
2. Set up the required parameter list expected for a read operation, using the input buffer address assumed to be in register 0, and set register 1 to point to the parameter list (stored in the I/O buffer area).



18 word	save area (implementation dependent)
34 word	buffer holding runtime message JOB CANCELLED DUE TO READ ERROR
35 words	used as an output buffer for testing purposes
10 word	request control block for read operations
10 word	request control block for write operations
2 words	first word gives read count data, second word gives write count data.
6 words	first 3 words used to hold read parameter list, second 3 words to hold write parameter list.
2 words	containing EBCDIC codes of "/*", "/&", "//".

Note: "request control blocks", "read count data" and "write count data" are explained in [2].

Fig. 5.7 : I/O buffer organisation

3. Call the /360 Supervisor to perform the read operation to completion.

4. Inspect the specified input buffer and test whether a control card has been read (card image commences with "/*", "/&" or "//").

   (a) If one has, then copy the run-time message stored in the I/O buffer area

   JOB CANCELLED DUE TO READ ERROR

   to the output listing and call the Supervisor to cancel the job.

   (b) If one has not, then continue since the read operation was successful.

Algorithm 5.3 (Performs a write operation)

1. Set up the required write operation parameter list using the output buffer address specified in register 0, and set register 1 to point to this list.

2. Call the /360 Supervisor to perform the write operation to completion.

5.9 Assignment Semantics

Two operations are required during the implementation of an assignment:

(i) The expression on the right hand side of the assignment must be computed. The "expression accumulator" is used to accumulate the partial and final results of evaluating an expression in strict "left to right" sequence.

(ii) Any variables in subscripted format must have code compiled to compute the value of the subscript, which will be an absolute run-time address. The "variable address accumulator" is used to accumulate the partial and final results of evaluating a subscript.

The routine which compiles code to compute a subscript value is the "ACCESS VARIABLE v" routine, which is passed a pointer "v" to the plex form of a subscripted variable. The routine which compiles code to evaluate on expression is the "COMPILE CODE FOR EXPRESSION e" routine, which is passed a pointer "e" to an expression plex. This routine follows the forward linked chain of operator/operand pairs, planting code to load the first operand (then to load its complement if the preceding unary operator is the code for minus) into the "expression accumulator". Then for each succeeding operator/operand pair, the operator is used to determine which machine operation is required to access the particular operand. If the operand is a subscripted variable, code is compiled (using the ACCESS VARIABLE .... routine) to evaluate the variable address in the "variable address accumulator" before code is planted to access the operand and perform the correct operation. If the operator is a shift operator, then the succeeding constant operand is shifted down arithmetic two bit positions to cancel the shift made when the plex was constructed (see §2.2), and the corresponding shift instruction is planted which actually incorporates the resulting constant in its structure.

e.g. ..... AU 2..... generates the machine instruction

"SLA expression accumulator, 2" (written in /360 Assembler mnemonics, see [5]).

All constants which are operands not following shift operators are addressed using the "constant base register" (but see below), and all scalars by using the "base register".

The only optimisations made to compiled code are the cases where the expression has a single positive constant operand of either

(i) zero, in which case a single machine instruction "subtract register" is planted to clear the "expression accumulator", or

(ii) less than 4096, in which case a "load address" machine instruction is generated to load the constant value into the "expression accumulator".

Note that in either of these cases, the constant operand is not accessed from the constant table.

Another routine is provided by the compiler to plant code to perform assignments. This is the "ASSIGN v, e" routine which obeys Algorithm 5.4.

Algorithm 5.4 (Assignment)

1. If "e" $\neq$ 0 then it is assumed to point to an expression plex, so compile code to evaluate the given expression in the "expression accumulator", i.e. call the COMPILE CODE .... routine.
2. If "v" = 0 then finish.
3. "v" is assumed to point to a variable plex, so compile code to store the "expression accumulator" into the given variable as follows:
   (a) If the variable is a semantic constant, then monitor a compile-time fault.
   (b) If the variable is a register, then generate a "load register" instruction.
   (c) If the variable is an ordinary scalar, then generate a "store" instruction.

(d) If the variable is a subscripted variable, then

(i) Call the ACCESS VARIABLE .... routine to evaluate the subscript in the "variable address accumulator", then

(ii) Generate a store instruction to store the "expression accumulator" at the address indicated by the "variable address accumulator".

5.10 For-Statement Semantics

The for-statement provided in the RCCT language is implemented as a static cycle, i.e. the increment and limit expressions are computed prior to entry into the cycle and their values stored in hidden locations (in the "scalar area"). The for-statement form which omits a [STEP CLAUSE] is treated as having an implied increment of 1 (see §5.14).

The sequence of actions performed at run-time by a for-statement is:

(i) Assign the value of the initial expression to the control variable.

(ii) Compute and save the increment and limit expressions.

(iii) Compare the control variable with the limit and, if the increment is positive (negative), branch out of the cycle if the control variable is greater than (less than) the limit.

(iv) Execute the statements contained in the cycle.

(v) Increment (decrement) the control variable by the positive (negative) increment and go to (iii). (This is called the "inter-cycle action" code).

Thus, the jump-out-of-cycle test is made prior to an execution of the cycle (which implies that if the initial value of the control variable exceeds (is less than) the limit, then the cycle would never be executed), and the updating of the control variable is performed after the execution of each cycle.

To generate code to obey the above sequencing involves making provisions for two possible end-of-cycle conditions. This means that the computed value of the increment expression must be used to set up a particular condition to be tested for at run-time, and this is costly in terms of the amount of code generated. Some minor optimisations in code generation can be made when the control variable is a register synonym, but there is the problem that three run-time locations (hidden from the user) must be reserved to store the computed increment and limit values, together with a code representing the condition to be tested for (known as the "mask").

Hence, code generation to obey the above sequence of code involves reserving three scalar locations and planting a lot of code corresponding to the for-clause. (The inter-cycle action code, described above, is inserted between (ii) and (iii) and is preceded by a branch instruction to ignore it when the for-statement is first entered). However, quite radical optimisations can be made to the generated code whenever the increment value, and hence the end-of-cycle testing condition, is known at compile-time. This is the case when the increment is a positive or negative constant.

When the increment is a positive constant less than 4096, then there is no need to compute and store this value since it can be added to the control variable, in the inter-cycle action code, directly,

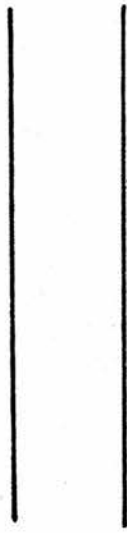
using a load-address machine instruction. In addition, because the sign of the increment is known there is no need to compute the branching condition ("mask" part of the branch instruction) for the end-of-cycle test, so the scalar location which would otherwise be required to hold this value, together with the code required to evaluate the condition, can be saved.

Thus there are eight possible ways to generate code corresponding to a for-clause. Four of them correspond to the case where the control variable is a scalar, and four to the case when it is a register synonym. The actual code generated by the four types of for-statement is summarised in figure 5.8.

When code is compiled for a for-clause, the forward referencing instruction to jump to the end-of-cycle is added to Control List 1 of the current CD block. Any FINISH EACH.... instructions found in the body of the cycle generate a branch instruction which is added to Control List 1, and any FINISH CURRENT.... instructions generate a branch instruction which is added to Control List 2. (A check is made that these statements lie within the scope of a for-statement having the same control variable identifier by chaining down the levels of control data comparing the control variable identifiers with the identifier stored in the relative CD block entry. This ensures that FINISH EACH/CURRENT.... references are added to the correct Control List as well as checking that their context is valid.)

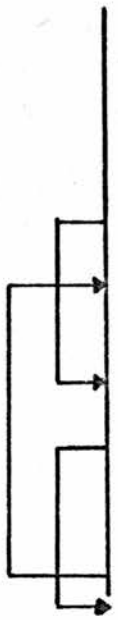
The inter-cycle action code has its starting address stored in the current CD block when the for-clause is compiled. Similarly, the "address" of the control variable (the first word of its identifier) is stored in the current CD block. Both these items are cleared from the CD block by the "PROCESS CONTROL...." routine when the end of the

a) e_2 constant and $0 \leq e_2 < 4096$



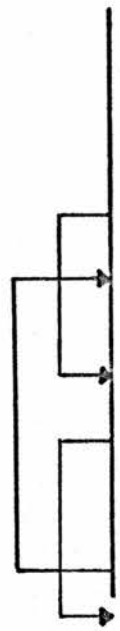
- Assign n = e_1
- Compute and save e_3, call it "limit"
- branch round inter-cycle-action code
- inter-cycle-action code - adds increment using load address instruction
- code to compare n and limit
- branch to end of cycle if n > limit
- code for cycle body
- branch to inter-cycle action code

b) e_2 constant and $e_2 \geq 4096$



- Assign n = e_1
- Compute and save e_3, call it "limit"
- Compute and save e_2, call it "increment"
- branch round inter-cycle action code
- inter-cycle action code - adds "increment" to n
- Compare n with "limit"
- branch to end-of-cycle if n > limit
- code for cycle body
- branch to inter-cycle action

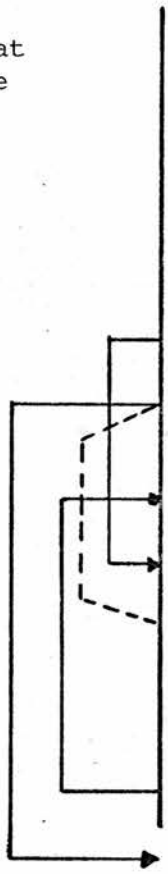
c) e_2 constant and $e_2 < 0$



- Assign n = e_1
- Compute and save e_3, call it "limit"
- Compute and save e_2, call it "increment"
- branch round inter-cycle action code
- inter-cycle action code -- adds "increment" to n
- Compare n with "limit"
- branch to end-of-cycle if n < limit
- code for cycle body
- branch to inter-cycle action code

Fig. 5.8 : Code generated by FOR n = e_1 STEP e_2 TO e_3 :....

d) e_2 undetermined at compile-time



Assign n = e_1

Compute and save e_3, call it "limit"

Compute and save e_2, call it "increment"

Compute and save "mask" from sign of "increment"

branch round inter-cycle action code

branch to end-of-cycle; condition part set to zero

inter-cycle action code

compare n with "limit"

code to use "mask" to execute jump-out-of-cycle branch instruction

code for cycle body

branch to inter-cycle action code

Fig. 5.8 : Code generated by FOR n = e_1 STEP e_2 TO e_3:..... (continued)

for-statement is reached. At the same time, a branch instruction is added to Control List 2 (to cause a branch to the inter-cycle action code when a cycle has been completed), then both Control Lists are filled in, and their list variables set to zero.

Note that figure 5.8d uses a dotted line notation to show how an Execute instruction is used to execute a one-instruction sub-routine consisting of the end-of-cycle testing branch [5].

5.11 If-Statement Semantics

Only one semantic routine is used to generate code for an if-clause. It has to plant code to compute the various conditions that can appear after an "IF", and to determine that a sequence of conditional [RCCT PIECE] form a valid if-clause.

Every conditional [RCCT PIECE] is reduced to the form (see §5.14):

[CONDITIONAL][EXPRESSION][COMPARATOR][EXPRESSION][TERMINATOR EX NL]

and code must be compiled to evaluate the condition. This code has the form:

(i) Compute the expression on the left hand side of the [COMPARATOR] in the "expression accumulator", then store this result in the "comparison register".

(ii) If the expression on the right hand side is the constant zero, then plant an instruction to test the sign of the value in the "comparison register", otherwise, plant code to compute the right hand expression and to compare the results held in the "comparison register" and "expression accumulator".

(iii) Plant an incomplete branch instruction which tests for the [COMPARATOR] condition, and add this instruction to Control List 1 of the current CD block.

The condition to be tested for can depend on the form of [CONDITIONAL] present, its context in an if-clause (determined by

inspecting the level status) and the terminator at the end of the conditional [RCCT PIECE].

Incomplete control references generated during compilation of an if-clause are filled in when their reference address is known. The semantics of an OTHERWISE: statement is an example of this, where Control List 1 can be filled in and another incomplete forward reference is added to it.

Figure 5.9 illustrates the code generated for the various forms of if-statement.

5.12 Code Generation

At present, object code is planted in the codestack by the

"PLANTCODE i, r, d, x, b"

routine, which is given five input parameters:

(i) "i" specifies a machine operation code. (The compiler declares a set of semantic constants corresponding to machine instructions, and this parameter will be one of these semantic constants at the call of this routine)

(ii) "r" identifies the first (register) operand of the instruction.

(iii) "d" specifies the displacement part of a second operand having a base/index/displacement form address (an RX-type instruction), or the displacement part of a second operand having a base/displacement form address (an RS-type instruction), or identifies the second (register) operand of an RR-instruction.

(iv) "x" identifies either the index register used in an RX-type instruction second operand specification, or the third (register) operand in an RS-type instruction, or is undefined for an RR-type instruction.

(v) "b" identifies the base register used to specify the address of a second operand in either an RX- or RS-type instruction, or is undefined for an RR-type instruction.

```
a)   IF condition : statement
b)   IF condition : statement : OTHERWISE : statement
c)   IF condition 1; OR IF condition 2; OR IF condition 3: statement
d)   IF condition 1; OR IF condition 2; OR IF condition 3: statement 1: OTHERWISE:
e)   IF condition 1; AND IF condition 2; AND IF condition 3: statement        statment 2
f)   IF condition 1; AND IF condition 2; AND IF condition 3: statement 1: OTHERWISE:
                                                                    statement 2
```

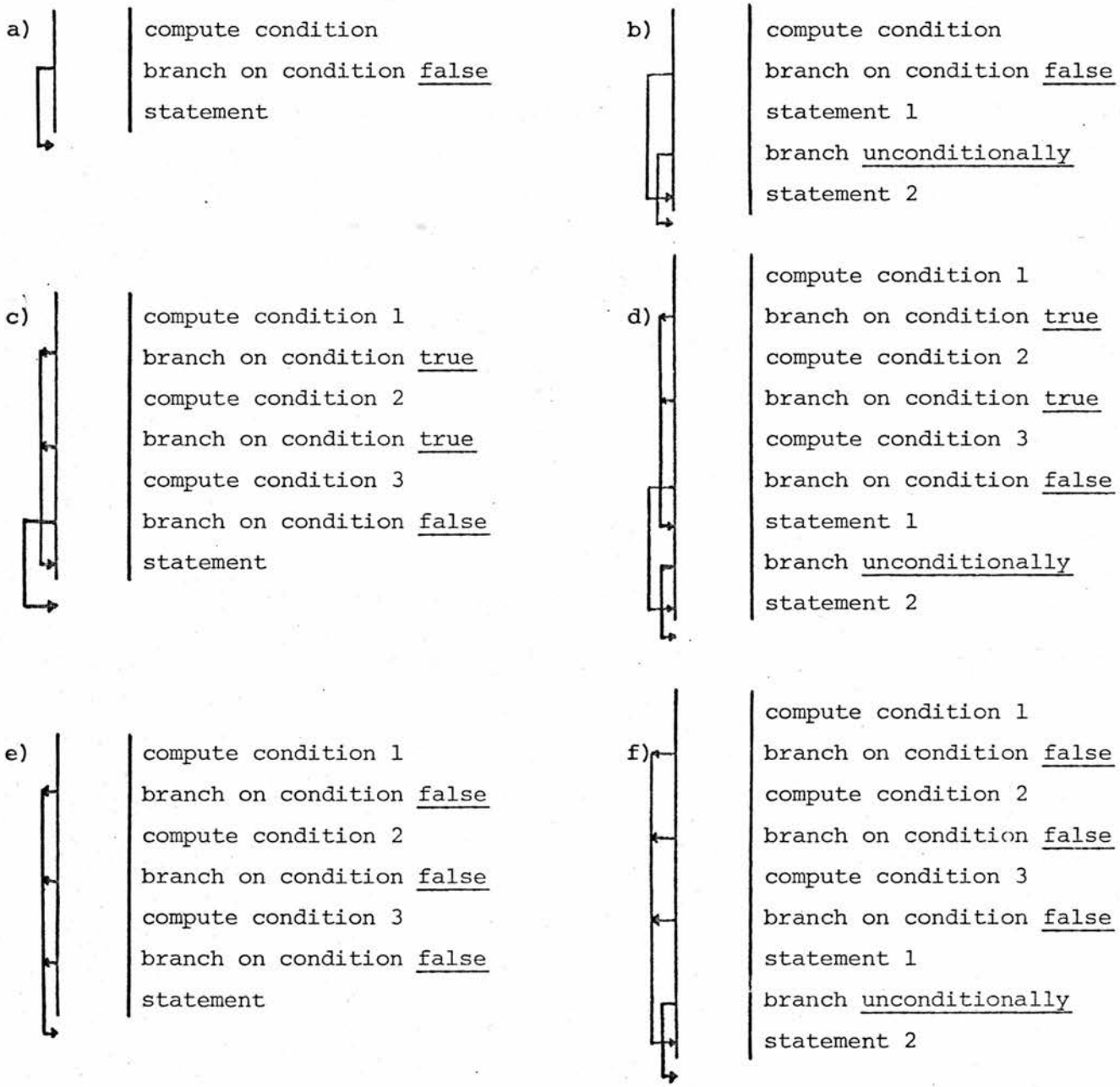


Fig. 5.9 : Code generated by if-statement

For an explanation of RR-, RX- and RS- type instructions, see [6]. The action taken by the PLANTCODE .... routine is:

(i) the operation code "i" is used to identify the length of the required instruction (the set of /360 instructions which can be generated by the compiler may be either 2 or 4 bytes in length).

(ii) the specified machine instruction is constructed and placed at the top of the codestack.

(iii) "code address" and "routine byte count" are updated by the length, in bytes, of the planted instruction.

Notably, the PLANTCODE .... routine performs no checking, such as testing whether the scalar area or constant table has overflowed (indicated by the condition "d" ≥ 4096). This checking is performed at a higher level in the code planting "hierarchy".

The routines that generate machine code form a hierarchical structure, with the PLANTCODE .... routine at the bottom and the semantic (code generation) routines at the top of the structure. Figure 5.10 illustrates this in terms of the actual routines involved, and with reference to this figure:

(i) At top level are the semantic routines. To generate code, these routines use the PLANT .... routine, which has the same parameter list as the PLANTCODE .... routine. Routines at this level are aware of the automatic segmentation techniques and can delay automatic segmentation using the "segmentation delay switch". This switch is typically set when contiguity of a sequence of generated instructions must be guaranteed. However, any routine which sets the switch must switch it off again before finishing. The compiler initialises the switch to be off (zero). The FINISH OFF .... routine is a special semantic routine which specifically wishes to avoid automatic segmentation, since the amount of code it generates is small, and the bulk of the work involved in automatic segmentation is performed by a call of the COMPLETE BRANCHING .... routine. Any routines generating code below this level use the PLANTCODE .... routine.

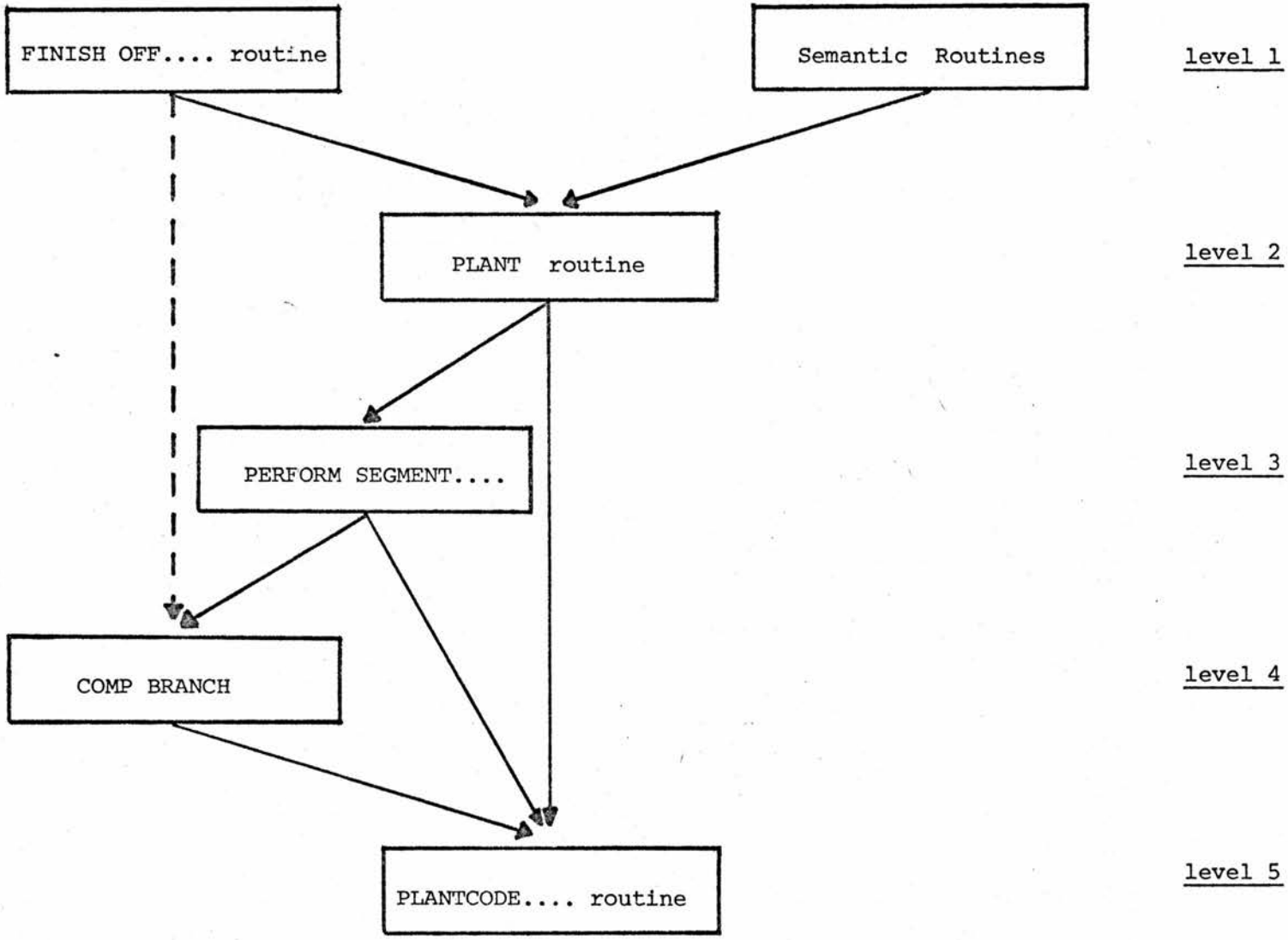


Fig. 5.10 : Hierarchy of code generation routines

(ii) At level two is the PLANT .... routine which simply passes its parameters to the PLANTCODE .... routine if either the "segmentation delay switch" is set, or the "routine byte count" does not exceed 4000. Otherwise, the input parameters are saved, automatic segmentation is invoked by calling the "PERFORM SEGMENTATION PROCEDURE", then the saved parameters are used to call the PLANTCODE .... routine, as before.

(iii) At the third level is the automatic segmentation routine "PERFORM SEGMENTATION PROCEDURE" which calls the COMPLETE BRANCHING .... routine, then plants an instruction to update the "routine index register", increments "index value" by 4000 and decrements "routine byte count" by 4000. This has the effect of declaring a new code segment to continue on from the previous one.

(iv) At level four the COMPLETE BRANCHING .... routine has the responsibility of filling in any incomplete branch instruction whose reference address is already known, of filling in any other (forward) references to the current top of the codestack, and planting the appropriate instructions to update the "routine index register" and branch again. This code is the segmentation code. The process involves checking the list variables associated with any labels or Control Lists. A non-zero list variable implies that there is at least one incomplete branch instruction to be filled in. In the case of a Control List (including the control list associated with FINISH instruction references) or an undefined label reference, the branch instruction is a forward reference to an (as yet) undefined point, so all instructions accessed by the particular list variable are filled in to point to the current free code position on the codestack, then code is planted to update the "routine index register" and branch again. The list variable is then updated to point to this single branch instruction. In the case of references to points defined in the old code segment (the one being completed), the address parts of all referencing instructions can be filled in to directly refer to the referenced point, without necessitating a"hop", and the appropriate list variables can be set to zero. A reference to a point defined in a previous code segment involves filling in these branch instructions to point to the current free code position, then planting code at this position to load the "routine index register" with the index value associated with (identifying) the code segment containing the referenced point, and planting code to branch to that point, then setting the list variable to zero.

The COMPLETE BRANCHING .... routine has one input parameter used as a switch to determine whether it was called by the FINISH OFF .... routine or the automatic segmentation routine. When the call is to perform automatic segmentation, the actions described above are taken, but when the call is to complete compilation of a routine there are two main differences:

(a) Incomplete forward references are noted and will cause one of the following error messages to be output, depending on the type of reference:

UNDEFINED LABEL

INCOMPLETE FOR-STATEMENT

INCOMPLETE IF-STATEMENT

If any of these messages are output, then the routine terminates without executing b).

(b) The Control List associated with FINISH instructions (if any) is filled in to the current codestack position, but no code is generated at that address. The FINISH OFF .... routine will plant code to finish off the routine at this address.

Note that, because of (b), the list variable associated with the FINISH Control List is the last list variable inspected by the COMPLETE BRANCHING .... routine. Also, if no segmentation code is required at a code segment boundary, then at run-time the boundary will be crossed due to normal sequencing of code. Otherwise, a branch instruction must be generated prior to the segmentation code to cause run-time sequential control to "hop" over the segmentation code.

(v) At the bottom level of the code planting hierarchy is the PLANTCODE .... routine.

5.13 Code generated by remaining basic statements

REPEAT statements and goto's simply generate a branch instruction. In the case of a REPEAT, a check is made that the statement is followed by at least one closing compound statement bracket (if not, an appropriate error message is output) and a check is made that the address of the starting point of the compound statement, held in the current CD block, is in the same code segment as the REPEAT statement (this address being held in the segment/word form). If a segment boundary has to be crossed (by a REPEAT branch), an instruction is planted to update the "routine index register" prior to planting a complete branch instruction. A goto generates one incomplete branch instruction which is not filled in until segmentation code is planted. Note that the implementation of a REPEAT always generates a backward reference.

The incomplete branch instruction generated by a goto is added to the Control list associated with the particular label.

Figure 5.11 illustrates the code generated to perform routine linkage, and figure 5.12 shows the code generated by the six remaining basic statement forms which have not already been defined:

(i) GO TO [NAME*/{,}] BY [EXPRESSION]

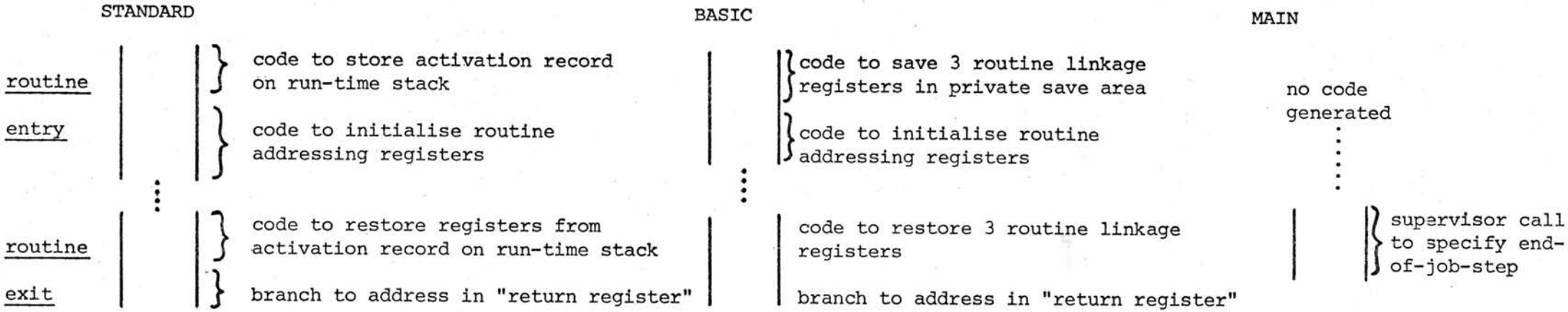
(ii) STACK [EXPRESSION */{,}] AT [NAME]

(iii) DESTACK [VARIABLE */{,}] AT [NAME]

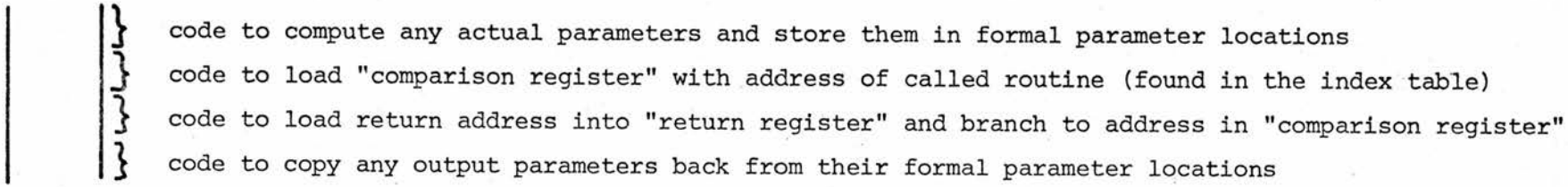
(iv) PERFORM [NAME]

(v) FINISH [NAME]

(vi) $\{\begin{matrix}\text{LET}\\\text{RESOLVE}\end{matrix}\}$ [PHRASE VARIABLE] $\{\frac{\equiv}{\text{INTO}}\}$ [MATCHING PHRASE EXPRESSION]



a) routine entry and exit code



b) routine call code

Fig. 5.11 : Code generated to perform routine linkage

a) GO TO [NAME *] BY [EXPRESSION]

segmentation delayed

- code to compute the expression
- code to load absolute address of next instruction into "variable address accumulator"
- code to branch forward using "expression accumulator" as an index, and "variable address accumulator" as a base
- list of branch instructions equivalent to a list of GO TO references to the list of named labels.

b) STACK [EXPRESSION */⟨,⟩] AT [NAME]

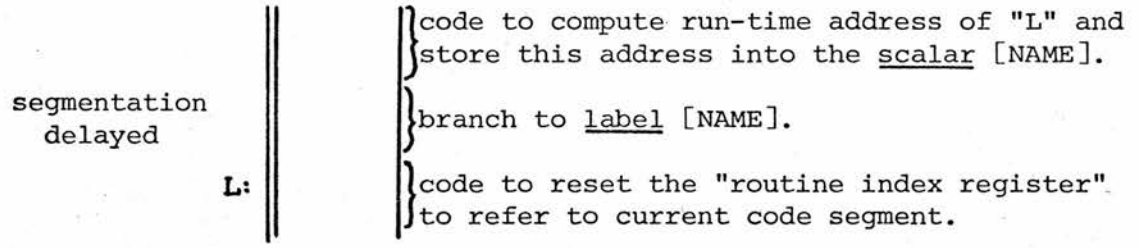
- code to load "comparison register" with scalar [NAME], then "comparison register" is used as an "index", otherwise, the [NAME] register is used as an "index".
- code to compute first expression in "expression accumulator"
- store result in address indicated by "index" register with displacement 0.
- code to compute succeeding expressions (if any) and store in consecutive locations indexed via the "index" register".
- code to update "index" register by correct displacement.
- if [NAME] is a scalar requiring automatic storage allocation, code to store "comparison register" into [NAME].

c) DESTACK [VARIABLE */⟨,⟩] AT [NAME]

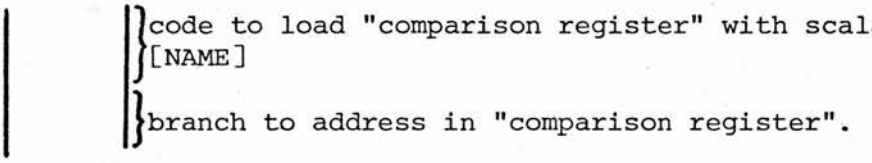
- code to load "comparison register" with [NAME] if it is scalar (as above)
- code to subtract correct displacement from "index" register.
- code to store "index" register in [NAME]if it is scalar.
- code to load "expression accumulator" with first value pointed to by "index" register.
- code to store result in first [VARIABLE].
- code to load successive values and store in successive [VARIABLE]'s.

Fig. 5.12 : Code generated for basic statement forms

d) PERFORM [NAME]



e) FINISH [NAME]



f) $\{\substack{\text{LET}\\\text{RESOLVE}}\}$ [PHRASE VARIABLE] $\{\substack{=\\\text{INTO}}\}$ [MATCHING PHRASE EXPRESSION]

If the [PHRASE VARIABLE] identifier is "p" and the list of phrase variable identifiers in the [MATCHING PHRASE EXPRESSION] are "p1", "p2",...."pn", then the code compiled is equivalent to the sequence of assignments: p1 = p(1); p2 = p(2);....;pn = p(n) except that "p" is used as an index to optimise the assignment code generated, since the value of "p" need only be loaded once.

Fig. 5.12 : Code generated for basic statement forms (continued)

Note that the code generated by a PERFORM [NAME] statement, say, is an example of a sequence of code which must be contiguous. Hence, the "segmentation delay switch" is set during the code generation for this statement. (In this case, contiguity is required since a relative run-time address is required).

5.14 The Primaries

The Primaries are the set of RCCT statements which are, in effect, macros standing for special sequences of basic statements. However, RCCT contains no explicit macro definition structure, so a method was devised to simulate this feature. The solution adopted was to place elements into the Format Table, manually, so that the Primaries are included in the definition of the syntax of valid [RCCT PIECE]'s. Semantic routines were then written whose function is to manipulate the elements in the resultant parse-tree structure into new parse-trees having a form compatible with the corresponding sub-statements for which the Primary is an abbreviation. The appropriate sub-statement semantic routines are then called. For an example of this technique see the "SIMPLE FOR STATEMENT" routine included in Appendix F.

When the Syntax Analyser parses an [RCCT PIECE], the result is a parse tree on the "Analysis Stack" whose head is indicated by the global "plexbase" (see §4.7). If the [RCCT PIECE] is a Primary then the corresponding semantic routine takes the following action:

(i) A new parse tree is constructed at the top of the "Analysis Stack" corresponding to the first sub-statement in the macro-expansion of the Particular Primary.

(ii) "plexbase" is set to point to the head of the new parse tree.

(iii) The appropriate semantic routine is called.

This cycle is then repeated for any further sub-statements in the appropriate macro-expansion.

e.g. the Primary: NEXT * [NAME]

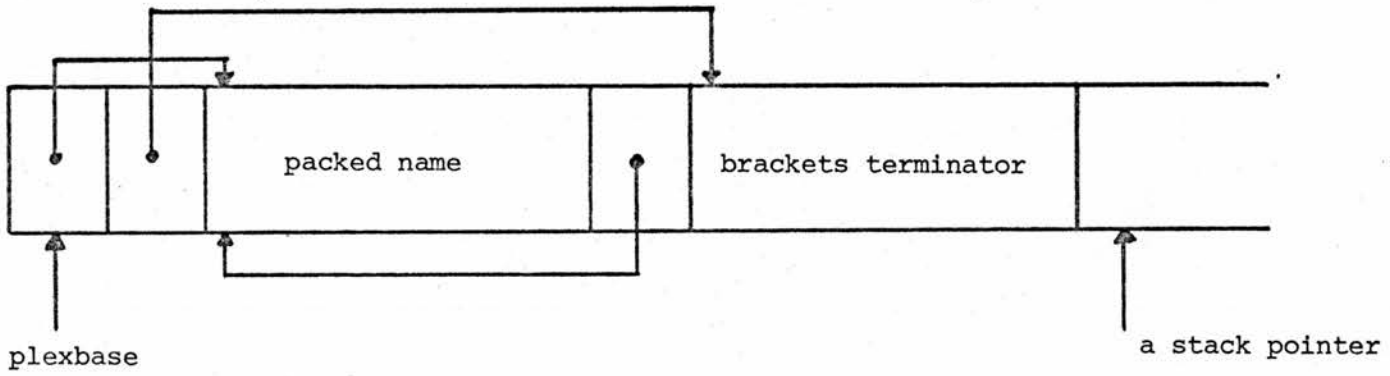
whose macro-expansion is: [NAME] = [NAME](1)

appears in the Format Table as: NEXT * [NAME][BRACKETS TERMINATOR]

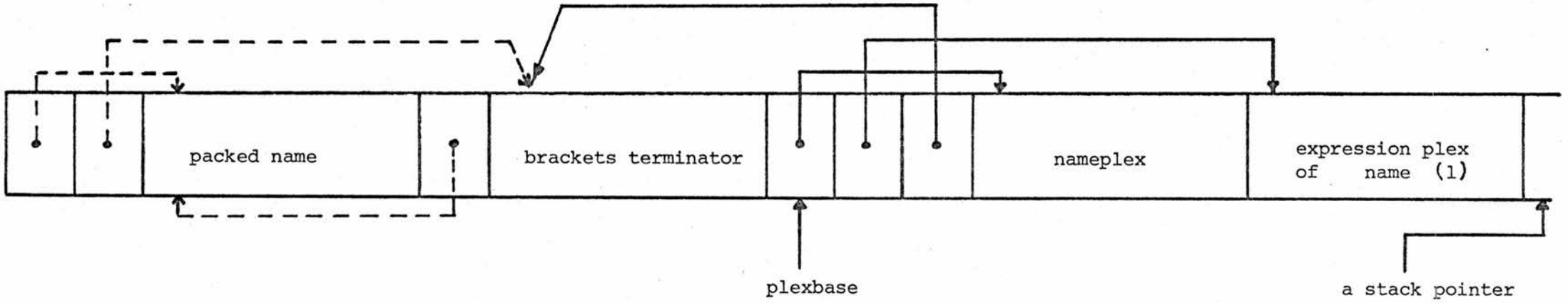
and the parsing routine generates the parse tree defined in figure 5.13a on the "Analysis Stack". Figure 5.13b shows the configuration of the "Analysis Stack" after the semantic routine has re-arranged the parse, altering "plexbase" to point to a new parse tree. The semantic routine then calls the assignment semantic routine which expects (and finds) "plexbase" to point to the parse tree associated with an assignment statement.

Figure 5.14 illustrates the macro-expansion of each Primary statement form i.e. identical code will be generated if the macro-expansion sequences appear in place of the Primaries. (Note that the small letter abbreviations for class names referred to in figure 4.9 have been used).

With reference to figure 5.14 note that if the SET [PHRASE VARIABLE].... Primary appears in an RCCT program, the assumption is made that the program has already defined a global identifier called "main stack front", and that this global will point to the top of a run-time stack defined by the program.



a) Parsing routine output on Analysis Stack



b) Analysis Stack configuration when control passes to the assignment semantics routine

Fig. 5.13 : Example of parse-tree manipulation performed when NEXT * [NAME] [BRACKETS TERMINATOR] is recognised

Primary	Macro-expression	
SET v	v = 1	
SET NO v	v = 0	
NEXT n	n = n + 1	
NEXT-n	n = n - 1	
NEXT*n	n = n(1)	
u v HAS n x	u v&n /= 0 x	
u NO e x	u e = 0 x	
u e x	u e /= 0 x	
u r x	r ; u reg.0 /= 0 x	(register zero used as a boolean switch)
REPEAT UNTIL efe	UNLESS efe : REPEAT	
CONDITION SATISFIED	reg.0 = 1; FINISH	register zero used as a boolean switch
CONDITION NOT SATISFIED	reg.0 = 0; FINISH	
IF p (i) =_ m :	IF (p) = i : LET p =_ m;	
SET p(i) = m	p = main stack front; STACK i, p_1, p_2,...,p_n AT main stack front (where "m" contains the "n" phrase variables p_1, p_2,...., p_n)	

Fig. 5.14 : Macro-expansion of Primaries

5.15 The "PROCESS STATEMENT" routine

The main program of the compiler is simply concerned with initialising and declaring workspace and data items required during compilation. As soon as this initialisation has been completed, control passes to the "PROCESS STATEMENT" routine which is the driving routine of the compiler and obeys Algorithm 5.5.

Algorithm 5.5

1. Call the Lexical Analyser to initialise the contents of the "input buffer".
2. If the global variable "endmet" is non-zero, then
   (a) Print the error message END OF PROGRAM UNSPECIFIED
   (b) Call the /360 Supervisor to cancel the job
3. Reset the "Analysis Stack" by setting "astackpointer" to point to its base, then set "plexbase" = "astackpointer".
4. Call the parsing routine.
5. If parsing is successful, then use the output parameter from the parsing routine to access the corresponding semantic routine, and execute this routine.
6. If the "input buffer" needs refilling, call the Lexical Analyser.
7. Go to 2.

5.16 The "TRANSFER CONTROL" routine

This routine is entered when the Lexical Analyser finds that it has just read in the master word END OF PROGRAM which should delimit the end of a valid RCCT program (see §3.4). The function of this routine is:

(i) To check that all referenced routines have been defined, by scanning all the entries in the routine dictionary checking that each routine plex has its "defined bit" set. For any undefined routine, the error message UNDEFINED ROUTINE is output.

(ii) To check that a MAIN routine has been defined. The semantic routine associated with routine headings at present sets a global variable called "start address" to hold the entry point of any MAIN routine. This global is initialised by the compiler to zero and overwritten each time a MAIN routine heading is found. Hence, the TRANSFER .... routine simply checks that "start address" is non-zero. If it is zero, the error message NO MAIN ROUTINE SPECIFIED is output.

(iii) To check whether the program is executable (which it should be if the "compile time fault detector" has not been set). If it is not, then an appropriate message is output (see Appendix B) and if the "codedump switch" is set, a call is made of a routine, called "DUMPCODE", which prints out the generated code in hexadecimal and assembler format. Then the job is cancelled. If the program is executable, an appropriate message is printed (see Appendix B) and, after performing the above codedump if the"codedump switch" is set, action is taken to transfer control to the object program.

In the current load-and-go testing environment, the action taken to run a program is as follows:

(i) Check that the index table entries contain the correct run-time routine base addresses.

(ii) Use the preset array dictionary elements to initialise the appropriate address constants in the constant table.

(iii) Initialise the run-time registers.

(iv) Having loaded the base address of the MAIN routine into the "routine base register", transfer control to this address.

CHAPTER 6

CONCLUSION

6.1 Development of the Compiler

There have been three distinct stages in the development of the compiler:

(i) Design
(ii) Coding
(iii) Testing

The Design stage has been summarised in §1.2. The Coding Stage turned out to be trivial since the translation of RCCT into PL360 was almost on a one-to-one code generation ratio. However, it was decided that the Testing should be as rigorous as possible, to minimise (or attempt to eliminate) the presence of bugs. Most of the faults found were due to erroneous translation from RCCT to PL360, resulting in syntax violations.

Testing the compiler routines was undertaken in a modular fashion in five stages:

(i) I/O and Lexical Analysis
(ii) Syntax Analysis
(iii) Semantic Processing (code generation)
(iv) The Primaries
(v) Executing compiled programs

In the first stage, the set of output routines were written and tested before the Lexical Analyser. The latter is a fairly complex routine which had to be thoroughly debugged before further testing could proceed. However, once this stage was completed the compiler had the ability to read and list card images and prepare the correct input for the Syntax Analyser.

Stage two involved first testing all the basic Recognition routines, ensuring that the top-down fastback algorithm was implemented correctly. Then came the tedious task of constructing the Recognition and Format Tables (manually) and adding them to the compiler data declarations. The Parsing routine was then coded and thoroughly tested. At this point in the development, the compiler had the ability to construct the parse-tree forms expected by the code generation routines.

Testing the code generation was performed in two phases:

(i) Generated code was listed on the line-printer
(ii) Generated code was placed in the "codestack"

This involved testing all the basic semantic routines and all the routines in the code generation hierarchy except PLANTCODE.... (see §5.12), in the first phase. In this phase, PLANTCODE.... simply constructed and printed line images. Once it was decided that each basic statement form (as opposed to Primaries) was generating the expected code, PLANTCODE .... was altered, and, in the second phase, retested in its final form. At this point in the development, the compiler was generating a complete program which only remained to be executed.

The fourth stage involved "plugging-in" the Primary [RCCT PIECE]'s into the Format Table and checking the code generated by them. Since the Primaries are simply concerned with creating different parse-tree forms on the "Analysis Stack", and the semantic routines they call were already tested, this stage was concerned primarily with checking the validity of the generated parse-trees.

The final stage of development was actually running test programs. Appendix E lists some test programs together with their output and an indication of the size and speed of execution.

Throughout the testing, the main problem has been the construction of test data sufficiently complex to cover all possible cases. Each stage of development has involved greater complexities in this problem, culminating in the execution of compiled programs. Whilst testing has been as thorough as possible, there is always the possibility that bugs may still exist. However, because of the power of the design language, the basic structure of the compiler is sound, and hence its integrity should be high.

6.2 Problems associated with the /360 architecture

Two main problems were found during the implementation of the compiler:

(i) The System /360 branch-to-address instruction
(ii) The byte-addressability of the /360.

The /360 branch-to-address instruction always denotes its address in the base/index/displacement format with a positive displacement of <4096 bytes. Thus, it specifies a forward branch of up to 4096 bytes relative to a location specified as the sum of the contents of two registers. Ideally, a forward or backward branch with no limit on the length of jump is required. However, since one is not available, a device was introduced to obviate the problem of "long" forward, or backward jumps. This was the segmentation of code and the execution of "hops" (multiple jumps) at execution time (see §5.2).

The byte-addressability of the /360 causes problems when implementing a word-oriented programming language. The solution adopted was the conversion of certain constant forms to multiples of four. However, this necessitated the introduction of an implementation-dependent constant, the hexadecimal bit-pattern constant (see §2.2). Even so, the problem is not hidden from the RCCT programmer because of the I/O facilities. It is a programs responsibility to convert card images read in in packed EBCDIC format into a compatible form i.e. each card image contains four eight-bit EBCDIC characters per word and these must be processed with care. Similarly, output line images must be converted into packed EBCDIC format (see §2.4).

6.3 Compatibility with the /360 Operating System

In the current load-and-go testing environment, the only interface between an RCCT object program and the /360 Supervisor program is during an I/O operation. The function of the I/O macros (see §5.8) is to achieve compatibility between object program register conventions and system conventions.

The whole configuration of an RCCT object program and the associated object program data space completely ignores standard IBM conventions for generating object programs. The standard convention is that each routine is compiled as a separate unit. However, this has the disadvantage that each compiled routine must necessarily include relocation information, its local constants,

and information about external references (address constants). The philosophy adopted by the RCCT compiler is that all (routine) base addresses are stored in one compact data area (the index table), and all constants are stored in one compact data area (the constant table). The only relocation information required is held in a simple list (the preset array dictionary). Hence the code corresponding to a routine is completed at compile-time and does not require any alterations prior to execution.

6.4 Conversion to a Standard Compiler

In its final form, the RCCT compiler will be converted from a load-and-go machine into a standard compiler. This implies that the output from the compiler should be in a form familiar to the Operating System. The proposed solution is that the output is organised into a sequential data set whose configuration is as shown in figure 6.1. Referring to this figure:

(i) The Initialisation Routine will be a section of code constructed by the compiler and preceded by a word containing the length of the data set (see below).

(ii) The PADICT will consist of the list of single word elements found in the preset array dictionary at the end of compilation, followed by a word containing a zero.

When a program is deemed to be executable, the "TRANSFER CONTROL" routine will have the responsibility of converting the compiler output into this configuration and copying it out onto disc as a sequence of 360-byte blocks, where the word preceding the Initialisation Routine contains a count of the total number of blocks needed. Now, to

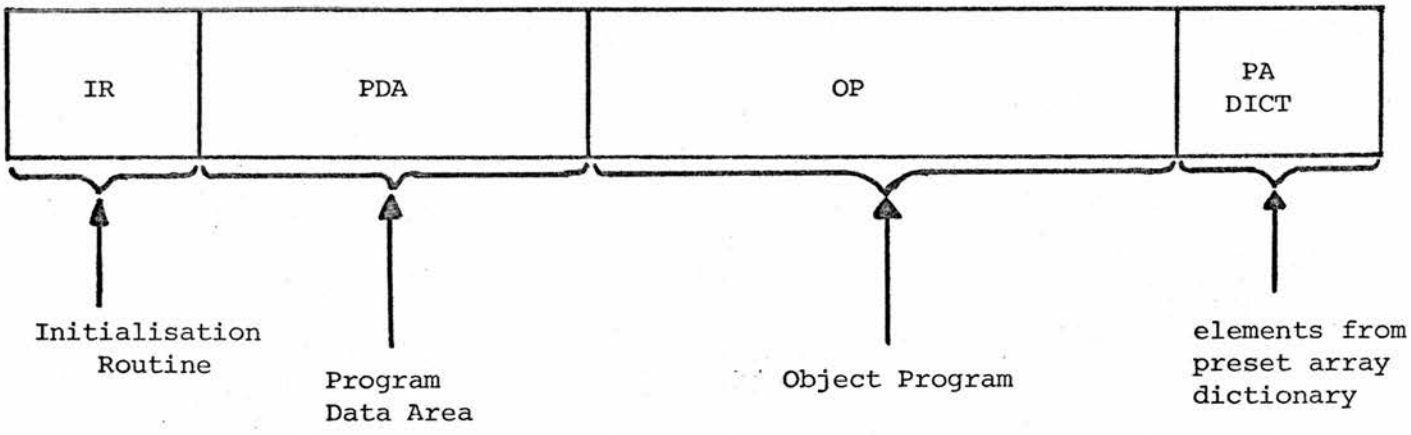


Fig. 6.1 : Output from proposed standard compiler

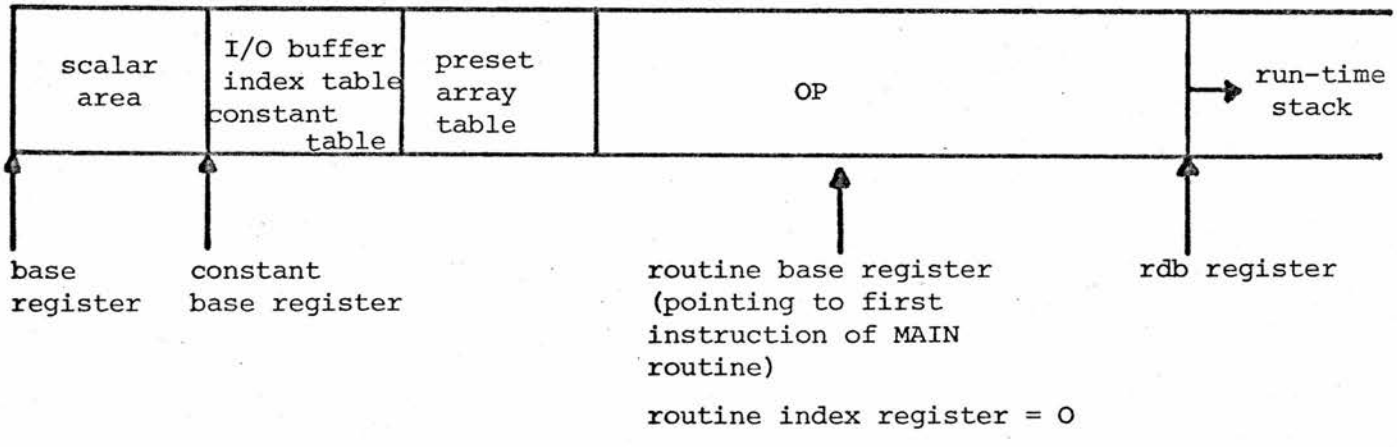


Fig. 6.2 : Main Memory configuration at start of execution of object program

execute the object program, a small (mini-loader) routine

is required whose function will be:

(i) To copy the first block of the disc data set into main memory.

(ii) To find out (from the block count) how many more (360-byte) blocks are needed.

(iii) To copy the remainder of the data set into main memory (so that the entire data set occupies contiguous locations).

(iv) To transfer control to the Initialisation Routine.

The Initialisation Routine must:

(i) Initialise the run-time registers.

(ii) Convert all index table entries to absolute run-time addresses (see below).

(iii) Use the preset array dictionary elements to initialise any preset array address constants, which will be contained in the constant table.

(iv) Transfer control to the main program (contained in the object program).

When control is transferred to the main program, the configuration of the RCCT program and addressing registers will be as in figure 6.2. (Note that the space occupied by the PA DICT will be used to define the start of the run-time stack).

The remaining alterations required in the compiler are:

(i) Instead of storing absolute base addresses in the index table at compile-time, logical i.e. relative addresses should be inserted (see note below).

(ii) At the end of compilation, the preset array dictionary (list) elements should be appended to the object program and the final entry in the index table should be a relative pointer to this list.

(iii) On recognising a MAIN routine, instead of setting "start address" to be the absolute address of the base of the MAIN routine

(a) Set "start address" = 1 (as a switch to indicate that a MAIN routine has been found).

(b) Insert the serial number of the routine into the Initialisation Routine (which will be held as a data array at compile-time). The Initialisation Routine will use this value to transfer control to the main program.

Note: Because an object program may generate an arbitrary amount of code, re-arrange the codestack as a 2 - or 3K word array, and at the end of compilation of a routine, copy out as many blocks of the codestack as possible into the disc data set, and shift the remainder of the code to the base of the codestack, updating the codestack pointer "code address" accordingly.

6.5 Concluding Remarks

The concept of compiler directives (see §3.4) has been borrowed from the PL360 language compiler where it was found extremely useful during program development. It is therefore suggested that the present (single) compiler directive be replaced by the following set of four directives (where underlining indicates the default value):

(i) £DUMP / £NODUMP — to specify that at the end of each routine, a codedump is made of all the machine code generated corresponding to that particular routine (or that codedumps should not be made) onto the line printer.

(ii) £LIST / £NOLIST — to specify that the input to the compiler should (or should not) be listed on the lineprinter.

If it is found that the constant table and/or scalar area and/or preset array table are too small, then an alternate version of the compiler could be constructed in which the scalar area was extended by 1K words (a further segment), and/or the constant table by 1K words, and/or the preset array table by a sufficiently large area. Then,the PLANTCODE .... routine would have the additional function of checking each instruction requiring a displacement part. If any displacement exceeds 1K words, then 1K words would be subtracted from it, and the data base register would be altered to refer to the next data segment i.e. another data register is required. This implies that one or two further permanent run-time register assignments would have to be made to refer to the extra data segment(s), and hence the users register availability would be further restricted. Also, the Initialisation Routine (§6.4) would need to be altered to take account of the resulting enlargement in the Program Data Area.

Another useful extension to the compiler would be the inclusion of further statements to allow I/O operations to and from disc and tape. At present, the machine* generated by the compiler, "the object compiler", only allows input from one source and output to one destination. The input source must be a device transmitting 80 - byte records, i.e. it may be exclusively either a tape unit or card reader. The output must, effectively, be in 133 - byte (formatted) line image records. For compatibility with the system, the following set of I/O statements could be implemented:

* i.e. virtual machine.

Statement	No of bytes transmitted	I/O device (and suggested unit)	
READ CARD TO e	80	card reader	(SYSIPT)
READ TAPE TO e	80	magnetic tape unit	(SYS002)
READ DISC TO e	360	disc	(SYS004)
WRITE LINE FROM e	133	line printer	(SYSOPT)
WRITE CARD FROM e	80	card punch	(SYSPCH)
WRITE TAPE FROM e	80	magnetic tape unit	(SYS003)
WRITE DISC FROM e	360	disc	(SYS006)

The implementation of each additional I/O statement would be in the same form as the two present statements i.e. additional I/O macros would have to be designed, and the additional statements would have to be added to the Format Table.

Register usage and memory configuration of the RCCT compiler and of an object compiler are summarized in Appendix D. The load-and-go compiler needs about 7K words of main memory for its code, about 5K words for working space, and a codestack (at present occupying 10K words, but this will be reduced to about 2K words in the final version of the compiler.) The compiler takes 56 seconds to be compiled by the PL360 compiler under the 44MFT Operating System.

Although the work described in this thesis refers to the load-and-go RCCT compiler, the problems involved in conversion to a standard compiler have been analysed in depth. The actual conversion should require very little coding effort.

REFERENCES

1 "IBM /360 Reference Data"
IBM Publication GX20-1703-9.

2,4 "IBM System /360 Model 44 Programming System
Guide to System Use" IBM Publication Form C28-6812-2.

3 "The Revised Compiler Compiler (A temporary description)"
R.B.E. Napper, Department of Computer Science,
University of Manchester.

5,6 "Assembler Language Programming: The IBM System /360"
G. Stuble. Addison Wesley pub.

7 "PL360 Programmers' Reference Manual"
A. Davie, Department of Computational Science,
University of St. Andrews, Technical Report No. CL/72/6.

8 "Compiling Techniques"
F.R.A. Hopgood. Macdonald/American Elsevier.

APPENDIX A

FORMAL SYNTAX OF RCCT

[PROGRAM] = [RCCT BLOCK */ [EOL]] [EOL] END OF PROGRAM [EOL]

[RCCT BLOCK] = GLOBAL [EOL] [DECLARATION */ξ,∃],
ROUTINE [EOL][RTYPESPEC] : [ROUTINE HEADING PART *] c
[EOL STATMENTS?]

[EOL STATEMENTS] = [EOL] [STATEMENTS]

[STATEMENTS] = [CONTROL STATEMENT] [EOL STATEMENTS ?],
[IMPERATIVE STATEMENT */(; [EOL ?])] [TERM STATEMENTS ?]

[TERM STATEMENTS] = [TERMINATOR] [STATEMENTS]

[TERMINATOR] = [EOL], ; [EOL ?]

[CONTROL STATEMENT] = [IF STATEMENT], [FOR STATEMENT]

[IMPERATIVE STATEMENT] = [L ?] [IMPERATIVE STATEMENT],
[COMPOUND STATEMENT],
LOCAL : [DECLARATION */ξ,∃],
PRESET ARRAY [IDENTIFIER] ([INTEGER]) = c
[CONSTANT */ξ,∃],
[P/(SET)?] [VARIABLE] = [EXPRESSION],
SET [Q/(NO)?] [VARIABLE]
NEXT [Q/(*,-)?] [IDENTIFIER],
[Q/(READ CARD TO, WRITE LINE FROM)] [EXPRESSION],
STACK [EXPRESSION */ξ,∃] AT [IDENTIFIER],
DESTACK [VARIABLE */ξ,∃] AT [IDENTIFIER],
GO TO [LABEL IDENTIFIER */ξ,∃] BY [EXPRESSION],
GO [P/(AND, TO)] [LABEL IDENTIFIER],
FINISH [Q/(CURRENT, EACH)] [IDENTIFIER],
PERFORM [IDENTIFIER],
FINISH [IDENTIFIER ?],
CONDITION [Q/(NOT) ?] SATISFIED,

```
                        REPEAT [REPEAT CONDITION ?],
                        SET [PHRASE VARIABLE] ([INTEGER]) =      c
                                [MATCHING PHRASE EXPRESSION],
                        LET [PHRASE VARIABLE] = [MATCHING PHRASE EXPRESSION],
                        RESOLVE [PHRASE VARIABLE] INTO [MATCHING     c
                                              PHRASE EXPRESSION],
                        [ROUTINE CALL PART *]

[L] = [LABEL] : [EOL ?]

[COMPOUND STATEMENT] = {[EOL ?] [STATEMENTS ?] [EOL ?]}

[REPEAT CONDITION] = UNTIL [SIMPLE CONDITION]

[FOR STATEMENT] = [L * ?] FOR [IDENTIFIER] = [EXPRESSION]   c
                  [STEP CLAUSE ?] TO [EXPRESSION]: [EOL ?]  c
                  [IMPERATIVE STATEMENT */ (; [EOL ?])]

[STEP CLAUSE] =  STEP [EXPRESSION]

[IF STATEMENT] = [L * ?] [IF CLAUSE] : [EOL ?][IMPERATIVE STATEMENT   c
                  */(;[EOL ?])] [OTHERWISE CLAUSE ?]

[OTHERWISE CLAUSE] = :[EOL ?] OTHERWISE : [EOL ?] [IMPERATIVE       c
                       STATEMENT */ (; [EOL ?])]

[IF CLAUSE] = [CONDITIONAL CLAUSE */(; [EOL ?] OR)],
              [CONDITIONAL CLAUSE */(; [EOL ?] AND)],
              IF [PHRASE VARIABLE] = [MATCHING PHRASE EXPRESSION]

[CONDITIONAL CLAUSE] = [Q/(IF, UNLESS)] [CONDITION]

[CONDITION] = [SIMPLE CONDITION],
              [VARIABLE] HAS [IDENTIFIER],
              [Q/(NO) ?] [EXPRESSION],
              [ROUTINE CALL PART *]

[SIMPLE CONDITION] = [EXPRESSION] [Q/(=, ≠, <, ≤, >, ≥)] [EXPRESSION]

[RTYPESPEC] = MAIN, BASIC [Q/(IF, IMP) ?], [STANDARD SPEC]

[STANDARD SPEC] = STANDARD [Q/(IF, IMP) ?], [Q/(IF, IMP)]

[ROUTINE HEADING PART] = [FORMAL PARAMETER], [ROUTINE ID PART]

[ROUTINE CALL PART] = [ACTUAL PARAMETER], [ROUTINE ID PART]
```

[ROUTINE ID PART] = [CAPITAL LETTER], ⦃,⦄

[FORMAL PARAMETER] = [IDENTIFIER], [PHRASE VARIABLE]

[ACTUAL PARAMETER] = [EXPRESSION], @ [VARIABLE], [Q/(@)?] c
[PHRASE VARIABLE]

[DECLARATION] = [IDENTIFIER] = [DECL RT PART], [IDENTIFIER]

[DECL RT PART] = [IDENTIFIER], [CONSTANT], A [INTEGER]

[EXPRESSION] = [Q/(+, -) ?][OPERAND] [OPTR OPND *?]

[OPTR OPND] = [Q/(+, -, &, |, ≠)] [OPERAND], [Q/(AU, AD, LU, LD)] c
[CONSTANT]

[OPERAND] = [VARIABLE], [CONSTANT]

[VARIABLE] = ([IDENTIFIER] [Q/(+, -)] [IDENTIFIER] [Q/(+, -)] c
[CONSTANT]),
([IDENTIFIER] [Q/(+, -)] [IDENTIFIER]),
[VAR A], [IDENTIFIER] [VAR A ?]

[VAR A] = ([IDENTIFIER] [Q/(+, -)] [CONSTANT]),
([IDENTIFIER]), ([CONSTANT])

[PHRASE VARIABLE] = [[CAPITAL LETTER] [PVSYMBOL *?] [IDENTIFIER]]

[PVSYMBOL] = NOT [EOL], ;, :,], [SMALL LETTER], BUT [SYSTEM SYMBOL]

[MATCHING PHRASE EXPRESSION] = [MPE PART *]

[MPE PART] = NOT [EOL], ;, :, BUT [PHRASE VARIABLE], [SYSTEM SYMBOL]

[LABEL IDENTIFIER] = [IDENTIFIER]

[IDENTIFIER] = [SMALL LETTER *] [DIGIT *?]

[LABEL] = [UNDERLINED SMALL LETTER *] [UNDERLINED DIGIT *?]

[CONSTANT] = [INTEGER], * [OCTAL DIGIT *],
#[Q/(#) ?] [HEX DIGIT *],
"[SYSTEM SYMBOL]", % [FIELD SPECIFIER]

[FIELD SPECIFIER] = [BIT SEQUENCE], ([BIT SEQUENCE */⦃,⦄])

[BIT SEQUENCE] = [INTEGER]↦[INTEGER], [INTEGER]

[INTEGER] = [DIGIT *]

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

[DIGIT] = [OCTAL DIGIT], 8, 9

[HEX DIGIT] = [DIGIT], a, b, c, d, e, f

[SMALL LETTER] = a, b, c, ...,z

[UNDERLINED SMALL LETTER] = a, b, c, ..., z

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

[CAPITAL LETTER] = A, B, C, ..., Z

Notes: i) [SYSTEM SYMBOL] is any single EBCDIC coded character, except the single quote character.

ii) [EOL] represents a new line terminator appearing any number of times in sequence.

APPENDIX B

DIAGNOSTICS PRODUCED BY COMPILER

1. Warning Messages

WARNING 01 DECLARATION POSITIONING	A LOCAL: declaration does not follow a routine heading.
WARNING 02 ODD QUOTECOUNT	The Lexical Analyser has found a card with an odd number of single quote symbols appearing on it.
WARNING 03 REDEFINITION	A routine is being redefined.

2. Error Messages

ERROR ... 01 COMMENT BRACKETING	The lexical analyser has detected an unmatched opening or closing comment bracket.
ERROR ... 02 DUPLICATE LABEL	An RCCT label has already been defined in the current routine.
ERROR ... 03 ILLEGAL CONTROL	An ambiguous IF-statement has been found.
ERROR ... 04 INCOMPLETE IF-STATEMENT	On completion of the previous routine, an IF-statement was found to be incomplete.
ERROR ... 05 INCOMPLETE FOR-STATEMENT	On completion of the previous routine, a FOR-statement was found to be incomplete.
ERROR ... 06 INVALID ROUTINE EXIT	Either FINISH appeared inside an IF-routine, or CONDITION (NOT) SATISFIED appeared within an IMP-routine.
ERROR ... 07 MISSING CLOSING BRACKET	On completion of the previous routine, a compound statement was found to be unclosed.

ERROR ... 08 NAME ALREADY DECLARED	A name has been declared again, this declaration is disregarded.
ERROR ... 09 TOO MANY SCALARS	The automatic allocation area for scalars has overflowed.
ERROR ... 10 NO BRACKET AFTER "REPEAT"	A REPEAT statement was not followed by a close compound statement bracket.
ERROR ... 11 NOT EXPECTING FOR-STATEMENT	A FOR-statement appears ambiguously in the scope of another FOR-statement or IF-statement.
ERROR ... 12 PUNCTUATION	The terminator following an imperative statement is invalid.
ERROR ... 13 ROUTINE TYPE	A routine is being defined with a different type to its previous call(s) or definition.
ERROR ... 14 LABEL REFERENCES	The routine contains too many label references.
ERROR ... 15 SEMANTIC CONSTANT ASSIGN	An attempt was made either explicitly or implicitly to assign a value to a semantic constant.
ERROR ... 16 STATEMENT LENGTH	The lexical analyser has found a line of RCCT code exceeding 255 characters
ERROR ... 17 SYNTAX	The statement violates the RCCT Syntax.
ERROR ... 18 TOO MANY CLOSING BRACKETS	An attempt was made to close an unopened compound statement.
ERROR ... 19 UNDEFINED LABEL	On completion of the previous routine, a label was found to be undefined.
ERROR ... 20 UNEXPECTED OTHERWISE	An OTHERWISE: statement was found in an invalid position.
ERROR ... 21 UNRECOGNISED CONTROL NAME	The control variable in a FINISH EACH or FINISH CURRENT statement is undefined, i.e. the statement does not appear in the scope of a FOR-statement.

ERROR ... 22 USE OF SEMANTIC CONSTANT	A PERFORM name or FINISH name or STACK pointer name was found to be a semantic constant.
ERROR ... 23 INVALID ROUTINE CALL	An attempt has been made to call a routine defined as (or assumed to be) an IMP-routine as if it were an IF-routine, or vice versa.
ERROR ... 24 UNDEFINED ROUTINE	At end of compilation, a routine was found to remain undefined.
ERROR ... 25 NO MAIN ROUTINE SPECIFIED	At end of compilation, no main routine has been specified.
ERROR ... 26 END OF PROGRAM UNSPECIFIED	The RCCT compiler has run out of data without finding an END OF PROGRAM master word.
ERROR ... 27 TOO MANY CONSTANTS	The automatic allocation area for constants has overflowed.

3. End Of Compilation Messages

EOC 01 NO ERRORS DETECTED EOC 02 EXECUTION }	These two messages appear if no errors have been found at the end of compilation. Control is transferred to the Object Code.
EOC 03 PROGRAM NON-EXECUTABLE	At end of compilation the compile-time fault detector was set indicating unrecoverable errors in compilation, hence the job is cancelled.

APPENDIX C

SYNTAX DEFINITION USED BY SYNTAX ANALYSER

```
[RCCT PIECE] = GLOBAL [EOL] [DECLARATION * / ⋹ , ⋺ ] [EOL],
               ROUTINE [EOL] [ROUTINE HEADING] [EOL],
               {[EOL?],
               [BRACKETS TERMINATOR],
               [LABEL] : [EOL?],
               OTHERWISE : [EOL?],
               CONDITION [Q/(NOT)?] SATISFIED [BRACKETS TERMINATOR],
               DESTACK [VARIABLE * /⋹, ⋺ ] AT [NAME] [BRACKETS TERMINATOR],
               FOR [NAME] = [EXPRESSION] [STEP CLAUSE?] TO [EXPRESSION] :      C
               [EOL?],
               FINISH [Q / (CURRENT, EACH)] [NAME] [BRACKETS TERMINATOR],
               [Q / (FINISH, PERFORM)] [NAME] [BRACKETS TERMINATOR],
               FINISH [BRACKETS TERMINATOR],
               GO TO [NAME * / ⋹. , ⋺ ] BY [EXPRESSION] [BRACKETS TERMINATOR],
               GO [P / (AND, TO)] [NAME] [BRACKETS TERMINATOR],
               LOCAL : [DECLARATION * / ⋹ , ⋺] [BRACKETS TERMINATOR],
               NEXT [Q / ( * , - ) ? ] [NAME] [BRACKETS TERMINATOR],
               PRESET ARRAY [NAME]  ([INTEGER]) = [CONSTANT * / ⋹ , ⋺ ]        C
               [BRACKETS TERMINATOR],
               REPEAT [REPEAT CONDITION?] [BRACKETS TERMINATOR],
               [P / (SET) ? ] [VARIABLE] = [EXPRESSION] [BRACKETS TERMINATOR],
               SET [Q / (NO) ? ] [VARIABLE] [BRACKETS TERMINATOR],
               STACK [EXPRESSION * / ⋹ , ⋺ ] AT [NAME] [BRACKETS TERMINATOR],
               [Q / (READ CARD TO, WRITE LINE FROM)] [BRACKETS TERMINATOR],
```

```
                [Q / (IF, UNLESS, ORIF, OR UNLESS, AND IF, AND UNLESS)]      C
                [CONDITION] [TERM EX NL],
                SET [PHRASE VARIABLE] ([INTEGER]) = [MATCHING PHRASE.        C
                EXPRESSION] [BRACKETS TERMINATOR],
                LET [PHRASE VARIABLE] ≡ [MATCHING PHRASE EXPRESSION]         C
                [BRACKETS TERMINATOR],
                RESOLVE [PHRASE VARIABLE] INTO [MATCHING PHRASE              C
                EXPRESSION] [BRACKETS TERMINATOR],
                IF [PHRASE VARIABLE] ([INTEGER]) ≡ [MATCHING PHRASE          C
                EXPRESSION] : [EOL?],
                [ROUTINE CALL PART*]  [BRACKETS TERMINATOR]
[ROUTINE HEADING] = [RTYPE SPEC] : [ROUTINE HEADING PART *]
[RTYPE SPEC] =  MAIN, BASIC [Q / (IF, IMP) ? ] , STANDARD [Q / (IF, IMP) ? ] ,
                [Q / (IF, IMP)]
[STEP CLAUSE] = ....
[REPEAT CONDITION] = ....
[CONDITION] = ....
[SIMPLE CONDITION] = ....
[ROUTINE HEADING PART] = ....
[ROUTINE CALL PART] = ....
[ROUTINE ID PART] = ....
[FORMAL PARAMETER] = ....
[ACTUAL PARAMETER] = ....
[BRACKETS TERMINATOR] = } [BRACKETS TERMINATOR],[TERMINATOR]
[TERM EX NL] = : [EOL?], ;[EOL?]
[TERMINATOR] = [EOL], [TERM EX NL]
[DECLARATION  = [NAME] ≡ [DECL RT PART], [NAME]
[DECL RT PART] = [NAME], [CONSTANT], A [INTEGER]
[EXPRESSION] = ....
[OPTR OPND] = ....
[OPERAND] = ....
[VARIABLE] = ([NAME] [Q / (+, -)] [NAME] [Q / (+, -)]  [CONSTANT]),
             ([NAME] [Q / (+, -)] [NAME]),
             [VAR A], [NAME] [VAR A?]
```

[VAR A] = ([NAME] [Q / (+, -)] [CONSTANT]), ([NAME]), ([CONSTANT])

[PHRASE VARIABLE] = ...

[PV SYMBOL] = ...

[MATCHING PHRASE EXPRESSION] = ...

[MPE PART] = ...

[NAME] = [SMALL LETTER *][DIGIT * ?]

[LABEL] = ...

[CONSTANT] = ...

[FIELD SPECIFIER] = ...

[BIT SEQUENCE] = ...

[INTEGER] = ...

[OCTAL DIGIT] = ...

[DIGIT] = ...

[SMALL LETTER] = ...

[UNDERLINED SMALL LETTER] = ...

[UNDERLINED DIGIT] = ...

[CAPITAL LETTER] = ...

Note: the "...." notation means that the definition is the same as in Appendix A.

APPENDIX D

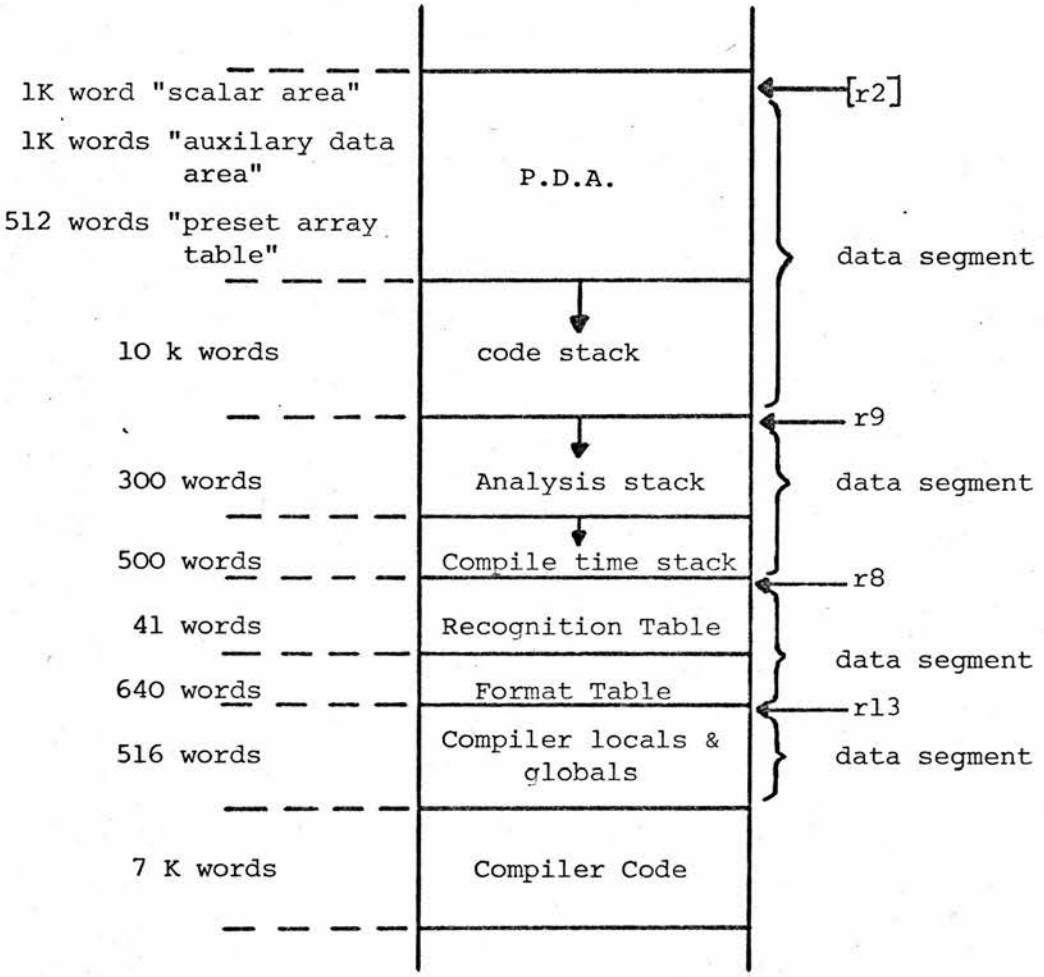
MEMORY AND REGISTER USAGE AT COMPILE AND RUN TIME
(in the load-and-go testing environment)

a) Compile-Time

i) Registers:

0, 1	used in I/O operations
2	return register (also used in initialisation in Main Program as a data base register)
3	"input symbol position"
4	"a stack pointer"
5	"compile time stack front"
6	"code address"
7	NOT USED in load-and-go system
8	data base register
9	data base register
10, 11, 12	working registers
13	data base register
14	I/O return register
15	routine base register

ii) Memory configuration:

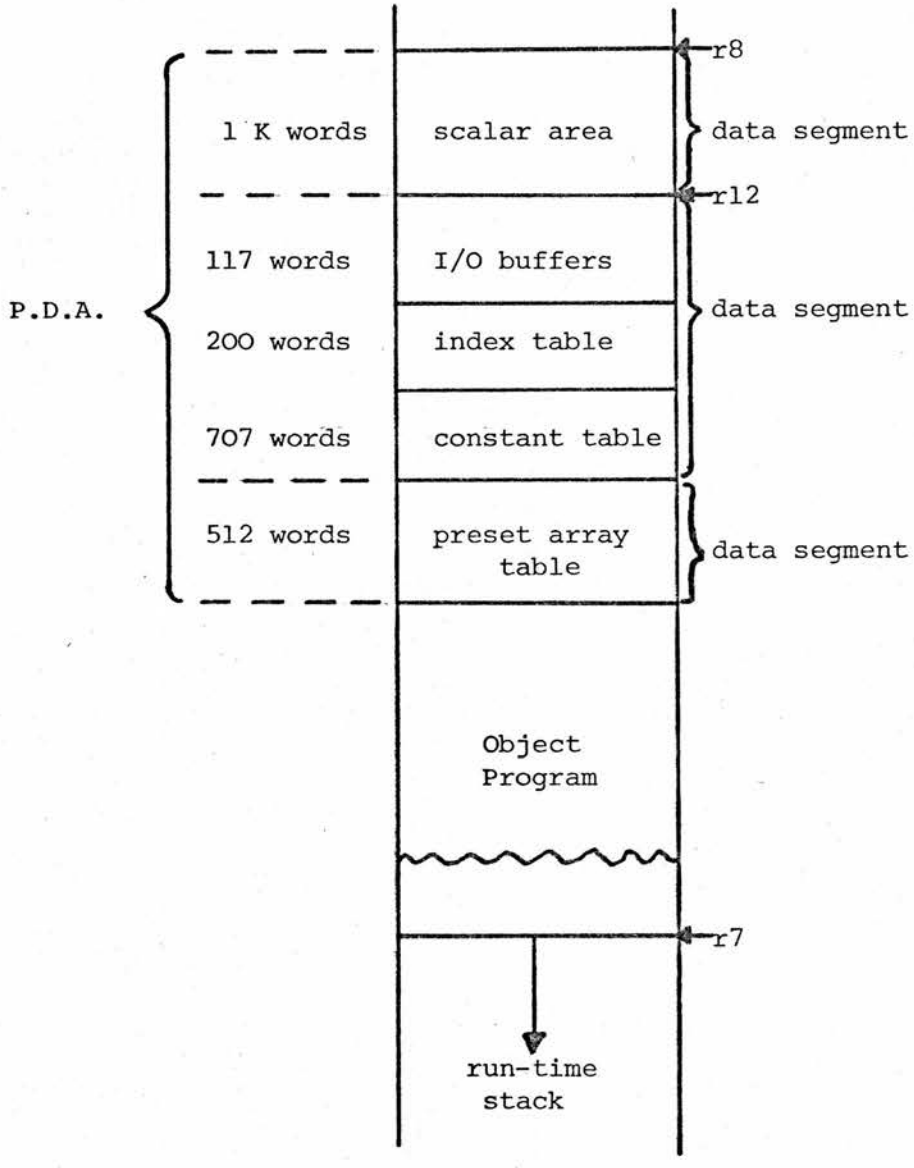


b) Run-Time

i) Registers:

0	used in I/O operations and to hold boolean result of IF-routine
1	available to the RCCT program
2	
3	
4	
5	
6	
7	"rdb register" (run-time stack pointer)
8	"base register"
9	"expression accumulator"
10	"variable address accumulator"
11	"comparison register"
12	"constant base register"
13	"routine index register"
14	"return register"
15	"routine base register"

ii) Memory Configuration



APPENDIX E

TEST PROGRAMS

The three test programs following were run using the final version of the RCCT compiler. This version consists of the load-and-go compiler described in the text with the additional modifications described in §6.4 required to convert it to a standard compiler. (A mini-loader has also been implemented as described in the text). These particular programs have been selected as examples because in addition to displaying some of the capabilities of the compiler they are actually being used in the bootstrapping of RCC onto the IBM 360-44.

The first program, PDTEST1, consists of a set of elementary output routines to be used during the initial steps of the bootstrapping operation for testing purposes. The main program constructs and prints a set of line images and then repeatedly reads in cards and copies them to the line printer. The program listing illustrates:

(i) The source program listing produced by including the "£LIST" compiler directive throughout (note the listing of automatically allocated local scalars - (AA) LOCALS, at the end of each routine).

(ii) The end-of-routine diagnostics message consisting of the routine SERIAL NO., LENGTH (in bytes) and NAME (the "#" symbol indicating a parameter position). This diagnostic format is always listed, whether "£LIST" or "£NOLIST" is in operation.

(iii) The format of dumped machine code specified by compiler directive "£DUMP" as a string of hexadecimal digits partitioned into groups of 8 with a 4-hexadecimal digit address on the left hand side.

(iv) The format of dumped machine code specified by compiler directive "£ASSM" in /360 Assembly code (for the routine "PPRINT e, f") with a 4-decimal digit address on the left hand side.

(v) The run time "JOB CANCELLED DUE TO READ ERROR" message caused by an attempt to read a card from an empty input file. The dumping of this message, causes the program to halt.

This program took ~0.02 seconds to compile, and ~0.01 seconds to execute.

The second program, PDTEST2, contains all the routines used in PDTEST1, which have not been listed (apart from the end-of-routine diagnostic messages), together with a set of routines corresponding to the RCC "CHAIN" routines. The main program tests out these routines. This program took ~0.02 seconds to compile, and ~0.01 seconds to execute.

The third program PDTEST3, consists of all the routines contained in the two previous examples (not listed) together with the routine corresponding to the RCC "READ NEXT LINE" routine. The main program repeatedly calls this routine and prints out the resulting chain. The job terminated after reading all the available data. Compile-time was ~0.03 seconds, and execution time was ~0.10 seconds.

```
THIS LINE PRINTED AT 12.00.06  ;

        PDTEST1PDTEST1PDTEST1PDTEST1PDTEST1PDTEST1PDTEST1PDTEST1PDTEST1PDTEST1PDTEST1PDTEST1PDTEST1PDTEST1P
        DTEST1PDTEST1PDTEST1PDTEST1PDTEST1PDTEST1PDTEST1PDTEST1PDTEST1PDTEST1PDTEST1PDTEST1PDTEST1PDTEST1PD
        TE           DTEST          DTEST1P            1PDT            DTEST          TEST1            T1PDTEST1  T
        ES            EST1           EST1PD            PDTE            TEST           ST1P             1PDTEST1   E
        ST  DTEST1PD  ST1P  EST1PDT  T1PDTEST1P  EST1PDTES  PDTEST1PDTEST1  TEST1PDT  T1PDTEST1  TEST1PDTEST1     S
        T1  TEST1PDT  T1PD  ST1PDTES  PDTEST1PD  ST1PDTEST  DTEST1PDTEST1P  EST1PDTEST1PDTEST1P  EST1PDTEST1PDT  T
        1P  EST1PDTE  1PDT  T1PDTEST  DTEST1PDT  T1PDTEST1  TEST1PDTEST1PD  T1PDTEST1PDTEST1PD  ST1PDTEST1PDTE  1
        PD            PDTE  1PDTEST1  TEST1PDTE  1PDTEST1P        TEST1PDTE          1PDTEST1PDT  T1PDTEST1PDTES  P
        DT           PDTES  PDTEST1P  EST1PDTES  PDTEST1PD        EST1PDTEST          DTEST1PDTE  1PDTEST1PDTEST  D
        TE  1PDTEST1PDTEST  DTEST1PD  ST1PDTEST  DTEST1PDT  T1PDTEST1PDTEST1PDTEST1   EST1PDTES  PDTEST1PDTEST1  T
        ES  PDTEST1PDTEST1  TEST1PDT  T1PDTEST1  TEST1PDTE  1PDTEST1PDTEST1PDTEST1PD  ST1PDTEST  DTEST1PDTEST1P  E
        ST  DTEST1PDTEST1P  EST1PDT  T1PDTEST1P  EST1PDTES  PDTEST1PDTEST1  TEST1PDT  T1PDTEST1  TEST1PDTEST1PD  S
        T1  TEST1PDTEST1PD           T1PDTEST1PD  ST1PDTEST            ST1P            1PDTEST1P  EST1PDTEST
        1P  EST1PDTEST1PDT           T1PDTEST1PDT  T1PDTEST1          T1PDT          1PDTEST1PD  ST1PDTEST1
        PDTEST1PDTEST1PDTEST1PDTEST1PDTEST1PDTEST1PDTEST1PDTEST1PDTEST1PDTEST1PDTEST1PDTEST1PDTEST1PDTEST1P
        DTEST1PDTEST1PDTEST1PDTEST1PDTEST1PDTEST1PDTEST1PDTEST1PDTEST1PDTEST1PDTEST1PDTEST1PDTEST1PDTEST1PD


/PDTEST1  JOB  ,P100    PETE DEWAR                                    11.59.04
A00I  WEDNESDAY   14 MAY       1975                                   11.59.04
/SYS004  ACCESS  PDTEST,DISK=COMS4A                                   11.59.05
/  EXEC  PDRCCT                                                       11.59.05
```

```
001                                    _'ROUTINE'?_
002     'BASIC':  'PLOAD' E
003          'LOCAL': PBR=_'A'12, BIT ONE SET=_##1, BIT TWO SET=_##2,  C_
004            BITS ONE AND TWO SET=_##3
005          OBSP=PBR(86)
006          'IF' OBSP<=##83:
007            (<'IF' OBSP 'HAS' BIT ONE SET;'AND IF' OBSP 'HAS' BIT TWO SET:
008                (<DUMMY=OBSP-BITS ONE AND TWO SET; SHIFT=0;
009                  MASK=##FFFFFF00 >) :
010              'OTHERWISE':
011                (<'IF' OBSP 'HAS' BIT TWO SET:
012                    (<DUMMY=OBSP-BIT TWO SET; SHIFT=8
013                      MASK=##FFFF00FF >):
014                 'OTHERWISE':
015                   (<'IF' OBSP 'HAS' BIT ONE SET:
016                       (<DUMMY=OBSP-BIT ONE SET; SHIFT=16
017                         MASK=##FF00FFFF >):
018                     'OTHERWISE':
019                       (<DUMMY=OBSP; SHIFT=24; MASK=##FFFFFF >) >) >)
020              'FOR' I=1 'TO' SHIFT: E=E'LU'1
021              PBR(DUMMY+52)=PBR(DUMMY+52)&MASK|E >)

SERIAL NUMBER = 000     LENGTH = 0138     NAME = PLOAD#

                                                          (AA) LOCALS :  I
                                                                         MASK
                                                                         SHIFT
                                                                         DUMMY
                                                                         OBSP
                                                                         E

0000    90DF8008  18FB1BDD  18AC5AA0  C4F85890  A0005090  80185890  80181889  41900083
0020    19B9472D  F1325890  80185490  C4FC1299  478DF060  58908018  5490C500  1299478D
0040    F0605890  80185B90  C5045090  801C1B99  50908020  5890C508  50908024  47FDF0D4
0060    58908018  5490C500  1299478D  F0BC5890  80185B90  C5005090  801C4190  00205090
0080    80205890  C5005090  802447FD  F0D45890  80185490  C4FC1299  478DF08C  58908018
00A0    5B90C4FC  50908010  41900040  50908020  5890C510  50908024  47FDF0D4  58908018
00C0    50908010  41900060  50908020  5890C514  50908024  41900004  50908028  18B95890
00E0    80205090  802C47FD  F0F641B0  00045AB0  80285080  80285980  802C472D  F10E5890
0100    80148990  00015090  801447FD  F0EA18AC  5AA08010  5AA0C518  5890A000  54908024
```

```
0120    56908014  18AC5AA0  801C5AA0  C5185090  A00098DF  800807FE


022                                    _'ROUTINE'_
023     'BASIC' : 'PSPACE' E
024          'LOCAL' : PBR=_'A'12
025          (_OUTPUT BUFFER SYMBOL POSITION AT)_PBR(86)=E'AD'2+PBR(86)

     SERIAL NUMBER = 004     LENGTH = 002A     NAME = PSPACE#

                                                                  (AA) LOCALS :  E

0000    90DF8030  18FB18DD  5890803C  8A900002  18AC5AA0  C4F85A90  A00018AC  5AA0C4F8
0020    5090A000  98DF8030  07FE0000


026                                    _'ROUTINE'_
027     'BASIC' : 'PNEWLINE' E
028          'LOCAL' : PBR=_'A'12
029          'WRITE LINE FROM' PBR+51
030          PBR(86)=0
031          BLANK=##40404040
032          'FOR' I=0 'TO' 32 : PBR(I+52)=BLANK
033          COUNT=E-1
034          'IF' COUNT :
035            (< 'FOR' I=1 'TO' COUNT : 'WRITE LINE FROM' PBR+51 >)

     SERIAL NUMBER = 008     LENGTH = 0108     NAME = PNEWLINE#

                                                                  (AA) LOCALS :  COUNT
                                                                                 I
                                                                                 BLANK
                                                                                 E

0000    90DF8040  18FB18DD  189C5A90  C51C1809  90DBC00C  41A0C15C  4170A028  18804180
0020    80034190  A0544110  A0649079  100018BF  0A050A06  18FB98DB  C00C1899  18AC5AA0
0040    C4F85090  A0005890  C5205000  80501B99  50908054  18B94190  00805090  805847FD
0060    F06B4180  00045AB0  80545080  80545980  80584720  F08C5890  805018AC  5AA08054
0080    5AA0C518  5090A000  47FDF062  5890804C  5B90C524  5090805C  5890805C  1299478D
00A0    F1024190  00045090  805418B9  5890805C  50908060  47FDF0C4  41B00004  5AB08054
00C0    50B08054  59B08060  472DF102  189C5A90  C51C1809  90DBC00C  41A0C15C  4170A028
00E0    18804180  80034190  A0544110  A0649079  100018BF  0A050A06  18FB98DB  C00C47FD
```

```
0100    F0B898DF  804007FE


036         _'ROUTINE'_
037     'BASIC:PPRINT TRUE HEX' E
038         'LOCAL' : PBR=_'A'12
039         I=##F0000000
040         'FOR' J=0 'TO' 7 :
041           (< DUMMY=E&I
042           'FOR' K=1 'TO' J'XOR'7 : DUMMY=DUMMY'LD'4
043           DUMMY=DUMMY+##F0
044           'IF' DUMMY>=##FA : DUMMY=DUMMY-##39
045           'PLOAD' DUMMY
046           PBR(86)=PBR(86)+##1
047           I=I'LD'4 >)

    SERIAL NUMBER = 00C     LENGTH = 00E6     NAME = PPRINTTRUEHEX#

                                                          (AA) LOCALS :  K
                                                                         DUMMY
                                                                         J
                                                                         I
                                                                         E

0000    90DF8064  18FB1BDD  5890C528  50908074  1B995090  807818B9  4190001C  5090807C
0020    47FDF030  41B00004  5AB08078  50B08078  59B0807C  472DF0E0  58908070  54908074
0040    50908080  41900004  50908084  18B95890  80785790  C52C509J  808847FD  F06A41B0
0060    00045AB0  80845080  808459B0  8088472D  F0825890  80808890  00045090  808047FD
0080    F05E5890  80805A90  C5305090  80805890  80801889  419000FA  19B9474D  F0AA5890
00A0    80805B90  C5345090  80805890  80805090  801458B0  C1D405EB  18AC5AA0  C4F85890
00C0    A0065A90  C4FC18AC  5AA0C4F8  5090A000  58908074  88900004  50908074  47FDF024
00E0    98DF8064  07FE0000


048                                 _'ROUTINE'_
049     'BASIC' : 'PPRINT SYM' E
050         'LOCAL' : PBR=_'A'12
051         'PLOAD' E'LD'2
052         PBR(86)=PBR(86)+##1

    SERIAL NUMBER = 010     LENGTH = 0038     NAME = PPRINTSYM#
```

```
                                                                    (AA) LOCALS :  E

0000    90DF808C  18FB18DD  58908098  88900002  50908014  58B0C1D4  05EB18AC  5AA0C4F8
0020    5890A000  5A90C4FC  18AC5AA0  C4F85090  A00098DF  808C07FE


053                                     _'ROUTINE'_
054     'BASIC' : 'PPRINT HEX' E
055          E=E'LD'2
056          'PPRINT TRUE HEX' E

        SERIAL NUMBER = 014     LENGTH = 0028     NAME = PPRINTHEX#

                                                                    (AA) LOCALS :  E

0000    90DF809C  18FB18DD  589080A8  88900002  509080A8  589080A8  50908070  58B0C1E0
0020    05EB98DF  809C07FE


057                                     _'ROUTINE'_
058     'BASIC' : 'PPRINT' E
059          'LOCAL' : PBR=_'A'12
060          E=E'AD'2
061          CHECK=PBR(86); PBR(86)=CHECK+PBR(85)
062          'IF' E<0 : SIGN= "-";E=-E: 'OTHERWISE' : SIGN=0
063          DUMMY=E
064          (< N=0
065          'IF' DUMMY>=##186A0:
066            (<DUMMY=DUMMY-##186A0; N=N+##2710
067            'IF' DUMMY>=##186A0:'REPEAT' >)
068          'IF' DUMMY>=##3E8:
069            (<DUMMY=DUMMY-##3E8;N=N+##64
070            'IF' DUMMY>=##3E8:'REPEAT' >)
071          'IF' DUMMY>=##A :
072            (< DUMMY=DUMMY-##A; N=N+##1; 'IF' DUMMY>=##A : 'REPEAT' >)
073          PBR(86)=PBR(86)-##1
074          'IF' PBR(86)<CHECK : 'GO TO' ERROR EXIT
075          'PLOAD' DUMMY|##F0
076          'IF' N>0 : DUMMY=N;'REPEAT' >)
077          'IF' SIGN/=0 :
078            (< PBR(86)=PBR(86)-##1
079            'IF' PBR(86)<CHECK : 'GO TO' ERROR EXIT
```

```
080           'PPRINT SYM' SIGN >)
081         PBR(86)=CHECK+PBR(85)
082         'FINISH'
083     _ERROR EXIT_: PBR(86)=CHECK
084         'FOR' I=1 'TO' PBR(85)'AU'2 :'PPRINT SYM'"*"
```

SERIAL NUMBER = 018 LENGTH = 023C NAME = PPRINT#

(AA) LOCALS : I
N
DUMMY
SIGN
CHECK
E

```
0000    90DF80AC  18FB18DD  58908088  8A900002  50908088  18AC5AA0  C4F85890  A0005090
0020    80BC5890  80BC18AC  5AA0C538  5A90A000  18AC5AA0  C4F85090  A0005890  80B81299
0040    47ADF05A  41900180  509080C0  58908088  13995090  80B847FD  F0601B99  509080C0
0060    58908088  509080C4  1B995090  80C85890  80C418B9  5890C53C  19B9474D  F0AA5890
0080    80C45B90  C53C5090  80C45890  80C85A90  C5405090  80C85890  80C418B9  5890C53C
00A0    19B9474D  F0AA47FD  F07E5890  80C41889  419003E8  19B9474D  F0E65890  80C45B90
00C0    C5445090  80C45890  80C85A90  C5485090  80C85890  80C418B9  419003E8  19B9474D
00E0    F0E647FD  F0BA5890  80C418B9  41900004  19B9474D  F1225890  80C45B90  C54C5090
0100    80C45890  80C85A90  C4FC5090  80C85890  80C418B9  41900004  19B9474D  F12247FD
0120    F0F618AC  5AA0C4F8  5890A000  5B90C4FC  18AC5AA0  C4F85090  A00018AC  5AA0C4F8
0140    5890A000  18B95890  80BC19B9  47ADF154  47FDF1E2  589080C4  5690C530  50908014
0160    58B0C1D4  05EB5890  80C81299  47CDF17C  589080C8  509080C4  47FDF068  589080C0
0180    12994780  F1C618AC  5AA0C4F8  5890A000  5B90C4FC  18AC5AA0  C4F85090  A00018AC
01A0    5AA0C4F8  5890A000  18B95890  80BC19B9  47ADF1B8  47FDF1E2  589080C0  50908098
01C0    58B0C1E4  05EB5890  80BC18AC  5AA0C538  5A90A000  18AC5AA0  C4F85090  A00047FD
01E0    F2365890  80BC18AC  5AA0C4F8  5090A000  41900004  509080CC  18B918AC  5AA0C538
0200    5890A000  86900002  509080D0  47FDF21C  41800004  5AB080CC  50B080CC  59B080D0
0220    4720F236  41900170  50908098  58B0C1E4  05EB47FD  F21398DF  80AC07FE
```

```
085                                 _'ROUTINE'_
086     'BASIC' : 'PPRINT' E,F
087         'LOCAL' : PBR=_'A'12
088         PBR(85)=F'AD'2; 'PPRINT' E; PBR(85)=##A
```

SERIAL NUMBER = 01C LENGTH = 003C NAME = PPRINT#,#

```
  0        STM   13,15, 212( 8)
  4        LR    15,11
  6        SR    13,13
  8        L      9, 228( 0, 8)
 12        SRA    9, 2
 16        LR    10,12
 18        A     10,1336( 0,12)
 22        ST     9,    0( 0,10)
 26        L      9, 224( 0, 8)
 30        ST     9, 184( 0, 8)
 34        L     11, 492( 0,12)
 38        BALR  14,11
 40        LA     9,  10( 0, 0)
 44        LR    10,12
 46        A     10,1336( 0,12)
 50        ST     9,    0( 0,10)
 54        LM    13,15, 212( 8)
 58        BCR   15,14
```

```
089          _'ROUTINE'_
090     'MAIN : RCCT TESTRUN'
091     'LOCAL' : IOBUFFER=_'A'3, I=_'A'4, RTSTACKPOINTER=_'A'7
092          I=249; J=256
093          'PPRINT : START'; 'PNEWLINE' 1
094          'PPRINT : TEST SPACE; PSPACE 5; PPRINT: ROUTINE'
095          'PNEWLINE' 1
096          'PPRINT':1**********************; 'PNEWLINE' 1
097          'PPRINT HEX' I; 'PNEWLINE' 1
098          'PPRINT': 2****; 'PNEWLINE' 2
099          'PPRINT SYM' I; 'PNEWLINE' 1
100          'PPRINT': 3****; 'PNEWLINE' 1
101          'PPRINT' I; 'PPRINT': **; 'PNEWLINE' 1
102          'PPRINT': 4****; 'PNEWLINE' 1
103          'PPRINT' I,3; 'PPRINT': ***; 'PNEWLINE' 1
104          'PPRINT' I,2; 'PPRINT': ??; 'PNEWLINE' 1
105          'PPRINT' J; 'PPRINT': ??; 'PPRINT' J,4; 'PPRINT': ??; 'PPRINT' J,3
106          'PPRINT': ??; 'PNEWLINE' 1
107          'PPRINT' -J; 'PPRINT': ??; 'PPRINT' -J,4; 'PPRINT': ??
```

```
08        'PPRINT' -J,3; 'PPRINT': ??; 'PNEWLINE' 1
09        'FOR' I=0 'TO' 127 : 'PPRINT SYM' I
10        'PNEWLINE' 1
11        'FOR' I=128 'TO' 255 : 'PPRINT SYM' I
12        'PNEWLINE' 1
13        'FOR' I=1 'TO' 10 : 'PPRINT' J,I; 'PPRINT': ??
14        'PNEWLINE' 1
15        'FOR' I=1 'TO' 10 : 'PPRINT' -J,I; 'PPRINT': ??
16        'PNEWLINE' 1
17        'PPRINT': 'END'; 'PNEWLINE' 1
18        IOBUFFER = RTSTACKPOINTER; RTSTACKPOINTER = 34 + RTSTACKPOINTER
19        'FOR' I=0 'TO' 33 : IOBUFFER(I) = ##40404040
20        (<'READ CARD TO' IOBUFFER+1; 'WRITE LINE FROM' IOBUFFER; 'REPEAT'>)

SERIAL NUMBER = 020     LENGTH = 0730     NAME = RCCTTESTRUN

                                                                    (AA) LOCALS :  J

0000    419003E4 18494190 04005090 80E858A0 C1584190 00E2429A C0D04190 00E3429A
0020    C0D14190 00C1429A C0D24190 00D9429A C0D34190 00E3429A C0D441A0 A00550A0
0040    C1584190 00045090 804C58B0 C1DC05EB 58A0C158 419000E3 429AC0D0 419000C5
0060    429AC0D1 419000E2 429AC0D2 419000E3 429AC0D3 41900040 429AC0D4 419000E2
0080    429AC0D5 41900007 429AC0D6 419000C1 429AC0D7 419000C3 429AC0D8 419000C5
00A0    429AC0D9 41A0A00A 50A0C158 41900014 50908030 58B0C1D8 05EB58A0 C1584190
00C0    00D9429A C0D04190 00D6429A C0D14190 00E4429A C0D24190 00E3429A C0D34190
00E0    00C9429A C0D44190 00D5429A C0D54190 00C5429A C0D641A0 A00750A0 C1584190
0100    00045090 804C58B0 C1DC05EB 58A0C158 419000F1 429AC0D0 4190005C 429AC0D1
0120    4190005C 429AC0D2 4190005C 429AC0D3 4190005C 429AC0D4 4190005C 429AC0D5
0140    4190005C 429AC0D6 4190005C 429AC0D7 4190005C 429AC0D8 4190005C 429AC0D9
0160    4190005C 429AC0DA 4190005C 429AC0DB 4190005C 429AC0DC 4190005C 429AC0DD
0180    4190005C 429AC0DE 4190005C 429AC0DF 4190005C 429AC0E0 4190005C 429AC0E1
01A0    4190005C 429AC0E2 4190005C 429AC0E3 4190005C 429AC0E4 4190005C 429AC0E5
01C0    4190005C 429AC0E6 41A0A017 50A0C158 41900004 50908040 58B0C1DC 05EB1894
01E0    50908098 58B0C1E8 05EB4190 00045090 804C58B0 C1DC05EB 58A0C158 419000F2
0200    429AC0D0 4190005C 429AC0D1 4190005C 429AC0D2 4190005C 429AC0D3 4190005C
0220    429AC0D4 41A0A005 50A0C158 41900008 50908040 58B0C1DC 05EB1894 50908098
0240    58B0C1E4 05EB4190 00045090 804C58B0 C1DC05EB 58A0C158 419000F3 429AC0D0
0260    4190005C 429AC0D1 4190005C 429AC0D2 4190005C 429AC0D3 4190005C 429AC0D4
0280    41A0A005 50A0C158 41900004 50908040 58B0C1DC 05EB1894 509080B8 58B0C1EC
02A0    05EB58A0 C1584190 005C429A C0D04190 005C429A C0D141A0 A00250A0 C1584190
02C0    00045090 804C58B0 C1DC05EB 58A0C158 419000F4 429AC0D0 4190005C 429AC0D1
02E0    4190005C 429AC0D2 4190005C 429AC0D3 4190005C 429AC0D4 41A0A005 50A0C158
```

```
0300    41900004  50908040  58B0C1DC  05EB1894  509080E0  41900000  509080E4  58B0C1F0
0320    05EB58A0  C1584190  005C429A  C0D04190  005C429A  C0D14190  005C429A  C0D241A0
0340    A00350A0  C1584190  00045090  804C58B0  C1DC05EB  18945090  80E04190  C0085090
0360    80E458B0  C1F005EB  58A0C158  4190006F  429AC0D0  4190006F  429AC0D1  41A0A002
0380    50A0C158  41900004  5090804C  58B0C1DC  05EB5890  80E85090  80B858B0  C1EC05EB
03A0    58A0C158  4190006F  429AC0D0  4190006F  429AC0D1  41A0A002  50A0C158  589080E8
03C0    5090B0E0  41900010  509080E4  58B0C1F0  05EB58A0  C1584190  006F429A  C0D04190
03E0    006F429A  C0D141A0  A00250A0  C1585890  80E85090  80E04190  00C05090  80E458B0
0400    C1F005EB  58A0C158  4190006F  429AC0D0  4190006F  429AC0D1  41A0A002  50A0C158
0420    41900004  5090804C  58B0C1DC  05EB5890  80E81399  509080B8  58B0C1EC  05EB58A0
0440    C1584190  006F429A  C0D04190  006F429A  C0D141A0  A00250A0  C1585890  80E81399
0460    5090B0E0  41900010  509080E4  58B0C1F0  05EB58A0  C1584190  006F429A  C0D04190
0480    006F429A  C0D141A0  A00250A0  C1585890  80E81399  509080E0  41900000  50908064
04A0    58B0C1F0  05EB58A0  C1584190  006F429A  C0D04190  006F429A  C0D141A0  A00250A0
04C0    C1584190  00045090  804C58B0  C1DC05EB  18991849  419001FC  509080EC  47FDF4E6
04E0    41B00004  1A4B5940  80EC472D  F4FE1894  50908098  58B0C1E4  05EB47FD  F4E04190
0500    00045090  804C58B0  C1DC05EB  41900200  18494190  03FC5090  80F047FD  F52441B0
0520    0D041A4B  594080F0  472DF53C  18945090  809858B0  C1E405EB  47FDF51E  41900004
0540    5090804C  58B0C1DC  05EB4190  00041849  41900028  509080F4  47FDF562  41B00004
0560    1A4B5940  80F4472D  F59E5890  80E85090  80E01894  509080E4  58B0C1F0  05EB58A0
0580    C1584190  006F429A  C0D04190  006F429A  C0D141A0  A00250A0  C15847FD  F55C4190
05A0    00045090  804C58B0  C1DC05EB  41900004  18494190  00285090  80F847FD  F5C441B0
05C0    00C41A4B  594080F8  472DF602  589080F8  13995090  80E01894  50908064  58B0C1F0
05E0    05EB58A0  C1584190  006F429A  C0D04190  006F429A  C0D141A0  A00250A0  C15847FD
0600    F5BE4190  00045090  804C58B0  C1DC05EB  58A0C158  419000C5  429AC0D0  419000D5
0620    429AC0D1  419000C4  429AC0D2  41A0A003  50A0C158  41900004  5090804C  58B0C1DC
0640    05EB1897  18394190  00881A97  18791B99  18494190  00845090  80FC47FD  F66441B0
0660    0D041A4B  594080FC  472DF67C  5890C520  18A31AA4  5090A000  47FDF65E  18935A90
0680    C5241809  90DBC00C  05F041A0  C15C4170  A0001880  4190A050  411CA058  90791000
06A0    18BF0A04  0A06185F  18FB5820  A05C1B33  43302000  89300008  43302001  41400004
06C0    19544780  F0544930  A0704780  F0544930  A0724780  F0544930  A0744770  F06C4170
06E0    A0284180  C0484190  A0544110  A0649079  10000A05  0A060A0F  98DBC00C  18931809
0700    90DBC00C  41A0C15C  4170A028  1886A180  80034190  A0544110  A0649079  1000186F
0720    0A050A06  18FB98D8  C00C47FD  F67C0A0E
```

21 _'END OF PROGRAM'_

```
  EOC     01  NO ERRORS DETECTED
SYS004  ACCESS  PDTEST                                          11.59.14
  EXEC  PDLOADER                                                11.59.14
ART
ST SPACE      ROUTINE
**********
```

```
****
      249**
****
49***
*??
      256?? 256??256??
     -256??-256??***??
ABCDEFGHI .<(+|&JKLMNOPQR £*);¬-/STUVWXYZ ,%_>?0123456789:#@'=" ABCDEFGHI .<(+|&JKLMNOPQR £*);¬-/STUVWXYZ ,%_>?012
ABCDEFGHI .<(+|&JKLMNOPQR £*);¬-/STUVWXYZ ,%_>?0123456789:#@'=" ABCDEFGHI .<(+|&JKLMNOPQR £*);¬-/STUVWXYZ ,%_>?012
??**??256?? 256??  256??   256??    256??     256??      256??       256??
??**??***??-256?? -256??  -256??   -256??    -256??     -256??      -256??
ND
    _'GLOBAL'_
    HEAD OF FREE CHAIN,C_
    MAIN STACK FRONT,C_
    START OF=_1,C_
    LINK FROM=_1,C_
    ITEM STACK FRONT,C_
    RCC CONTINUATION SYMBOL
    _'GLOBAL'_
    RCC MARKED SYMBOL EDITING SYMBOL,C_
    RCC LOCAL LINE NUMBER,C_
    FREE CHAIN POSITION=_HEAD OF FREE CHAIN,C_
    ISO NEWLINE=_10,C_
    MARKED SYMBOL MARKER=_%29,C_
    ISO SPACE=_32,C_
    RDB=_'A'7
    _'ROUTINE'_
BASIC : DELETE CHAIN' CHAIN
    'IF NO' CHAIN: 'FINISH'
    OLD HEAD OF FREE CHAIN= HEAD OF FREE CHAIN
    HEAD OF FREE CHAIN = START OF (CHAIN)
    LINK FROM((_END OF )_ CHAIN) = OLD HEAD OF FREE CHAIN
    CHAIN =0
    _'ROUTINE'_
BASIC : ADD CHAIN' C1 'BEFORE CHAIN' C2
    'IF NO' C1: 'FINISH'
    'IF NO' C2: C2=C1 :'OTHERWISE' :
    (< FIRST LINK OF C2= START OF (C2)
    LINK FROM((_END OF)_ C2) = START OF (C1)
    LINK FROM((_END OF)_ C1) = FIRST LINK OF C2 >)
    C1=0
    _'ROUTINE'_
BASIC : ADD CHAIN' C1 'AFTER CHAIN' C2
```

```
    'IF' C2 : (< FIRST LINK OF C2= START OF (C2)
      LINK FROM((_END OF)_ C2) = START OF(C1)
      LINK FROM((_END OF)_ C1)= FIRST LINK OF C2>)
    C2=C1; C1=0
    _'ROUTINE'_
BASIC: SPLIT CHAIN' C 'AFTER' LAST LINK IN FIRST CHAIN    C_
         'INTO' C1 'AND' C2
    'IF' LAST LINK IN FIRST CHAIN= (_END OF)_ C:
      C2=0 :'OTHERWISE':
    (< C2= (_END OF)_ C
    START OF FIRST CHAIN=START OF (C)
    START OF (C2)= LINK FROM(LAST LINK IN FIRST CHAIN)
    LINK FROM(LAST LINK IN FIRST CHAIN)= START OF FIRST CHAIN>)
    (_ END OF)_ C1= LAST LINK IN FIRST CHAIN
    C=0
    _'ROUTINE'_
BASIC : ADD' V 'BEFORE CHAIN' C
    'IF' LINK FROM (HEAD OF FREE CHAIN)=-1: 'PNEWLINE' 1;
      'PPRINT: CHAIN OVERFLOW; PNEWLINE 1; FINISH'
    OLD HEAD OF FREE CHAIN= HEAD OF FREE CHAIN
    HEAD OF FREE CHAIN = LINK FROM( HEAD OF FREE CHAIN)
    (OLD HEAD OF FREE CHAIN)= V
    'IF NO' C: C= OLD HEAD OF FREE CHAIN: 'OTHERWISE':
      LINK FROM( OLD HEAD OF FREE CHAIN)= START OF (C)
    START OF (C)= OLD HEAD OF FREE CHAIN
    _'ROUTINE'_
BASIC : ADD' V'AFTER CHAIN' C
    'ADD' V 'BEFORE CHAIN' @C
    C = LINK FROM (C)
    _'ROUTINE'_
BASIC : INITIALISE FREE CHAIN'
    HEAD OF FREE CHAIN= RDB
    'FOR' I=0 'STEP' 2 'TO'298:
      'SET NO' HEAD OF FREE CHAIN (I);
      HEAD OF FREE CHAIN(I+1)= HEAD OF FREE CHAIN+I+2
    _'ROUTINE'_
BASIC : PRINT OUT CHAIN' CHAIN
    'IF'CHAIN=0:'PPRINT: CHAIN EMPTY; PNEWLINE 3; FINISH'
    DUMMY=CHAIN
    (<'PPRINT'DUMMY; 'PSPACE'2;'PPRINT' (DUMMY); 'PSPACE'2;
    'PPRINT' (DUMMY+1); 'PNEWLINE' 1
    'NEXT'* DUMMY
    'IF' DUMMY /= CHAIN : 'REPEAT' >)
    'PNEWLINE' 5
```

********** JOB CANCELLED DUE TO READ ERROR **********

AOAI-SVC CANCEL
LD PSW FF25000F4000B8D8

********** ST.ANDREWS UNIVERSITY COMPUTING LABORATORY - 44MFT/SPOOLER END-OF-JOB RECORD **********
JOB NAME : PDTEST1 DATE : 75134 START TIME : 11.59.05 ELAPSED : 00.00.14
CPU TIME : 00.00.03 WAIT TIME : 00.00.06 SYSTEM OVERHEAD : 00.00.03
PAGES PRINTED : 12 CARDS PUNCHED : 0 CARDS READ : 208
A GRAND TOTAL OF 858 INPUT/OUTPUT REQUESTS WERE HANDLED FOR THE JOB.
THE AVAILABLE CORE WAS : 112 K OF WHICH AT LEAST 33 K WAS USED
THE APPROXIMATE COST OF THIS JOB WAS £ 0.20
THIS JOB WAS EXECUTED IN THE BACKGROUND PARTITION
THIS LINE PRINTED AT 12.00.59 ;

```
THIS LINE PRINTED AT 11.39.21  ;
        PDTEST2PDTEST2PDTEST2PDTEST2PDTEST2PDTEST2PDTEST2PDTEST2PDTEST2PDTEST2PDTEST2PDTEST2PDTEST2PDTEST2PDTEST2P
        DTEST2PDTEST2PDTEST2PDTEST2PDTEST2PDTEST2PDTEST2PDTEST2PDTEST2PDTEST2PDTEST2PDTEST2PDTEST2PDTEST2PDTEST2PD
        TE            DTEST         DTEST2P           2PDT            DTEST          TEST2           T2PDT
        ES             EST2          EST2PD           PDTE             TEST           ST2P           2PDT
        ST  DTEST2PD  ST2P  EST2PDT  T2PDTEST2P  EST2PDTES  PDTEST2PDTEST2  TEST2PDT  T2PDTEST2  TEST2PDTE  2PDTES
        T2  TEST2PDT  T2PD  ST2PDTES  PDTEST2PD  ST2PDTEST  DTEST2PDTEST2P  EST2PDTEST2PDTEST2P  EST2PDTEST2PDTEST
        2P  EST2PDTE  2PDT  T2PDTEST  DTEST2PDT  T2PDTEST2  TEST2PDTEST2PD  T2PDTEST2PDTEST2PD  ST2PDTEST2PDTEST2
        PD            PDTE  2PDTEST2  TEST2PDTE  2PDTEST2P       TEST2PDTE       2PDTEST2PDT  T2PDTEST2PDTEST2P
        DT           PDTES  PDTEST2P  EST2PDTES  PDTEST2PD       EST2PDTEST       DTEST2PDTE  2PDTEST2PDTEST2P
        TE  2PDTEST2PDTEST  DTEST2PD  ST2PDTEST  DTEST2PDT  T2PDTEST2PDTEST2PDTEST2  EST2PDTES  PDTEST2PDTEST2  T
        ES  PDTEST2PDTEST2  TEST2PDT  T2PDTEST2  TEST2PDTE  2PDTEST2PDTEST2PDTEST2PD  ST2PDTEST  DTEST2PDTEST  DTE
        ST  DTEST2PDTEST2P  EST2PDT  T2PDTEST2P  EST2PDTES  PDTEST2PDTEST2  TEST2PDT  T2PDTEST2  TEST2PDTES  PDTES
        T2  TEST2PDTEST2PD          T2PDTEST2PD  ST2PDTEST          ST2P           2PDTEST2P  EST2PDTES
        2P  EST2PDTEST2PDT         T2PDTEST2PDT  T2PDTEST2          T2PDT         2PDTEST2PD  ST2PDTEST
        PDTEST2PDTEST2PDTEST2PDTEST2PDTEST2PDTEST2PDTEST2PDTEST2PDTEST2PDTEST2PDTEST2PDTEST2PDTEST2PDTEST2PDTEST2P
        DTEST2PDTEST2PDTEST2PDTEST2PDTEST2PDTEST2PDTEST2PDTEST2PDTEST2PDTEST2PDTEST2PDTEST2PDTEST2PDTEST2PDTEST2PD


PDTEST2  JOB  ,P100    PETE DEWAR                                    11.11.46
.001  WEDNESDAY   14 MAY       1975                                 11.11.46
SYS004  ACCESS  PDTEST,DISK=COMS4A                                  11.11.46
  EXEC  PDRCCT                                                      11.11.46
```

SERIAL NUMBER = 000 LENGTH = 0138 NAME = PLOAD#

SERIAL NUMBER = 004 LENGTH = 002A NAME = PSPACE#

SERIAL NUMBER = 008 LENGTH = 0108 NAME = PNEWLINE#

SERIAL NUMBER = 00C LENGTH = 00E6 NAME = PPRINTTRUEHEX#

SERIAL NUMBER = 010 LENGTH = 0038 NAME = PPRINTSYM#

SERIAL NUMBER = 014 LENGTH = 0028 NAME = PPRINTHEX#

SERIAL NUMBER = 018 LENGTH = 023C NAME = PPRINT#

SERIAL NUMBER = 01C LENGTH = 003C NAME = PPRINT#,#

(AA) LOCALS : F
E

```
089         _'GLOBAL'_
090         HEAD OF FREE CHAIN,C_
091         MAIN STACK FRONT,C_
092         START OF=_1,C_
093         LINK FROM=_1,C_
094         ITEM STACK FRONT,C_
095         RCC CONTINUATION SYMBOL


096         _'GLOBAL'_
097         RCC MARKED SYMBOL EDITING SYMBOL,C_
098         RCC LOCAL LINE NUMBER,C_
099         FREE CHAIN POSITION=_HEAD OF FREE CHAIN,C_
100         ISO NEWLINE=_10,C_
101         MARKED SYMBOL MARKER=_%29,C_
102         ISO SPACE=_32,C_
103         RDB=_'A'7


104         _'ROUTINE'_
105     'BASIC : DELETE CHAIN' CHAIN
106         'IF NO' CHAIN: 'FINISH'
107         OLD HEAD OF FREE CHAIN= HEAD OF FREE CHAIN
108         HEAD OF FREE CHAIN = START OF (CHAIN)
109         LINK FROM((_END OF )_ CHAIN) = OLD HEAD OF FREE CHAIN
110         CHAIN =0

    SERIAL NUMBER = 020      LENGTH = 004A      NAME = DELETECHAIN#

                                                    (AA) LOCALS :  OLDHEADOFFREE
                                                                   CHAIN


111         _'ROUTINE'_
112     'BASIC : ADD CHAIN' C1 'BEFORE CHAIN' C2
113         'IF NO' C1: 'FINISH'
114         'IF NO' C2: C2=C1 :'OTHERWISE' :
115         (< FIRST LINK OF C2= START OF (C2)
116         LINK FROM((_END OF)_ C2) = START OF (C1)
117         LINK FROM((_END OF)_ C1) = FIRST LINK OF C2 >)
```

```
18          C1=0
```

SERIAL NUMBER = 024 LENGTH = 0070 NAME = ADDCHAIN#BEFORECHAIN#

(AA) LOCALS : FIRSTLINKOFC2
C2
C1

```
19          _'ROUTINE_'
20      'BASIC : ADD CHAIN' C1 'AFTER CHAIN' C2
21          'IF NO' C1: 'FINISH'
22          'IF' C2 : (< FIRST LINK OF C2= START OF (C2)
23            LINK FROM((_END OF)_ C2) = START OF(C1)
24            LINK FROM((_END OF)_ C1)= FIRST LINK OF C2>)
25          C2=C1; C1=0
```

SERIAL NUMBER = 028 LENGTH = 006C NAME = ADDCHAIN#AFTERCHAIN#

(AA) LOCALS : FIRSTLINKOFC2
C2
C1

```
26          _'ROUTINE'_
27      'BASIC: SPLIT CHAIN' C 'AFTER' LAST LINK IN FIRST CHAIN    C_
28              'INTO' C1 'AND' C2
29          'IF' LAST LINK IN FIRST CHAIN= (_END OF)_ C:
30            C2=0 :'OTHERWISE':
31          (< C2= (_END OF)_ C
32          START OF FIRST CHAIN=START OF (C)
33          START OF (C2)= LINK FROM(LAST LINK IN FIRST CHAIN)
34          LINK FROM(LAST LINK IN FIRST CHAIN)= START OF FIRST CHAIN>)
35          (_ END OF)_ C1= LAST LINK IN FIRST CHAIN
36          C=0
```

SERIAL NUMBER = 02C LENGTH = 0076 NAME = SPLITCHAIN#AFTER#INTO#AND#

(AA) LOCALS : STARTOFFIRSTC
C2
C1
LASTLINKINFIR

```
137         _'ROUTINE'_
138     'BASIC : ADD' V 'BEFORE CHAIN' C
139          'IF' LINK FROM (HEAD OF FREE CHAIN)=-1: 'PNEWLINE' 1;
140            'PPRINT: CHAIN OVERFLOW; PNEWLINE 1; FINISH'
141          OLD HEAD OF FREE CHAIN= HEAD OF FREE CHAIN
142          HEAD OF FREE CHAIN = LINK FROM( HEAD OF FREE CHAIN)
143          (OLD HEAD OF FREE CHAIN)= V
144          'IF NO' C: C= OLD HEAD OF FREE CHAIN: 'OTHERWISE':
145            LINK FROM( OLD HEAD OF FREE CHAIN)= START OF (C)
146          START OF (C)= OLD HEAD OF FREE CHAIN

    SERIAL NUMBER = 030      LENGTH = 0126      NAME = ADD#BEFORECHAIN#

                                                (AA) LOCALS :  OLDHEADOFFREE
                                                               C
                                                               V

147         _'ROUTINE'_
148     'BASIC : ADD' V'AFTER CHAIN' C
149          'ADD' V 'BEFORE CHAIN' @C
150          C = LINK FROM (C)

    SERIAL NUMBER = 034      LENGTH = 003C      NAME = ADD#AFTERCHAIN#

                                                (AA) LOCALS :  C
                                                               V

151         _'ROUTINE'_
152     'BASIC : INITIALISE FREE CHAIN'
153          HEAD OF FREE CHAIN= RD8
154          'FOR' I=0 'STEP' 2 'TO'298:
155            'SET NO' HEAD OF FREE CHAIN (I);
156            HEAD OF FREE CHAIN(I+1)= HEAD OF FREE CHAIN+I+2

    SERIAL NUMBER = 038      LENGTH = 006A      NAME = INITIALISEFREECHAIN

                                                (AA) LOCALS :  I
```

```
157         _'ROUTINE'_
158     'BASIC : PRINT OUT CHAIN' CHAIN
159         'IF'CHAIN=0:'PPRINT: CHAIN EMPTY; PNEWLINE 3; FINISH'
160         DUMMY=CHAIN
161         (<'PPRINT'DUMMY; 'PSPACE'2;'PPRINT' (DUMMY); 'PSPACE'2;
162         'PPRINT' (DUMMY+1); 'PNEWLINE' 1
163         'NEXT'* DUMMY
164         'IF' DUMMY /= CHAIN : 'REPEAT' >)
165         'PNEWLINE' 5

    SERIAL NUMBER = 03C      LENGTH = 0128      NAME = PRINTOUTCHAIN#

                                                          (AA) LOCALS :  DUMMY
                                                                         CHAIN


166             _'ROUTINE'_
167     'MAIN' : 'RCCT TESTRUN'
168         'INITIALISE FREE CHAIN'
169         C1 = 0
170         'PRINT OUT CHAIN' C1
171         'ADD' 1 'BEFORE CHAIN' @C1
172         'PRINT OUT CHAIN' C1
173         'ADD ' 3 'BEFORE CHAIN' @C1
174         'ADD' 5 'BEFORE CHAIN' @C1
175         'PRINT OUT CHAIN' C1
176         'SET NO' C2
177         'ADD' 2 'AFTER CHAIN' @C2
178         'PRINT OUT CHAIN' C2
179         'ADD' 4 'AFTER CHAIN' @C2
180         'ADD' 6 'AFTER CHAIN' @C2
181         'PRINT OUT CHAIN' C2
182         'ADD CHAIN' @C1 'AFTER CHAIN' @C2
183         'PRINT OUT CHAIN' C1;  'PRINT OUT CHAIN' C2
184         DUMMY = C2
185         'NEXT*' DUMMY; 'NEXT*' DUMMY
186         'SPLIT CHAIN' @C2 'AFTER' DUMMY 'INTO' @A 'AND' @B
187         'PRINT OUT CHAIN' C2;  'PRINT OUT CHAIN' A;  'PRINT OUT CHAIN' B
188         'DELETE CHAIN' @A;  'PRINT OUT CHAIN' A
```

SERIAL NUMBER = 040 LENGTH = 0204 NAME = RCCTTESTRUN

(AA) LOCALS : B
A
DUMMY
C2
C1

89 _'END OF PROGRAM_'

EOC 01 NO ERRORS DETECTED
SYS004 ACCESS PDTEST 11.11.55
EXEC PDLOADER 11.11.55
MAIN EMPTY

12829	1	12829
12829	1	12833
12833	5	12831
12831	3	12829
12835	2	12835
12839	6	12835
12835	2	12837
12837	4	12839

AIN EMPTY

12829	1	12835
12835	2	12837
12837	4	12839
12839	6	12833
12833	5	12831
12831	3	12829

IAIN EMPTY

12837	4	12835
12835	2	12837

12829	1	12839
12839	6	12833
12833	5	12831
12831	3	12829

IAIN EMPTY

;

11.11.58

A 36

ST.ANDREWS UNIVERSITY COMPUTING LABORATORY - 44MFT/SPOOLER END-OF-JOB RECORD

JOB NAME : PDTEST2 DATE : 75134 START TIME : 11.11.47 ELAPSED : 00.00.13

CPU TIME : 00.00.03 WAIT TIME : 00.00.00 SYSTEM OVERHEAD : 00.00.02

PAGES PRINTED : 8 CARDS PUNCHED : 0 CARDS READ : 197

A GRAND TOTAL OF 705 INPUT/OUTPUT REQUESTS WERE HANDLED FOR THE JOB.

THE AVAILABLE CORE WAS : 112 K OF WHICH AT LEAST 33 K WAS USED

THE APPROXIMATE COST OF THIS JOB WAS £ 0.16

THIS JOB WAS EXECUTED IN THE BACKGROUND PARTITION

THIS LINE PRINTED AT 11.39.48 ;

```
THIS LINE PRINTED AT 10.42.24  ;
        PDTEST3PDTEST3PDTEST3PDTEST3PDTEST3PDTEST3PDTEST3PDTEST3PDTEST3PDTEST3PDTEST3PDTEST3PDTEST3PDTEST3PDTEST3PDTEST3P
        DTEST3PDTEST3PDTEST3PDTEST3PDTEST3PDTEST3PDTEST3PDTEST3PDTEST3PDTEST3PDTEST3PDTEST3PDTEST3PDTEST3PDTEST3PDTEST3PL
        TE              DTEST          DTEST3P             3PDT              DTEST           TEST3            T3PDT
        ES               EST3           EST3PD              PDTE              TEST            ST3P             3PDT
        ST  DTEST3PD  ST3P  EST3PDT  T3PDTEST3P  EST3PDTES  PDTEST3PDTEST3  TEST3PDT  T3PDTEST3  TEST3PDTE  3PDTES
        T3  TEST3PDT  T3PD  ST3PDTES  PDTEST3PD  ST3PDTEST  DTEST3PDTEST3P  EST3PDTEST3PDTEST3P  EST3PDTEST3PDTES1
        3P  EST3PDTE  3PDT  T3PDTEST  DTEST3PDT  T3PDTEST3  TEST3PDTEST3PD  T3PDTEST3PDTEST3PD  ST3PDTEST3PDTEST3
        PD            PDTE  3PDTEST3  TEST3PDTE  3PDTEST3P       TEST3PDTE          3PDTEST3PDT  T3PDTEST3PDTEST3
        DT            PDTES  PDTEST3P  EST3PDTES  PDTEST3PD       EST3PDTEST          DTEST3PDTE  3PDTEST3PDTEST3P
        TE  3PDTEST3PDTEST  DTEST3PD  ST3PDTEST  DTEST3PDT  T3PDTEST3PDTEST3PDTEST3  EST3PDTES  PDTEST3PDTEST3PDT
        ES  PDTEST3PDTEST3  TEST3PDT  T3PDTEST3  TEST3PDTE  3PDTEST3PDTEST3PDTEST3PD  ST3PDTEST  DTEST3PDTEST3PDTE
        ST  DTEST3PDTEST3P  EST3PDT  T3PDTEST3P  EST3PDTES  PDTEST3PDTEST3  TEST3PDT  T3PDTEST3  TEST3PDTE  3PDTES
        T3  TEST3PDTEST3PD          T3PDTEST3PD  ST3PDTEST          ST3P          3PDTEST3P  EST3PDTES
        3P  EST3PDTEST3PDT          T3PDTEST3PDT  T3PDTEST3          T3PDT          3PDTEST3PD  ST3PDTEST3
        PDTEST3PDTEST3PDTEST3PDTEST3PDTEST3PDTEST3PDTEST3PDTEST3PDTEST3PDTEST3PDTEST3PDTEST3PDTEST3PDTEST3PDTEST3PDTEST3P
        DTEST3PDTEST3PDTEST3PDTEST3PDTEST3PDTEST3PDTEST3PDTEST3PDTEST3PDTEST3PDTEST3PDTEST3PDTEST3PDTEST3PDTEST3PDTEST3PD

/PDTEST3  JOB  ,P100    PETE DEWAR                                    10.35.22
A001  WEDNESDAY   14 MAY       1975                                   10.35.22
/SYS004  ACCESS  PDTEST,DISK=COMS4A                                   10.35.22
/  EXEC  PDRCCT                                                       10.35.22
```

SERIAL NUMBER = 000 LENGTH = 0138 NAME = PLOAD#

SERIAL NUMBER = 004 LENGTH = 002A NAME = PSPACE#

SERIAL NUMBER = 008 LENGTH = 0108 NAME = PNEWLINE#

SERIAL NUMBER = 00C LENGTH = 00E6 NAME = PPRINTTRUEHEX#

SERIAL NUMBER = 010 LENGTH = 0038 NAME = PPRINTSYM#

SERIAL NUMBER = 014 LENGTH = 0028 NAME = PPRINTHEX#

SERIAL NUMBER = 018 LENGTH = 023C NAME = PPRINT#

SERIAL NUMBER = 01C LENGTH = 003C NAME = PPRINT#,#

SERIAL NUMBER = 020 LENGTH = 004A NAME = DELETECHAIN#

SERIAL NUMBER = 024 LENGTH = 0070 NAME = ADDCHAIN#BEFORECHAIN#

SERIAL NUMBER = 028 LENGTH = 006C NAME = ADDCHAIN#AFTERCHAIN#

SERIAL NUMBER = 02C LENGTH = 0076 NAME = SPLITCHAIN#AFTER#INTO#AND#

SERIAL NUMBER = 030 LENGTH = 0126 NAME = ADD#BEFORECHAIN#

SERIAL NUMBER = 034 LENGTH = 003C NAME = ADD#AFTERCHAIN#

SERIAL NUMBER = 038 LENGTH = 006A NAME = INITIALISEFREECHAIN

SERIAL NUMBER = 03C LENGTH = 0128 NAME = PRINTOUTCHAIN#

```
66        _'ROUTINE'_
67    ' IMP: READ NEXT LINE' NEWLINE CHAIN
68        'LOCAL': TOP CHAR FIELD =_ ##FF000000,  C_
69     IC FOR QUOTE =_ #7D, UNDERLINED SPACE =_ #AC
70        'LOCAL': UNDERLINE MARKER =_ %7,    C_
71    ISO FOR EQUALS=_ 61, ISO FOR COMMA =_ 44,      C_
72    FOUR SPACES =_ ##40404040, CONVERTER =_ %0->5, ISO FOR N =_ 78, C_
73    ISO FOR B =_ 66
74        'LOCAL': CHAR POSITION=_'A'1, COUNT=_'A'2,    C_
75          CURRENT SYMBOL WORD=_'A'3, SYMBOL=_'A'4,    C_
76          SYMBOL POSITION=_'A'5, FREE POSITION=_'A'6
77        'PRESET ARRAY' CONVERSION TABLE(0)=          C_
78             32,65,66,67,68,69,70,71,72,73,36,46,60,40,43,124,38,   C_
79             74,75,76,77,78,79,80,81,82,33,35,42,41,59,94,45,47,83,84,  C_
80            85,86,87,88,89,90,32,44,37,95,62,63,48,49,50,51,52,53,      C_
81             54,55,56,57,58,96,64,39,61,34
82        START OF BUFFER AREA = ITEM STACK FRONT + 20
83        START OF FIRST LINE = ITEM STACK FRONT + 40
84        START OF LINE = START OF FIRST LINE
85    _READ NEXT LINE_ : FREE POSITION = START OF LINE
86        'SET NO' UPPER
87        'READ CARD TO ' START OF BUFFER AREA
88        UPPER BOUND = 17
89        (<(< 'IF' START OF BUFFER AREA(UPPER BOUND) = FOUR SPACES:
90          (<'IF NO' UPPER BOUND: 'GO AND' READ NEXT LINE >);
91          UPPER BOUND = UPPER BOUND -1;'REPEAT'>)>)
92        CURRENT SYMBOL WORD = (START OF BUFFER AREA)
93        CHAR POSITION = 1; COUNT = 1
94    _READ NEXT SYMBOL_: 'IF' CHAR POSITION =5:
95        (<'IF' COUNT=UPPER BOUND +1: 'GO TO' END OF LINE
96        CURRENT SYMBOL WORD = START OF BUFFER AREA(COUNT)
97        CHAR POSITION = 1; 'NEXT' COUNT >)
98        SYMBOL= CURRENT SYMBOL WORD&TOP CHAR FIELD 'LD' 22
99        CURRENT SYMBOL WORD = CURRENT SYMBOL WORD 'LU' 8
00        'NEXT' CHAR POSITION
01        'IF' SYMBOL = IC FOR QUOTE :
02          (<'IF' UPPER:'SET NO'UPPER:'OTHERWISE':'SET'UPPER
03          'GO AND' READ NEXT SYMBOL >)
```

```
204         'IF' SYMBOL= "_":
205           (<'IF' FREE POSITION=START OF LINE:
206           (FREE POSITION)= UNDERLINED SPACE;'NEXT' FREE POSITION:
207           'OTHERWISE':
208           (FREE POSITION-1)=(FREE POSITION-1) | UNDERLINE MARKER
209           'GO AND' READ NEXT SYMBOL >)
210         DISPLACEMENT = SYMBOL& CONVERTER
211         'IF' SYMBOL>="'A'";'AND IF'SYMBOL<="'Z'";'AND IF''NO'UPPER:
212           (FREE POSITION)=CONVERSION TABLE(DISPLACEMENT)+32:'OTHERWISE':
213           (FREE POSITION)=CONVERSION TABLE(DISPLACEMENT)
214         'NEXT' FREE POSITION; 'GO AND' READ NEXT SYMBOL
215     _END OF LINE_:(_FIRST REMOVE SPACES AT END OF LINE AND CHECK FOR C_
216           EMPTY LINE )_
217         (<'IF' FREE POSITION= START OF LINE:'GO AND' READ NEXT LINE
218         'IF' (FREE POSITION-1)=ISO SPACE: FREE POSITION=FREE POSITION-1;
219           'REPEAT'   >)
220         'IF' (FREE POSITION -1)= RCC CONTINUATION SYMBOL:
221           'SET' START OF LINE=FREE POSITION-1;'GO AND' READ NEXT LINE
222           (_CONTINUING ON FROM PEVIOUS IMAGE LESS THE CONTINUATION SYMBOL)_
223         'IF' RCC MARKED SYMBOL EDITING SYMBOL (_FOR SYSTEM SET AS   C_
224           OPEN COMMENT )_;'AND IF' FREE POSITION-START OF FIRST LINE>=4:
225         (<(< 'FOR'SYMBOL POSITION=FREE POSITION-3'STEP'-1'TO'     C_
226                         START OF FIRST LINE+1:
227           (< 'IF' (SYMBOL POSITION)= RCC MARKED SYMBOL EDITING SYMBOL;
228         'AND IF'(SYMBOL POSITION+1)=ISO FOR N;
229             'AND IF' (SYMBOL POSITION+2)= ISO FOR B:
230               'GO AND' MARK SYMBOLS >)  >)>)
231     _PUT LINE ONTO CHAIN_: 'NEXT' RCC LOCAL LINE NUMBER
232         HEAD OF CHAIN = FREE CHAIN POSITION
233         'SET' SYMBOL POSITION= START OF FIRST LINE
234         (< (FREE CHAIN POSITION)=(SYMBOL POSITION); 'NEXT' SYMBOL POSITION
235         FREE CHAIN POSITION=(FREE CHAIN POSITION+1)
236         'REPEAT UNTIL' SYMBOL POSITION= FREE POSITION  >)
237         NEWLINE CHAIN= FREE CHAIN POSITION
238         (FREE CHAIN POSITION)= ISO NEWLINE
239         'NEXT'* FREE CHAIN POSITION
240         START OF ( NEWLINE CHAIN)= HEAD OF CHAIN
241         'FINISH'
242     _MARK SYMBOLS_:  START OF EDIT REQUEST= SYMBOL POSITION
243         'SET' SS (_START OF NEXT SUBSTITUTION SEQUENCE E.G. @=A, )_ C_
244             = SYMBOL POSITION+3
245         (<(<'IF' SS+3>FREE POSITION (_I.E. SEQUENCE OVERFLOWS LINE )_;
```

```
0246          'OR IF' (SS+1) /= ISO FOR EQUALS: 'GO AND' ABORT MARKING
0247        SUBSTITUTION SYMBOL= (SS);
0248        MARKED SYMBOL= (SS+2) | MARKED SYMBOL MARKER
0249        'FOR' SYMBOL POSITION=START OF FIRST LINE'TO'START OF EDIT   C_
0250                    REQUEST -1:
0251          (<'IF' (SYMBOL POSITION)= SUBSTITUTION SYMBOL:
0252            'SET' (SYMBOL POSITION)= MARKED SYMBOL >)
0253        'IF' SS+3=FREE POSITION (_ CORRECT END OF EDITING SEQUENCE)_:
0254          (_REMOVE SEQUENCE)_ FREE POSITION= START OF EDIT REQUEST;
0255          'GO TO' END OF LINE
0256        (_AND BEWARE IF THERE IS ANOTHER EDITING SEQUENCE ON THE LINE)_
0257        'IF' (SS+3)= ISO FOR COMMA (_CORRECT SEPARATOR BETWEEN   C_
0258                      SUBSTITUTION SEQUENCES  )_:
0259          SS=SS+4; 'REPEAT' (_FOR NEXT SUBSEQUENCE)_     >)>)
0260        (_ OTHERWISE:)_
0261    _ABORT MARKING_: (_DO NOT REMOVE EDITING SEQUENCE FROM LINE,  C_
0262        AND (BUT) DO NOT ALTER ANY MARKING ALREADY DONE  )_
0263        'GO AND' PUT LINE ONTO CHAIN

    SERIAL NUMBER = 040     LENGTH = 04BA     NAME = READNEXTLINE#

                                                    (AA) LOCALS :  MARKEDSYMBOL
                                                                   SUBSTITUTION
                                                                   SS
                                                                   STARTOFEDITR
                                                                   HEADOFCHAIN
                                                                   DISPLACEMENT
                                                                   UPPERBOUND
                                                                   UPPER
                                                                   STARTOFLINE
                                                                   STARTOFFIRST
                                                                   STARTOFBUFFE
                                                                   NEWLINECHAIN

0264        _'ROUTINE'_
0265    'MAIN' : 'RCCT TESTRUN'
0266        !INITIALISE FREE CHAIN'
0267        ITEM STACK FRONT=RDB+300
0268        RDB=RDB+1000
0269        RCC CONTINUATION SYMBOL=227 (_SMALL C UNDERLINED)_
0270        RCC MARKED SYMBOL EDITING SYMBOL=191 (_QUERY UNDERLINED)_
```

```
0271          RCC LOCAL LINE NUMBER=0
0272          (<'READ NEXT LINE' @NEWLINE CHAIN
0273          'PPRINT' RCC LOCAL LINE NUMBER; 'PNEWLINE' 1
0274          'PRINT OUT CHAIN' NEW LINE CHAIN
0275          'DELETE CHAIN' @NEWLINE CHAIN
0276          'REPEAT' >)

        SERIAL NUMBER = 044      LENGTH = 008A      NAME = RCCTTESTRUN

                                                    (AA) LOCALS :  NEWLINECHAIN

0277          _'END OF PROGRAM'_


    EOC      01  NO ERRORS DETECTED
//SYS004  ACCESS  PDTEST                                          10.35.33
//  EXEC  PDLOADER                                                10.35.33
        1
     13191           10        13037
     13037           32        13039
     13039           32        13041
     13041           32        13043
     13043           32        13045
     13045           32        13047
     13047           73        13049
     13049           70        13051
     13051           32        13053
     13053          114        13055
     13055           99        13057
     13057           99        13059
     13059           32        13061
     13061          109        13063
     13063          111        13065
     13065          101        13067
     13067           32        13069
     13069           72        13071
     13071           65        13073
     13073           83        13075
     13075           32        13077
     13077          114        13079
     13079           99        13081
     13081           99        13083
     13083           32        13085
```

13089	117	13091
13091	116	13093
13093	105	13095
13095	110	13097
13097	101	13099
13099	32	13101
13101	109	13103
13103	97	13105
13105	114	13107
13107	107	13109
13109	101	13111
13111	114	13113
13113	58	13115
13115	227	13117
13117	32	13119
13119	32	13121
13121	102	13123
13123	114	13125
13125	105	13127
13127	103	13129
13129	32	13131
13131	227	13133
13133	32	13135
13135	32	13137
13137	32	13139
13139	32	13141
13141	32	13143
13143	32	13145
13145	32	13147
13147	32	13149
13149	32	13151
13151	32	13153
13153	105	13155
13155	32	13157
13157	104	13159
13159	111	13161
13161	112	13163
13163	101	13165
13165	32	13167
13167	116	13169
13169	104	13171
13171	105	13173
13173	115	13175
13175	32	13177
13177	119	13179
13179	111	13181

```
13185          115          13187
13187           58          13189
13189           59          13191



    2
13077           10          13037
13037           32          13039
13039           32          13041
13041           32          13043
13043           97          13045
13045           32          13047
13047           32          13049
13049           32          13051
13051           32          13053
13053           32          13055
13055           32          13057
13057           32          13059
13059           98          13061
13061           99          13063
13063          104          13065
13065          117          13067
13067          114          13069
13069          114          13071
13071           97          13073
13073          121          13075
13075           58          13077



    3
13127           10          13037
13037          116          13039
13039          104          13041
13041   -536870816          13043
13043           32          13045
13045          108          13047
13047   -536870816          13049
13049          116          13051
13051          116          13053
13053   -536870816          13055
```

13059	-536870816	13061
13061	32	13063
13063	115	13065
13065	104	13067
13067	111	13069
13069	117	13071
13071	108	13073
13073	100	13075
13075	32	13077
13077	98	13079
13079	-536870816	13081
13081	32	13083
13083	114	13085
13085	-536870816	13087
13087	112	13089
13089	108	13091
13091	97	13093
13093	99	13095
13095	-536870816	13097
13097	100	13099
13099	32	13101
13101	105	13103
13103	110	13105
13105	32	13107
13107	116	13109
13109	104	13111
13111	105	13113
13113	115	13115
13115	32	13117
13117	115	13119
13119	114	13121
13121	105	13123
13123	110	13125
13125	103	13127

4

13103	10	13037
13037	32	13039
13039	32	13041
13041	32	13043
13043	32	13045
13045	32	13047

13051	115	13053
13053	112	13055
13055	97	13057
13057	99	13059
13059	101	13061
13061	115	13063
13063	32	13065
13065	32	13067
13067	97	13069
13069	32	13071
13071	97	13073
13073	32	13075
13075	97	13077
13077	32	13079
13079	102	13081
13081	111	13083
13083	114	13085
13085	32	13087
13087	97	13089
13089	32	13091
13091	32	13093
13093	191	13095
13095	78	13097
13097	66	13099
13099	97	13101
13101	61	13103

5

13133	10	13037
13037	32	13039
13039	32	13041
13041	32	13043
13043	32	13045
13045	32	13047
13047	32	13049
13049	32	13051
13051	116	13053
13053	114	13055
13055	121	13057
13057	32	13059
13059	65	13061
13061	80	13063

```
13067          82        13069
13069          32        13071
13071      -   67        13073
13073          65        13075
13075          83        13077
13077          69        13079
13079          32        13081
13081          65        13083
13083          32        13085
13085          65        13087
13087          32        13089
13089          65        13091
13091          32        13093
13093  -536870851        13095
13095         110        13097
13097         105        13099
13099          32        13101
13101         108        13103
13103         111        13105
13105         119        13107
13107         101        13109
13109         114        13111
13111          32        13113
13113          99        13115
13115  -536870851        13117
13117         115        13119
13119         101        13121
13121          32        13123
13123  -536870851        13125
13125          32        13127
13127  -536870851        13129
13129          32        13131
13131  -536870851        13133
```

```
    6
13133          10        13037
13037          32        13039
13039          32        13041
13041          32        13043
13043          32        13045
13045          32        13047
13047          32        13049
```

13053	114	13055
13055	121	13057
13057	32	13059
13059	85	13061
13061	80	13063
13063	80	13065
13065	69	13067
13067	82	13069
13069	32	13071
13071	67	13073
13073	-536870851	13075
13075	83	13077
13077	69	13079
13079	32	13081
13081	-536870851	13083
13083	32	13085
13085	-536870851	13087
13087	32	13089
13089	-536870851	13091
13091	32	13093
13093	97	13095
13095	110	13097
13097	100	13099
13099	32	13101
13101	108	13103
13103	111	13105
13105	119	13107
13107	101	13109
13109	114	13111
13111	32	13113
13113	99	13115
13115	97	13117
13117	115	13119
13119	101	13121
13121	32	13123
13123	97	13125
13125	32	13127
13127	97	13129
13129	32	13131
13131	97	13133

```
13039  -536870853      13041
13041  -536870853      13043
13043         32       13045
13045        119       13047
13047        104       13049
13049         97       13051
13051        116       13053
13053         32       13055
13055        104       13057
13057         97       13059
13059        112       13061
13061        112       13063
13063  -536870853      13065
13065        110       13067
13067        115       13069
```

```
    8
13099         10       13037
13037         32       13039
13039         32       13041
13041         32       13043
13043         32       13045
13045         32       13047
13047         32       13049
13049        109       13051
13051  -536870877      13053
13053        108       13055
13055        116       13057
13057        105       13059
13059        112       13061
13061        108       13063
13063  -536870875      13065
13065         32       13067
13067  -536870877      13069
13069  -536870791      13071
13071  -536870875      13073
13073  -536870791      13075
13075         32       13077
13077        111       13079
13079        102       13081
13081         32       13083
13083        102       13085
```

```
13089  -536870875      13091
13091          32      13093
13093  -536870875      13095
13095          32      13097
13097  -536870875      13099


    9
13067          10      13037
13037          72      13039
13039  -536870828      13041
13041          80      13043
13043          69      13045
13045          32      13047
13047          84      13049
13049  -536870828      13051
13051          32      13053
13053          83      13055
13055          85      13057
13057          67      13059
13059          67      13061
13061          69      13063
13063          69      13065
13065          68      13067


   10
13079          10      13037
13037         109      13039
13039         117      13041
13041         108      13043
13043         116      13045
13045         105      13047
13047         112      13049
13049         108      13051
13051  -536870810      13053
13053          32      13055
13055         117      13057
13057         115      13059
13059  -536870810      13061
```

```
13065          111        13067
13067  -536870875        13069
13069           32        13071
13071  -536870875        13073
13073          114        13075
13075  -536870810        13077
13077          100        13079
```

********** JOB CANCELLED DUE TO READ ERROR **********

OAI-SVC CANCEL
D PSW FF25000F4000872E

********** ST.ANDREWS UNIVERSITY COMPUTING LABORATORY - 44MFT/SPOOLER END-OF-JOB RECORD **********
JOB NAME : PDTEST3 DATE : 75134 START TIME : 10.35.23 ELAPSED : 00.00.28
CPU TIME : 00.00.13 WAIT TIME : 00.00.03 SYSTEM OVERHEAD : 00.00.04
PAGES PRINTED : 17 CARDS PUNCHED : 0 CARDS READ : 314
A GRAND TOTAL OF 1188 INPUT/OUTPUT REQUESTS WERE HANDLED FOR THE JOB.
THE AVAILABLE CORE WAS : 112 K OF WHICH AT LEAST 33 K WAS USED
THE APPROXIMATE COST OF THIS JOB WAS £ 0.34
THIS JOB WAS EXECUTED IN THE BACKGROUND PARTITION

THIS LINE PRINTED AT 10.43.26 ;

APPENDIX F

PARTIAL COMPILER LISTING

```
.
.
.
    GLOBAL
asp ≡ a stack pointer, isp ≡ input symbol position, ct stack front ≡          c
compile time stack front
(Recognition) ROUTINE
IMP : TERMINATOR
    a = (isp)
    IF a = ";"; OR IF a = ":"; OR IF a = eol : (a global constant             c
    indicating the newline terminator)
        STACK a AT asp; recognised = 1:
    OTHERWISE : recognised = 0

(Recognition) ROUTINE
IMP : TERM EX NL
    a = (isp)
    IF a = ";"; OR IF a = ":": STACK a AT asp; recognised = 1:
    OTHERWISE : recognised = 0
```

```
(Recognition) ROUTINE
IMP : BRACKETS TERMINATOR
      count = 0
      {IF (isp) = ">"; AND IF isp(1) = ")":
           NEXT count; isp = isp + 2; REPEAT}
      STACK count AT asp
      TERMINATOR; IF NO recognised: NEXT - asp (to discard the                c
      Analysis Stack entry)
      (note that the TERMINATOR routine sets the recognition switch)

(Recognition) ROUTINE
IMP : SHIFT OPERATOR
      a1 = (isp); a2 = isp (1)
      IF a1 = "A"; OR IF a1 = "L":
          {{IF a2 = "U"; OR IF a2 = "D":
                {IF a1 = "A" : code = 6:
                OTHERWISE : code = 8
                IF a2 = "D" : NEXT code
                STACK code AT asp; isp = isp + 2; recognised = 1; FINISH} }}
recognised = 0
```

```
(Recognition) ROUTINE

IMP : OPERATOR

     a = (isp) ; SET NO code

     IF a = "+" : code = 1:

     OTHERWISE : {{{{ IF a = "-": code = 2:

          OTHERWISE : {{{ IF a = "&" : code = 3:

               OTHERWISE : {{ IF a = "|": code = 4:

                    OTHERWISE : {IF a = "X";AND IF isp(1) = "O";        C
                                AND IF isp(2) = "R": code = 5; isp = isp + 2
                                } }} }}} }}}}

     IF code : (asp) = code; NEXT isp; recognised = 1:

     OTHERWISE : recognised = O


(Recognition) ROUTINE

IMP : COMPARATOR

     a = (isp); code = o

     IF a = "<": NEXT isp; code = 3:

     OTHERWISE : {{IF a = ">": NEXT isp; code = 5:

          OTHERWISE :{ IF a = "/": NEXT isp; code = 1} }}

     IF (isp) = "=": NEXT isp; NEXT code

     IF code ≠ O: (it contains the correct code for a comparator)

          STACK code AT asp; recognised = 1:

     OTHERWISE : recognised = O
```

```
(Recognition) ROUTINE
IMP : INTEGER
     IF (isp) < "0"; OR IF (isp) > "9": recognised = 0:
     OTHERWISE:
         {{integer = (isp) - "0"; NEXT isp
          (scan for more decimal digits)
          {IF (isp) ≥ "0"; AND IF (isp) ≤ "9":
               integer = integer AU  3 + integer + integer; (has the        c
               effect of multiplying by 10)
               integer = integer + (isp) - "0"; (adds the next digit)
               NEXT isp; REPEAT}
          STACK (the plex) first bit set, integer  AU 2 AT asp
          recognised = 1}}

              ROUTINE
IMP : SPECIFIER (This routine looks for either a single [INTEGER]          c
     or [INTEGER]→[INTEGER] (see CONSTANT routine))
     INTEGER; IF recognised :
          {{{{backtrack = isp
               IF (isp) = "-"; AND IF isp(1)= ">":
                    {{{isp = isp + 2
                         INTEGER; IF recognised:
                              {{n = (asp - 1) - (asp - 3) + 1 (giving the    c
                                width of the bit field)
```

```
                    s = 1
                    IF n ≤ o : isp = backtrack; asp = asp - 2;     c
                    GO TO other option
                    IF n = 1 : asp = asp - 2; GO TO other option
                    FOR i = 2 TO n : S = S LU 1|1
                    shiftup = (asp - 3) AD 2
                    IF shiftup > o:
                         {FOR i = 1 TO shiftup: s = s LU 1}
                    (asp - 5) = (asp - 5) | s
                    asp = asp - 4
                    (recognised = 1 from the call of INTEGER above)
                    FINISH}}
               isp = backtrack}}}
          s = 1
          other option: shiftup = (asp - 1) AD 2
          IF shiftup > o: {FOR i = 1 TO shiftup : s = s LU 1}
          (asp - 3) = (asp - 3) | s
          asp = asp - 2
          (recognised = 1 from the first call of INTEGER)}}}}
     (otherwise recognised = 0)
```

```
(Recognition) ROUTINE

IMP : CONSTANT

(integer?) INTEGER; IF recognised : FINISH
     STACK (first word of a constant plex) first bit set AT asp
     backtrack = isp

(octal number?) IF (isp) = "*":
     {{NEXT isp
     IF (isp) < "0"; OR IF (isp) > "7": GO AND notrecognise
      s= (isp) - "0"
     NEXT isp
     {IF (isp) ≥ "0"; AND IF (isp) ≤ "7":
          s = s AU 3 + (isp) - "0"; (adds in the next octal digit)
          NEXT isp; REPEAT}
     s = s AU 2
     recogise : STACK s AT asp; recognised = 1; FINISH}}

(hex number or bit pattern?)
     IF (isp) = "#":
          {{{{NEXT isp
               IF (isp) = "#": NEXT isp; local switch = 1:
               OTHERWISE : local switch = 0
               IF (isp) < "a"; AND IF (isp) > "f":
                    {{{IF (isp) < "0"; AND IF (isp) > "9": GO AND not recognise
                    s = (isp) - "0"}}}:
               OTHERWISE: s = (isp) - "a" + 10
```

```
                NEXT isp
                {{{IF (isp) ≥ "0"; AND IF (isp) ≤ "9":
                     s = s LU 4 + (isp) - "0":
                OTHERWISE:
                     {{IF (isp) ≥ "a"; AND IF (isp) ≤ "f":
                          s = s LU 4 + (isp) - "a" + 10:
                     OTHERWISE:
                          {IF local switch = 0: s = s&first bit exterminator AU 2
                          GO AND recognise} }}
                NEXT isp; REPEAT}}}  }}}}

(system symbol?) IF (isp) = """:
     {{NEXT isp
     symbol = (isp); next symbol = isp (1)
     IF next symbol ≠ """:
          {IF symbol ≠ """: GO AND not recognise
          symbol = internal code for black (a global constant)
          NEXT isp} :
     OTHERWISE: isp = isp + 2
     s = symbol AU 2; GO AND recognise}}

(field specifier?) IF (isp) = "%":
     {{{STACK 0 AT asp
          SPECIFIER; IF recognised: FINISH
          IF (isp) = "(":
```

```
        {{repeat: NEXT isp
            SPECIFIER; IF recognised:
                {IF (isp) = ")": NEXT isp; FINISH
                IF (isp) = ",": GO AND repeat} }}
        asp = asp - 1 (to discard the location holding a zero) }}}
not recognised: asp = asp - 1
            isp = backtrack
            recognised = 0

(Recognition) ROUTINE
IMP: NAME
    old stack position = asp
    IF (isp) < "a"; OR IF (isp) > "z": recognised = 0; FINISH
    bytecount = 2
    new word = (isp)
    {NEXT isp
    IF (isp) ≥ "a"; AND IF (isp) ≤ "z":
        PERFORM character load; REPEAT}
    {IF (isp) ≥ "0"; AND IF (isp) ≤ "9":
        PERFORM character load; NEXT isp; REPEAT}
    (asp) = new word
    (code the first byte of the packed name)
    (old stack position) = asp + 1 - old stack position + 4 - bytecount    c
        LU 24 | (old stack position)
    asp (1) = old stack position
    asp = asp + 2
    recognised = 1
    FINISH
```

```
character load:  IF bytecount ≠ 4: NEXT bytecount;
                                   new word = new word LU 8 | (isp):
                 OTHERWISE: STACK new word AT asp;
                            new word = (isp);
                            bytecount = 1
                 FINISH character load
```

```
(Recognition) ROUTINE
IMP : LABEL
     backtrack = isp; reset = asp
     IF (isp) = "_":
          {{NEXT isp
          NAME; IF recognised:
               {IF (isp) = "_": NEXT isp;
                    (recognised = 1)
                    FINISH}
          isp = backtrack; asp = reset}}
     recognised = 0
```

```
(Recognition) ROUTINE
IMP : SPECIAL NAME
     NAME; IF recognised:
          {dummy = (asp - 1)
           SUBSTITUTE IDENTIFIER PLEX AT dummy
           asp = dummy + 2}
     (note that the recognition switch is set by NAME)
```

```
(Recognition) ROUTINE

IMP : CONDITIONAL
     backtrack = isp
     switch = 0
     PERFORM conditional recognition
     IF switch = 0:
          {{{IF (isp) = "O"; AND IF isp (1) = "R":
               {code = 2; isp = isp + 2; PERFORM conditional recognition}:
          OTHERWISE:
               {{IF (isp) = "A"; AND IF isp (1) = "N"; AND IF isp (2) = "D":
                    {code = 4; isp = isp + 3; PERFORM conditional recognition}:
               OTHERWISE : recognised = 0; FINISH}}
          IF switch = 0: isp = backtrack; recognised = 0; FINISH
          (asp - 1) = (asp - 1) + code}}}
     recognised = 1; FINISH

conditional recognition:  IF (isp) = "I"; AND IF isp (1) = "F":
                               isp = isp + 2; code = 0:
                          OTHERWISE:
                          {IF (isp) = "U"; AND IF isp (1) = "N"; AND IF      c
                               isp (2) = "L";
                          AND IF isp (3) = "E"; AND IF isp (4) = "S";        c
                               AND IF isp(5) = "S":
                               isp = isp + 6; code = 1:
                          OTHERWISE : FINISH conditional recognition}
                          STACK code At asp; switch = 1; FINISH conditional  c
                               recognition
```

```
(Recognition) ROUTINE
IMP : PHRASE VARIABLE
      backtrack = isp; reset = asp
      IF (isp) = "<"; AND IF isp (1) = "_":
           {{{isp = isp + 2
                IF (isp) ≥ "A"; AND IF (isp) ≤ "Z":
                   {{TERMINATOR; IF recognised : GO AND not recognise
                     IF (isp) = ">"; AND IF isp (1) = "_": GO AND not recognise
                     IF phrase variable switch = 0 : SPECIAL NAME:
                     OTHERWISE : NAME
                     IF recognised:
                         {IF (isp) = ">"; AND IF isp (1) = "_":
                               isp = isp + 2; FINISH (recognised)
                          GO AND not recognise}
                     NEXT isp (to ignore the PVSYMBOL)
                     REPEAT}}
                not recognised: isp = backtrack; asp = reset}}}
           recognised = 0

(Recognition) ROUTINE
IMP : MATCHING PHRASE EXPRESSION
      reset = asp
      phrase variable count = 0
      switch = 0
      NEXT asp ( reserves a location to hold the phrase variable count)
     [[ [ PHRASE VARIABLE; IF recognised : switch = 1;
                                           NEXT phrase variable count;
                                           REPEAT }
```

```
        a = (isp)
        IF a ≠ ":"; AND IF a ≠ ";"; AND IF a ≠ eol: switch = 1;
                                                    NEXT isp;
                                                    REPEAT}}
        IF switch = 0 : asp = reset; recognised = 0:
        OTHERWISE : (reset) = phrase variable count; recognised = 1

(Recognition) ROUTINE
IMP : DECLARATION
        reset = asp; local error (switch) = 0
        NAME; IF recognised:
            {{{{SEARCH 0 DICTIONARY FOR (asp - 1) GIVING PLEX ADDRESS @ a
            IF a : SET local error (switch) (since the NAME should not be there)
            backtrack = isp
            IF (isp) = "="; AND IF isp (1) = "_":
                {{{isp = isp + 2
                (Look for register synonym)
                IF (isp) = "A":
                    {{NEXT isp
                    INTEGER; IF recognised:
                        {(asp - 1) = (asp - 1) AD 2 (to cancel the bit shift)
                        a = (asp - 1)
                        IF a ≥ ##0; AND IF a ≤ ##F:
                            (asp - 2) = a | third bit set;
                            (asp - 1) = 0;
```

```
                    asp = asp - 2;
                    GO AND recognise
                asp = asp - 2}
            isp = backtrack
            GO AND construct plex}}
        SPECIAL NAME;
        IF NO recognised : CONSTANT
        IF recognised : asp = asp - 2; GO AND recognise
        isp = backtrack}}}
    construct plex: (i.e. allocate storage to hold the scalar)
    (asp) = next free rt location
    NEXT next feee rt location
    asp (1) = 0
    recognise;
    IF local error = 0 : ADD (asp - 1) TO 0 DICTIONARY WITH PLEX      c
            ADDRESS asp; (reset) = (asp);
                reset (1) = asp (1):
    OTHERWISE : (reset ) = (error) fourth bit set
    asp = reset + 2
    recognised = 1}}}}
(otherwise not recognised)

(Recognition) ROUTINE
IMP : VARIABLE
    reset = asp; backtrack = isp; count = 1 (of total number of operands    c
    in a subscripted variable) switch = 0
```

```
SPECIAL NAME;
IF recognised:
     {backtrack = isp; switch = 1
     IF (isp) ≠ "(" : FINISH (recognised)
     NEXT isp
     (asp) = (asp - 2); asp (1) = (asp - 1)
     (asp - 2) = second bit set (descriptor bit); (asp - 1) = 0
     count = 2
     asp (2) = "+"
     asp = asp + 3
     GO AND test for endings}
IF (isp) ≠ "(" : FINISH (unrecognised)
NEXT isp
STACK (descriptor plex) second bit set, 0 AT asp
SPECIAL NAME; IF recognised:
     {IF (isp) = "+"; OR IF (isp) = "-":
          STACK (isp) AT asp;
          NEXT isp; count = 2; GO AND test for endings
     IF (isp) = ")": GO AND recognise
     GO AND not recognise}
test for endings: CONSTANT
IF recognised:
     {IF (isp) = ")" : GO AND recognise
     GO AND not recognise}
SPECIAL NAME
IF recognised :
     {{{IF (isp) = ")" : GO AND recognise
     IF (isp) = "+"; OR IF (isp) = "-" :
```

```
                {{NEXT count
                STACK (isp) AT asp
                NEXT isp
                CONSTANT; IF recognised :
                    {IF (isp) = ")" : GO AND recognise} }} }}}
    not recognise: isp = backtrack
    IF switch = 0 : asp = reset; recognised = 0; FINISH (unrecognised)
    (reset) = reset (2); reset (1) = reset (3)
    asp = reset + 2
    recognised = 1; FINISH (recognised)
    recognise: NEXT isp
    (reset) = (reset) | count (places the count of number of operands      c
        into the first word of the variable plex)
    recognised = 1

(Recognition) ROUTINE
IMP : EXPRESSION
    reset = asp; link = asp; backtrack = isp; NEXT asp
    (Deal with any preceeding unary operator)
    IF (isp) = (unary) "+" : (asp) = 1; NEXT isp:
    OTHERWISE : {IF (isp) = (unary) "-" : (asp) = 2; NEXT isp :
                    OTHERWISE : (assume unary plus) (asp) = 1}
    switch = 0 (indicating that no operands have been found yet)
```

```
test for variable (operand) : NEXT asp; VARIABLE
     UNLESS recognised : {CONSTANT; UNLESS recognised : GO AND finish}
test for more: switch = 1 (indicating that another operand has been found)
     (set link for previous element) (link) = reset; link = reset
     (then update) reset = asp; backtrack = isp; NEXT asp
     (look for an operator)
     SHIFT OPERATOR
     IF recognised:
          {{SPECIAL NAME;
          IF recognised : (check that it is a semantic constant)
               {UNLESS (asp - 2) HAS first (constant) bit set : GO AND finish}:
          OTHERWISE:
               {CONSTANT; UNLESS recognised : GO AND finish}
          (Adjust the constant to cancel the shift made when it was created)
          (asp - 1) = (asp - 1) AD 2
          GO AND test for more}}
     OPERATOR; IF recognised : GO AND test for variable (operand)
finish: (Set the final link) (link) = 0
     asp = reset; isp = backtrack; recognised = switch

(Recognition) ROUTINE
IMP : ROUTINE CALL
     reset = asp; backtrack = isp; s = 0
     CHECK RSYNTAX FOR @ s PARAMETERS GIVING @ parmcount
     recognised = s
```

```
     UNLESS (syntax) recognised:
          {isp = backtrack; reset = asp
          PACK ROUTINE NAME ON ANALYSIS STACK, 0
          SUBSTITUTE RPLEX AT reset WITH parmcount PARAMETERS
          asp = reset + 3}

(Recognition) ROUTINE
IMP : ROUTINE HEADING
     backtrack = isp; reset = asp
     (look for routine type specification)
     recognised = 0
     IF (isp) = "M"; AND IF isp (1) = "A"; AND IF isp (2) = "I";
     AND IF isp (3) = "N" : tag = 2; isp = isp + 4; recognised = 1 :
     OTHERWISE :
          {{IF (isp) = "B"; AND IF isp (1) = "A"; AND IF isp (2) = "S";
          AND IF isp (3) = "I"; AND IF isp (4) = "C" :
               tag = 4; isp = isp + 5; recognised = 1 :
          OTHERWISE :
               {tag = 0
               IF (isp) = "S"; AND IF isp (1) ; "T"; AND IF isp (2) = "A";
               AND IF isp (3) = "N"; AND IF isp (4) = "D"; AND IF isp (5) = "A";
               AND IF isp (6) = "R"; AND IF isp (7) = "D" :
                    isp = isp + 8; recognised = 1}
          (look for IF/IMP spec)
          IF (isp) = "I"; AND IF isp (1) = "F":
               NEXT tag; isp = isp + 2; recognised = 1 :
```

```
        OTHERWISE :
            {IF (isp) = "I"; AND IF isp (1) = "M"; AND IF isp (2) = "P" :
                isp = isp + 3; recognised = 1} }}

IF recognised (routine type specification) :
    {{{{{{tag = tag LU 27 (stores the tag bits in a form compatible with     c
        a routine plex)
    IF (isp) = ":" :
        {{{{{NEXT isp; s = 1; dummy = isp
        CHECK RSYNTAX FOR @ s PARAMETERS GIVING @ parmcount
        IF s ≠ 0 :
            {{{{isp = dummy; link = asp
            PACK ROUTINE NAME ON ANALYSIS STACK, 1
            SEARCH 2 (routine) DICTIONARY FOR link GIVING PLEX ADDRESS @  a
            IF a = 0 : (the routine is not on the dictionary)
                {{{(construct a routine plex)
                (asp) = first two bits set | tag | current serial number
                NEXT current serial number
                IF tag LD 29 ≠ 0 : (BASIC routine)
                    asp (1) = next free rt location LU 16 | parmcount;
                    next free rt location = next free rt location + 3 :
                OTHERWISE : (STANDARD routine) asp (1) = parmcount
                IF NO parmcount : asp (2) = 0 :
                OTHERWISE : asp (2) = next free rt location
                ADD link TO 2 (routine) DICTIONARY WITH PLEX ADDRESS asp
                asp = asp + 3
```

```
      IF parmcount ≠ o :
           {{dummy = reset
           {SUBSTITUTE IDENTIFIER PLEX AT dummy + 1
           dummy = (dummy)
           IF (dummy) ≠ o : REPEAT} (adds the formal parameters  c
           to the identifier dictionary)}}
      dummy = asp ~ 3}}}:
OTHERWISE :
{{{IF (a) HAS first bit set : (print redefinition warning)....
   IF (a) & tags extractor ≠ tag : (print routine type          c
   error message)....
   (a) = first two bits set | tag | (a)
   IF parmcount ≠ o :
     {{(construct formal parameter identifier plexes and        c
        insert in the identifier dictionary)
     address = a (2)
     asp (1) = o
     dummy = reset
     {(asp) = address
     ADD dummy + 1 TO o DICTIONARY WITH PLEX ADDRESS asp
     NEXT address
     dummy = (dummy)
     REPEAT UNTIL (dummy) = o} }}
dummy = a}}}
```

```
                (Now copy the routine plex back to the old configuration      c
                 of the Analysis Stack)
                FOR i = 0 TO 2 : reset (i) = dummy (i)
                asp = reset + 3
                recognised = 1; FINISH}}}}
          asp = reset}}}}}
     isp = backtrack}}}}}}
recognised = 0
```

```
(Semantics) ROUTINE
IMP : ASSIGNMENT (referred to as SEM001 in PL360)
     CHECK BLOCK SYNTAX GIVING RESULT @ a
IF a : ASSIGN (plexbase), plexbase (1);
     PROCESS CONTROL AFTER IMPERATIVE plexbase (2)
```

```
(Semantics) ROUTINE
IMP : SET UP CONTROL DATA (Semantics of "{".  Referred to as SEM002 in PL360)
     LOCAL : ccd ≡ current control data
CHECK BLOCK SYNTAX GIVING RESULT @ a
IF a :{{IF ccd (2) = 0 : (NO forward link, so set up a new CD block on       c
     the CT stack)
     (ct stack front) = ccd;
     ccd (1) = ct stack front;
     ccd = ct stack front;
     ct stack front = ct stack front + 8;
     ccd(1) = 0
     OTHERWISE : ccd = ccd (1)
```

```
(initialise elements in CD block, inserting segment/word address in          c
     fifth word)
FOR i = 2 TO 7 :
     {IF i ≠ 5 : ccd (i) = 0 :
     OTHERWISE : ccd (i) = index value LU 16 | routine byte count}
NEXT brackets count}}

(Semantics) ROUTINE
IMP : CONDITIONALS (referred to as SEM003 in PL360)
     LOCAL : ea ≡ expression accumulator, cr ≡ comparison register,          c
             ccd ≡ current control data
CHECK BLOCK SYNTAX GIVING RESULT @ a
IF a : {ASSIGN 0, plexbase (1) (compiles code to compute the left           c
     hand side expression)
     dummy = plexbase (3)
     (Inspect right hand side expression to see if optimisations can be made)
     IF (dummy) = 0; AND IF dummy (1) = 1 (code for "+");
     AND IF dummy (2) HAS first bit set (the single operand is a constant);
     AND IF dummy (3) = 0 (the constant is zero) :
          PLANT load and test register, ea, ea, 0, 0 :
     OTHERWISE : (usual case)
          PLANT load register, cr, ea, 0, 0 ;
          ASSIGN 0, plexbase (3) (compiles code to compute r.h.s. expression);
          PLANT compare register, cr, ea, 0, 0
```

```
     terminator = plexbase (5)
     conditional = (plexbase)
     IF (conditional) HAS bottom (least significant) bit set:
          SET switch (to indicate an UNLESS form) :
     OTHERWISE : SET NO switch (to indicate an IF form)
     (conditional) = (conditional) LD 1
     level status = ccd (2)
     IF (conditional) = 0 : (a plain IF/UNLESS form is present)
     {IF level status ≠ 0 (IMP) : PERFORM error sequence
     dummy = 0; (an input parameter to) PERFORM plant branch instruction
     IF (terminator) = ":" : ccd (2) = 4 (sets level status = IFIMP) :
     OTHERWISE : ccd (2) = 1 (sets level status = IF)} :
OTHERWISE :
     {{{IF (conditional) = 1 : (an OR IF/UNLESS form is present)
          {{IF level status ≠ 1 (IMP); AND IF level status ≠ 3 (ORIF) :
               PERFORM error sequence
          IF level status = 1 :
               {(code to alter previous branch instruction)....}
          IF (terminator) = ":" :
               {dummy = 1; PERFORM plant branch instruction
               ccd (2) = 4 (sets level status = IF IMP)}:
          OTHERWISE :
               {switch = switch ≠ 1; dummy = 0;
               PERFORM plant branch instruction
               ccd (2) = 3 (sets level status to OR IF)} }}:
```

```
     OTHERWISE : (an AND IF/UNLESS form is present)
          {{IF level status ≠ 1; AND IF level status ≠ 2 (AND IF):
               PERFORM error sequence
          dummy = 0; PERFORM plant branch instruction
          IF (terminator) = ":" : ccd (2) = 4 (sets level status = IFIMP) :
          OTHERWISE : ccd (2) = 2 (sets level status = AND IF) }} }}}
FINISH
error sequence : (Print error message ILLEGAL CONTROL)....
     FINISH error sequence
plant branch instruction : comparator address = plexbase (2)
     c = (comparator address)
     IF  c = 1 : mask = 8 :
     OTHERWISE : {{{{IF c = 2 : mask = 6 :
          OTHERWISE :
          {{{IF c = 3 : mask = 4 :
               OTHERWISE :
               {{IF c = 4 : mask = 12 :
                    OTHERWISE :
                    {IF c = 5 : mask = 2 :
                         OTHERWISE : mask = 10} }} }}} }}}}
     IF switch = 0 : mask = mask ≠ 14
     PLANT branch on condition, mask, 0, 0, 0 (incomplete branch instruction  c
          planted)
     IF dummy ≠ 0 : FULL IN ccd + 3 (control list 1) TO routine byte count
     FINISH plant branch instruction}
```

```
(Semantics) ROUTINE
IMP : LABEL (Semantics of label : , referred to as SEM004 in PL360)
     CHECK BLOCK SYNTAX GIVING RESULT @ a
IF a : {[SEARCH 1 (label) DICTIONARY FOR (plexbase) GIVING PLEX                c
     ADDRESS @ a
     IF a = 0 : (the label is not in the dictionary, so construct a            c
          label plex and add it to the label dictionary)
          (asp) = first (defined) bit set;
          asp (1) = index value LU 16 | routine byte count; (segment/word      c
          address);
          ADD (plexbase) TO 1 DICTIONARY WITH PLEX ADDRESS asp :
     OTHERWISE : (the label is on the dictionary)
          [IF (a) HAS first (defined) bit set : (Print DUPLICATE LABEL          c
          error) .... :
          OTHERWISE : (set defined bit of label plex)
               (a) = (a) | first (defined) bit set;
               (then insert label address in plex)
               a (1) = index value LU 16 | routine byte count} }}

(Semantics) ROUTINE
IMP : OTHERWISE (Semantics of OTHERWISE : , referred to as SEM005 in PL360)
     LOCAL ccd ≡ current control data
CHECK BLOCK SYNTAX GIVING RESULT @ a
IF a : [UNLESS ccd (2) = 7 (level status = EXPECTING OTHERWISE) :
     (Print UNEXPECTED OTHERWISE error) .... :
```

```
     OTHERWISE : ccd (2) = 5 (sets level status = OTHERWISE IMP);
          PLANT branch on condition, 15, 0, 0, 0;
          FILL IN ccd + 3 (control list 1) TO routine byte count;
          ADD code address - 1 TO ccd + 3}

(Semantics) ROUTINE
IMP : GO TO (Semantics of GO TO/AND, referred to as SEM006 in PL360)
CHECK BLOCK SYNTAX GIVING RESULT @ a
IF a: { PLANT GO TO BRANCH FOR (plexbase)
     PROCESS CONTROL AFTER IMPERATIVE plexbase (1)}

          ROUTINE
STANDARD IMP : PLANT GO TO BRANCH FOR p (Subsidiary routine to GOTO and SWITCH)
     PLANT branch on condition, 15, 0, 0, 0
     SEARCH 1 (label) DICTIONARY FOR p GIVING PLEX ADDRESS @ a
     IF a = 0 : (construct a label plex and add it to the label dictionary)
          (asp) = code address - 1;
          asp (1) = 0;
          ADD p to 1 DICTIONARY WITH PLEX ADDRESS asp :
     OTHERWISE :
          ADD (branch instruction at) code address - 1 TO (list variable      c
          in plex indicated by) a
```

```
(Semantics) ROUTINE
IMP : SWITCH (Semantics of Switch statement, referred to as SEM007 in PL360)
CHECK BLOCK SYNTAX GIVING RESULT @ a
IF a : {{ASSIGN 0, plexbase (1) (compiles code to evaluate the expression)
     SET segmentation delay switch
     PLANT branch and link register, variable address accumulator, 0, 0, 0
     PLANT branch on condition, 15, 0, variable address accumulator,        c
     expression accumulator;
     dummy = (plexbase); number of labels = (dummy)
     FOR i = 1 TO number of labels :
          {dummy = dummy (1) LD 26 + dummy + 1
          PLANT GO TO BRANCH FOR (dummy)}
     SET NO segmentation delay switch}}

(Semantics) ROUTINE
IMP : GLOBAL DECLARATION (referred to as SEM008 in PL360)
     PERFORM 1 (global) DECLARATION SEMANTICS
     instruction count = smallest negative number (initialised by compiler)

(Semantics) ROUTINE
IMP : LOCAL DECLARATION (referred to as SEM009 in PL360)
     CHECK BLOCK SYNTAX GIVING RESULT @ a
     IF a :
          {PERFORM 0 (local) DECLARATION SEMANTICS
          IF instruction count ≠ 0 : (Print DECLARATION POSITIONING warning)....
          instruction count = - 1
          PROCESS CONTROL AFTER IMPERATIVE plexbase (1)}
```

```
                ROUTINE
IMP : PERFORM switch DECLARATION SEMANTICS (Subsidiary to declaration routines)
     IF switch ≠ 0 (global declaration) :
          global dictionary head = dictionary head;
          old ct stack front = ct stack front
     (Check given identifier plexes for any double declaration faults)
     dummy = (plexbase); count = (dummy)
     FOR i = 1 STEP 2 TO count AU 1 + 1 :
          {IF dummy (i) HAS fourth (error) bit set :
               (PRINT NAME ALREADY DECLARED error) ....}

(Semantics) ROUTINE
IMP : PRESET ARRAY (referred to as SEM010 in PL360)
     CHECK BLOCK SYNTAX GIVING RESULT @ a
     IF a :
          {{{SEARCH 0 (identifier) DICTIONARY FOR (plexbase) GIVING PLEX     c
          ADDRESS @ a
          IF a : (Print NAME ALREADY DECLARED error)....:
          OTHERWISE :
               {{(place zero in constant table) constant table(current       c
               free constant position) = 0
               (construct identifier plex and add to identifier dictionary)
               (asp) = first (constant) bit set | preset array bit set
               asp (1) = current free constant position
```

```
        ADD (plexbase) TO O DICTIONARY WITH PLEX ADDRESS asp
        dummy = plexbase (2); count (of number of constants) = (dummy)
        link = dummy + 1 (points to first constant plex)
        (Add constants to preset array table)
        FOR i = 1 TO count :
            {address = i - 1 + pa current free constant position
            preset array table (address) = link (1); link = link + 2}
        (Add an element to preset array dictionary)
        (ct stack front) = O
        dummy = plexbase (1); integer = dummy (1)
        ct stack front (1) = pa current free constant position -      c
        integer AU 16 | current free constant position
        IF preset array dictionary head = O :
            preset array dictionary head = ct stack front :
        OTHERWISE : (preset array dictionary tail) = ct stack front
        preset array dictionary tail = ct stack front
        ct stack front = ct stack front + 2
        (Update globals)
        NEXT current free constant position
        pa current free constant position = pa current free constant   c
        position + count}}
    PROCESS CONTROL AFTER IMPERATIVE plexbase (3)}}}

(Semantics) ROUTINE
IMP : REPEAT (referred to as SEMO11 in PL360)
    CHECK BLOCK SYNTAX GIVING RESULT @ a
    IF a :
```

```
{{{(check for following bracket)
dummy = (plexbase); rbracket count = (dummy)
IF NO rbracket count :
     (Print NO BRACKET AFTER REPEAT error)....:
OTHERWISE :
     {{(inspect address of start of compound statement to see     c
     if it is in the same segment)
     dummy = current control data (5)
     segment = dummy LD 16; word = dummy & bottom half word extractor
     SET segmentation delay switch
     IF index value ≠ segment :
          {(Put segment number in constant table and plant code   c
          to load index register with segment value)
          (asp) = first (constant) bit set; asp (1) = segment
          ACCESS CONSTANT asp GIVING POSITION @ p
          PLANT load, routine index register, p, 0, constant      c
          base register}
     PLANT branch on condition, 15, word, routine index register, c
     routine base register
     SET NO segmentation delay switch
     PROCESS CONTROL AFTER IMPERATIVE (plexbase)}} }}}
```

(Semantic) ROUTINE

IMP : FOR STATEMENT (with STEP, referred to as SEM012 in PL360)

```
LOCAL : ccd ≡ current control data, cr ≡ comparison register,          c
ea ≡ expression accumulator, br ≡ base register, vaa ≡ variable        c
accumulator, rir ≡ routine index register, rbr ≡ routine base register
CHECK BLOCK SYNTAX GIVING RESULT @ a
IF a :
    {{{{IF ccd (2) ≠ 0 (level status ≠ IMP) :
        (Print UNEXPECTED FOR-STATEMENT error) ....:
    OTHERWISE :
        {{{(place control variable into CD block)
        dummy = (plexbase); SUBSTITUTE IDENTIFIER PLEX AT dummy
        ccd (7) = (dummy)
        IF (dummy) HAS register bit set :
            local switch = 1; register = (dummy) & register bit        c
            exterminator :
        OTHERWISE : local switch = 0; register = cr
        ASSIGN (initial value to control variable) dummy, plexbase (1)
        IF local switch = 0 : PLANT load register, register, ea, 0, 0
        address limit = next free rt location; NEXT next free rt location
        (compile code to compute and store limit expression)
        (asp) = address limit; ASSIGN asp, plexbase (3)
        SET segmentation delay switch
        (Inspect increment expression setting "switch" to control      c
        the code compiled henceforth)
```

```
dummy = plexbase (2)
IF (dummy) ≠ 0 (more than one operand);
     OR UNLESS dummy (2) HAS first (constant) bit set :      c
     switch = 3 :
OTHERWISE : (single constant operand)
     {{IF dummy (1) = 2 (negative) : switch = 2 :
     OTHERWISE :
          {IF dummy (3) ≥ 1024 : switch = 1 :
          OTHERWISE : constant = dummy (3); switch = 0} }}
dum = 0
IF switch ≠ 0 : {{(compile code to compute and store increment)
     address inc = next free rt location; NEXT next free rt    c
     location
     (asp) = address inc; ASSIGN asp, plexbase (2)
     IF switch = 3 :
          {(compile code to compute and store branch mask)
          PLANT load and test register, ea, ea, 0, 0
          PLANT branch on condition, 2, routine byte count     c
          + 12, rbr, rt
          PLANT load address, vaa, 64, 0, 0
          PLANT branch on condition, 15, routine bytecount + 8  c
          rir, rbr
          PLANT load address, vaa, 32, 0, 0
          PLANT store, vaa, next free rt location, 0, br
          address mask = next free rt location
          NEXT next free rt location
          dum = 1} }}
```

```
IF local switch = 0 : dum = dum + 4 : OTHERWISE :                    c
dum = dum + ## A
PLANT branch on condition, 15, routine byte count + dum,             c
rir, rbr
     (instruction to branch round inter-cycle action code)
IF switch = 3 : (plant the branch instruction which is               c
executed by an EX instruction in the case where the                  c
sign of the increment is not known till run-time)
branch out address = code address;
PLANT branch on condition, 0, 0, 0, 0
..
(store address of inter-cycle action code in CD block)
ccd (6) = index value LU 16 | routine byte count
..
(plant inter-cycle action code)
IF switch = 0 : PLANT load address, cr, constant, 0, 0 :
     OTHERWISE:  PLANT load, cr, address inc, 0, br
IF local switch = 0 : PLANT add, cr, ccd (7) (control                c
variable), 0, br;
          PLANT store, cr, ccd (7), 0, br :
     OTHERWISE : PLANT add register, register, cr, 0, 0
(plant instruction to compare control variable with limit            c
(setting  condition code))
PLANT compare, register, address limit, 0, br
IF switch = 3 :
     {(plant instruction to execute previous branch instruction)
     PLANT load, vaa, address mask, 0, br
     IF local switch = 0:  dum = ## 18 : OTHERWISE : dum = ## 12
     PLANT execute, vaa, routine bytecount-dum, rir, rbr} :
```

```
            OTHERWISE :
                {(plant instruction to branch on +ve or -ve)
                branch out address = code address
                IF switch = 2 : dum = 4 : OTHERWISE : dum = 2 (sets      c
                mask of branch)
                PLANT branch on condition, dum, 0, 0, 0}
            SET ccd (2) (level status) = 6 (CYCLE IMP)
            ADD branch out address TO ccd + 3 (Control List 1 in CD block)
            SET NO segmentation delay switch}}} }}}}
```

(Semantics) ROUTINE

IMP : SIMPLE FOR STATEMENT (without STEP, referred to as SEM013 in PL360)

```
    (This routine has the same function as a Primary semantic routine      c
    in that it manipulates the given parse on the Analysis Stack into
    the parse form expected by the FOR STATEMENT semantics routine,
    then calls it)
    CHECK BLOCK SYNTAX GIVING RESULT @ a
    IF a : {old plexbase = plexbase; plexbase = asp
        STACK (oldplexbase), oldplexbase (1), plexbase + 5, oldplexbase (2), c
        0, 1, first bit set, 4 AT asp
        FOR STATEMENT}
```

(Semantics) ROUTINE

IMP : STACK (stacking semantics routine, referred to as SEM014 in PL360)

```
    CHECK BLOCK SYNTAX GIVING RESULT @ a
    IF a :
```

```
{{{{{CHECK STACK POINTER GIVING @ register USED@  naddress        c
AND SET @ (validity) switch
IF switch ≠ - 1 :
{{{{dummy = (plexbase); expression address = dummy + 1
limit = (dummy) (gives total no. of expressions)
displacement = 0
FOR i = 1 to limit :
     {{{ASSIGN 0, expression address (compiles code to compute     c
     an expression, leaving result in expression accumulator)
     PLANT store, expression accumulator, displacement, 0, register
     NEXT displacement
     UNLESS i = limit :
          link = expression address;
          {{(chain down the expression plex to set "expression      c
          address" to point to the next expression)
          IF (link) = 0 :
               {dummy = link + 2;
               IF (dummy) HAS descripter bit set :
                    expression address = (dummy) & descripter bit   c
                    exterminator;
                    expression address = expression address AU 1 +  c
                    expression address;
                    expression address = expression address + 1 AU 2 :
```

```
                    OTHERWISE : expression address = dummy + 2
                    FINISH CURRENT i}
                link = (link)
                REPEAT}} }}}
        (plant instruction to update stack pointer)
        PLANT load address, register, displacement, 0, register
        IF switch = 0 : (the stack pointer is not a register)
            PLANT store, register, naddress, 0, br
        PROCESS CONTROL AFTER IMPERATIVE plexbase (2)}}}} }}}}}
```

(Semantics) ROUTINE

```
IMP : DESTACK (semantics of destacking, referred to as SEM015 in PL360)
    CHECK BLOCK SYNTAX GIVING RESULT @ a
    IF a :
        {{{{CHECK STACKPOINTER GIVING @ register USED @ naddress AND        c
        SET @ switch
        IF switch ≠ - 1 :
            {{{(find how many variables are to be destacked, and plant      c
            code to retract the stack pointer by this no. of locations)
            dummy = (plexbase)
            (asp) = first bit set; asp (1) = (dummy)
            ACCESS CONSTANT asp GIVING POSITION @ p
            PLANT subtract, register, p, 0, constant base register
            IF switch = 0 : PLANT store, register, naddress, 0, base register
```

```
(first) variable address = dummy + 1; displacement = 0
FOR i = 1 TO (dummy) :
     (plant code to assign to a variable)
     {{PLANT load, expression accumulator, displacement, 0,    c
     register
     ASSIGN variable address, 0
     NEXT displacement
     UNLESS i = (dummy) :
          {(move "variable address" to point to next variable)
          IF (variable address) HAS descriptor bit set :
               variable address = (variable address) &         c
               descripter bit exterminator;
               variable address = variable address AU 1 +      c
               variable address + 1 AU 2 :
          OTHERWISE : variable address = variable address + 2} }}
PROCESS CONTROL AFTER IMPERATIVE plexbase (2)}}} }}}}
```

ROUTINE

```
STANDARD IMP : CHECK STACK POINTER GIVING register USED naddress AND SET switch
     (This routine is auxiliary to the STACK and DESTACK Semantic routines.  c
     Its function is to declare the stackpointer to the compiler to obtain   c
     a plex form.  The plex is then checked.  If it is a semantic constant,  c
     a fault is monitored and the output parameter "switch" set to - 1.      c
     If it is a register, then that "register" is passed back to the calling c
     routine, together with a "switch" setting of 1.  Otherwise, "register"  c
```

```
    indicates the "comparison register", "switch" is set to zero, and        c
    an instruction is planted to load the "comparison register" with the     c
    stack pointer value.  ("register" is subsequently used as an index       c
    into the stack)}
    nameaddress  = plexbase (1); SUBSTITUTE IDENTIFIER PLEX AT nameaddress
    IF (nameaddress) HAS first (constant) bit set : switch = - 1; (code      c
    to generate the error USE OF  SEMANTIC CONSTANT).... :
    OTHERWISE :
    {IF (nameaddress) HAS register bit set : register = (nameaddress) &      c
    register bit exterminator; switch = 1 :
    OTHERWISE : register = comparison register; switch = 0; naddress =       c
    (nameaddress); PLANT load, register, naddress, 0, base register}

(Semantics) ROUTINE
IMP : FINISH (referred to as SEM016 in PL360)
    (This routine undertakes the semantics of a "FINISH" statement.          c
    A check is first made that the currently-being-defined routine           c
    is IMP : if not then an error is indicated, otherwise a routine          c
    is called to plant the branch instruction and add it to the              c
    control list associated with "FINISH" references)
CHECK BLOCK SYNTAX GIVING RESULT @ a
IF a :
    {{(Find currently-being-defined routine's plex in routine dictionary)
    link = routine dictionary head
    repeat : IF link ≠ 0 :
```

```
        {UNLESS link (2) HAS second (currently being defined) bit set:
            link = (link); GO AND repeat}
    if imp tag bit = link (2) LD 27 & bottom bit set
    IF if imp tag bit (is set) : (print INVALID ROUTINE EXIT error)....:
    OTHERWISE : PERFORM FINISH SEMANTICS}}
```

ROUTINE

```
STANDARD IMP : PERFORM FINISH SEMANTICS
    (This is an auxiliary routine to the "FINISH" and "CONDITION (NOT)    c
    SATISFIED" ROUTINES.  Its job is to plant an incomplete branch        c
    instruction and add it to the Control List associated with            c
    FINISH refs., then to process control)
    PLANT branch on condition, 15, 0, 0, 0
    link = current control data
    {(find top level of control data) IF (link) ≠ 0 : link = (link) ;     c
    REPEAT}
    ADD code address  - 1 TO link - 1
    PROCESS CONTROL AFTER IMPERATIVE (plexbase)
```

(Semantics) ROUTINE

```
IMP : RESOLVE (referred to as SEM017 in PL360, this routine handles       c
the semantics of RESOLVE/LET p INTO/≡ m y, (using class abbreviations).   c
Code generated is equivalent to the assignment sequence:                  c
    P1 = p(1); p2 = p(2);.... but optimises by only accessing             c
    p once and using it as an index)
```

```
CHECK BLOCK SYNTAX GIVING RESULT @ a
IF a :
    {{{(check whether there are any p.v.s. in the m.p.e.  If not, then      c
    the instruction does not generate any code)
    dummy = plexbase (1); count = (dummy)
    IF count ≠ 0 :
        (plant code to load "register" with phrase variable)
        {{phrase var address = (plexbase)
        register = variable address accumulator
        pvplex  = (phrase var address)
        IF pvplex HAS first (constant) bit set :
            ACCESS CONSTANT phrase var address GIVING POSITION @ p;
            PLANT load, register, p, 0, constant base register :
        OTHERWISE :
            {IF pvplex HAS register bit set :
                register = pvplex & register bit exterminator :
            OTHERWISE :
                PLANT load, register, pvplex, 0, base register}
        (plant code to perform sequence of assignments)
        address = dummy - 1; displacement = 0
        FOR i = 1 TO count :
            [NEXT displacement; address = address + 2
            PLANT load, expression accumulator, displacement, 0, register
            ASSIGN address, 0} }}
    PROCESS CONTROL AFTER IMPERATIVE plexbase (2)}}}
```

```
(Semantics) ROUTINE
IMP : PERFORM NAME (referred to as SEM018 in PL360)
     (Semantics of "PERFORM n")
     LOCAL : cr ≡ comparison register, rir ≡ routine index register,        c
     br ≡ base register
CHECK BLOCK SYNTAX GIVING RESULT @ a
IF a : {{SEARCH 0 (identifier) DICTIONARY FOR (plexbase) GIVING             c
PLEX ADDRESS @ a
     IF a = 0 : {(make a new entry in dictionary)
          (asp) = next free rt location; asp (1) = 0; NEXT next free rt location
          a = asp
          ADD (plexbase) TO 0 (identifier) DICTIONARY WITH PLEX ADDRESS asp} :
     OTHERWISE : {IF (a) HAS first (constant) bit set : (Print USE OF        c
     SEMANTIC CONSTANT error)....; FINISH}
     SET segmentation delay switch
     PLANT branch and link register, cr, 0, 0, 0 (to load run-time address   c
     of next instruction into cr)
     (plant code to update return address and store into scalar n)
     IF (a) HAS register bit set :
          PLANT load address, cr, 10, 0, cr;
          PLANT load register, (a)& register bit exterminator, cr, 0, 0 :
     OTHERWISE : PLANT load address, cr, 12, 0, cr;
          PLANT store, cr, (a), 0, br
```

```
     PLANT GO TO BRANCH FOR (plexbase) (plants a jump as for GO TO n)
     (plant code to reset index value (stored in routine index register)    c
     on returning from the PERFORM sequence)
     IF index value = 0 :
          PLANT subtract register, rir, rir, 0, 0 :
     OTHERWISE :
          {(asp) = first (constant) bit set; asp (1) = index value
          ACCESS CONSTANT asp GIVING POSITION @ p
          PLANT load, rir, p, 0, constant base register}
     SET NO segmentation delay switch
     PROCESS CONTROL AFTER IMPERATIVE plexbase (1)}}

(Semantics) ROUTINE
IMP : FINISH NAME (referred to as SEM019 in PL360)
     (This routine performs semantics of "FINISH n".  A check is          c
     made that "n" is not a semantic constant, and if so code is          c
     planted to load the"comparison register" with scalar "n" and         c
     to branch to that address)
CHECK BLOCK SYNTAX GIVING RESULT @ a
IF a : {{SUBSTITUTE IDENTIFIER PLEX AT (plexbase)
       a = (plexbase)
          IF (a) HAS first (constant) bit set :
               (print USE OF SEMANTIC CONSTANT error) .... :
```

```
        OTHERWISE :
            {IF (a) HAS register bit set :
                register = (a) & register bit exterminator :
            OTHERWISE :
                PLANT load, comparison register, (a), 0, base register;
                register = comparison register
            PLANT branch on condition register, 15, register, 0, 0}
        PROCESS CONTROL AFTER IMPERATIVE plexbase (1)}}
```

(Semantics) ROUTINE

```
IMP : FINISH EACH (referred to as SEM020 in PL360)
    CHECK BLOCK SYNTAX GIVING RESULT @ a
    IF a : PERFORM FINISH 0 (EACH) SEMANTICS
```

(Semantics) ROUTINE

```
IMP : FINISH CURRENT (referred to as SEM021 in PL360)
    CHECK BLOCK SYNTAX GIVING RESULT @ a
    IF a : PERFORM FINISH 1 (CURRENT) SEMANTICS
```

ROUTINE

```
STANDARD IMP : PERFORM FINISH switch SEMANTICS
    DUMMY = (plexbase)
    SUBSTITUTE IDENTIFIER PLEX AT dummy
    PLANT branch on condition, 15, 0, 0, 0 (an incomplete branch instruction)
    (Check CD blocks for succeeding levels of control data to see if the    c
    given control variable name is present.)
```

```
    link = current control data
    {{IF (dummy) = link (7) (the CD block entry reserved to hold          c
    control variable) :
        {IF switch = 0 (FINISH EACH) : dummy = 3 (Control List 1) :
        OTHERWISE (FINISH CURRENT) : dummy = 4 (Control List 2)
        ADD code address - 1 TO link + dummy
        GO AND finish}
    link = (link)
    REPEAT UNTIL link = 0}}
    (print UNRECOGNISED CONTROL VARIABLE error)....
finish : PROCESS CONTROL AFTER IMPERATIVE plexbase (1)

(Semantics) ROUTINE
IMP : CLOSING CS BRACKETS (referred to as SEM022 in PL360)
    (This routine handles the semantics of a sequence of closing compound  c
    statement brackets, followed by any terminator)
    CHECK BLOCK SYNTAX GIVING RESULT @ a
    IF a : PROCESS CONTROL AFTER IMPERATIVE (plexbase)

(Semantics) ROUTINE
IMP : ROUTINE HEADING (referred to as SEM023 in PL360)
    (This routine performs the code generation corresponding to a         c
    routine heading.  A routine plex is found on the Analysis Stack.      c
    The action taken is to insert "code address" into the top level       c
    CD block then to store the "code address" into the respective index table c
    entry.  Then:                                                         c
```

```
(a)  A MAIN routine generates no code, but sets "start address".        c
(b)  A BASIC  routine generates code to save the routine addressing     c
     registers, then to initialise the routine base and index           c
     registers.                                                         c
(c)  A STANDARD routine generates code to store an activation           c
     record on the run-time stack, then to initialise the routine       c
     base and index registers.)
(Insert code address into CD block) current control data (5) = code address
(Extract tag bits from routine plex)
rplex = (plexbase); tag bits = (rplex) & tag bits extractor LD 27
(Make entry in index table)
serial number (of routine) =(rplex) & tag field exterminator
index table (serial number) = code address
IF tag bits HAS second bottom (MAIN) bit set : start address = code     c
address :
OTHERWISE :
     {{{{IF tag bits < 4 (STANDARD) : (plant code to save an activation  c
     record)
          {{{PLANT store multiple, 1, 15, 4, rdb register
          PLANT load address, expression accumulator, serial number, 0, 0
          PLANT store, expression accumulator, 68, 0, rdb register
          PLANT store, rdb register, 72, 0, rdb register
          PLANT load address, rdb register, 72, 0, rdb register}}} :
```

```
OTHERWISE (BASIC) :
    {{{IF rplex (1) > ## FFFF : svaddress = rplex (1) LD 16 :
     OTHERWISE : (reserve 3 locations for save area)
          {{svaddress = next free rt location
          next free rt location = next free rt location + 3
          (Alter routine plex) link = routine dictionary head
          {IF link (1) = (rplex) :
               link (2) = svaddress LU 16 | link (2) :
          OTHERWISE : link = (link) ; REPEAT} }}
     (plant code to save routine addressing registers (which are      c
     contiguous) into the BASIC routine savearea, "svaddress")
     PLANT store multiple, routine index register, svaddress, routine  c
     base register, base register
(plant code to initialise routine base and index registers)
PLANT load register, routine base register, comparison register, 0, 0
     (note that base address of routine will be in the comparison     c
     register)
PLANT subtract register, routine index register, routine index        c
register, 0, 0}}}}
SET instruction count (global) = - 1 (so that when control returns to  c
the PROCESS STATEMENT routine, it will be incremented by 1 to become zero)
```

(Semantics) ROUTINE

IMP : ROUTINE CALL (referred to as SEMO24 in PL360)

```
     (This routine handles the semantics of an IMP routine call.  A call  c
     is made of the TEST ROUTINE TYPE routine to check that the given      c
     routine plex is that of an IMP routine, then a call is made of the    c
     COMPILE ROUTINE CALL CODE .... routine to actually generate code.)
```

```
CHECK BLOCK SYNTAX GIVING RESULT @ a
IF a :
    {TEST ROUTINE (pointed to by) (plexbase) TYPE IS 0 (IMP)
    COMPILE ROUTINE CALL (plexbase) CODE
    PROCESS CONTROL AFTER IMPERATIVE plexbase (1)}

          ROUTINE
STANDARD IMP : TEST ROUTINE p TYPE IS given if imp tag bit
    (This routine is called by either ROUTINE CALL, or the routine      c
    associated with the semantics of an IF routine call condition.      c
    Its function is to inspect the if imp tag bit of the routine plex   c
    on the Analysis Stack, comparing it with a given if imp tag bit     c
    value.  If they differ then an error condition is monitored since   c
    an IF/IMP routine is being called as an IMP/IF routine respectively.)
    (Skip to the end of the chain of actual parameter plexes)
    link = p; dummy = p
    {dummy = (dummy) & top (I/O marker) bit exterminator
    IF dummy ≠ 0 : link = dummy; REPEAT}
    (Extract the if imp tag bit) if imp tag bit = link (1) & if imp bit c
    extractor LD 27
    IF if imp tag bit ≠ given if imp tag bit : (print INVALID ROUTINE
    CALL error)....
```

ROUTINE

```
STANDARD IMP : COMPILE ROUTINE CALL n CODE
    (This routine plants code generated by an IF or IMP routine call)
    LOCAL : ea ≡ expression accumulator, br ≡ base register, cr ≡ c
    comparison register
    link = n (Access the routine plex)
    {IF (link) ≠ 0 : link = (link) & top bit exterminator; REPEAT}
    routine plex (address) = link + 1
    first (formal) parameter (address) = routine plex (2)
    parameter (address) = first parameter
    (plant code to compute all actual parameters and to store into formal c
    parameter locations) link = n
    {{{{IF (link) = 0 :
        {{{(is it an I/O i.e. variable parameter?)
        IF (link) HAS first (constant) bit set : (yes)
            {{IF link (1) HAS register bit set : (scalar register variable)
            register = link (1) & register bit exterminator :
            OTHERWISE : {register = ea
                IF link (1) HAS descriptor bit set : (subscripted variable)
                    ACCESS VARIABLE link + 1;
                    PLANT load, ea, 0, 0, vaa :
                OTHERWISE (ordinary scalar variable) :
                    PLANT load, ea, link (1), 0, br} }} :
        OTHERWISE (an input only parameter i.e. an expression plex) :
            ASSIGN 0, link + 1; register = ea }}};
```

```
        PLANT store, register, parameter, 0, br;
        NEXT parameter;
        link = (link) & top bit exterminator;
        REPEAT}}}}
(Plant code to load base address of called routine from index table        c
into the "comparison register" (cr) then to jump to that address)
PLANT load, cr, (routine plex) & tag field exterminator + index area, 0,   c
constant base register
PLANT branch and link register, return register, cr, 0, 0
(Rescan the Actual parameter plexes, compiling code to copy back           c
any I/O marked parameters from their formal parameter location)
    parameter (address) = first (formal) parameter (address); link = n
    {{{IF (link) ≠ 0 :
        {{IF (link) HAS first (I/O marker) bit set :
            {IF link (1) HAS register bit set :
                PLANT load, link (1) & register bit exterminator,      c
                parameter,0, br :
            OTHERWISE : PLANT load, ea, parameter, 0, br;
                ASSIGN link + 1, 0} }};
        NEXT parameter; link = (link) & top bit exterminator; REPEAT}}}
```

Notes to Appendix F :

The remaining routines contained in the compiler are listed by name (together with their PL360 names) as follows. A cross-reference is also included to indicate where, in the thesis text, further explanation of a routine may be found.

	RCCT name	PL360 name	Explanation
1.	GET CODE BASE	GETCODEBASE	This routine loads the base address of the compiler code into the global variable "code base"
2.	READ	READ	§3.1
3.	WRITE	WRITE	§3.1
4.	NEWLINE	NEWLINE	§3.1
5.	SPACE	SPACE	§3.1
6.	PRINT	PRINT	§3.1
7.	PRINT COLON	PRINT_COLON	§3.1
8.	MESSAGE INTERPRETER	MESSINT	§3.2
9.	SEARCH i DICTIONARY FOR x GIVING PLEX ADDRESS a	SEARCH_DICT	§4.4
10.	ADD x TO i DICTIONARY WITH PLEX ADDRESS a	ADD_DICT	§4.4
11.	SUBSTITUTE IDENTIFIER PLEX AT p	SUBPLEX	§4.4
12.	SUBSTITUTE RPLEX AT p WITH n PARAMETERS	SUBRPLEX	§4.4
13.	ADD a TO l	ADDTO	§5.3
14.	FILL IN l TO a	FILLIN	§5.3
15.	PLANTCODE i, r, d, x, b	PLANTCODE	§5.12

	RCCT name	PL360 name	Explanation
16.	ACCESS CONSTANT c GIVING POSITION p	ACCESSCONST	§5.7
17.	COMPLETE BRANCHING BY s	COMPBRANCH	§5.2
18.	FINISH OFF PREVIOUS ROUTINE	FINISHOFF	§5.5
19.	PERFORM SEGMENTATION PROCEDURE	PERSEGPROC	§5.12
20.	PLANT i, r, d, x, b	PLANT	§5.12
21.	DUMPCODE	DUMPCODE	§5.16
22.	TRANSFER CONTROL	TRANSFER	§5.16
23.	PERFORM LEXICAL ANALYSIS	LEX_SCAN	§3.4
24.	PERFORM SYNTAX ANALYSIS GIVING RESULT r	SYNTAXSCAN	§4.7
25.	PACK ROUTINE NAME ON ANALYSIS STACK, s	PACKRNAME	§4.5
26.	CHECK RSYNTAX FOR s PARA-METERS GIVING parmcount	CHECKRSYNT	§4.5
27.	CHECK BLOCK SYNTAX GIVING RESULT r	CHECKBSYNT	§5.4
28.	ACCESS VARIABLE v	ACCESSVAR	§5.9
29.	PROCESS CONTROL AFTER IMPERATIVE p	PROCONTROL	§5.3
30.	COMPILE CODE FOR EXPRESSION e	COMPCODE	§5.9
31.	ASSIGN v, e	ASSIGN	§5.9
32.	PROCESS STATEMENT	PROCESS	§5.15
33.	- main program -		§5.15

Primary Semantics are handled by the PL360 routines:

SETSEMANT

NEXTSEMANT

IFSEM

PERSEMCOND

SEMO25

SEMO26

.

.

.

.

SEMO39

I/O Semantics are handled by the PL360 routines:

PERIOSEM

SEMO40 (handles READ CARD TO e)

SEMO41 (handles WRITE LINE FROM e)

APPENDIX G

BRIEF SUMMARY OF THE RCC METALANGUAGE

(It is not intended to give a complete description of the RCC formal syntax definition notation here, but only to show those features which appear in this thesis.)

The RCC metalanguage differs from B.N.F. in a number of ways. The former has attempted to retain the power of B.N.F. whilst adding a number of features to enhance the flexibility of syntactic definition.

Firstly, the B.N.F. metasymbols "<", ">", "::=" and "|" are represented in RCC by the metasymbols "[", "]", "=" and ",", respectively. Secondly, syntactic class names are represented by upper-case letter strings in RCC rather than the lower-case letter strings of B.N.F.

e.g. The B.N.F. definition:-

<identifier> ::= <letter><identifier>|<letter>

is written in RCC as:-

[IDENTIFIER] = [LETTER][IDENTIFIER], [LETTER]

It is sometimes convenient to name a class by a sequence of any valid symbols rather than an upper-case identifier. This is allowed in the RCC metalanguage, but brackets "ⴑ" and "ⴒ" must be used instead of "[" and "]".

e.g. The user may write:-

{,} instead of [COMMA]

The RCC metasymbol "?" is used to denote the possible occurrence of the given class.

e.g. The B.N.F. definition:-

<terminator> ::= ;<eol>|;

is written in RCC as:-

[TERMINATOR] = ;[EOL?]

The RCC metasymbol "*" is used to denote a list of one or more occurrences of the given class. If "?" follows the "*" metasymbol then a possibly empty list of occurrences of the given class is denoted.

e.g. [LABEL] = [LETTER *][DIGIT *?]

is equivalent to the set of B.N.F. definitions:-

<label> ::= <letter string><digit string>|<letter string>

<letter string> ::= <letter><letter string>|<letter>

<digit string> ::= <digit><digit string>|<digit>

The RCC metalanguage allows classwords to be qualified by parameters. A parameter is a sequence of classwords and/or terminal symbols, the only restriction being that any classwords in it must not themselves be parametric. A parameter follows a classword, is preceded by a "/", and enclosed in round brackets if there is more than one symbol or classword in the sequence. Further detail is not required at this point since only two uses of parameters are made in the syntactic description of RCCT in this thesis, and are as follows:-

i) If the parameter appears with a class using the "*" metasymbol, it is taken to be the separating string. Thus [DECLARATION */ξ,ϑ] defines a syntactic class consisting of a list of [DECLARATION]'s separated by ξ,ϑ's.

ii) Two system defined classes [P] and [Q] are available, either of which may have a parameter consisting of a sequence of terminal symbols only (- if there are commas in the sequence they are interpreted as separating alternatives). These classes allow simple definitions to be made local to a phrase.

e.g. The user may write

[Q/(*, -)]

directly in a phrase rather than, say, using

[STAR OR DASH]

and then defining

[STAR OR DASH] = *, -

The only difference between [P] and [Q] is that [Q] leaves a record on the syntax tree whereas [P] does not.

The final form used in the text is the NOT....BUT sequence. Following the NOT metasymbol is a sequence of phrases separated by commas. A similar sequence follows the BUT metasymbol. This form then defines a single class. A sentence in this class is defined to be any sentence of the phrases following the BUT sequence excepting any sentence of the phrases following the NOT sequence.

e.g. In B.N.F.:-

<octal digit> ::= 0|1|2|3|4|5|6|7

<digit> ::= <octal digit>|8|9

In RCC this could be defined either as above, or:-

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

[OCTAL DIGIT] = NOT 8,9, BUT [DIGIT]