The COMPL language and operating system

By A. G. Fraser* and J. D. Smart†

A description is given of the Compiler and Operating System used in the preparation of the NEBULA compiler for the I.C.T. Orion computer. The article describes the facilities available within the system, but does not go into any detail concerning the actual implementation of the system.

1. Introduction

The NEBULA compiler for the I.C.T. Orion computer (Braunholtz, 1961) is probably one of the largest software projects yet undertaken outside the United States. The compiler itself contains about 250,000 words of program, and was written by a group which, at the later stages, contained some 28 programmers. At an early stage in the design of the compiler it was realized that it would be necessary to employ a compiler if the task were to be completed successfully. In practice the early decision to use the computer as a tool in the preparation of the NEBULA compiler has been the one main feature without which the compiler would certainly have failed.

The early problems of technical design were later enhanced by the needs of a group of programmers working on a common program. This latter situation led to the need for extensive standardization in all aspects of program writing, development and maintenance. Every effort was necessary to ensure compatibility between the various components of the one program, and to this had to be added the controlled flexibility needed to develop and assemble the component parts of the system. These considerations gave rise to the design of an operating system which contained, and was integral with, the compiler group's own compiler.

The source language used by the NEBULA team is called COMPL and it is this specially designed language and associated operating system which is described here.

The COMPL system was designed so that it could be used in the final NEBULA compiler itself. The total integration between the two systems has led to a number of operating inefficiencies, but it has significantly assisted both the development and maintenance of the NEBULA compiler. The design of the COMPL language itself reflects the needs of the NEBULA compiler, and for this reason it must be regarded as a special-purpose tool. In spite of this the language, and indeed the entire system, possess properties which could make it interesting to other groups.

No creation, when it eventually emerges, will satisfy its creator, and the tortuous history of development which the COMPL system has undergone during the past four years has left it in a form that leaves much to be desired. The reader must therefore regard it as a well-worn tool which was designed to do a specific job of work.

2. The COMPL system

The COMPL system and its compiler are integral parts of a whole, and consequently the language which is used for program preparation merges with the language of directives that control the system. No clear boundary can, therefore, be drawn between the two components. For the purposes of this article the individual procedure statements in a COMPL procedure (or subroutine) will be described under the heading of the COMPL programming language, and all else will be given under the heading of the COMPL system.

The system itself is built around its storage organization and it is the latter which is the subject of a major part of the hidden complexity in the systems programs. The principal storage space "known to the user" is a large single-level store of about 256,000 words of 48 bits each. This store is commonly referred to as the "data store."

The entire configuration comprising system program and store is best thought of as having a continuous existence. When the system is not actually running on the computer it is held on magnetic tape in the form which it last reached. The process of unloading the system program and data onto magnetic tape is known as *dumping* and the process of re-establishing the system on the computer is known as *restoring*. The system can be dumped under operator control at any time.

Overall control of the system is achieved by means of a series of directives which are read into the machine from paper tape. The system reads one directive and then carries out the action requested before reading the next directive. One such directive is:

PERFORM Test Program; BEGIN 2.

This directive instructs the system to find the procedure called Test Program and enter it at entry point 2. There is no STOP instruction in the system but all procedures terminate with the EXIT statement which causes control to return to the higher level. In the case of the above example control returns to the operating system when EXIT is obeyed in the test procedure. The system will then proceed to read and obey the next directive from paper tape.

In exceptional circumstances the operator may interrupt the operation of the system by typing a message on

* *I.C.T. Ltd., Bracknell, Berks.* (now at *University Mathematical Laboratory, Corn Exchange St., Cambridge*).
† *I.C.T. Ltd., Bracknell, Berks.*

the console Flexowriter. This message causes the system to halt the current job, whatever it might be, and to read and obey one directive typed on the console Flexowriter. Having performed the requested action the system continues with the job previously interrupted.

The directives that may be given to the system are many and varied but one whose effect has already been described is DUMP. The other directives that are available are listed in Section 6. One important directive which should be mentioned here, however, is the COMPL procedure. The system treats an entire procedure as if it were one directive, and the effect of carrying out this "pseudo-directive" is to compile the procedure and to store it within the system. Having this way "defined" a new procedure the user is free to call upon it in any way he wishes.

When a procedure is compiled the object code subroutine is left in the data store from where it can be obeyed if necessary. However, this is not usually its final resting place since there is a second storage area reserved and designed specifically for program. This second store is the CHAPTER store. In this second store COMPL object program is stored in 512-word chapters, and it is from the chapter store that a procedure is more frequently obeyed.

The process of clearing out procedures from the data store and putting them into chapters is initiated by the directive UPDATE. This is effectively a "spring cleaning" operation which is usually done at regular intervals. Much effort has been put into the design of the system in an attempt to make it do that which is expected of it even in extreme circumstances. One example of this is the fact that a procedure can be executed whether or not an update has been done since it was compiled, in spite of the fact that the two modes of execution are vastly different from an internal viewpoint. The extra effort required to implement this policy of rigid adherence to a uniform design has been proved repeatedly to be sound and, indeed, many cases of common failure can be traced to points where this policy has not been properly carried out.

There is a group of directives which are available to the COMPL programmer which enable him to use and test the compiled procedure. PERFORM is a directive that has already been mentioned, and the other directives are directly linked to this. The directive NEST allows the user to set up the 7 nest words (working-space registers described in Section 3) and this setting will apply when the next PERFORM is obeyed. MONITOR is a directive which allows the user to make use of the trace and check-point monitoring facilities that have been built into the system.

There are two trace facilities available.

(*a*) The system will print one line on the monitoring peripheral whenever a PERFORM or EXIT is obeyed.
(*b*) The system will produce a train of symbols on the monitoring peripheral which describe the path taken by the object program. Since the user does not usually know the details of the object code this trace is in terms of the source language branching in the original COMPL version. A single letter (N for NO or Y for YES) is printed each time that a COMPL source language condition is evaluated. G is printed if a GO TO statement is obeyed, and the program address of a labelled sentence is printed whenever such a sentence is obeyed.

The check-point monitoring facility relies upon check points written by the programmer in his original COMPL version of the program. This, in fact, has proved to be the weakness of the system, and it would have been much more valuable had it been possible to insert check points after the procedure had been compiled. Nevertheless the facility does provide a useful range of printing styles that can be output on the monitoring peripheral. The adequacy of this facility is demonstrated by the way that check-point monitoring has been used regularly in place of the other methods of output for test results. The design of this facility was assisted materially by the discipline which was imposed on programmers in the use of the store, most data being held in a standard and recognizable format.

The MONITOR directive itself sets up the styles of monitoring which apply when the next PERFORM is obeyed. In addition to the style of printing the user can monitor on selected procedures and suppress any monitoring action on the others.

The remaining directives fall into four major groups.

(*a*) Directives for the administration of the compiled procedures (e.g. CHAPTER which allocates a specified procedure to a specified chapter in the chapter store).
(*b*) Special facilities developed directly in association with the NEBULA compiler. These are aids to the administration and development of the NEBULA compiler while being assembled.
(*c*) Directives which operate on and in terms of the COMPL object program and data store (e.g. LIST allows the user to read a list structure into the data store).
(*d*) Directives for "class definition." (See Section 5 for a description of this specialized part of the system.)

3. The storage system

There are principally three components to the COMPL storage system. These are:

(*a*) the data store
(*b*) the nest
(*c*) the chapter store

The data store contains about 256,000 words which are addressed by the integers starting at 4096. The content of word 12305 is referred to as W(12305) so that the statement SET W(12305) = 26 causes the integer 26 to be set up as the value contained in word

12305. Similarly to negate this we write SET W(12305) = −W(12305).

In addition to the data store there is a small store of seven words known as the *nest*. These words are referred to by the integers 0 to 6 inclusive, and the content of nest word 6 is written as N(6).

An extension to the notation described above allows a reference to the data store to be written as W(N(5)) or even as W(W(W(N(5)))) where the number of the data store word referred to is to be taken as the value of the item specified within the brackets.

Both the nest and data store can be referred to in assignment statements and conditions, but there are also special procedure statements which are associated with the special roles played by these two stores.

Associated with the nest are the statements PUSH DOWN and POP UP. The actions of these statements associate the nest with a "stack" of copies of the nest. Each level in the stack contains one copy of each of the seven nest words, and a new level is made in the stack when the PUSH DOWN statement is obeyed. PUSH DOWN does not change the content of the seven nest-words but merely arranges to place a copy of the nest onto the top of the stack. POP UP ALL has the reverse effect and copies from the stack to the nest before removing the top layer of the stack. The statement

POP UP 0 ; 1 ; 2 ; 6

causes only N(0), N(1), N(2) and N(6) to be reset from the stack although the entire top level, consisting of all seven words, is removed from the stack.

The administration of the data store is the responsibility of the operating system, and each COMPL program is expected to communicate with the system when changing its store requirements. For this purpose the FIND and FREE statements are provided. No program may use any part of the data store which has not been allocated to it by the system, and it is also the program's responsibility to tell the system when it has finished with a particular segment of store.

The statement FINDR BLOCK 6 INTO N(2) will cause the system to allocate 6 words of data store to the program and the number or "address" of the first of the 6 words is placed in N(2). Thus the program may now assign the value 3 to the first and last of these words by writing

SET W(N(2)) = 3 THEN SET W(N(2) + 5) = 3.

When finished with, these words of store are relinquished by writing

FREE N(2) ; BLOCK 6.

To obtain a measure of efficiency in an otherwise random system of store allocation one may write

FINDR BLOCK 6 INTO N(2) ; NEAR N(3)

which tells the system that an increase in running speed is to be expected if the 6 words allocated to the program

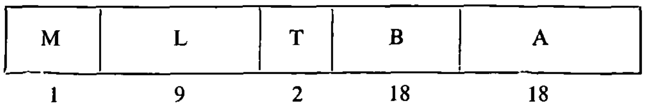

Fig. 1.—The list word format

are physically near the data store word whose address is in N(3).

Since the locations in the data store are allocated by the system and not by the programmer there is a system of symbolic addresses available to the program writer. A data store address can be represented by a *label* such as LE6 or LA2. The data in the store can then be referred to by writing, for example, W(LA2 + 9). The system will handle any program that contains a reference of this type and when, ultimately, a data store address is assigned to that label then the appropriate value is written into the COMPL program.

The word length employed throughout is 48 bits, and for many purposes it is the programmer who decides how he will store his data. However, the COMPL system is specially organized to handle one particular format and this format is known as the *list word*. A list word is a 48-bit word sub-divided as shown in **Fig. 1.** The COMPL programmer may refer to the five sub-fields of a word directly. For example, NA(2) refers to the 18-bit subfield called A in the nestword N(2). Similarly one may refer to WT(N(3)). The A and B fields will usually hold the addresses of other list words and the M and T fields have conventional meanings which are associated with A and B, respectively.

A *main chain*, for example, is a set of list words in which the first contains the address of the second in its A field, the second contains the address of the third in its A field and so on. (See **Fig. 2.**) The last word in a main chain is recognizable because it has M = 0, whereas all other words in the chain will have M = 1. The address of the main chain is the same as the address of the first word on the chain.

The interpretation of the B field depends upon the value of T. Commonly, B will contain the address of some other main chain, and in this case T will be either 0 or 2. If T = 3 then B will be a single data field and will not usually be the address of another list word. The value T = 1 is used when a large volume of data has to be associated with one list word, and in this case the following B words will contain these data. The L field has no special meaning but can contain any data. The programmer may thus form his own complex pattern of interconnected list words. Such structures are normally referred to as *tree structures*. In the tree structure shown in **Fig. 3** the shaded areas represent data which have no special effect on the shape of the tree structure itself.

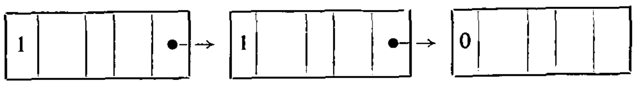

Fig. 2.—A main chain

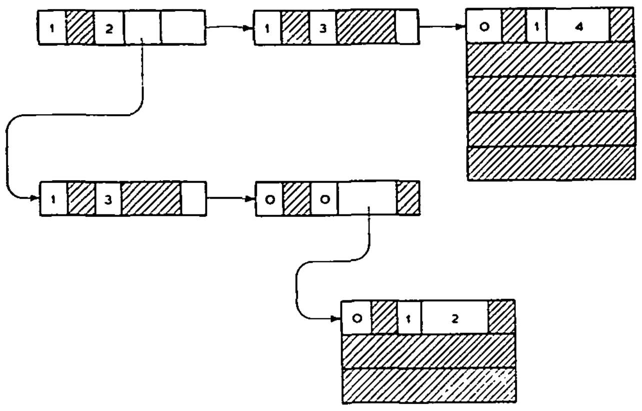

Fig. 3.—A tree structure

There are special instructions which can be used only on list words and tree structures since they assume that the conventions described above are adhered to. For example FREE N(3); TREE is equivalent to using FREE ; BLOCK on every entry in the tree whose address is in N(3).

The language also includes functions which operate upon tree structures. One such function is LABELW as in

SET N(3) = LABELW [6 ; N(2)]

The action of this statement is to search the main chain whose address is to be found in N(2) and to find the first list word on the chain for which L = 6. The address of this list word is then put into N(3). If there is no word with L = 6 on the specified main chain then N(3) will be set to zero.

The third level of store is the chapter store which is a series of 512-word blocks of chapters. Each chapter contains an integral number of procedures and is used solely for this purpose; the COMPL programmer may not use any part of a chapter as working space. The user, in fact, knows very little about the chapter system and in general he is only concerned with the allocation of specific procedures to chapters. This latter chore is not done automatically since it materially affects the running speed of the system, and no simple mechanical system for producing an optimum solution to this problem was found.

4. The COMPL programming language

The NEBULA compiler is built up from about 1,000 individual subroutines, most of which were compiled from COMPL. These subroutines were written and tested independently by the different members of the team. To avoid administrative difficulties and accidents all procedures have equal priority in the system and all must adhere to the same protection rules. In particular every procedure must have a unique name, it must have no more than four declared entry points, and no procedure can enter another except at a declared entry point. Storage protection for data between one procedure and another is facilitated by the free store mechanism in the data store and by the PUSH DOWN mechanism for the nest. As a discipline aimed at a healthy programming style, procedures are restricted in length to a maximum of about 500 words of object code.

```
COMPILING PROCEDURE [count words]
[Begin 0]
  SET N(3) = 0.
[Begin 1]
  IF N(2) = 0 THEN EXIT.
[L]
  SET N(3) = N(3) + 1.
  IF WT (N(2)) = 1 THEN SET N(3) = N(3) + WB(N(2))
  OTHERWISE If WT(N(2)) ≠ 3
                THEN PUSH DOWN
                THEN SET N(2) = WB(N(2))
                THEN PERFORM count words; Begin 1
                THEN POP UP 2.
  IF WM(N(2)) = 1 AND WA(N(2)) ≠ 0
                THEN SET N(2) = WA(N(2))
                THEN GO TO L.
EXIT.
END OF PROCEDURE.
```

Fig. 4.—A COMPL procedure which counts in N(3) the number of words held in a tree whose address is held in N(2)

The format of a COMPL procedure is illustrated in Fig. 4. It will be seen that the name of the procedure is given in the heading, and the entry points are marked by writing as program labels. The principal syntactic unit is the sentence which may consists of one simple statement or may be conditional as illustrated by the fourth and fifth sentences in Fig. 4. Most statements in the example will be self explanatory once the user has acquainted himself with the method of referring to data. The interpretation of a conditional sentence is such that only one sequence of statements is obeyed, and this sequence is that which follows the first true conditional expression. Conditions are formed using the symbols $=$, $\neq$, $>$, $\not>$, $<$ and $\not<$ and are compounded using AND and OR. Arithmetic expressions may be used almost anywhere that an integer is permitted. All expressions are evaluated using integer arithmetic and may use the symbols +, —, *, /, ⟨+⟩ (logical OR), ⟨*⟩ (logical AND), and ⟨≠⟩ (logical not equivalent). Sub-expressions may be written in parenthesis if required.

Certain functions are also available in the language and these may appear as the operand in any arithmetic expression. All functions are specified by the function name, and the operands, which are listed after the name, are enclosed in brackets thus:

LABELW [2 ; N(6)]

This particular function searches a list and, indeed, most COMPL functions are non-arithmetic in this way. All, however, produce a single integer result.

The selection of functions and statements which are to be found in the language fall into the following

```
DEFINE CLASS sentence: 8 = IF: Condition: THEN: action,
                              alternative?
                              TO USE condition comp
                              OR statement, extra statement . . .
                                TO USE
                              Statement comp.
DEFINE CLASS alternative: 9 = OTHERWISE: Sentence.
DEFINE CLASS extra statement: 10 = THEN: statement.
DEFINE CLASS action: 11 = statement, extra statement . . . TO
                              USE
                              Statement comp.
```

Fig. 5(a).—Class definition of a sentence

groups and reflect the fact that the language is a special purpose tool designed for one particular purpose.

(*a*) Facilities required in any procedural language (e.g. simple assignment, control transfer and subroutine execution).
(*b*) Facilities required to communicate with the storage system (e.g. FIND, FREE, PUSH DOWN and POP UP).
(*c*) Facilities aimed at handling COMPL tree structures (e.g. LABELW and FREE TREE).
(*d*) Facilities introduced as special macro operations because they appear frequently in the NEBULA compiler.
(*e*) Facilities associated with Syntax analysis. (See Section 5).

A full list of the procedure statements and functions is given in Section 6. The facilities, under heading (*d*), which were designed specifically for the NEBULA translation process have not been included because they must be discussed in the much larger subject of the NEBULA translation itself. Some facilities which should strictly come under (*d*) have been isolated and are discussed in detail in Section 5. This is because the process of syntax analysis itself may be of more general interest.

5. Syntax analysis

Both the NEBULA and the COMPL compilers employ the same first stage of compilation which is referred to as *syntax analysis.* The COMPL system includes a general purpose syntax analysis program which matches a sample sentence against a set of syntactic rules which are held in the store as a tree structure and is similar to a scheme devised at Manchester for Atlas 1 (Brooker, 1960). These rules are referred to as FORMAT TREES and define the form of a correctly constructed sentence.

The format tree is in practice a COMPL tree structure but there is an equivalent written form which is called a *class definition.* Each format has a name which is the class name, and the class definition is in effect the source language for a compiler that produces format trees in lieu of object program.

In Section 2 it was stated that the COMPL system automatically translated any procedure that was presented to it. This is also true for any class definition

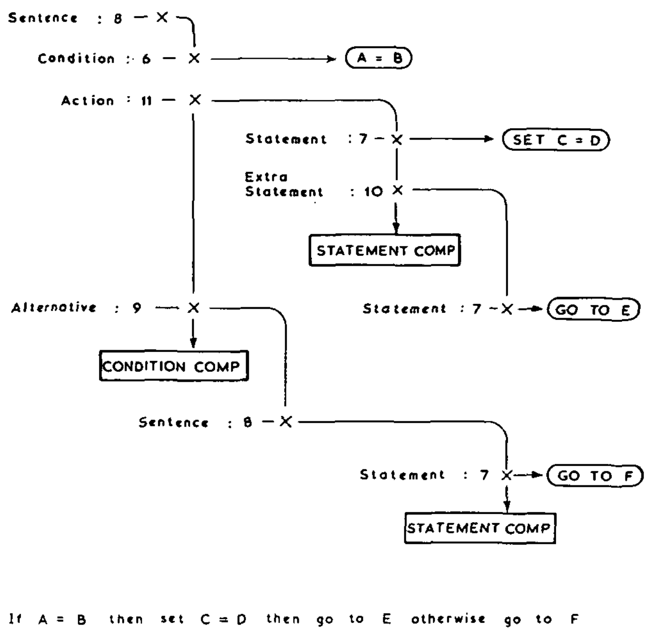

Fig. 5(b).—Analysis tree for a sentence

except in this case a format tree and not a procedure is stored in the data store.

A class definition is illustrated in **Fig. 5(a).** In this example a definition is given for the class SENTENCE in terms of the simpler concepts of STATEMENT and CONDITION. Having compiled the definition of the class SENTENCE and all the classes which that definition requires (e.g. CONDITION and STATEMENT) the program can use a standard subroutine to test any sequence of input source language to see if it is a valid sentence. If it is valid then an analysis tree is produced which contains in its shape an interpretation of the original source language. It is this analysis tree which both the NEBULA and COMPL compilers use in their respective translation processes. The analysis tree is a COMPL tree structure and a pictorial representation of part of such a tree is shown in **Fig. 5(b).**

Since analysis trees and their manipulation are common features in the NEBULA compiling process certain conventions are used and special facilities are available to handle these trees.

A statement specially designed for producing an analysis tree is GEN. For example:

GEN Sentence = {If A = B then set C = D then go to E otherwise go to F} INTO N(1).

This statement generates the tree shown in Fig. 5(*b*) and puts its address in N(1). An important refinement to this facility is shown in the following example:

GEN Sentence = {If Condition [N(2)] then set C = D} INTO N(1).

In this case it is assumed that a COMPL procedure has previously put the address of the analysis tree for

```
COMPILING PROCEDURE [condition comp]
[Begin 0] PUSH DOWN THEN SET NB(5) = NB(5) + 1.
          SET N(1) = WB(N(1)).
          PERFORM PROCEDURE (ETB[W(N(1))]).
          IF NM(6) = 1 THEN GO TO fail.
          SET  N(1) = WA(N(1))  THEN  PERFORM  PRO-
            CEDURE (ETB[W(N(1))]).
          IF NM(6) = 1 THEN GO TO fail.
          IF WM(N(1)) = 0 THEN EXIT.
          SET N(1) = WB(WA(N(1))).
          PERFORM PROCEDURE (ETB)[W(N(1))]).
[fail]    POP UP 1;2;3;4;5 THEN EXIT.
END OF PROCEDURE.
```

Fig. 5(c).—A COMPL procedure showing a method of controlling the compilation of a conditional sentence, defined by the definitions in Fig. 5(a). On entry N(1) is the address of the analysis tree of such a sentence and NB(5) is the current value of a branch count, i.e. a count of the number of conditional sentences found connected by OTHERWISE. In this case the procedure would use itself recursively

a condition in N(2) and now wishes to generate a sentence embodying this condition. The use of a class name as a function in this way is known as a *parameter*.

By convention, in any analysis tree the terminal A field of a main chain (i.e. the A field of the word with M = 0) can indicate the procedure to be used to process this chain. This information is compiled into the corresponding FORMAT TREE by the TO USE option on a class definition as shown in Fig. 5(*a*), the effect on the analysis tree being shown in Fig. 5(*b*). This information is easily obtained from an analysis tree by use of the function ETB, as in

SET N(2) = ETB [W(N(1))].

This function requires as its operand a whole word and if the T field of the given word is three then the value of the function is the B field of this word. If the T field is two or zero then the B field is taken to be the address of a main chain and the value of the function is then the value of the terminal A field of this main chain. Thus, if in the above example N(1) is the address of the analysis tree for the Sentence shown in Fig. 5(b) then N(2) will be the address of the procedure *Condition Comp*. Furthermore, if this statement is followed by the statement "Perform procedure (N(2))" the procedure *Condition Comp* will be entered with N(1) giving the address of the analysis tree to be processed. As an example of this **Fig.** 5(c) shows a method in which the procedure *Condition Comp* could compile, using an analysis tree produced from the definitions in Fig. 5(*a*).

This facility provides a valuable switching mechanism which is controlled by the source language.

An important extension to this facility is given by the COMPILE statement, as in:

```
COMPILE Sentence = {If A = B then set C = D then
                    go to E otherwise go to F}.
```

This statement is equivalent to the statements

```
PUSH DOWN.
GEN Sentence = {If A = B
                then set C=D
                then go to E
                otherwise go
                to F} INTO
                NA(1).
PUSH DOWN THEN PER-
FORM PROCEDURE
  (ETB[W(N(1))]).
POP UP 1;2;3;4;5 THEN
  FREE NA(1); TREE.
POP UP 1;2;3;4;5.
```

Note that this sequence assumes that N(0) and N(6) will be changed by the compilation involved and, in fact, they are expected to indicate the results of the compilation.

A COMPILE statement, like GEN, allows the use of parameters.

These facilities in the COMPL programming language, together with the ability to compile new class definitions provide a system which is easily extended and is a potentially powerful mechanism for handling character strings.

One rather specialized example of the use to which this device can be put should perhaps be mentioned here. This is the class MESSAGE which is used solely in GEN and COMPILE statements to produce messages for output. Besides accepting a text in plain language it allows the text to be given in a parametric form as shown in the following example:

```
COMPILE message = {error in Identifier [N(2)] at
                   sentence Number [N(5)] — Mes-
                   sage [N(6)]}.
```

This example assumes that N(2) contains the address of the analysis tree of an Identifier, e.g. a NEBULA source language name, that N(5) contains an integer, and that, previously, a message had been GEN'd into N(6). Thus, if N(2) points to the analysis tree of the name "FRED" and N(5) = 10 and N(6) had been set up by the statement "GEN Message = {incorrect format} INTO N(6)", the execution of the above COMPILE will produce the message:

```
ERROR IN FRED AT SENTENCE 10—INCORRECT
                FORMAT
```

6. Summary of directives, statements and functions

6.1 COMPL directives

WAIT

This terminates a paper tape and instructs the system to disengage the tape reader and then call for a further paper tape.

```
                COMPL
            LIST LA6
            0     0          1234
            11    2
            LA8   12         3    LA7
                  13         0
                        14   3    4567
                        END  1    0
                  END        0    LA6
            LA7   15         3    9876
                  END        0    999
```

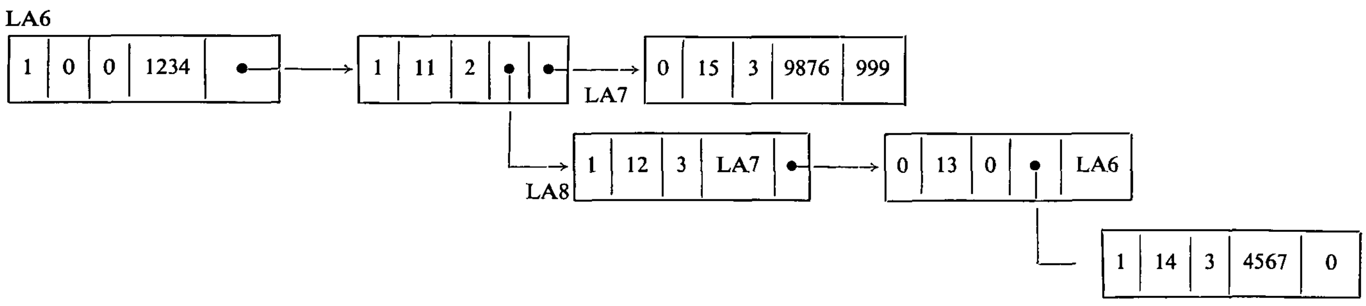

Fig. 6(a).—Input of a tree structure using LIST directive

DUMP AGF 3

This instructs the system to make a complete dump of itself on a magnetic tape called

NEBULA/DUMP/AGF/3

UPDATE AGF 3

This directive instructs the system to carry out the "spring-cleaning" operation which updates the chapter store in the light of all changes that have been made since the last update. The operation terminates with a dump.

READ NEBULA/TESTS/272/-

This directive instructs the system to suspend reading from the current paper tape without unloading it, and to search for another paper tape on a separate reader. The new paper tape will have the heading DOCUMENT NEBULA/TESTS/272. When the system has finished reading the new paper tape it returns to the one on which reading had been suspended.

END

This directive terminates a secondary paper tape and causes the system to return to the primary tape. (See READ.)

SET LA2 = 123

This directive allows the user to make an explicit label setting. A more advanced form of this is frequently useful: SET LA2 = LA5 A2 B causes LA2 to be set to the value WB(WA(WA(LA5))).

FREE LA2

This clears a label setting and allows that label to be used again.

LIST LA6

This directive is the first line of a tree structure and the system is asked to store a copy of that tree structure in the data store. The label LA6 will then be set to be the address of the first word in this tree.

The directive is followed by the list words punched one per line usually as 3 integers representing the L, T and B fields. The words in the tree are automatically linked together by the system with the appropriate settings for M and A. The terminal M and A values for any one branch of the tree are specified as integers on the line which begins END. If T = 2 then the subsidiary main chain can follow immediately and the B field will be filled in by the system. If T = 1 then the contents of the block of B words must be specified on the following lines. An example of a LIST directive is given in **Fig. 6(a).**

FREELIST LA6

This directive frees the store used by the tree structure whose address is given by LA6.

ALTER LA6 A3 L

This directive allows the content of one data store field to be changed. In this case the field to be changed is WL(WA(WA(WA(LA6)))) and the new value of this field will be given on the line following the ALTER directive.

MONITOR TRACE PROG A/PROG B/PROG C

This directive specifies the style of monitoring to be used when a subsequent PERFORM directive is obeyed. The directive can specify two different monitoring styles, and for each a list of procedures on which this monitoring should take place is given. The two styles are TRACE and CHECKPOINT.

PERFORM TEST PROG: BEGIN 3 and PETRACE

This directive instructs the system to obey the procedure called TEST PROG at entry point 3. While doing this it should trace all PERFORM and EXITS. Alternatively the directive can specify "AND CHECKPOINT". In this case procedural monitoring is switched on and those procedures mentioned in a previous MONITOR directive will produce the monitoring specified when obeyed.

NEST N(1) = 2
N(2) = LISTW [0|2|3|LA2|0]

This directive instructs the system to set up the nestwords as specified and to store these values for use on execution of the next PERFORM directive. In this case N(0), N(3), N(4), N(5) and N(6) are cleared and N(1), N(2) are set up as specified (see function LISTW for setting of N(2)).

This is a useful directive for testing procedures in isolation and this, with the LIST and monitoring facilities, provide a comprehensive development aid.

6.1.2 Administration of COMPL procedures

COMPILING PROCEDURE [count words]

An example of a COMPL procedure is shown in Fig. 4, and the operation of this directive is to cause such a procedure to be compiled. The compilation is terminated by END OF PROCEDURE. If any errors are found during compilation the system completes the current pass of compilation in order to find the maximum possible number of errors and then leaves the procedure in a *cancelled* state.

COMPILE FROM Nebula/Compl/Lib/AGF/1/Test-proc/–

In order to improve the speed of COMPL compilation an off-line job can be used to carry out the syntax analysis of a COMPL procedure. This job uses the same syntactic rules as the COMPL system but the resultant analysis trees are stored on magnetic tape. The COMPL system can be directed to compile, from this form of input data, by using the COMPILE FROM directive. In the above example, the analysis trees for the procedure identified as Testproc are retrieved from the magnetic tape called Nebula/Compl/Lib/AGF/1 for compilation. The off-line job provides a fast system for syntax checking a COMPL procedure before compilation.

CANCEL Prog A/Prog B/Prog C

This directive cancels an unwanted procedure. It removes all the object program compiled for these procedures, in this case Prog A, Prog B and Prog C, and leaves them marked as "referred to but not compiled yet".

ERASE Prog A/Prog B

This directive removes all trace of procedures from the system, and is used to erase procedures that are no longer required or that have been incorrectly named. In the above case procedures Prog A and Prog B will be erased.

REFLIST Prog A = ADD Prog B THEN REMOVE Prog C

For the purpose of improving the running speed of COMPL programs each procedure has associated with it a list of the procedures which it performs. This list is usually constructed during compilation and the REFLIST directive is used in certain circumstances to make additions and corrections. The directive, in the above example, will cause procedure Prog A to record that it now refers to Prog B but no longer refers to Prog C.

CHAPTER abcd = ADD Prog A AND Prog B THEN REMOVE Prog C THEN ASSEMBLE

In the chapter store each 512-word block, or chapter, is identified by a four-character name, e.g. abcd as shown above. A chapter directive operates on the single chapter specified by the name given. In the above example procedures Prog A and Prog B are assigned to the chapter abcd and the procedure Prog C, which was previously assigned to this chapter, is now removed from it. Finally, the system is instructed to assemble this chapter in the chapter store. The operations requested by chapter directives are held over by the system and the actual changes to the chapter store only take place when an UPDATE directive is given. If a change has been made to a procedure that is in a chapter, e.g. it has been CANCELLED or compiled, or a CHAPTER directive specifies some change to a chapter, as above, then the UPDATE directive will automatically "dismantle" the chapter, and after the necessary changes have been made, the chapter will be "assembled".

Since chapters are stored serially in ascending values of chapter names the situation can arise whereby a chapter stored in one section of the chapter store is required by other chapters which are physically distant from it. To overcome this a copy is made of a chapter containing frequently used procedures, and the copy is placed near to those procedures that use it. This is done by the COPY OF directive as in CHAPTER xyz3 = COPY OF abcd.

6.1.3 Directives for class definitions

DEFINE CLASS alternative : 9 = $\underline{\text{OTHERWISE}}$ Sentence

The general purpose and operation of this directive is described in Section 5, and Fig. 5(*a*) shows some examples of the use of this directive. This directive also makes it possible to modify a definition previously given, e.g. DEFINE CLASS Sentence : 8 = CHANGE : Condition Comp = $\underline{\text{UNLESS}}$: Condition : $\underline{\text{THEN}}$: action, alternative $\overline{\text{TO USE}}$ Unlesscond Comp.

This example causes the alternative of the class sentence which was TO USE the procedure *Condition Comp* to be replaced by the format given. It is also possible to define a class in such a way that certain formats are not allowed as in DEFINE CLASS consonant : 12 = letter NOT vowel.

DELETE CLASS sentence

The operation of this directive is self-explanatory in that it simply removes the format trees associated with the class specified.

6.2 Summary of statements

COMPILE Sentence = {Set N(1) = W(N(2))}.

The operation and use of this statement has been explained in Section 5.

COPY W(N(1)) to W(N(6)) ; 6

This statement is used to copy a block. In the example given the block of 6 words, the first of which is W(N(1)), is copied to the block starting at W(N(6)). In order to clear a block the source may be written as ZERO.

EXECUTE Prog A ; Begin 1

This statement is equivalent to PERFORM then EXIT, and transfers control from one procedure to another without storing a link. See PERFORM for further details.

EXIT

This returns control to the procedure which entered the current one by a PERFORM statement. If EXECUTE was used to enter a procedure then control will be returned to the procedure above that.

FINDR TREE 10 INTO N(1) ; NEAR NB(6)

This statement requests storage space from the system and cannot ask for more than 128 words at a time. In the above example, 10 words of store are found, near the data store address in NB(6) and the address of this space is put into N(1). Since a TREE is requested the space is supplied as a main chain. Alternatively the space can be requested as a BLOCK as described in Section 3. In that case the requested space is supplied completely empty.

FREE N(1) ; TREE

This returns storage space that is no longer required to the system. The space to be returned may be a BLOCK of a specified length as described in Section 3 or may be, as above, a TREE. When freeing a TREE all substrings pointed to by a word with T = 2 are freed but T = 0 is treated as T = 3, i.e. the B field is regarded as a data field.

GEN Sentence = {If Condition [N(2)] then set N(1) = W(N(6))} INTO NA(1)

The operation and use of this statement has been explained in Section 5.

GO TO fail

This causes a transfer of control to the sentence identified by the label specified, in this case to the sentence labelled FAIL. (See Fig. 5(*c*).) A GO TO statement may not specify a BEGIN label.

PACK N(3) INTO W(N(4)) ; 1 ; 9

This statement will pack data into a non-standard field of a nestword or data store word. The field to be used is specified by two integers, the first being the position of the most-significant bit in the word; the second gives the length of the field. The above example is equivalent, therefore, to

SET WL(N(4)) = N(3).

PERFORM Prog A ; Begin 1

This statement transfers control to the procedure specified having stored a link to be used when EXIT is obeyed. In the example given procedure *Prog A* is entered at *begin* 1. Alternatively the procedure can be specified by an operand, as in PERFORM PROCEDURE (ETB[W(N(1))]). In this case the operand specifies the data store address which the system uses to identify a procedure internally. (See Section 9.)

POP UP 1 ; 2 ; 4 ; 5

This statement, together with PUSH DOWN controls the operation of the "stack"; see Section 3. This POP UP statement removes the top level from the stack resetting nestwords N(1), N(2), N(4) and N(5). POP UP ALL resets all the nestwords, and POP UP just removes the top level of the stack. No procedure may "pop up" more times than it has "pushed down" but may "push down" more often than it "pops up". In this latter case the EXIT instruction will cause the system to execute the necessary POP UP statements to reset the stack but will not modify the nestwords.

PUSH DOWN

This simply stores the current values of the seven nestwords on the top of the stack without altering them.

SET WB(N(1)) = N(2) + NL(3) * WB(NB(4))

This is used to assign a value to a word, or part of a word in the data store or in the nest.

6.3 Functions

LISTW [1|WL(N(5))|3|WA(NB(5))|N(4)]

The purpose of this function is to set up a word in the standard listword format. The value of this function is therefore the 48-bit quantity specified by the MLTB and A fields given. Field overflow is ignored everywhere.

ENDW [W(LA6)]

The value of this function is the contents of the penultimate word in the main chain whose address is given. Thus the least-significant 18 bits of this value give the address of the last word in the main chain. The last word of a main chain is indicated by having M = 0 or by A = 0 if M = 1. If the above example is applied to the tree in Fig. 6(*a*) it will be seen that the value of the function, in this case, is

LISTW [1|11|2|LA8|LA7].

ETB [W(WA(LA6))]

The value of this function is the terminal A field of the substring addressed by the word given. Again, using Fig. 6(*a*), the value in the above example would be LA6. A complete description of the use and operation of this function was given in Section 5.

LABELW [15 ; W(LA6)]

The purpose of this function is to scan the main chain whose address is given in order to find the first word which has an L field of the value specified. If no such word exists the value of the function is zero. Applying the above example to Fig. 6(*a*) would give the value as LA7.

LSHIFT [W(N(2)) ; 15]

This function provides a logical shift up and its value is the result of shifting up the word given by the number of places specified.

RSHIFT [W(N(2)) ; 15]

This provides an unrounded arithmetical shift down and is specified as the function LSHIFT. As these two functions represent different types of shift it should be noted that in either case a negative shift can be used to shift in the opposite direction.

NFIELD [W(N(4)) ; 1 ; 9]

This function permits the extraction of non-standard fields from the nest and data store words. The position and length of the field is specified as in the PACK statement. The above example is equivalent to WL(N(4)).

STRINGR [10|WB(N(5)) : N(4)]

This function is used to introduce a tree structure to the system and its value is the address of the tree created. The expression between square brackets represents a main chain where the elements on the chain are separated by semicolons. The terminal A field of this chain is given by the last entry in the square brackets; the terminal M is zero if this is preceded by colon and one if preceded by semicolon. The LTB fields of each word of the chain can be specified in the following ways

L|T.B or L|TB or LT.B or LTB.

Thus the above example, if the result is to be held in N(1), is equivalent to
FINDR BLOCK 1 into N(1).

SET W(N(1)) = LISTW [0|10|0|WB(N(5))|N(4)].

A more complex example of the use of this function is given in **Fig. 6(b).**

VALUE [N(1)]

This function is provided to enable the analysis tree for an integer to be converted to its binary value. An integer is defined as Digit, Digit . . ., i.e. any number of digits but at least one. The operand between the square brackets gives the address of such an analysis tree, and the value of the function is the value of this integer as a binary number.

```
SET  N(1) = STRINGR[1234 ; 11|2.STRINGR[12|3.LA7 ; 13|
STRINGR [14|3.4567 ; 0] : LA6] ; 63.9876 : 999].
```

Fig. 6(b).—Creation of tree structure shown in Fig. 6(a) by use of STRINGR function

7. The implementation of the COMPL system

For reasons which are purely historical the operating system is broken down into two distinct parts: the COMPL system proper and the CIL system. All those operating facilities which are associated with the data store and the COMPL object program together with the mechanism for controlling the peripheral and the interface with the computer operator form the CIL system. The COMPL system itself handles all the remaining facilities including the initiation of the compilation processes.

The user of this two-level system must know which directives are handled by which system and must use extra directives to change from the one system to the other. The directive COMPL causes the CIL system to enter the COMPL system, and the directive COP has the opposite effect. This unfortunate split in responsibilities has frequently given rise to operating errors and is one of the less desirable features of the system which result from pure administrative difficulties within the compiler team.

The CIL system itself is again divided in two. The operations which control the execution of procedures, the data store and, in general, the environment for a COMPL object program are all the responsibility of the CIL interpreter. The remaining features of the system which include practically all the directives which are recognized in the CIL system are handled by a program called COP. This split in the design of the system is logically necessary since it is the CIL interpreter that transforms the Orion computer into a machine with large data store and chapter store, etc., and COP is merely one of the programs held within this artificial machine. The COMPL system proper is also held in the form of a COMPL object program.

The mechanics of the COMPL and COP system programs are reasonably straightforward and do not justify any special discussion in this context. The operations of the interpreter, however, may well be of interest to the reader.

The interpreter provides three essential facilities: The storage organization of the COMPL system, the mechanism for handling the COMPL object program and the interface with the operator. The first and second of these are described in Sections 8 and 9, respectively.

During normal operations the entire COMPL system is controlled by a series of directives punched into paper tape. This paper tape input channel, like the monitoring peripheral, is controlled by the lower-level subroutines of the system, and these are written in such a way that the entire input stream, or monitoring output stream, can be switched from one peripheral to another by operator action. The other peripherals of the system are controlled by standard routines also, so that organizational changes which affect the use of peripherals can be made by operating on the central routines of the system.

One instance of the value of this technique is provided by the DUMP and RESTORE facilities which have been built into the system. These facilities will operate at any time and are able to do so because the CIL system can recognize the position and status of all devices in use by the program. The DUMP directive specifies a particular magnetic tape and onto this tape will be written a copy of all the program and data held within the system. It is thus possible to use this tape at any subsequent date to reinitiate the system at the point which it had reached just prior to the dump.

The practice of making dumps is normally introduced to combat the effects of machine failure, but in the COMPL system DUMP plays two additional and important roles. It provides the continuity between one day's operations and the next, and it provides a mechanism which is used by programmers to isolate the system for investigation or further development work.

Two dumping methods are therefore included. A "Master" dump is initiated by the directive DUMP MAST and this causes the system to make a dump on the next tape in a cycle of master dump tapes. These are used for dumps made for security reasons and for the provision of continuity between computer runs. "Private" dumps are initiated by the programmer for his own purposes by writing "DUMP initials/number", where the initials and number specify a privately owned magnetic tape. A variety of checks are included in the system to guard against errors in operating and to ensure that the cycle of MASTER dumps is not accidentally destroyed.

8. Implementation of the storage system

The COMPL system is written to run on an Orion computer with at least 8K of core store, 2 magnetic drums, 4 magnetic tape decks, a paper tape reader and a line printer (or paper tape punch). The way in which the equipment is used has been heavily conditioned by the desire to make the system compatible with the eventual NEBULA requirements.

The main components of the conceptual system have already been described; they are the chapter store (capable of holding 250,000 words of program), the nest, and the data store (of 256,000 words). It is the task of the system's store-handling facility to interpret the actual configuration so that it performs like the conceptual system.

The store-handling facility is essentially a set of subroutines which sit permanently in the core store and which have been linked together by an interpreter. The decision to use an interpretive language as the object code of the COMPL compiler was largely taken in order to effect some reduction in the size and complexity of the NEBULA compiler program. The use of an interpretive scheme made it very much simpler to write the COMPL compiler and also allowed more advanced program development facilities to be written into the system.

The interpreter handles the data store in blocks of 128 words each and the chapter store in blocks of 512 words each. The two stores are handled quite separately and each has allotted to it a certain fraction of the core store and drum store.

The data store blocks are allocated to specific block positions on the pre-addressed magnetic tapes. There must be at least two pre-addressed tapes but if more are available the operator may allocate these to the system thus increasing its running speed. The 2048 data store blocks are distributed equally among the pre-addressed tapes, and therefore if more tape decks are available the amount of data held on any one tape will be proportionately reduced. The drum space which is used to hold data store blocks can be varied, but to make a change to this is not a simple operator action.

Space is usually allocated on the drum for about 100 blocks and in addition there is room for 12 blocks in the core store. A "learning" program is used to control the transfer of data store blocks between the core, the drum and the magnetic tape. The strategy used by this program is similar to that used in the Atlas 1 drum learning program (Kilburn *et al.*, 1962).

When the system has to bring a new block into the core store it will usually write the least frequently used block from the core to the drum. At the same time it may find that it has to clear a space on the drum which means that the least recently used drum block must in turn be written back onto magnetic tape. There are a few special cases which are recognized; in particular the system does not write back onto drum or tape a block which has not been changed in value.

Similarly free blocks are not transferred from one part of the machine to another. It has also been found desirable to "look ahead" by clearing space in both core and drum before that space is actually required. This latter operation, however, has not yet been taken as far as it could, and there are strong arguments in favour of clearing significantly more space on the drum than is necessary at any particular time so that the clearing operation can work on batches of blocks.

The chapter store is handled by the interpreter's procedure entry and exit subroutines. Whenever control passes from one procedure to another the system ensures that the chapter holding the destination procedure is first brought into the core store. The core and drum are divided into blocks which each contain one chapter, and all the chapters are held on one magnetic tape. In addition each chapter is allotted to one particular core position and one particular drum position. The system then initiates drum and tape transfers on this basis by putting the required chapter in the core store if it is not already there, and by using the drum wherever possible.

The free store mechanism, which was mentioned in Section 3, is complicated by the fact that the data store is handled in blocks. The problem is made even more complex when one attempts to minimize the peripheral transfers involved when handling the free store. The solution adopted is to design the system so that the free store requests are met, if possible, in a block held already in the core. If this is not possible then a partly filled block which is held on the drum is used, and if no such block exists then the space is allocated in a previously free block (peripheral transfers for which are suppressed automatically). To do this it is necessary to hold a free store pointer in an extra word carried around with each block, and in addition a value E is computed which gives an approximate measure of the sizes of the free store areas in the block. Two tables are then held in the core store: one gives the value of E for each block currently held on the drum, and the other contains a 1-bit marker which indicates whether or not a particular block is free.

To facilitate program development certain extra development aids have been included in the store handling system. On the one hand it is possible for the operator to turn on a trace which gives details of the peripheral transfers as they are initiated, and in addition there are built in checking facilities which ensure that free store is not used illegally. This latter trap has been most valuable, as has the trap which is based on the fact that the data store locations with small integer addresses are never made available to the programmer. Additional optimization has been made possible by allowing the programmer to specify a data store address near to which a newly found word should be for maximum efficiency. To assist in the detection of "store thieves" it is possible to print out at any time the number of words which are currently classified as being free store.

The system of program development aids also includes a standard post mortem which is initiated if the system should fail or if the COMPL programmer should come up against a trap. This post mortem gives printed details of the nest, the free store, those parts of the chapter and data stores which are currently in use, and some extra details about the procedure currently being executed. It has been found that this single post mortem is usually enough to allow a fault to be diagnosed correctly. The printout is a little lengthy, however, but a single post mortem facility which meets nearly all occasions and which can be initiated simply and quickly has proved to be most valuable.

Section 3 also mentioned the seven nest words. These seven words are held permanently in the core store although the stack with which they are associated is held in the data store. The stack is held in a series of data store blocks and is used to hold subroutine links as well as copies of the nest. There is a trap, however, which prevents the user using the top entry on the stack in the wrong way.

In addition to the storage facilities known to the COMPL programmer there are eight additional registers which are available and which are used by the COMPL object program as working space to hold intermediate results. These too are held permanently in the core store.

9. The COMPL object program

The COMPL compiler compiles into a language called CIL, and it is this language which is interpreted by the CIL interpreter. The CIL procedures may contain machine orders and in some of the more critical parts of the compiler this is indeed the case. In general, however, the CIL instruction is used because it can be produced from a COMPL program and because a CIL instruction is considerably more compact than the equivalent Orion instructions. This latter situation arises mainly through the need to include data store handling program in addition to the arithmetic or logical operation itself.

The CIL procedure is a unit which operates in its own local store and is therefore completely relocatable. This is achieved by holding with each procedure a "directory" which is effectively a conversion table for converting an internal program address into a data store address. The directory also contains the program addresses of the four entry points to the procedure so that it is not necessary to know the internal format of a procedure when writing a PERFORM statement.

The procedure's internal addresses are specified as, for example, P(176) and this notation can be extended in the same way as a reference to the nest or data store. One can therefore write PA(176) to refer to the A subfield in P(176). The range of permitted program addresses is larger than the permitted maximum procedure size. The program locations that can be addressed are P(128) to P(4095) but no one procedure is normally permitted to contain more than about 500 words of program. The locations P(1) to P(7) correspond to the locations N(0) to N(6), and the extra registers used by the object program but unknown to the COMPL programmer are P(8) to P(15) or alternatively X(0) to X(7).

The COMPL object program can write into the procedure's own local storage space, although this facility is denied to the COMPL programmer. The reason for this restriction on the COMPL programmer is, like several other restrictions, designed to enforce a uniform and healthy approach to program writing.

The CIL instructions which are used in most COMPL object programs can be distinguished from the machine-code instructions by the absence of the most-significant bit. If the user attempts to obey a CIL instruction after a series of machine instructions the hardware automatically enters the interpreter, and similarly the CIL interpreter will transfer control to any machine order that it is required to obey.

The CIL instruction is essentially three-address and occupies 48 bits. The written format is, for example:

```
3  X(1)  N(1)  WA(NB(2))
```

This instruction sets X(1) = N(1) * WA(NB(2)). In general the format of an instruction is

function X-address Y-address Theta,

where the X-address specifies an X register or Nest word, the Y-address specifies any program address (and thus any X register or nest word as well), and Theta is an operand which is evaluated in 48 bits. The operand may be a simple integer or may be a data store reference plus a program address reference as in the above example.

The function list is somewhat arbitrary and is the result of a somewhat chequered history of development. In total there are 64 available function codes which are organized in several different groups. Examples of each group are given in **Fig. 7.**

The program development aids mentioned in Section 1 have their equivalent forms in the CIL object program. The trace on branches in a program is controlled by two 1-bit fields T1 and T2. If T1 = 1 the instruction was labelled in the COMPL version and if T2 = 1 the instruction is the "fail" jump in a conditional expression. In theory a third bit "T3" would be needed to mark the "success" jump in a conditional expression, but since only two bits are available the combination T1 + T2 is taken to have this meaning and therefore the occasional occurrence of T1 + T2 on a fail jump that was labelled gives an inappropriate printout. The check point monitoring facility also uses one bit in the CIL instruction. If this bit is set then associated with that instruction will be an entry in a "Monitoring Directory". This latter directory is part of the main directory associated with the procedure and which was described earlier in this section. The monitoring directory entry contains the program address of the instruction together with a series of codes specifying the monitoring styles to be used. Essentially these are a four-character text plus two requests for printouts of sections of the store using the addresses in the CIL instruction as the operands where necessary. For example the entry

P(324) 5 9 MON2

causes "MON2" to be printed followed by the volume of free store available (Style 5) and then the tree whose address is Theta in the associated CIL instruction (Style 9).

CIL Instruction				*Interpretation*
3	X(1)	N(2)	WA(N(3))	Set X(1) = N(2) * WA(N(3)).
10	0	P(312)	WB(N(1))	Set P(312) = WB(N(1)).
12	0	P(312)	W(NA(1))	Set WA(P(312)) = W(NA(1)).
16	0	N(3)	N(4)	Add N(3) to the integer part of the Y address in the next instruction and add N(4) to the integer part of theta in the next instruction.
17	N(1)	P(314)	N(3)	Jump to P(314) if N(1) > N(3).
24	N(2)	258	W(12784)	Note that 258 = 4*64 + 2. Put in N(2) the 4-bit field starting 2-bits down from the top of W(12784).
26	N(2)	258	12784	Put N(2) into the 4-bit field starting 2-bits down from the top of W(12784).
30	N(2)	N(3)	7	Find R Block 7 into N(2); near N(3)
33	N(2)	0	5	Free N(2); Block 7.
36	0	2	172846	Perform procedure 172846 at Begin 2.
38	0	0	0	Push Down.
44	N(6)	N(3)	16	Set N(6) = LABELW[16 ; N(3)].
50	3	N(2)	W(6172)	Note that 3 refers to the subfield L where subfields are numbered in the order ABTLM. Set NL(2) = NL(2) + W(6172).
59	N(1)	P(324)	6	Copy a block of 6 words from P(324) to W(N(1)).

Fig. 7.—Some CIL instructions

The procedure directory can therefore be seen to be the one important link between the body of the procedure and the external world. The procedure is, in fact, identified solely by the address of its directory, and the cross references within the system are implemented in terms of this one address. The directories, therefore, are never relocated, and when a procedure is cancelled and a new version is compiled it is only the content of the directory which is changed. This one device is, therefore, the corner stone to the administrative and technical mechanisms which were designed to ensure compatibility between the various segments of the NEBULA compiler during all stages of its assembly and development.

10. Acknowledgement

The work on the COMPL compiler was carried out by the Nebula Programming Branch in I.C.T. and the authors wish to express their gratitude to I.C.T. for permission to publish this article.

References

BRAUNHOLTZ, T. G. H., FRASER, A. G., and HUNT, P. M. (1961). "NEBULA: A Programming Language for Data Processing," *Computer Journal*, Vol. 4, p. 197.

I.C.T. NEBULA Reference Manual.

BROOKER, R. A., and MORRIS, D. (1960). "An Assembly Program for a Phrase Structure Language," *Computer Journal*, Vol. 3, p. 168.

BROOKER, R. A., and MORRIS, D. (1961). "Some Proposals for the Realization of a Certain Assembly Program," *Computer Journal*, Vol. 3, p. 220.

KILBURN, T., EDWARDS, D. B. G., LANIGAN, M. J., and SUMNER, F. H. (1962). "One Level Storage System," *I.R.E. Transactions on Electronic Computers*, April 1962.