AN AUTOMATIC PROGRAMMING SYSTEM TO GENERATE PAYROLL PROGRAMS

Thesis submitted in Fulfilment of the

Requirements for the Degree of

MASTER OF SCIENCE

Rhodes University

by

ELIZABETH VERA CATHERINE FIELDING

March 1978

ACKNOWLEDGEMENTS

I wish to express my gratitude to my supervisor Prof. M.H. Williams, who provided the idea for this project. I appreciated the assistance and guidance which he gave me, in particular his help during the initial planning and design stages.

The cooperation of the Officer in Charge of Rhodes University Computing Centre, Mr. R.M. Urry, was also greatly appreciated, as it considerably hastened the testing of the system.

In addition, I wish to thank Mrs. J. Urry for the time and patience she devoted to typing the bulk of this thesis, as well as Mrs. S. Hoffman for her contribution to the typing and Mr. H.L. Ossher and Mr. A.R. Bulmer for their help.

Finally, I acknowledge the financial assistance of the C.S.I.R., as I was in receipt of a C.S.I.R. bursary while this work was carried out.

C O N T E N T S

1 INTRODUCTION

The purpose of this project was to try to investigate one approach to the problem of automatically generating programs from some specification. Rather than following the approach which requires the user to define his problem using some formulation, it was decided to look at a class of problems that have similar solutions, but have many variations, and to try to design a system capable of obtaining user requirements and generating solutions tailored to these requirements. The aim was to design the system in such a way that it could be extended to cater for other classes of problems, so that eventually a system which could automatically generate program solutions for a range of problems might be developed.

Since the class of payroll problems has the property that user requirements may vary greatly, but the basic tasks of all payroll programs are similar, this class of problems proved to be ideal for the purposes of this work. Also, the fact that payroll programs are extremely common and are of considerable importance to most organizations[7], means that a system solving problems of this type would have wide application.

The current approach to the general solution of payroll problems is to construct a package for which the user has to provide parameters defining his problem. To avoid problems and complexities like those involved in setting up parameters for packages, it was decided that the automatic generating system should communicate directly with the user in determining his requirements. This decision was influenced by the idea that " the ultimate objective in automatic programming is a system that can carry on a natural language dialogue with the user (especially a non-programmer) about his requirements and then produce an appropriate program for him"[9]. However, the system is not so ambitious as to allow a free dialogue with the user, rather, he is presented with a set of questions in English which he has to answer.

Further drawbacks and problems with packages are that the user has in

general no control over the final product. The solutions are unnecessarily large as they contain routines for facilities which are not required, and they are also extremely difficult to alter. The system designed overcomes these disadvantages by generating a unique solution tailored to the requirements of a user. The solution is consequently easy to understand and is therefore simple to alter if required. All of these points are amplified in Chapter 2, which provides an overview of the problem and the solution approach.

The system has been designed as a set of programs, the chief of which are a Question-Answering Program, which is described in Chapter 4, a Format Editor, which is the subject of Chapter 5, and a Program Generator, which is discussed in Chapter 6. Chapter 3 of this thesis is devoted to a discussion of the notation which is used.

2 GENERAL VIEW OF THE PROBLEM AND SOLUTION APPROACH

2.1 OBJECTIVE

The object of this project was to try to design and implement a system to generate payroll programs automatically. It was decided to investigate the problem of generating payroll programs because the logic of a payroll program is fairly straightforward and thus does not complicate the task of trying to generate such programs. Also, payroll programs are of vital importance, as is borne out by the fact that there are more payroll programs than computers in existence,[7] so solution of the problem would clearly be extremely useful. The design of the system was, however, done in such a manner that it can be extended to produce solutions to other types of problems by redefining the data.

2.2 LANGUAGE TO BE GENERATED

The choice lay between generating a payroll program in a high level language, such as COBOL, or a low level language. Whilst a program in a low level language may be more efficient in terms of speed and size, it is machine-dependent and is difficult for a user to alter if he so wishes. A program in a high level language is both transportable and more easily readable.

Although efficiency is obviously important, COBOL was chosen to be the generated language, as this would enable the user to make amendments readily and it would also have a more general application. This choice satisfies the main aim of the system, namely, to produce a correct program tailored to the user's requirements.

2.3 WHAT INPUT

Once the decision to generate a high level language had been taken, the next problem to be considered was what form the input to the generating system should take. Clearly, in order to be able to generate a payroll program tailored to a user's requirements, information as to these requirements has to

be obtained from him in some way.

A conventional approach to providing a general solution to payroll problems is to have a "package" or set of routines which provides all possible facilities which might be required .[5,13] The user must then specify values for a set of parameters to define the solution providing the particular facilities which he desires. The parameters required by certain packages currently being marketed are extremely complicated and difficult to define, and have, in some instances, taken years to set up correctly.

Because of the complexity of specifying parameters, the idea of using a "question-answering" approach to obtain information from the user was considered. Using this approach, the user provides the information necessary to generate the program he requires by answering selected questions from a set of questions in a conversational manner. If these questions are clearly and simply stated, providing answers should be a simple matter, and so it was decided to adopt this approach in the hope that users would find it quick, efficient and easy to use.

2.4 STATEMENT OF THE PROBLEM

The problem of designing a system capable of generating a complete COBOL program tailored to a user's requirements by presenting him with a set of questions and obtaining his answers, resolved into the following sub-problems :

1) The problem of deciding what information is actually required in order to be able to generate a program.

2) The design of a suitable set of questions to be presented to the user, formulated in such a way that the information necessary to generate a program is extracted in a clear, concise manner and in a suitable order.

3) The design of a system to present the set of questions to the user and to process his replies.

4) The generation of a program skeleton based on user answers to questions.

5) The design of a system to obtain information concerning desired input and output formats.

6) The conversion of the program skeleton to programming language statements and the combination of these with format information to produce a final program.

2.5 BRIEF DESCRIPTIONS OF THE SUBPROBLEMS

2.5.1 INFORMATION REQUIRED

As is reflected by subproblems (4) and (5) above, two types of information are required in order to generate a program, viz.

(a) information enabling a program skeleton to be developed, from which, in the case of the problem under consideration, it is possible to produce COBOL PROCEDURE DIVISION statements.

(b) information defining input and output to the program, the formats of these and any processing such as sorting to be done to this information.

The way in which this information is obtained and is processed is discussed more fully in later chapters. The program responsible for obtaining information of type (a) above, as well as information of type (b) with the exception of format specifications, is referred to as the "Question-Answering Program". The program which obtains the remaining information of type (b) is called the "Format Editor".

2.5.2 DESIGN OF THE QUESTIONS

As there were no guidelines to follow, the design of the questions had to be done by trial and error. It was an iterative process which involved determining the needs of a payroll program, formulating a set of questions to determine the user's requirements in respect of these needs, and then revising and expanding the original needs and repeating the whole process until eventually a decision was taken to halt as most facilities seemed to have been provided for by the questions which had been developed.

Although this was probably the most time-consuming part of the project, the process cannot be easily described or formalized.

2.5.3 INTERFACE WITH THE USER

In using a conversational approach to obtain information from the user, the particular path followed through the set of questions is dependent on the answers provided by him at each stage. Thus questions concerning facilities which the user has indicated are not required, are bypassed. This approach is therefore more efficient than presenting the user with a questionnaire to complete as he does not have to furnish any information that is not directly concerned with his requirements. As queries on a questionnaire would usually be in a condensed form, the conversational approach to questioning is superior as questions can be clearly and explicitly presented. An additional advantage of interacting with the user is that his responses can be tested immediately and he can be invited to resubmit any answers found to be incorrect. Also, by seeing the path that the questions are taking, the user has greater opportunities to discover previous mistakes, and the system provides error recovery by allowing the user to backtrack to the last question which he knows he answered correctly.

The questions were stored effectively as a linked structure on a disc file, and the Question-Answering Program operates by reading a question from the file, displaying it on the console and accepting the user's reply and interpreting it. The Question-Answering Program sets up as output two files viz. a file containing the skeleton of the program to be generated, and a file containing information required by the Format Editor.

2.5.4 GENERATION OF A PROGRAM SKELETON

Stepwise refinement is used in generating a final program skeleton. Initially, a very general basic program skeleton and a set of structures were defined. The program skeleton, or flowchart, and the structures each consist of a set of linked elements or boxes, which may take parameters.

Among its other functions, the Question-Answering Program is responsible for developing the program skeleton in the process of interpreting user replies to questions. The question answers are used to select structures with which

to refine the program skeleton step by step, and to provide parameter values for the structures used in expanding skeleton elements. Expansion operations consist of replacing a skeleton element by one or more copies of a particular structure or linking one or more copies of a structure onto a skeleton element.

2.5.5 EXTRACTION OF INPUT/OUTPUT FORMATS

After information defining inputs and outputs of the generated program has been obtained from the user, there remains the problem of determining the formats in which program information is to be input and output. Once again, a conversational approach was used to present the user with the names of the various files, reports etc., which he has specified and to obtain his format specifications for these. The way in which this is done is described in Chapter 5, and the program responsible for this function is the Format Editor, as has been mentioned.

2.5.6 FINAL PROGRAM GENERATION

The generation of the final COBOL program proved to be fairly straightforward, and this process is performed by a program called the "Program Generator". Certain portions of the COBOL program to be produced were determined to be fixed and the code for these was supplied as data which simply have to be listed at appropriate points in the generation process. Apart from this, the generation process consists of converting the final program skeleton to COBOL source statements to provide most of the PROCEDURE DIVISION and combining this with the format specification results to produce a final complete COBOL program. The format specification results comprise the whole of the INPUT-OUTPUT SECTION and DATA DIVISION, as well as additional necessary PROCEDURE DIVISION statements.

2.6 STRUCTURE OF THE SYSTEM

The system designed to solve the problems described above consists of a set of programs and a set of disc files.

The programs include

1) an initializing program - to set up question data, codestring and fixed data and other data.

2) the Question-Answering Program

3) the Format Editor

4) the Program Generator

5) and three listing programs to list the question data in a readable form and enabling intermediate structures in the process of producing the final program to be listed.

The disc files include files for

1) the question data

2) other data required by the Question-Answering Program

3) codestrings for the Format Editor

4) fixed COBOL strings for the Program Generator

5) flowchart skeleton data for the Program Generator

6) Format Editor data

7) and four files to store FILE-CONTROL, WORKING-STORAGE SECTION, FILE SECTION and PROCEDURE DIVISION code produced by the Format Editor and used by the Program Generator.

The relationship between these system components is depicted in Figure 2.1.

2.7 IMPLEMENTATION LANGUAGE

The language which was used to implement the system was PASCAL. The choice was influenced by the facts that the Question-Answering Program uses recursion in interpreting question answers, which is not available in FORTRAN, and the Program Generator and Format Editor handle character data which is extremely cumbersome in FORTRAN. A procedure-oriented language leads to a far more structured and modular system and the programs comprising the system were more easily written in this type of language and are more readable.

ALGOL could have been used, as the structure-defining facilities of

PASCAL were not required and the filing system was not found to be altogether satisfactory. However, the ALGOL compiler in use at Rhodes University is unreliable.

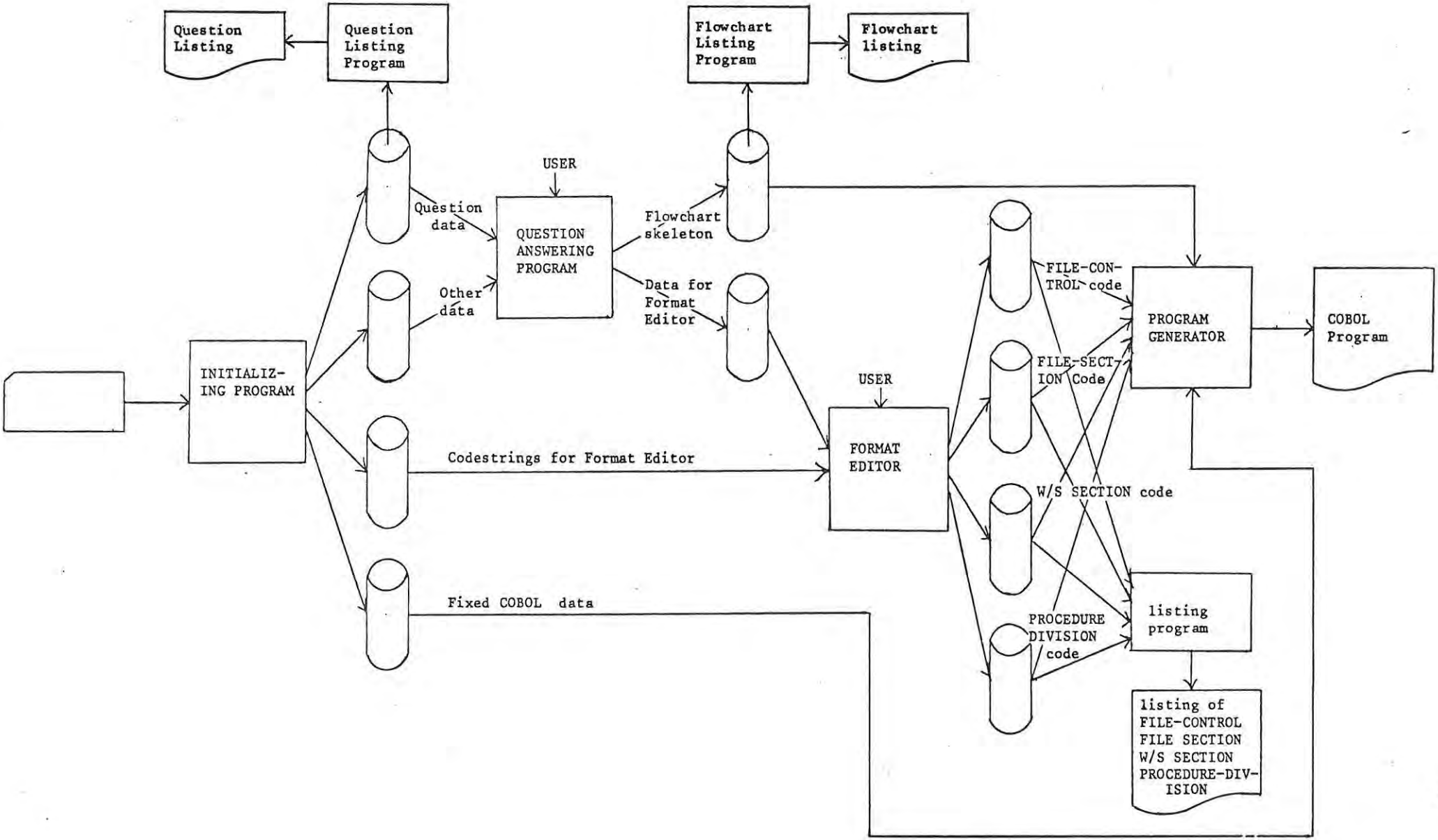


Figure 2.1

STRUCTURE OF THE SYSTEM

3 DEFINITION OF NOTATION AND CLASSIFICATION OF INPUTS AND OUTPUTS

3.1 INTRODUCTION

Before the Question-Answering Program and the Format Editor are discussed in Chapters 4 and 5 respectively, it is necessary to define the notation which is used to formulate part of the data for each question, and to describe the results of investigating the types of inputs and outputs of the final program to be generated.

3.2 NOTATION USED IN FORMULATING QUESTION DATA

The data for each question consists of three components. The first component is the question text and the remaining two components each consist of a sequence of actions and are in effect procedures which are executed by the Question-Answering Program. The purpose of these components and the manner in which they are used in processing are described in Chapter 4, the object of this section being to define the notation which is used to specify the actions comprising these components.

3.2.1 DATA STRUCTURES USED IN THE NOTATION

The types of data structures referred to in the following specification of the notation in which actions are written are mentioned only briefly at this point. This is because the actual data structures which have been implemented in the Question-Answering Program are described in full in section 4.2.

The data structures appearing in the description of the notation are :

(i) constants - a constant may have a character, string, real, logical or integer value.

(ii) variables - these may be of type character, string, real, logical, integer, typebox or pointer. The last two types are used in actions concerned with flowchart generation. A variable name consists of a sequence of alphanumeric characters.

(iii) stacks - a stack is a linked structure, the elements of which consist of pointers to element values or to other linked stacks, or are the actual values themselves. Values may be of different types e.g. integer, real, string etc.

(iv) a flowchart skeleton - this is a linked structure, representing a program skeleton.

(v) flowchart structures - these are linked structures used in flowchart expansion operations described below.

3.2.2 DESCRIPTION OF THE NOTATION [22]

The actions which may be performed are

(i) Push down an item i onto stack S : $i \downarrow S$

(ii) Access current element of stack S, placing the resulting item in i : $S \uparrow i$

(iii) Push down stack T onto stack S as a single element : $T \Downarrow S$

(iv) Access current element of stack S, creating stack T from the resulting element : $S \Uparrow T$

(v) Search a stack S for a particular item i : $\mathcal{S}$(S, i, success action/ fail action)

(vi) Assign a value to a variable, a : $a \Leftarrow i$

(vii) Get element n of stack S : $\mathcal{G}$(n, S, access operation)

(viii) Merge stack S onto stack T (used only when the elements of both stacks are, in turn, stacks) : $\mathcal{M}$(S,T)

(ix) Add stack T onto element n of stack S (used only when element n of stack S is itself a stack) : $\mathcal{A}$(n,S,T)

(x) Replace element n of stack S by stack T, thus increasing the number of elements of S : $\mathcal{R}$(n,S,T)

(xi) Loop n times, performing some action : $\mathcal{L}$(n, action)

(xii) Take each element of stack S in turn, and perform some action : @ (S, access operation, action)

(xiii) Move on to a later question, question n : goto Qn

(xiv) Empty a stack S : $\mathcal{L}$(S)

(xv) Conditional actions : if Boolean expression then action else action fi

(xvi) Accept input until a terminator is read, and process this input : $\mathcal{I}$ (read operation, action)

(xvii) Output a stack S, element by element, and perform some action at each step : $\mathcal{O}$(S, action) (This is used only with stacks whose elements are pointers to string values).

(xviii) Examine the flowchart skeleton to find the first occurrence of any one of a list of flowchart elements (or box types), and then perform some operation on the located box : $\mathcal{F}$(list of boxes, action)

(xix) Scan the flowchart skeleton to find every occurrence of each box specified in a list of flowchart box types, and perform some action each time a box is located : $\forall$ (list of boxes, action)

(xx) Develope the flowchart skeleton by expanding a box pointed to by a pointer variable, box, into a structure or sequence of structures : box → box type. The right hand side of this action has a variety of forms which will be illustrated in section 4.3.4.

(xxi) Expand the flowchart skeleton by inserting one structure or a sequence of structures after a box pointed to by a pointer variable, box : box ← box type. Again, the right hand side of this action has a variety of forms which are illustrated in section 4.3.4.

(xxii) A special action to analyse an input string (this is described in section (4.3.3) :

Analyse(stack, stack, stack, logical variable)

(xxiii) The null action : -

In addition to these actions a number of functions and procedures are used. These are described later on in the present section.

In each of the actions listed above, "action", where it occurs, may be a single one of the listed actions or a series of these actions separated by semi-colons. Actions (i) and (iii) are referred to as stack operations and actions (ii) and (iv) are access operations.

Values which may be pushed down onto a stack using ↓ , may be any variable values or constants. The value of the variable remains unaltered after its value has been stacked. The effect of this operation is depicted in Figure 3.1.

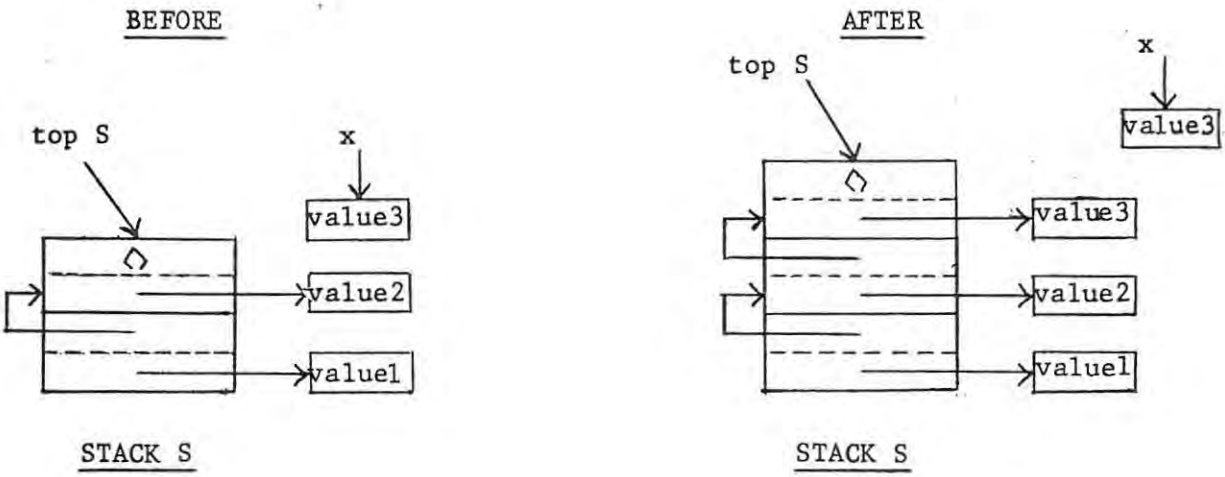


Figure 3.1 : x ↓ S

The stack operation ↓

The operation of accessing an element of a stack, using ↑ , is used only within the action portion of actions such as $\mathcal{G}$ and @, and is not destructive - i.e. the value of a particular stack element is obtained for use, but the stack itself is left unchanged. This operation therefore relies on either $\mathcal{G}$ or @ to set the number of the particular element which is required to be accessed. The value obtained may be allocated to a variable. The operation is illustrated in Figure 3.2.

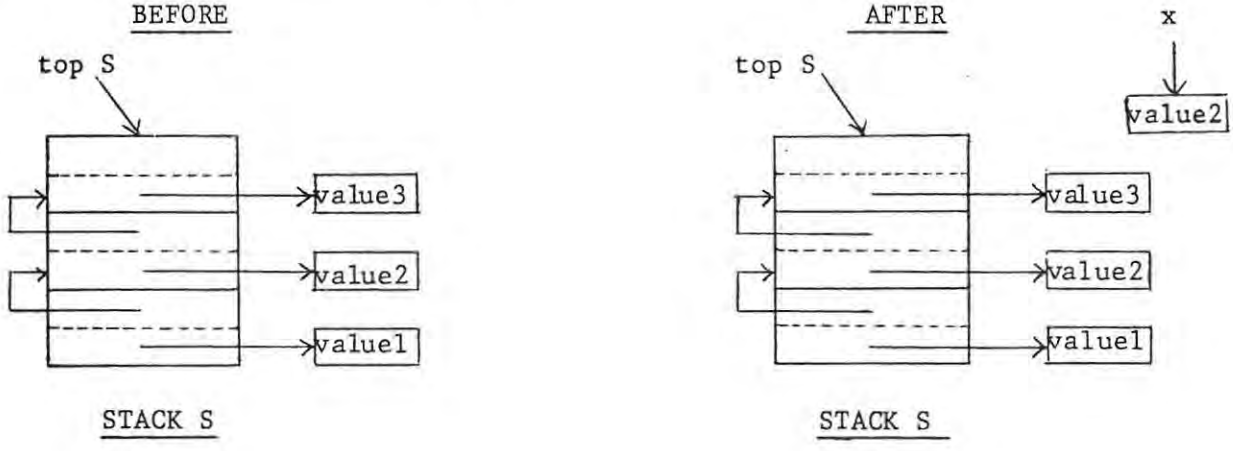


Figure 3.2 :S↑x (where element number has been set to 2)

The access operation ↑

The effect of pushing down a whole stack onto another stack as a single element is shown below, in Figure 3.3. It can be seen from the diagram that the stack which is used to push down another stack

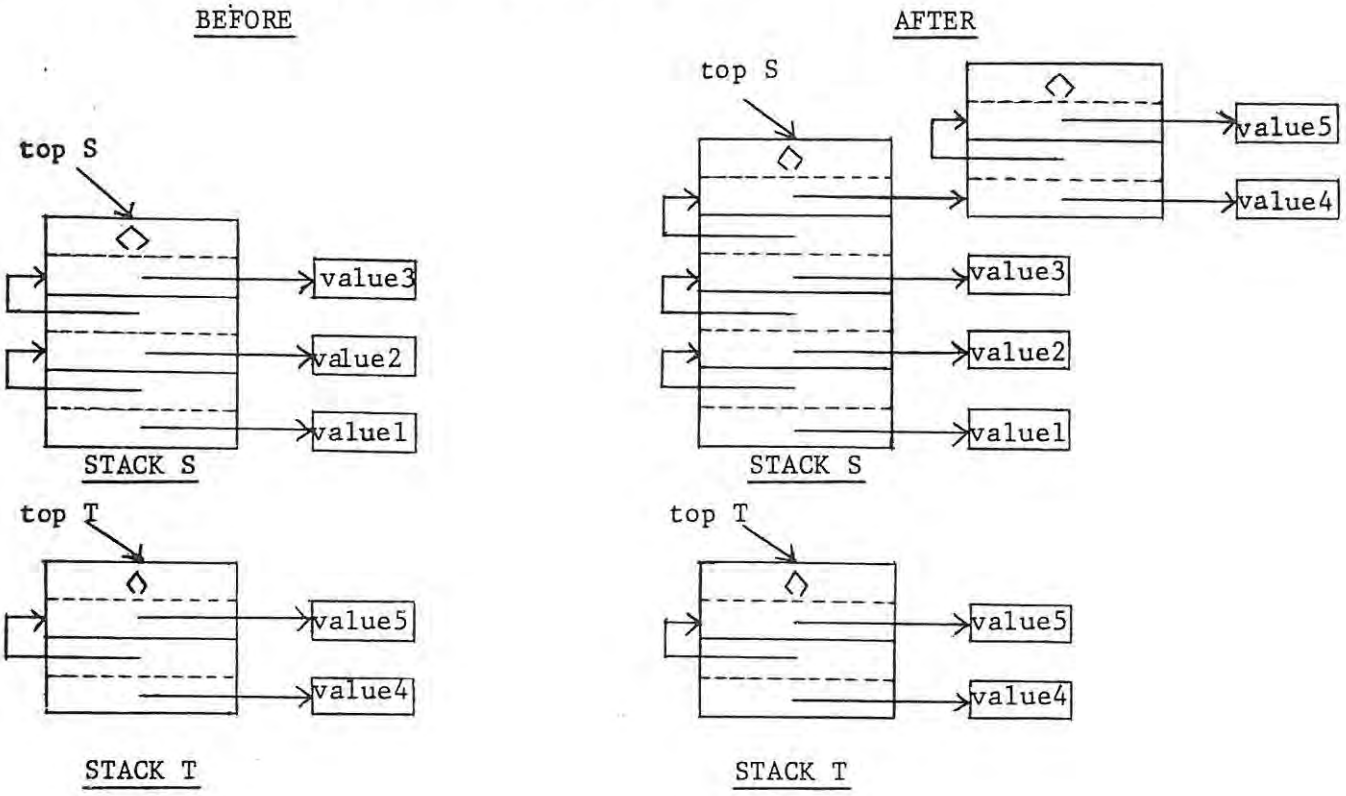


Figure 3.3 : T ⇓ S

The stack operation ⇓

remains unchanged.

The operation of accessing an element of a stack, and obtaining a stack, is, like the ↑ operation, used only within the action portions of the $\mathcal{G}$ and @ actions, as it also requires the number of the particular stack element which is to be obtained to have been set previously. This action is also not destructive as the original stack is left unchanged after an element has been obtained from it to create another stack. The operation is depicted in Figure 3.4.

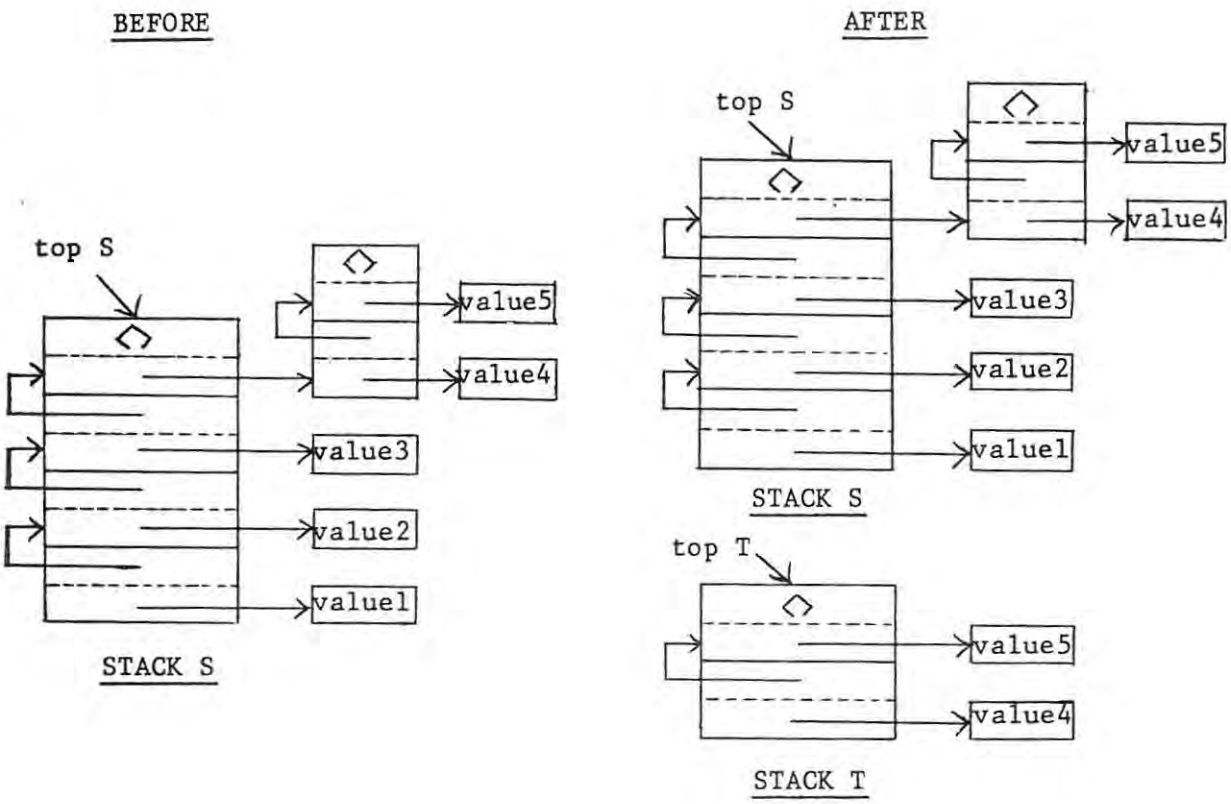


Figure 3.4 : S ⇑ T (where element number has been set to 4)

The unstack operation ⇑

The action $\mathcal{S}$(S,i, success action/fail action) searches stack S from the bottom up until it finds the value i. This action is used only on stacks where none of the elements is in turn a stack. If item i is found, then the action(s) denoted by "success action" are performed. Otherwise, if i is not found, the "fail action" is executed. It has associated with it a reserved variable, searchpointer, which is set to the element number of the required item if it is found.

The assignment action, ⇐ may be used to assign a constant value or a result returned by a function to a variable.

The $\mathcal{y}$ action is used to access, or rather to obtain a copy of a part-

icular element of a stack. It operates by setting the number of the required element and then by performing one of the access operations, ↑ or ⇑ to obtain the particular element.

The merge action, $\mathcal{M}(S,T)$ is used to combine two stacks which have the same number of elements, and, each of whose elements are also stacks. It has the effect of adding each element of stack S onto the corresponding element of stack T. The stack of items comprising an element of stack S is added onto the stack of items constituting the corresponding element of stack T as a number of separate items. This operation is shown in Figure 3.5.

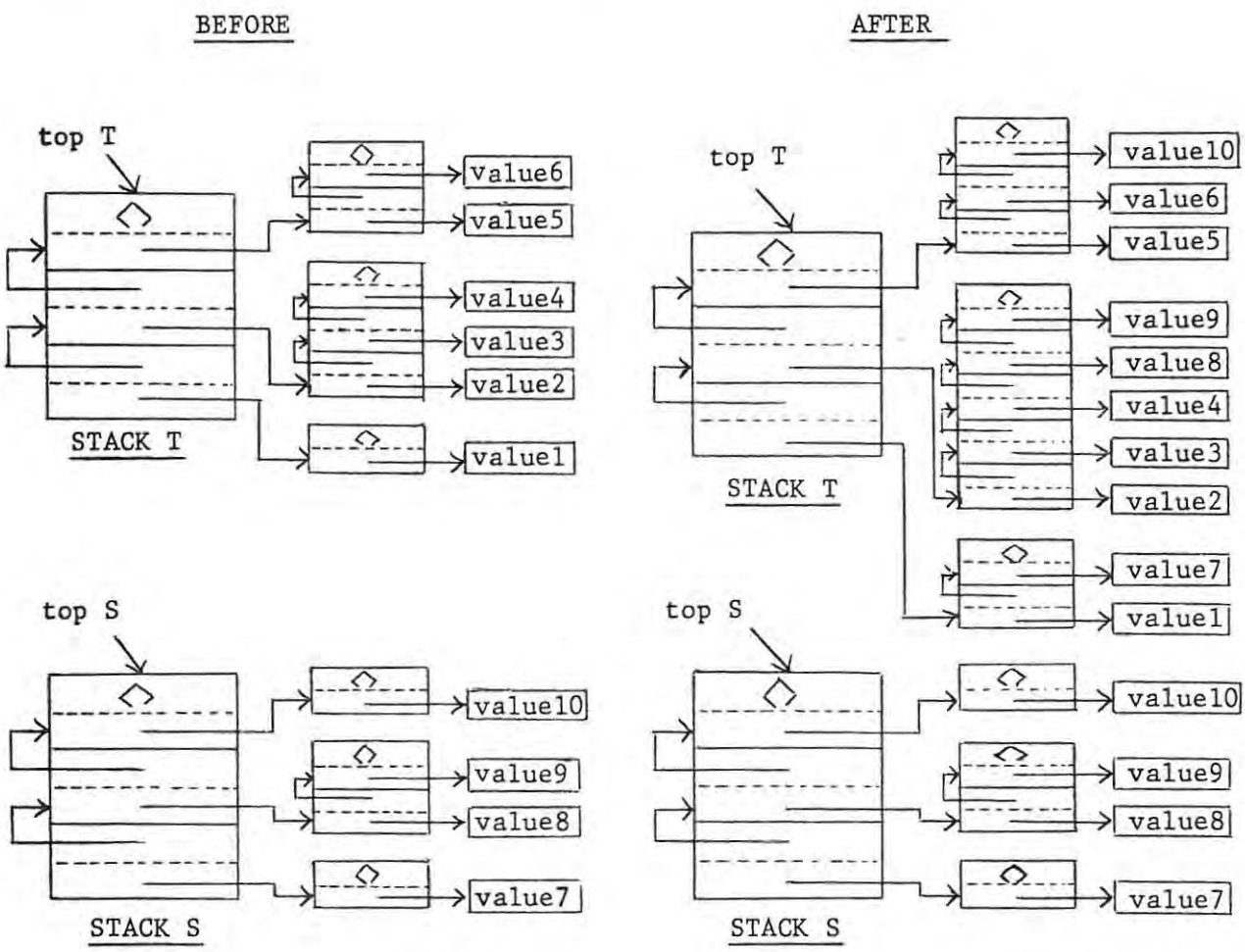


Figure 3.5 : $\mathcal{M}(S,T)$

The Merge Operation

The Add action, $\mathcal{A}(n,S,T)$ has the effect of adding stack T onto the stack pointed to by element n of stack S, as a number of separate elements. The effect of this is shown in Figure 3.6 below.

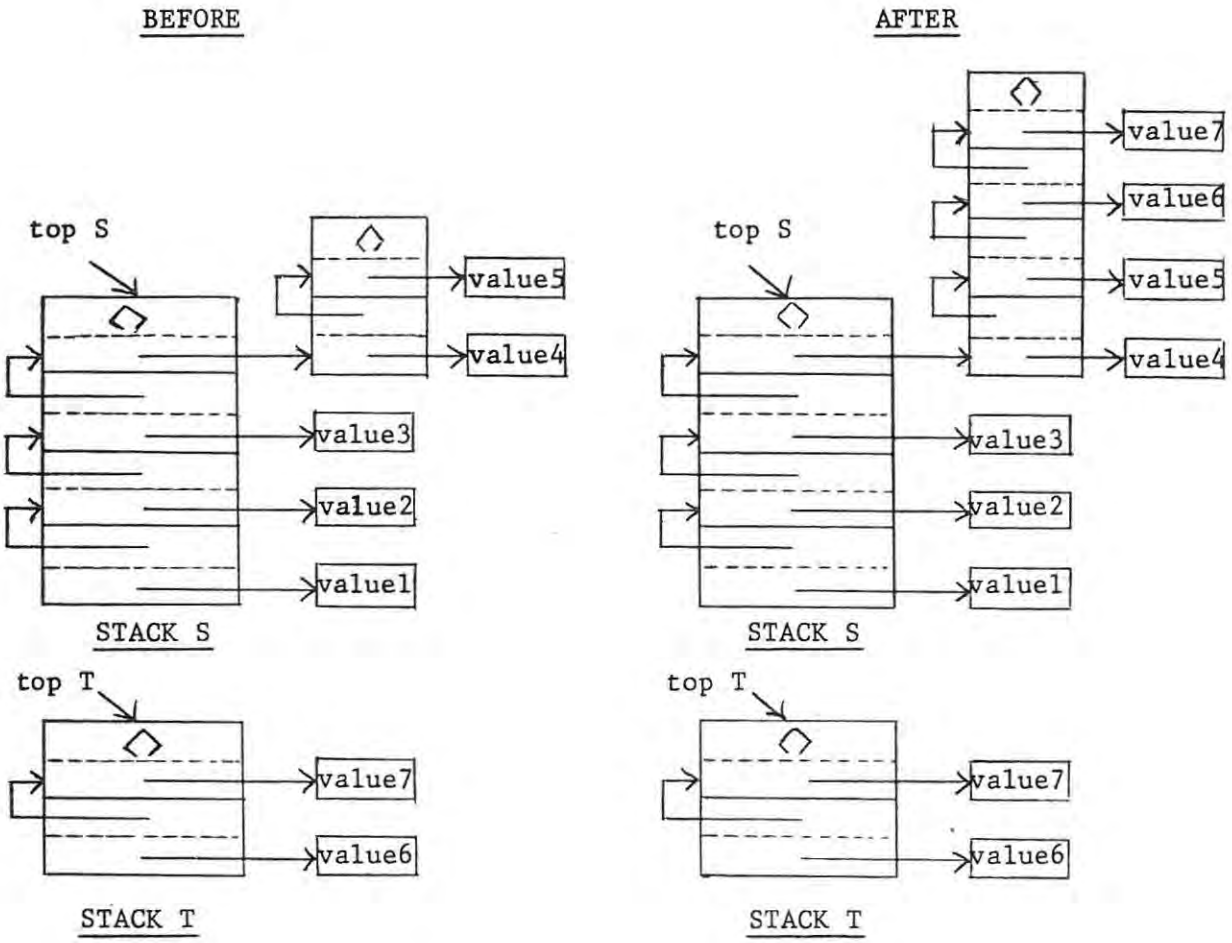


Figure 3.6 A(n,S,T) (where n has been set to 4)

The Add Operation

The Replace action, R(n,S,T) replaces element n of stack S by the stack T, by inserting the contents of stack T, as a number of separate elements, into the position in stack S which was formerly occupied by element n. The result of performing this action is shown in Figure 3.7.

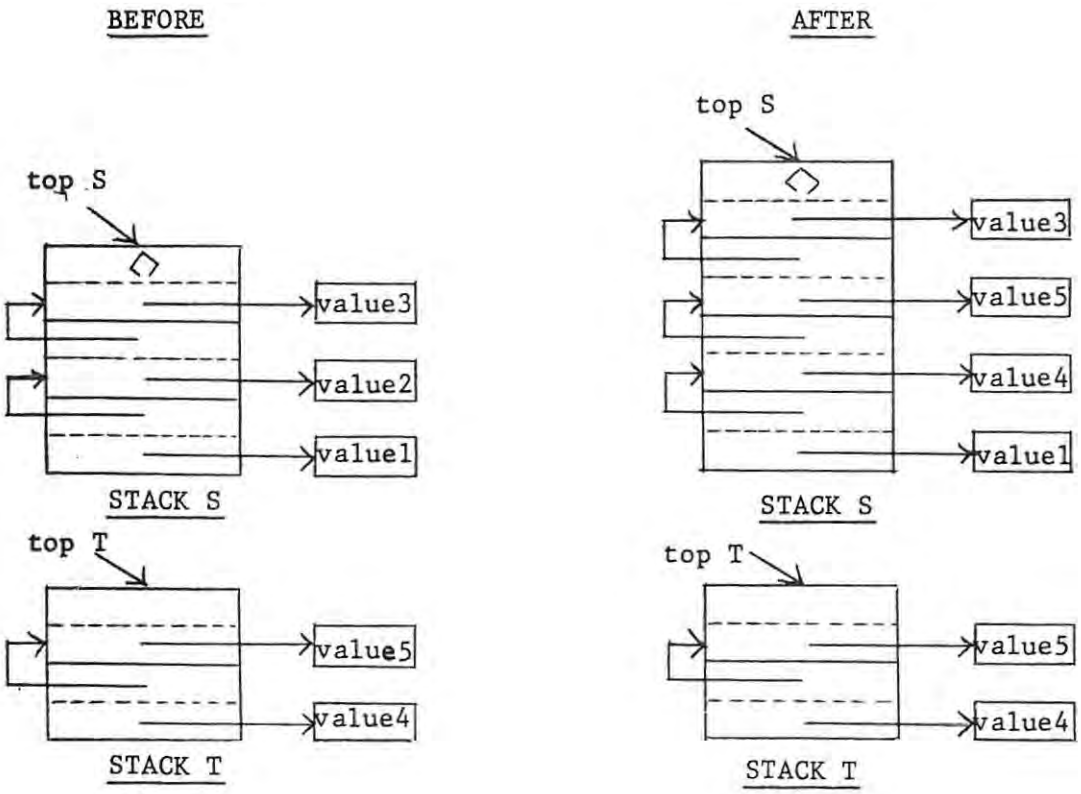


Figure 3.7 $\mathcal{R}$(n,S,T) (where n has been set to 2)
The Replace Operation

The action $\mathcal{L}$, $\mathcal{L}$(n, action) where n is an integer variable or constant, enables a particular action or actions to be performed a specified number (n) times. It has a reserved variable associated with it called loopcount, which contains the iteration number each time the action is performed. The variable loopcount is therefore equivalent to the control variable in a FORTRAN DO or an ALGOL or PASCAL for statement. The value of n is the range, and this action therefore has the same effect as a simple DO or for statement where the control variable starts at one and is incremented by one at each iteration.

The action @ , @ (S, access operation, action) takes each element of a stack in turn and enables some action to be performed with this element.

The elements are not actually removed from the stack which remains unchanged at the end of the process. The access operations used may be ↑ and ⇑. @ has associated with it a reserved variable, foreachcount, which is incremented by one each time an element is accessed and is initially zero.

The action goto, with the form goto Qn causes the Question-Answering Program to stop processing the current question and to scan the question data file until Qn is encountered. This question is then presented to the user for him to answer. Thus this action enables questions to be skipped over and is used to control the path followed through the questions.

As the $\mathcal{y}$, @, ↑ and ⇑ operations do not actually remove items from stacks, an action was introduced which enables a stack to be emptied once there is no further use for its contents. This has the form $\mathcal{b}$(S) and has the effect of reducing a stack, S in this case, to an empty stack.

The next action to be described is the conditional action. It has the form

if condition then action else action fi

The condition may be a Boolean expression containing logical variables and the operators and and or and not (not is used as function as is described later in this section), or it may be a relation comparing variables and/or constants using the operators =, #, >= and <=. The actions after the then and else may be any of the actions described, or may be a series of actions separated by semi-colons.

The actions concerned with accepting user input and with outputting information other than the question text to assist the user in answering questions are $\mathcal{I}$ and $\mathcal{O}$. $\mathcal{I}$, of the form $\mathcal{I}$(read operation, action) is used in conjunction with a function which does the actual reading. The read operation generally consists of assigning the result returned by a read function to a variable. The action will loop and continue performing the read operation and then the action specified until a terminator is detected during the execution of the read function.

The action used to output information on the user's console is $\mathcal{O}$, with the

the form $\mathcal{O}$(S, action). This causes all the elements of stack S to be written out, one by one, on the console. The action specified is performed each time the value of an element is printed. This action is used only with stacks which have elements that are pointers to string values, and not with stacks whose elements point to other stacks. The action performed may be any of the actions listed previously, including $\mathcal{O}$. $\mathcal{O}$ has associated with it two reserved variables, outcount and outpointer. The variable outcount will contain the number of the last element that has been written, and outpointer will point to the value of this element at any stage.

The last actions listed are all responsible for developing the flowchart skeleton of the program to be generated. Each of these actions will be described briefly here.

An action has been defined which enables one particular flowchart element or box, or the first occurrence of one of a list of box types to be "found" or located in the program skeleton. This is the $\mathcal{F}$ action, with the form $\mathcal{F}$(box type list, action). It has associated with it a reserved variable, box, which is set to point to the appropriate flowchart box, once it has been located.

The other action which scans the flowchart skeleton is the $\forall$ action, with the form $\forall$(box type list, action). This enables every single occurrence of each box of the types specified in the box type list to be located, and the specified action is performed each time a box is located. It also sets the reserved variable, box, each time a box of the right type is located.

The actions $\rightarrow$ and $\leftarrow$ are used for expanding a flowchart box, by replacement of the box with one or more copies of a structure or by insertion of one or more copies of a structure after the box concerned, respectively. The left hand side of both of these actions is always a variable which has been set to point to the flowchart box on which the action is to operate. Usually, these actions are used in the action part of $\mathcal{F}$ or $\forall$, and the pointer variable which appears on the left hand side is the variable, box, which has been set by the $\mathcal{F}$ or $\forall$ action.

The right hand sides of both → and ← can have a variety of forms, but a structure type will always be present. This may be followed by a list of parameters, separated by commas and enclosed in brackets. A parameter may be a variable name, a stack name or a parameter indicator of the form %i, where i is a positive integer. The parameter indicators are used to pick up appropriate parameter values from the flowchart box being expanded, and to pass them across to the structure by which the box is being replaced or which is being linked onto the box. If more than one of the parameters are stacks, then these stacks must all have the same number of elements. Another option which exists is that the box type on the right hand side may be preceeded by * , indicating that several copies of the structure are to be used in the flowchart expansion. If the asterisk is preceeded by an integer variable, then, the number of copies used is the value of that variable. Otherwise, there will be as many copies of the structure as there are elements of the stack(s) used as parameters. If an asterisk is used then either the integer variable described must be present, or, at least one of the parameters must be the name of a stack. Examples of how and with what effect these different expansion forms are used appear in section 4.3.4.

Lastly, a number of functions and procedures were defined, to be used in conjunction with these actions. These are mainly concerned with read operations, but perform other tasks as well. All the functions which read in information, will set a terminator for use by the $\int$ action if a predefined terminator character is read.

Briefly, the procedures and functions and their tasks are :

(i) readchar - this reads a single character and sets a terminator if the character read is the terminator character.

(ii) readstring - this reads a string of characters and sets a terminator if the first character read is the terminator character. Spaces may not be present in a string as a space or comma must be used to indicate the end of a string. The character $ may be used within a string to represent a space character.

(iii) readcat - this is used in conjunction with the action Analyse and is described in Appendix 6.

(iv) storea - this has the form storea (stringvar). It stores the contents of the string variable constituting its parameter and returns a pointer to the position at which the string has been stored.

(v) convert - this is used to convert a user's answer to a character string. It returns a pointer to the resulting string. It takes two parameters, the number of a conversion table which maps characters or integers onto strings, and a character or integer variable which contains the user's answer.

(vi) concat - this takes two parameters, a stack name and a string constant. The elements of the stack must all be pointers to string values. The function concat then takes each of the element values of the stack and concatenates them to form one string by inserting the string constant in between the string values. The value returned is a pointer to the single created string. The stack remains unchanged.

(vii) errorsub - when an incorrect answer is detected, this is called to inform the user that his answer is incorrect.

(viii) curr - this function is used to return a pointer to the nth occurrence of a particular type of flowchart box, where n is the current value of the reserved variable, searchpointer. It is thus used in the action part of $\mathcal{S}$. It takes one parameter, a pointer to the first occurrence of a box of the particular type.

(ix) not - this may be used in expressing the condition part of a conditional statement. It takes as a parameter a logical expression consisting of logical variables separated by and and or.

(x) noelmnts - this takes one parameter, a stack name, and returns the number of elements in this stack.

(xi) constring - this function constructs a string from its parameter values and returns a pointer to the constructed string. It takes as parameters string constants and pointers to strings and concatenates them to form a single new string.

(xii) storestring - this stores a string constant and returns a pointer to the position at which the string has been stored. It takes a string constant as a parameter.

(xiii) charstring - this takes one parameter, an integer variable or a real variable. It converts the value of this variable to a string of characters and returns a pointer to the string thus created.

(xiv) writestring - this procedure writes out a character string on the user's console. It takes one parameter, a pointer to the string to be written.

(xv) readno - this function reads a number, returning a real value if a decimal point is encountered during the reading, and an integer value otherwise.

3.3 TYPES OF INPUT AND OUTPUT

The description of notation and definitions which this chapter provides will now be concluded by giving the results of investigating the inputs and outputs of the final program to be generated. These results are of relevance to the formulation of the question data and to the Question-Answering Program (in terms of what information it stores and outputs) as well as to the Format Editor.

The information which acts as input to or output from a payroll program may be classified into three categories:

(i) Information

(ii) Files

(iii) Reports

Each of these categories will now be described in turn, and, in the process, a picture presented of what input/output information has to be obtained and what strategies have been adopted.

3.3.1 INFORMATION

It was assumed that punched cards would be a medium on which data could be input, and information is the name given to data which is read from cards by the generated program. The possible different types of information which may act as input are :

(i) editing information - this information is concerned with editing the contents of the payroll master file.

(ii) periodic information - this information may have to be provided each time the payroll program is run to enable payroll calculations to be performed.

(iii) cancelled information - this provides information concerning cancelled cheques or payslips to enable reconciliation to be performed.

(iv) setup information - this consists of information necessary to set up the records of a disc file.

It clearly depends on the user's answers to the questions which of these types of information are present in a generated program. Information may be verified as it is read, if the user specifies this.

As well as being read from cards, and possibly being verified, information may be involved in some form of processing before being used, and so four information processing codes have been defined. Any one of these is associated with each information name after the appropriate code has been determined from

question answers. The information codes and what they signify are :

1 - implies that information is "left". That is, it is simply read from cards and is not processed prior to use.

2 - indicates that the information is to be stored on a disc file as it is read.

3 - indicates that the information must be sorted.

4 - indicates that the information is to be used to set up a disc file and must therefore be read into a file record.

The way in which the questions concerning information are presented, allows the following processing of and by the different types of information in the generated program. The options present in the final program depend on the user's specifications.

(i) Edit information : This information is read in and may be verified during the reading. The information is used to edit the payroll master file as it is read. Any verification errors cause an edit request to be ignored and an error to be reported and the program to halt at the end of the edit, before performing any payroll calculations.

(ii) Periodic information : This information is read, may be verified while it is being read, and is stored on a file. The file may then be sorted. If there are errors during verification, the payroll program will halt before sorting takes place, or, if no sorting was specified, before the payroll calculations are begun.

(iii) Cancelled Information : This information is read in and may be verified during the reading. If it is required to be sorted, it is stored on a file during the reading, after which this file is sorted and then used for reconciliation. If the information is not required to be sorted, it is used for reconciliation immediately, as it is read. Any errors during verification cause the cancelled information record to be ignored, but the reconciliation continues.

(iv) Setup Information : This information is read in, may be verified during

the reading, and is stored on a file which may then be sorted.

3.3.2 FILES

The next type of inputs and/or outputs to or from the generated payroll program are files. These are considered to be different from files used in the processing of information and mentioned above. Possible files which may be present in the generated programs are :

(i) a payroll master file - the file containing the records of all the employees.

(ii) a reconciliation file - the file on which all unclaimed pay information is stored.

(iii) a setup file - a file on which setup information is stored.

The user's responses to questions concerning necessary files determine which files are present in a generated program.

Depending on what facilities are required in the generated program, each file may have different versions. For example, a file may act as input in one form and may be output in another form, and so, each file has one of seven possible version codes associated with it. The appropriate version code for a file is determined from the user's answers to questions by the Question-Answering Program.

The values and meanings of file version codes are :

1 - indicates that only one version of the file is required.

2 - indicates that an old version of the file is also required.

3 - indicates that a new version of the file is also required.

4 - signifies that both old and new versions of the file are also required.

5 - signifies that a temporary file and a sort file version are also required.

6 - indicates that the file is to be used as a setup file.

7 - the file is to be used as a setup file, and, in addition, a temporary

file and a sort file version are also required.

Both file version codes and information processing codes are determined by the Question-Answering Program and are used by the Format Editor. Their use by the Format Editor is described in sections 5.4.2 and 5.5.2 respectively.

3.3.3 REPORTS

The final type of outputs of the generated program consists of reports. After considering the types of reports generated by payroll-type programs, it was concluded that, for the purpose of designing structures with which to expand the flowchart skeleton to enable reports to be produced, three categories or varieties of reports exist.

These categories are :
(i) Full list reports
(ii) Partial list reports
(iii) Summarized reports

A full list report is a report which lists all the employees, and an example of a report of this type is a payroll register.

A partial list report is a report which contains information concerning only employees who fall into particular categories. An example of this is a list for a bank of all employees whose salary cheques are paid into their accounts at the particular bank. A single cheque of the value of all the applicable employees' salaries would be made out to the bank, and this cheque would be accompanied by the list of applicable employees so that their individual salaries could be paid into their accounts.

A summarized list is really a breakdown of values of quantities over some categories and consists of a listing of totals and subtotals. An example of this might be a breakdown of rent and tax over the departments of an organization. This report would then contain subtotals for rent for each department, and for tax for each department, as well as totals of rent and tax. All these subtotals and totals would be listed under appropriate headings.

Both full list reports and partial list reports may be subdivided into groups, and these groups may be subtotalled. Totals of quantities appearing in reports of these types may also have to be printed at the end of the report. The questions concerning reports are responsible for determining which of these options are required in a particular report.

As report names are established during the questioning of the user, with each report is associated a unique identity number. The use of these identity numbers will become apparent in the following description of what producing reports involves. This description maps out the strategy decided upon for generating the report-printing section of the final program.

The production of a report consists of two stages. The first stage is the preparation of the report, and the second stage is the actual writing of the report. In order to prepare a full list report, at the end of the payroll calculations for each employee, all quantities which appear in a particular report are written away to a file, called the report file, as one record. An item which will be present in the record written is the identity number of the report with which the information is to be associated. Any subtotals and totals of quantities which are to be printed are also incremented. This procedure is followed for each full list report. For a partial list report, the information contained in the report for a particular employee is written to the report file as for a full list report, and any subtotals and totals are incremented only if the employee has been determined to be a member of a category which must appear in the report. The preparation stage for a summarized list report consists only of accumulating quantities into appropriate totals and subtotals, to be printed at a later stage.

The second stage in producing a report consists of actually writing the report. This is done when the whole of the payroll master file has been read and all payroll calculations have been completed. The writing of a report may be preceeded by sorting the quantities to appear in it into some specified order in the case of full and partial list reports. Much of the code necessary to print a report is produced by the Format Editor, and this is described in detail

in Chapter 5.

To write a full or partial list report that does not have to be sorted simply involves scanning the report file, record by record, and, each time a record with the appropriate identity number is encountered, producing a line of the report by using the code produced by the Format Editor. If the report has to be sorted before being written, the generated program calls the supplied COBOL SORT PROCEDURE to extract the records of the report from the report file by using the identity number, and to sort them onto a file. This file can then be read and used record by record to write the particular report, again using the code produced by the Format Editor.

The writing of a summarized report simply involves writing out the totals and subtotals which were accumulated in readiness during the preparation stage. Once more, code generated by the Format Editor is used in the writing of reports of this type.

4 THE QUESTION-ANSWERING PROGRAM

4.1 INTRODUCTION

The Question-Answering Program is responsible for obtaining information necessary to generate a program meeting the requirements of a user, by presenting him with a set of questions. This program uses a table-driven approach, as is used in Computer Assisted Instruction [1,12], in questioning the user.

The table data consists of the set of questions which is set up on a file by the initializing program, ready for use by the Question-Answering Program. Each question is stored in a form which has three components i.e. :

(i) the question text

(ii) the "expected answer" component

(iii) the "action" component

The expected answer and action components each consist of a sequence of actions which are formulated in a manner described in Chapter 3.

The Question-Answering Program uses this information in the following way. The data for a question is read from the file and the question text is then displayed on the user's console. The expected answer component of the question data is then used to accept the user's reply, and, once the reply has been correctly obtained, the action component data is used to interpret the reply and to perform the appropriate action. The action performed may cause the Question-Answering Program to move on to a question stored at a later position on the file, or it may continue to the next question on the file.

4.2 DATA STRUCTURES IMPLEMENTED IN THE QUESTION-ANSWERING PROGRAM

4.2.1 CONSTANTS

Constants of type string, character, integer, logical and real are used as parameters for functions or in assignment or stack or access actions and conditions

4.2.2. VARIABLES

Apart from the reserved variables which are associated with the different actions (cf. section 3.2.2), variables are used in the actions. The types of variables allowed are : string, character, integer, logical, real, typebox and pointer. A typebox variable is used to hold a box type value, and a pointer variable to store a pointer to a box in the flowchart skeleton. Variables were implemented by declaring an array for each type of variable allowed, with size equal to the number of variables of that particular type. The value of a variable of a particular type occupies one element of the appropriate array. The name of a variable may be 1 - 32 alphanumeric characters.

4.2.3. STACKS AND OTHER STRUCTURES

Two kinds of stacks are used. These will be referred to as local stacks and global stacks. The reason for having two types of stacks is that they are used for different purposes. Local stacks are used as temporary working storage space, and hold information for the duration of the processing of only one question. These stacks are automatically cleared at the end of each question, before the next question is presented to the user. Global stacks, on the other hand, are used to store information that is permanent, or that is to be kept during the processing of several questions before the stacks involved may be cleared. Information stored in a global stack may be added to over a number of questions.

Local stacks of type integer and character, which may store integer and character values respectively, have been implemented. Global stacks may be used to store integers or pointers to string values or pointers to other global stacks. A global stack element which is itself a stack may not have pointers to stacks as values of its elements i.e. the depth of "stacking" is a maximum of two.

The stack and access operations which may be performed on local stacks are only $\downarrow$ and $\uparrow$. However, global stacks may be operated on using $\downarrow$, $\uparrow$, $\Downarrow$ and $\Uparrow$

depending on the type of elements they have or must receive.

Local stacks were implemented by declaring a two-dimensional array for each stack type, and using one row of this array to represent each local stack of the appropriate type. In addition, an array was declared for each stack type, to hold pointers to the tops of each of the local stacks. This is illustrated in Figure 4.1. Local stacks are identified by names consisting of 1 - 32 alphanumeric characters.

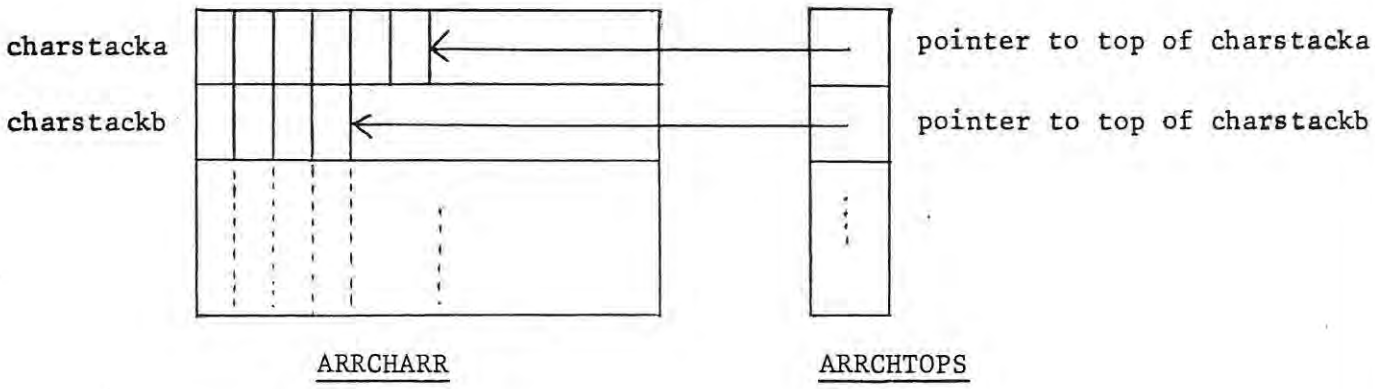


Figure 4.1

Storage of local stacks of a particular type

Global stacks were more complicated to implement because the values of elements can be pointers to further global stacks. The structures used to represent them are shown in Figure 4.2. Global stacks are identified by numbers, and a global stack number is used to reference array GSTACKS. Also shown in Figure 4.2 is the way in which a global stack, number 2, with 3 elements might be stored. Two of the elements are pointers to string values and the third points to another stack whose elements are pointers to two string values.

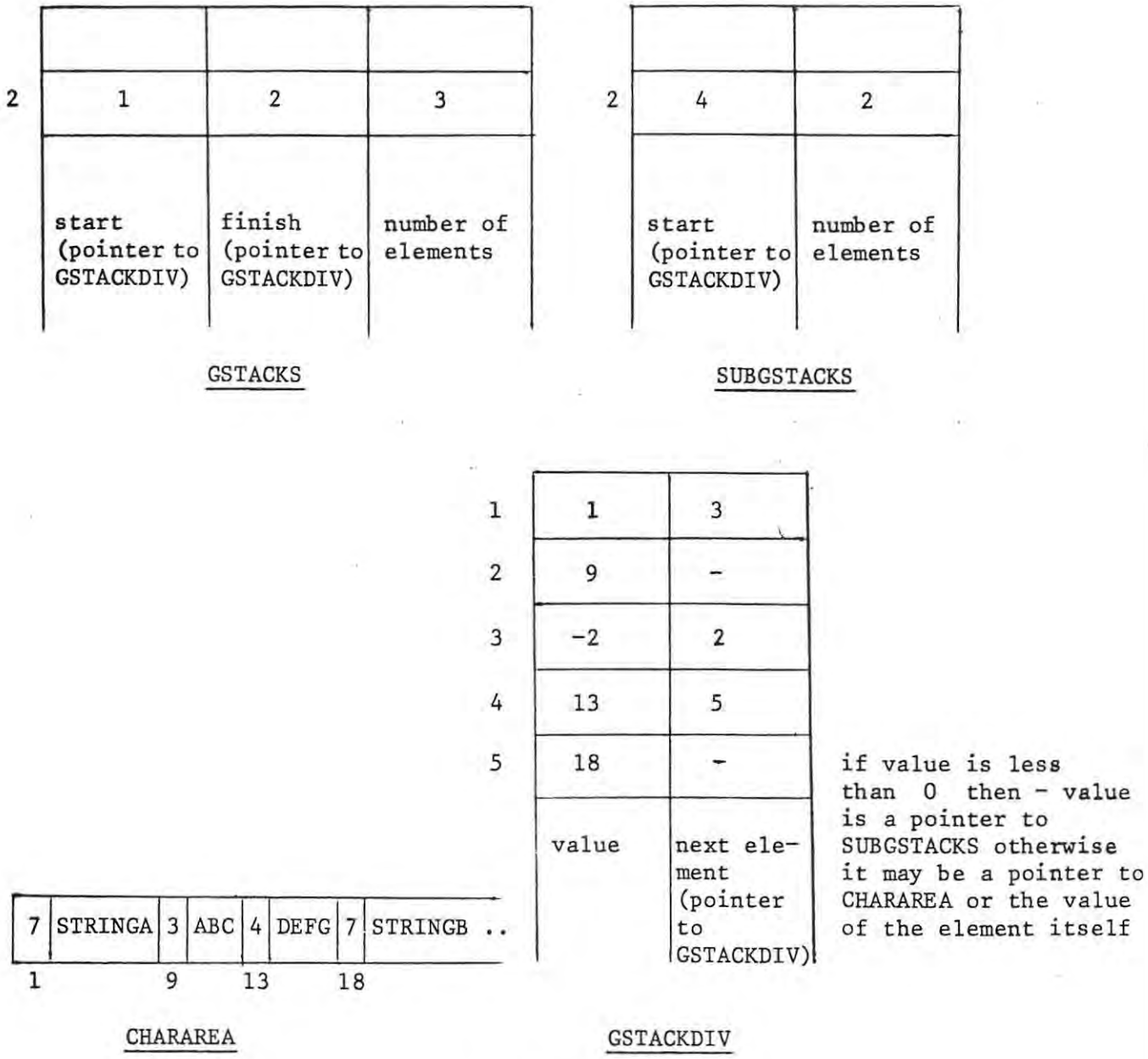


Figure 4.2

Structures representing global stacks

The values of the elements of the global stack stored as shown in Figure 4.2 could be represented as

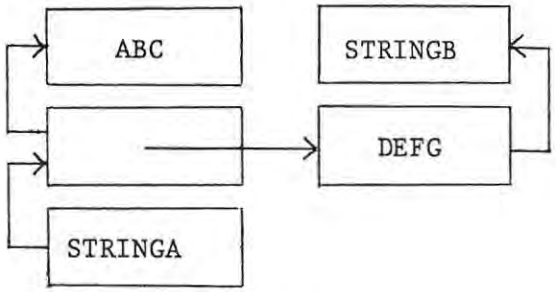


Figure 4.3

The remaining structures necessary to store information are an array in which the flowchart skeleton is stored, and arrays containing information concerning

flowchart structures. The format of these arrays and what information is stored in them is shown in Figure 4.4.

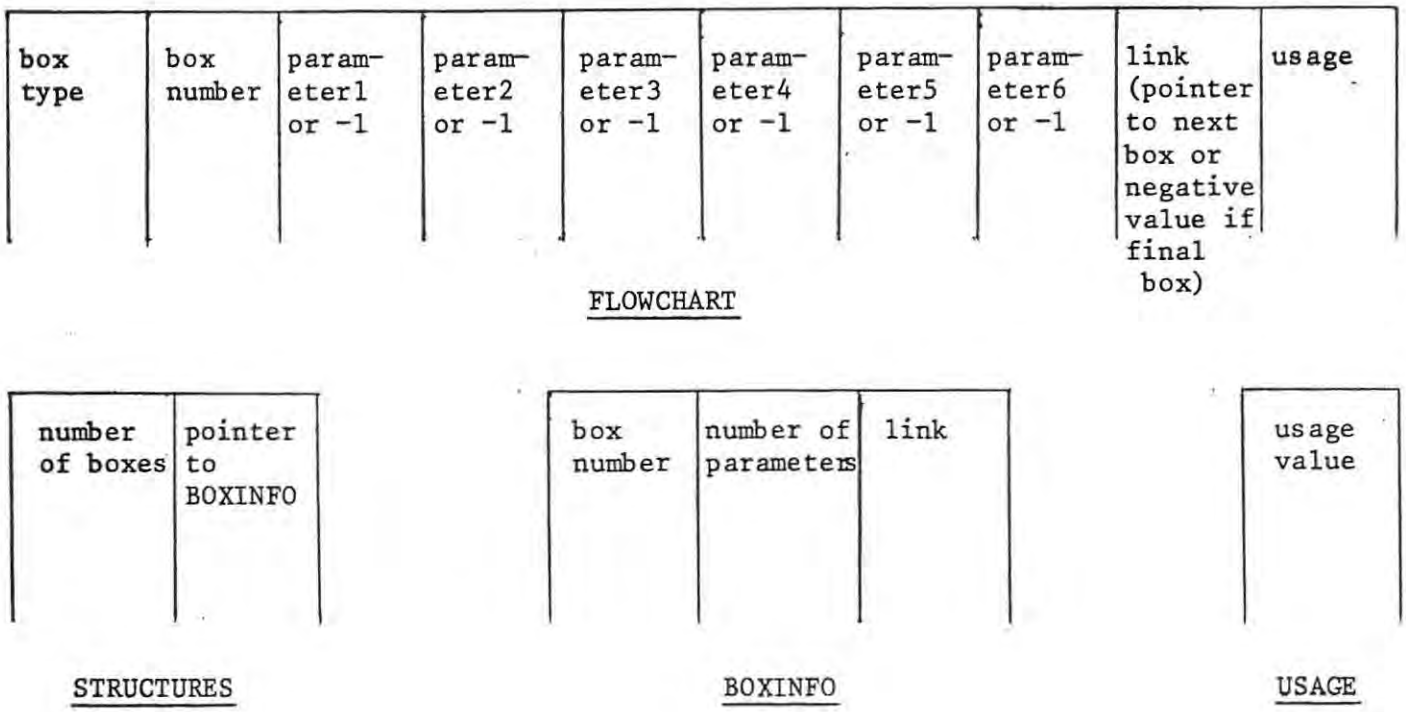


Figure 4.4

Information for each structure is stored in one row of array STRUCTURES and in as many rows of array BOXINFO as there are boxes in the structure. Array USAGE has one element for each type of structure, in which is stored the usage value of that structure. The purpose of usage values will be described in Chapter 5.

4.3 FORMAT AND INTERPRETATION OF THE QUESTION DATA

4.3.1 SETUP PROCESSING

The notation described in section 3.2 was used to formulate the expected answer and action components of each question when the questions were being designed. The actions were subsequently converted by hand to numeric code values, in readiness for interpretation by the Question-Answering Program. The particular code values which are used to represent the different actions and to indicate variable types, etc. were chosen arbitrarily. A full list of these codes is presented in Appendix 1 for reference.

An example of the numeric code equivalent of an action is shown to give a picture of the encoding process. The action :

if x > B then error else x ↓ anstack fi is coded as

63 78 91 X 27 B 64 76 ERROR 65 32 91 X 39 ANSTACK 66

where 63 represents if

78 represents >

91 indicates that a character type variable name is to follow

x is the name of a variable of type character

27 indicates that a character constant is to follow

B is the character constant

64 represents then

76 indicates that a function or procedure name is to follow and must be evaluated

ERROR is the name of the procedure to be executed

65 represents else

32 represents ↓

39 indicates that the name of a local stack of type character is to follow

ANSTACK is the name of the character-type local stack.

and 66 represents fi

In order to facilitate their interpretation, expressions are written in prefix form and then converted to numeric code. This is shown in the above example.

In its final form, before any processing by programs, the data for each question therefore consists of :

(i) A string of characters, starting with the letter Q and terminated by the character * . This constitutes the question text.

(ii) A sequence of numeric code values determining actions and indicating variable and stack types, as well as variable names, procedure names, function names and constant values. This sequence starts with the code

200 which marks the beginning of the expected answer component, and is terminated by the code 201.

(iii) A sequence of numeric code values, variable and other names and constant values comprising the action component. The code 100 is used to indicate the start of the action component, and the code 101 to indicate its end.

One of the three listing programs mentioned in section 2.6 may be used to list out the data on the line printer. It converts the numeric codes to the characters they represent and is useful for verifying the data and for obtaining a complete listing of all the questions in a readable form.

In being presented to the initializing program to be set up on a file for use by the Question-Answering Program, the question data is preceeded by a list of declarations. These are declarations of all the reserved variables as well as all variable, local stack and function names which appear in the question data. The declarations take the form :

type declaration variable name,variable name, ... variable name;

The type declaration part may be : string, chvar, intvar, realvar, logical, pointer, chstack, funvar, nostack and typebox, indicating types string, character, integer, real, logical, pointer, character stack, function or procedure, integer stack and typebox respectively. As the initializing program uses a case statement with the type declarations as labels, chvar, intvar and realvar had to be used as type declarations rather than char, integer and real, as the latter are reserved words in Pascal. Each different type declaration starts on a new line in column 1, and the declaration quesbegin must appear on the line immediately before the first question.

The initializing program reads the declarations and analyses them, to produce a symbol table containing each variable name and an associated unique code value. Variable names which have been declared more than once are listed as errors. The codes for string variables are in the range 300 .. 399, for character variables are in the range 400 .. 499 and so on through integer, logical, .. etc. up to typebox variables which range from 1200 to 1426.

(The codes used to encode actions range between 1 and 99, and the codes 100 and 101 and 200 and 201 are used to mark the beginning and end of the action and expected answer component data respectively). These codes are used to access the appropriate element (in the case of a variable) or row (in the case of a local stack) of the array containing the values of the variable or stack.

After processing the declarations, the initializing program reads the question data and every time a variable, local stack or function or procedure name is encountered, it is looked up in the symbol table and replaced by the appropriate code value in the data. The data is output to a disc file set aside to store the question data. In addition, the initializing program performs any other conversions necessary to convert all expected answer and action component data to integer values, such as converting string and character constants to their numerical equivalents. Thus the expected answer and action components of the question data which is stored on disc consist of a sequence of numeric codes only.

4.3.2 THE QUESTION TEXT

As has been mentioned, the question text data consists of a string of characters, starting with the letter Q and terminated by an asterisk. The data for each question always starts with the number of the question and usually contains a heading, which indicates the subject of the question. The heading is then followed by the body of the question which phrases the question itself. When a question is presented to a user, the question text is listed, character by character on the user's console until the character "*" is encountered. The number of characters to be printed on a line may vary, depending on the console to be used, and so this figure is read as data by the Question-Answering Program.

In order to be able to control the layout of the text on the console screen, the character ↑ has been used to force a new line whenever it is encountered during the listing of the question text. It may be followed by an integer,

which is taken to be the number of character positions by which the new line must be indented. Otherwise the new line will start in the first character position.

4.3.3 THE EXPECTED ANSWER COMPONENT

All the information necessary for a user to be able to answer a particular question may be presented in the question text. In this case, the expected answer component is responsible for inviting the user to type his reply, accepting the answer and possibly checking it and, if it is found to be incorrect, requesting the user to retype his answer.

Alternatively, not all information which the user needs to be able to answer a question may be provided by the question text. The expected answer component may be required to present supplementary information to the user to aid him in his response, or this component may be responsible for extracting the user's replies in a particular order or manner. For example, the expected answer component may have to list out the contents of a global stack, element by element and accept a response after each element is listed.

The types of answers that can be given and the way in which answers may be requested will be described, and then the working of the expected answer component will be illustrated by examples taken from the questions.

Types of Answer

The types of answer that a user may have to type in response to a question are :

1) Character
2) Number
3) String
4) Category

An answer of type character, as its name implies, consists of a single character. An answer of type number may be either an integer or a real number, the latter if it contains a decimal point. A string answer may consist of

1 - 30 characters and may contain any of the letters A - Z and the digits 0 - 9 and the character '-'. As strings may not contain spaces, the character '$' must be used to represent a space as was mentioned in section 3.2.2. A category-type answer will be described in the examples presented later in this section.

For answers of types 1 - 3, the answer required for a particular question may be a single answer - i.e. a single character or number or string. Alternatively, a list of characters or of numbers or of strings may be required. What is needed will be evident from the question, but there is no danger of a user providing only a single answer where a list should have been given, as further answers will be requested. In order to provide a list of answers of a particular type, the user can type in a series of items (i.e. characters or numbers or strings), each separated by a comma and terminated by typing a terminator after the last comma. Alternatively, he can type in one item and wait to be prompted before typing the next item, and, when he wishes to end his answers, type the terminator in response to a request for an answer. The terminator which is used can be set to whatever value is desired as it is read as part of the data by the Question-Answering Program. The character "%" is the value currently provided in the data.

Manner of Requesting Answers

There are a variety of different ways in which the user can be invited to provide answers to questions.

The question may be phrased so that a simple Y or N (for yes or no) should be provided at the end of the listing of the question text, or a global stack might be listed, and a Y or N might have to be typed next to each element of the stack as it is listed, to provide some information about each element. Alternatively, a list of options may be presented either in the question text listing, or by the listing of a stack by the expected answer component. Such a list will always be numbered or have letters identifying each element, and the user must indicate his selection(s) from the provided list by typing in one or

more of the identifying characters or numbers.

The examples which follow illustrate the functioning of the expected answer component and provide an idea of the different ways in which answers may be obtained from the user and where the different types of answer are required. The questions are taken from the set of questions which is presented to the user by the Question-Answering Program.

Examples of Expected Answers

Selected questions have been taken and are presented here and in Appendix 6 to illustrate various facets of the expected answer components of questions. Only the question text and the expected answer components are shown.

Q1 TYPE OF PROGRAM

IS THIS SYSTEM TO BE :

A) A PERIODIC PAYROLL SYSTEM

B) A SPECIAL-PURPOSE OR ONE-SHOT PROGRAM?

This question provides an example of the user's answer having to be chosen from a list contained in the question text. Only one answer is expected - the letter A or the letter B.

The expected answer component is simply responsible for accepting and checking the user's answer, and has the form :

```
EXPANS
x ⇐ readchar
if  x > B  then error else x ↓ anstack fi
```

The answer is read, checked to see if it exceeds the letter B, and, if it is found to be correct is stacked on the answer stack for futher use.

The next example also shows how an answer must be selected from a list provided by the question text, but in this case a list of characters may be given.

Q9 WHAT ADDITIONS

WHAT ADDITIONS ARE THERE TO BASIC PAY/STANDARD SALARY?

A) OVERTIME

B) ANNUAL BONUS

C) HOLIDAY BONUS

D) OTHER BONUS

E) COMMISSION

F) ABOVE STANDARD SALARY

G) HOSTEL ALLOWANCE

H) OUTSIDE GRANTS

I) ENTERTAINMENT ALLOWANCE

J) TRAVELLING ALLOWANCE

K) UNIVERSITY ALLOWANCE

L) SUBSISTENCE ALLOWANCE

M) BURSARY

N) NONE

O) OTHER?

EXPANS

$\jmath$ (x ⇐ readchar, if x > 0 then error fi; x ↓ anstack; if x # N then convert(C1,x) ↓ G11 fi)

The user must choose which options are applicable and then type his selection, either as a list or one by one as he is prompted. The $\jmath$ operation causes items to be read until the terminator is detected. If a character greater than the letter O is given as an answer by the user, it will be rejected and he will be invited to retype his answer. A correct reply is added to the answer stack, anstack and is also converted using the function convert, and the result returned by this function is stored in global stack 11.

This is an appropriate point to describe the use and working of the function convert. It can be seen that this function takes two parameters. The

first parameter is the number of a conversion table to be used, and the second parameter is the item that is to be converted, using the conversion table.

The data structures used by the function convert are shown in Figure 4.5 below.

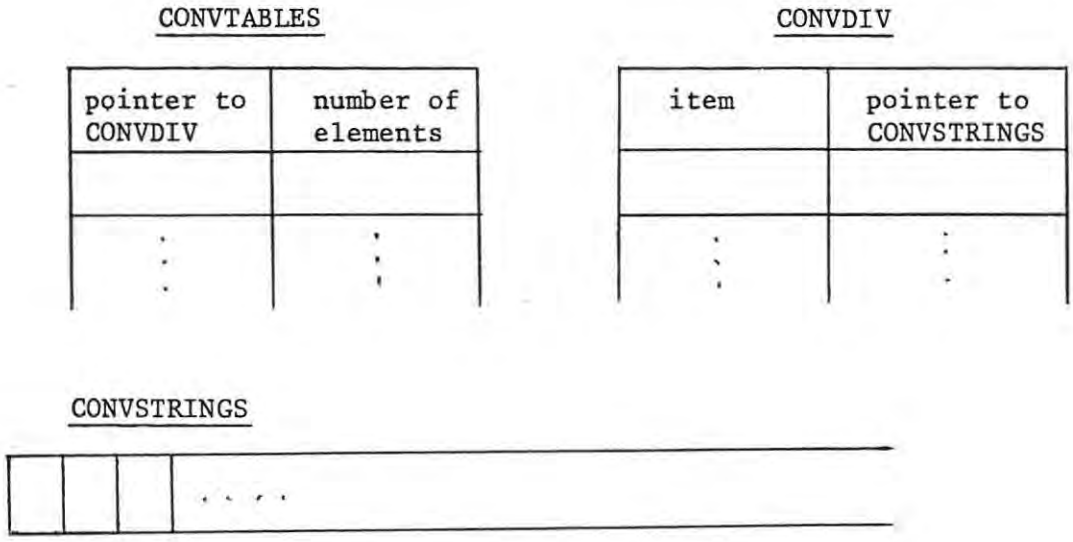


Figure 4.5

Data structures used by the function convert

There are nine conversion tables identified by the numbers 1 to 9. These tables map characters or numbers onto character strings. For example, conversion table 1, which is referenced in the question under consideration, maps the letters A to O onto the data or field names OVERTIME, ANNUAL-BONUS etc. The conversion table number is used by the function convert to access the correct row of CONVTABLES. The first column of CONVDIV is then scanned, from the row pointed to by the pointer in CONVTABLES until all the elements of the conversion table have been examined or until the value of an element in this column has the same value as the item to be converted i.e. as parameter 2. If no value equal to that of the second parameter is found, an error is flagged. Once the row of CONDIV holding the value equal to that of parameter 2 has been found, the string in CONVSTRINGS pointed to by the second element in that row is accessed and transferred to the character area. The value returned by the

function is a pointer to this string in the character area.

Q10 OTHER ADDITIONS

WHAT OTHER ADDITIONS ARE THERE?

EXPANS

$\int$ (z ⇐ readstring, storea(z) ↓ G12)

The user must type a list of strings, terminated by the terminator, % . Each string is read into a string variable z. The function storea, which takes a string variable as a parameter, transfers the contents of the string variable to the character area, CHARAREA, and returns a pointer to the position of this string in the character area.

The following question was chosen to illustrate the situation where the contents of a global stack are listed, element by element and an answer list must be given next to each of these elements. The question text provides a list of options from which the user may select his answers.

Q16 FREQUENCY OF ADDITIONS

AN ADDITION MAY BE CALCULATED IF

A) IT IS A WEEKEND

B) IT IS A FORTNIGHTEND

C) IT IS A MONTHEND

D) IT IS A QUARTEREND

E) IT IS A HALFYEAREND

F) IT IS A TAXYEAREND

G) IT IS A YEAREND

H) IT IS A FINANCIALYEAREND

I) IF A SPORADIC INDICATOR HAS BEEN SET

J) EVERY TIME THE PAYROLL PROGRAM IS RUN.

THE ADDITIONS ARE LISTED BELOW, USE THE ABOVE LIST TO INDICATE NEXT TO EACH ITEM WHICH FREQUENCIES ARE APPLICABLE. IF J IS SPECIFIED, DO NOT SELECT ANY OF THE OTHER OPTIONS AS WELL.

EXPANS

$\mathcal{O}$ (G15, $\mathcal{J}$ (x ⇐ readchar, if x > J then error fi;
if x # J then convert (C4,x) ↓ G13 fi);
if no (G13) # 0 then concat (G13,AND) ↓ G14; $\mathcal{b}$ (G13) else NULL ↓ G14 fi)

The global stack 15 contains pointers to the names of all the additions which are calculated with the same frequencies for all employees. These are listed one by one by the $\mathcal{O}$ action. After each addition name has been printed, a list of characters is read by the action $\mathcal{J}$. Each character is checked and, if it is not equal to the letter J, is converted into an appropriate string using conversion table 4 and the resulting pointer is stacked onto global stack 13. If J is not selected, then after the whole answer list for an addition has been read and converted, all the strings pointed to by the elements of global stack 13 are concatenated into a single string by the function concat (cf. section 3.2.2). The pointer to the resulting string is placed in global stack 14. Global stack 13 is then cleared, ready to receive answers for the next addition. If J were selected, a null value is placed in global stack 14. Thus, assuming that global stack 14 was initially empty, it will have the same number of elements as global stack 15 at the end of executing this expected answer, and information in corresponding elements of these stacks will be related. The strings produced in global stack 14 will be statements of the form FREQUENCY (1) IS NOT EQUAL TO 1, FREQUENCY (2) IS NOT EQUAL TO 1, etc. where FREQUENCY (1) refers to a weekend indicator, FREQUENCY (2) to a fortnightend indicator etc. The contents of global stack 14 will later be used in conditions to test whether frequency flags are set or not, and to branch past an addition calculation if the appropriate flag has not been set, or to eliminate frequency flag tests (in the case of null-valued elements). Further examples of expected answer components may be found in Appendix 6.

The examples which have been considered should provide a reasonable picture of the variety of ways in which answers can be elicited from the user, and thus should give some indication of the usefulness and flexibility of the notation

which was used for expressing the actions of the expected answer component of each question.

4.3.4. THE ACTION COMPONENT

The main tasks of the action components of the questions are to expand the flowchart skeleton and to select the next question which is to be presented to the user. This component may also be responsible for the storing of further information.

These tasks are not necessarily confined to the action component. In some instances, where it is simplerand more efficient, the flowchart expansion may be performed by the expected answer component. The next question to be processed may also be chosen during the execution of the expected answer component. However, in general, the action component is responsible for these functions.

When the flowchart expansion actions → and ← were listed in section 3.2.2., it was stated that they would be illustrated in this section. As was done in the previous section, selected questions from the set of questions will be presented and discussed in turn. This is not only to illustrate the actions → and ← , by giving examples of their different permissible forms, but also to demonstrate the strategy of flowchart expansion and how a path is chosen through the questions. Some of the questions which will be discussed here are the same as those discussed in the previous section.

It should be remembered that a "top down" approach was followed for developing the flowchart and this should be brought out in the following examples and descriptions. For a complete list of all the flowchart structures used in the flowchart development, as well as the initial program skeleton, consult appendix 2.

Once again, consider question 9 :

Q9 WHAT ADDITIONS

WHAT ADDITIONS ARE THERE TO BASIC PAY/STANDARD SALARY?

A) OVERTIME

B) ANNUAL BONUS

C) HOLIDAY BONUS

.
.
.

N) NONE

O) OTHER?

EXPANS

$\mathcal{J}$(x ⇐ readchar, if x > 0 then error fi; x ↓ anstack;
if x # N then convert (C1,x) ↓ G11 fi)

ACTION

$\mathcal{S}$(anstack, N, no-additions ⇐ true; $\mathcal{F}$(B13, box → LAMBDA);
$\mathcal{F}$(B14,box →*AD); goto Q13/ -)
$\mathcal{F}$(B13, cptr ⇐ box, box →* AE (G11))
$\mathcal{S}$(anstack, A, overtime-pointer ⇐ curr(cptr);
overptr ⇐ searchpointer/ -)
$\mathcal{S}$(anstack, 0, other-addition-pointer ⇐ curr(cptr);
otherptr ⇐ searchpointer; goto Q10 / -)
if overtime-pointer # 0 then goto Q11 else @(G11, G11 ↑ y, y ↓ G1);
goto Q13 fi

A previous expansion of the flowchart will have introduced the structure B into the flowchart skeleton. Boxes B13 and B14 and structures AD and AE may be represented diagramatically as shown in Figure 4.6.

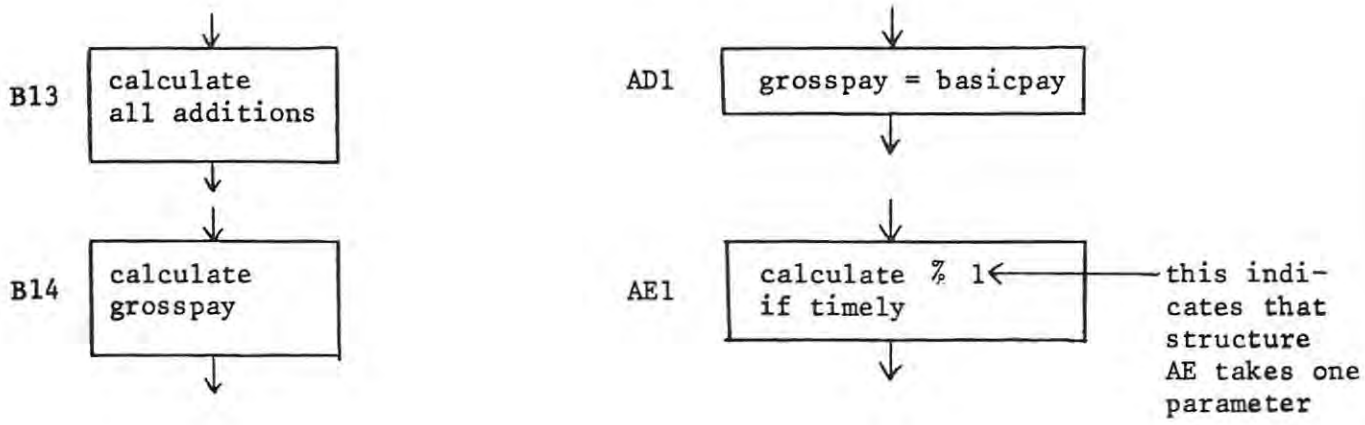


Figure 4.6

The expected answer component sets up local stack anstack and global stack 11 to contain the user's answers and their string equivalents respectively.

The action component searches the answer stack to see whether no additions are required to be calculated. If this is the case, box B13 is located in the flowchart skeleton by the ℱ action which sets a pointer to it. The → action in the action part of ℱ causes B13 to be removed from the flowchart skeleton by altering the link of the box which points to B13 to have the value of the link of B13 itself. Box B14 is then located by another ℱ action and replaced by structure AD which has thus refined box B14 and determined the calculation to be performed.

Should some addition names have been selected, box B13 is located by a ℱ, the pointer variable cptr is set to point to its position and it is then expanded into as many structures of type AE as there are elements of global stack 11. The effect of the action B13 →* AE(G11) may be diagramatically represented as depicted in Figure 4.7, (assuming A, B and O were selected):

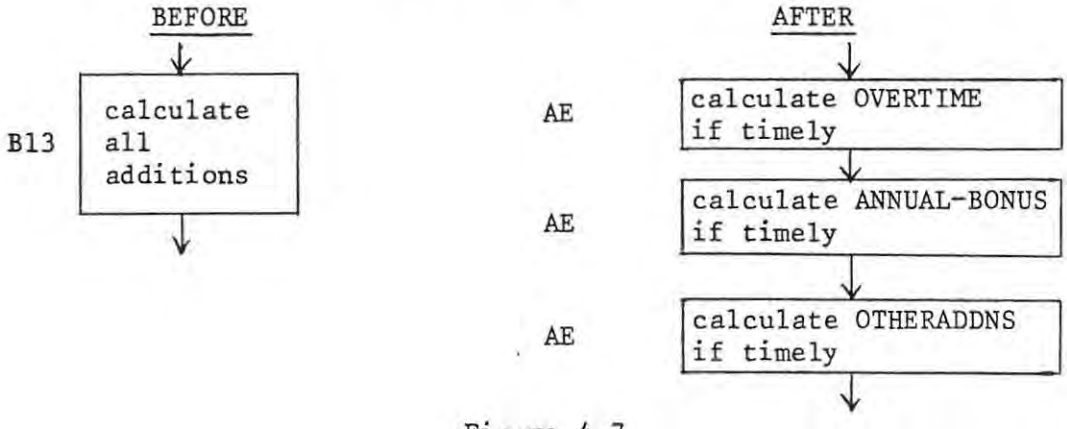


Figure 4.7

Each time a type of structure is used in a flowchart expansion, the usage number associated with the structure type is incremented by 1. If overtime was selected as an addition, the next action sets the pointer variable overtime-pointer to point to the flowchart box which is concerned with calculating overtime, and the integer variable overptr to the number of the element of global stack 11 which points to the addition OVERTIME. Similarly, if there are to be other additions, two variables, other-addition-pointer and otherptr respectively, are set and also Q10 is selected as the next question to be processed and is branched to as it obtains the names of these additions. If there are no other additions but overtime must be further investigated a branch is made to Q11, otherwise the contents of global stack 11 are stored in global stack 1 (the global stack in which fieldnames are accumulated) and Q13 is then presented.

Question 10 will be looked at next to show the further development of the flowchart and global stacks.

Q10 OTHER ADDITIONS

WHAT OTHER ADDITIONS ARE THERE?

EXPANS

$\mathcal{J}$ (z ⇐ readstring, storea(z) ↓ G12)

ACTION

$\mathcal{R}$(otherptr, G11, G12)

other-addition-pointer →* AE (G12)

$\mathcal{B}$ (G12)

if overtime-pointer # 0 then goto Q11 else @(G12,G12 ↑ y,y ↓ G1);

goto Q13 fi

If the user specifies that there are additions other than the options presented in Q9 then Q10 is accessed and presented. This question requests the other addition names and stores them in the character area, stacking the resulting pointers in global stack 12. At this stage global stack 11 will contain the addition names and otherptr will have been set to the number of the element of

this stack which points to the name other-additions. This pointer is used to replace this element by the elements of global stack 12. Similarly other-addition-pointer will have been set to point to the flowchart box concerned with other additions, and is therefore used to expand this box into as many structures of type AE as there are elements of global stack 12. Global stack 12 has no further use and its contents are then cleared. Thereafter, the next question to be dealt with is selected in much the same way as the choice is made at the end of Q10 and is then accessed. If no further addition names are to be obtained the contents of global stack 12 are stacked onto global stack 1 as fieldnames.

The condition of the flowchart at the end of this question (assuming two further addition names were provided) is shown in Figure 4.8. Global stack 11 will contain pointers to

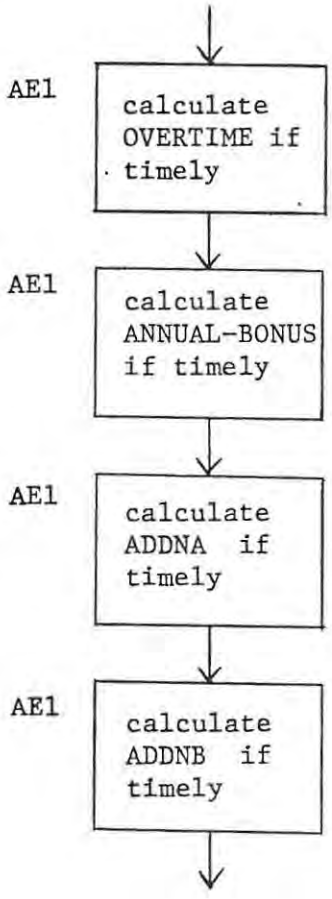


Figure 4.8

all the addition names obtained at this stage. The addition OVERTIME would be the subject of the next question presented if the user's answers were as assumed.

Question 16 is also reconsidered, as, not only does it carry the line of development of the flowchart further, but it also illustrates the use of a %i type of parameter in a → action, as well as showing the ∀ action.
Assume that the addition OVERTIME has been replaced by TIME-AND-A-HALF and DOUBLE-TIME by intervening questions, and the flowchart has been expanded, in the manner of Q10. Assume further that all the additions are calculated with the same frequency for all employees, as in this case global stack 15 will contain pointers to the names of all the additions.

Q16 FREQUENCY OF ADDITIONS

AN ADDITION MAY BE CALCULATED IF

A) IT IS A WEEKEND

B) IT IS A FORTNIGHTEND

C) IT IS A MONTHEND

.
.
.

J) EVERY TIME THE PAYROLL PROGRAM IS RUN .

THE ADDITIONS ARE LISTED BELOW. USE THE ABOVE LIST TO INDICATE NEXT TO EACH ITEM WHICH FREQUENCIES ARE APPLICABLE. IF J IS SPECIFIED DO NOT SELECT ANY OF THE OTHER OPTIONS AS WELL.

EXPANS

```
θ (G15, ℊ(x ⇐ readchar, if x > J then error fi; if x # J
then convert (C4,x) ↓ G13 fi); if no (G13) # 0 then concat (G13,AND) ↓ G14;
      ℓ(G13)  else  NULL ↓ G(14) fi )
```

ACTION

```
∀(AE, box → AI(G14, %1, %1))
∀(AI1, ℊ(forallcount,G14, G14 ↑ y); if y = NULL then box → λ fi)
ℓ(G14)
```

The effect of the expected answer component has been discussed and it need only be reiterated that it produces in global stack 14 the pointers to a number of strings, or null values to be associated with each element of global stack 15.

Assume that the relevant part of the flowchart skeleton and the structure AI have the forms depicted in Figure 4.9.

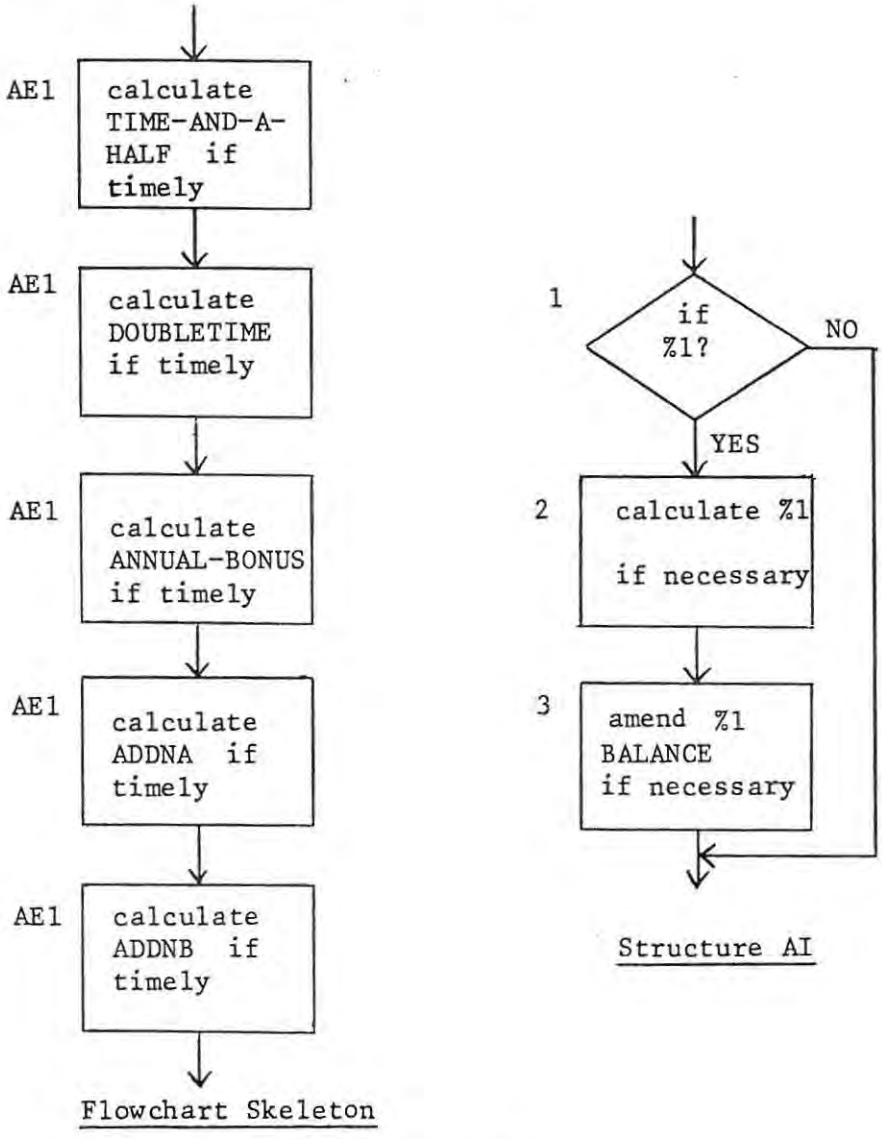


Figure 4.9

Then the effect of the action will be to expand the flowchart to the form displayed in Figure 4.10.

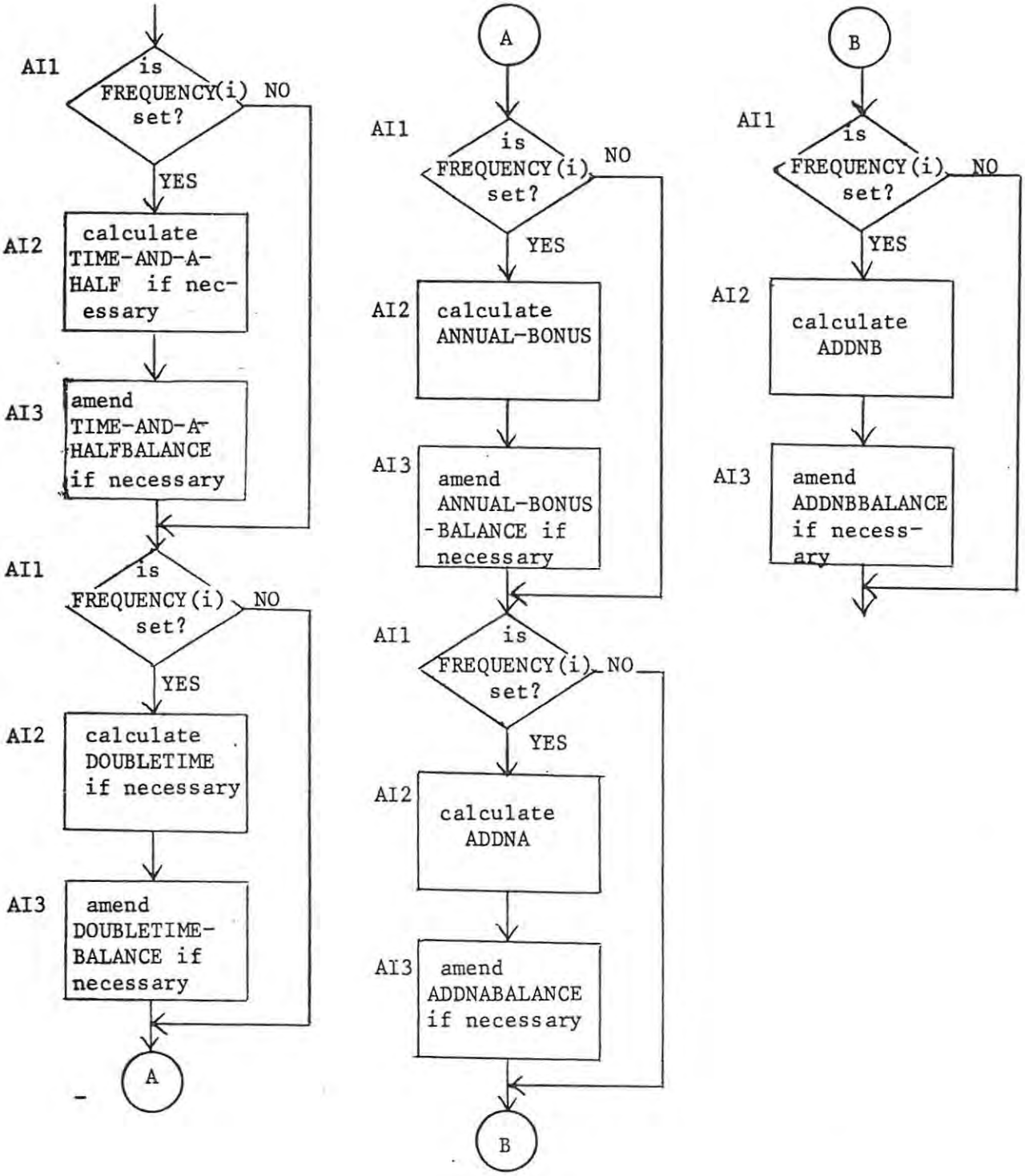

AI1
is FREQUENCY(i) set?
NO
YES
AI2
calculate TIME-AND-A-HALF if necessary
AI3
amend TIME-AND-A-HALFBALANCE if necessary
AI1
is FREQUENCY(i) set?
NO
YES
AI2
calculate DOUBLETIME if necessary
AI3
amend DOUBLETIME-BALANCE if necessary
A
A
AI1
is FREQUENCY(i) set?
NO
YES
AI2
calculate ANNUAL-BONUS
AI3
amend ANNUAL-BONUS-BALANCE if necessary
AI1
is FREQUENCY(i) set?
NO
YES
AI2
calculate ADDNA
AI3
amend ADDNABALANCE if necessary
B
B
AI1
is FREQUENCY(i) set?
NO
YES
AI2
calculate ADDNB
AI3
amend ADDNBBALANCE if necessary

Figure 4.10

Considering the following question further developes the flowchart and determines two categories of additions :

Q17 TO WHOM ARE ADDITIONS APPLICABLE

EACH ADDITION MAY BE APPLICABLE TO ALL EMPLOYEES OR IT MAY BE RESTRICTED TO CERTAIN TYPES OF EMPLOYEE. THE ADDITIONS ARE LISTED BELOW, NEXT TO EACH TYPE Y IF IT IS APPLICABLE TO ALL EMPLOYEES AND N OTHERWISE.

EXPANS

θ (G11, x $\Leftarrow$ readchar, if x # Y and x # N then error fi;
x $\downarrow$ anstack; if x = N then outpointer $\downarrow$ G13 fi)

ACTION

$\forall$ (AI2 & EE2, if ans = Y then box $\rightarrow$ AJ(%1) else box $\rightarrow$ AK(%1) fi)

δ (anstack, N, -/ if tax then goto Q21 else if pension then goto Q22
else goto Q23 fi fi)

The expected answer component should be self-explanatory at this point. Assume that the flowchart has the form depicted in Figure 4.10 at the end of discussing question 16 and that structures AJ and AK may be depicted as shown in Figure 4.11.

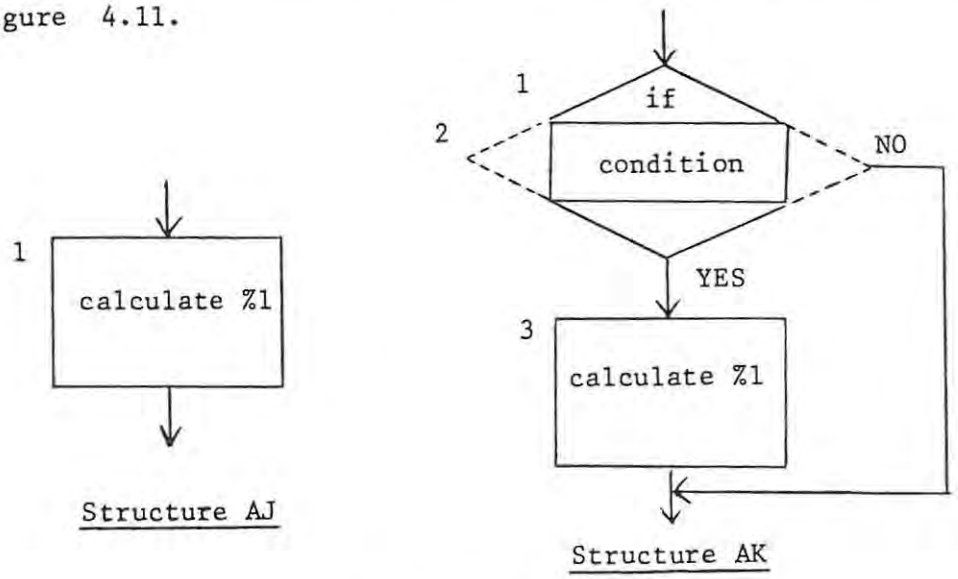


Figure 4.11

Then, if ANNUAL-BONUS and ADDNA are indicated to be restricted, the

flowchart representation will take on the form shown in Figure 4.12. Boxes of type EE2 are used for flowchart expansion by question 15 and would not have been introduced in the flowchart under the assumptions that have been made in the examples presented.

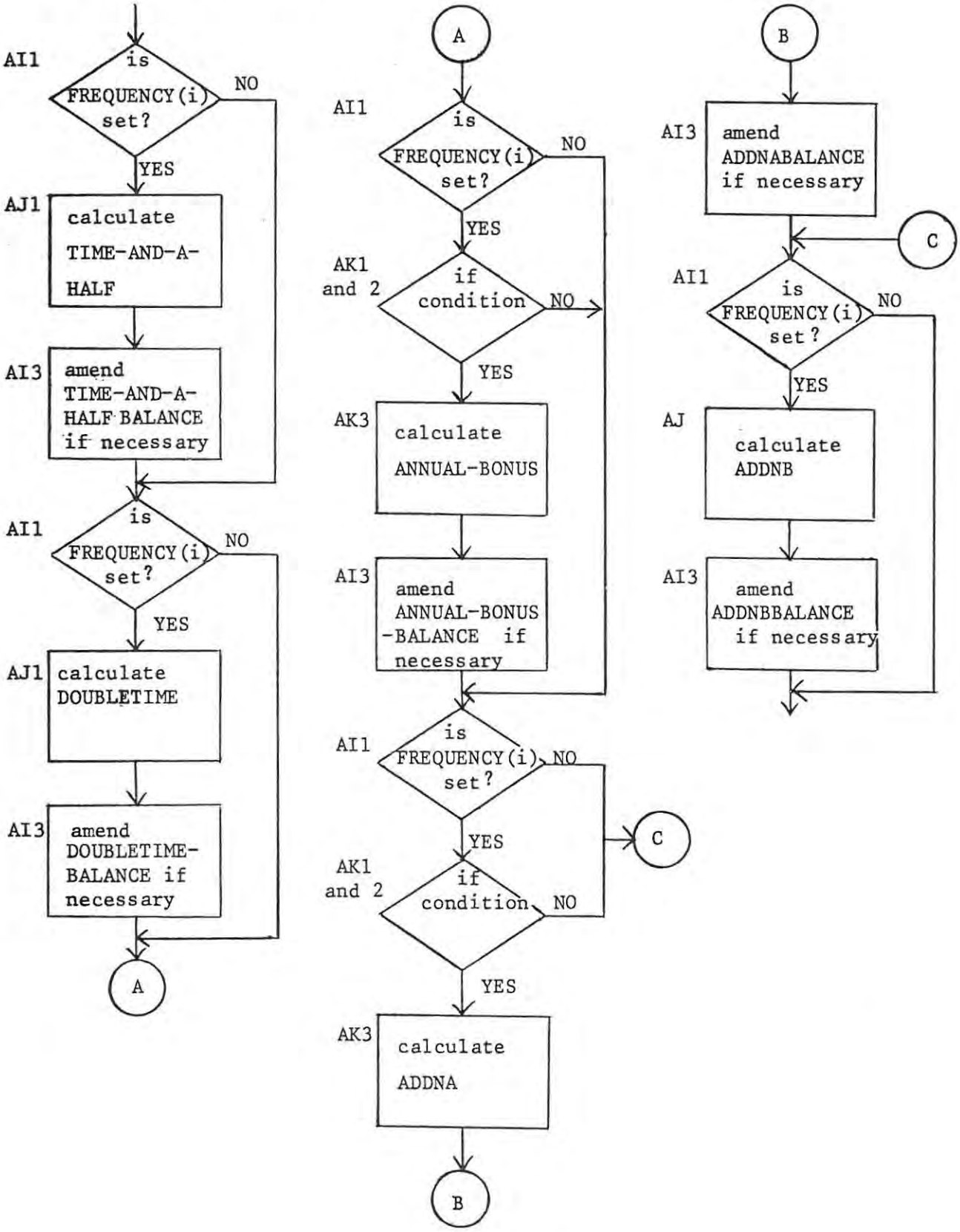


Figure 4.12

A jump will now be made to look at question 22, in order to give an example of the use of the ← action.

Q22 ADDITIONS TO BE USED IN CALCULATING PENSION

THE ADDITIONS ARE LISTED BELOW. TYPE Y NEXT TO EACH ADDITION WHICH IS TO BE INCLUDED IN THE GROSS PAY TOTAL FROM WHICH PENSION PAYMENT IS CALCULATED AND N NEXT TO EACH WHICH IS NOT TO BE INCLUDED.

EXPANS

$\mathcal{O}$ (G11, x ⇐ readchar, if x # Y and x # N then error else x ↓ anstack fi)

ACTION

$\mathcal{S}$ (anstack, N, storestring (NONPENSIONADDNS) ↓ G1; $\mathcal{F}$(B3, box ← AU);

$\mathcal{F}$(AQ&AR, box ← AV) / $\mathcal{F}$(AQ&AR, box ← AN))

$\forall$(AS1 & AT1, if ans = N then box → AX(%1,%1) fi)

At this stage, continuing with the assumptions of the flowchart previously developed, boxes AK3 and AJ will all have become structures AS or AT depending on whether the additions concerned are taxable or not respectively. These structures have the form shown in Fig. 4.13.

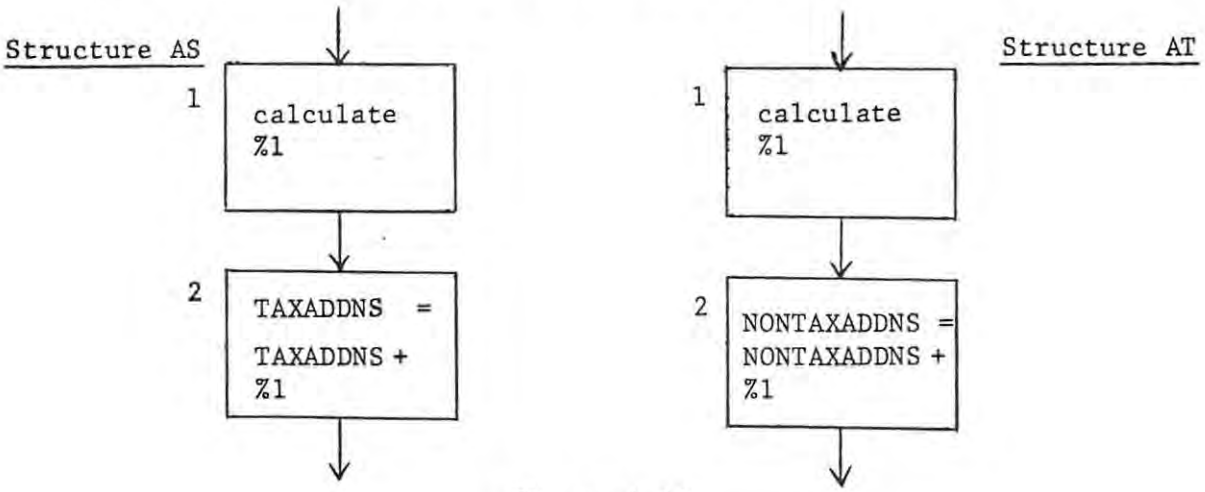
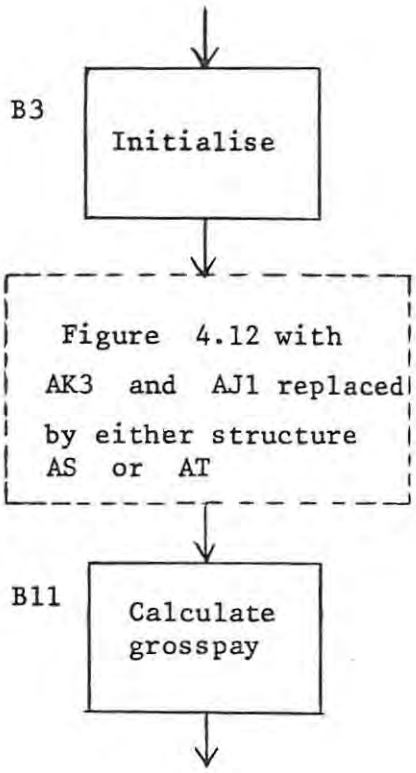


Figure 4.13

Assume that the flowchart at this stage has the form depicted in Figure 4.14 and that the structures used in the action may be represented diagramatically as shown in this figure.



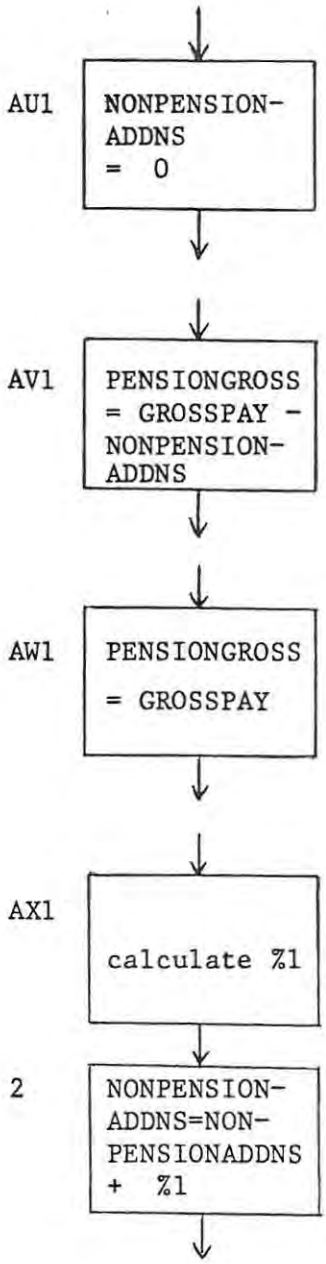
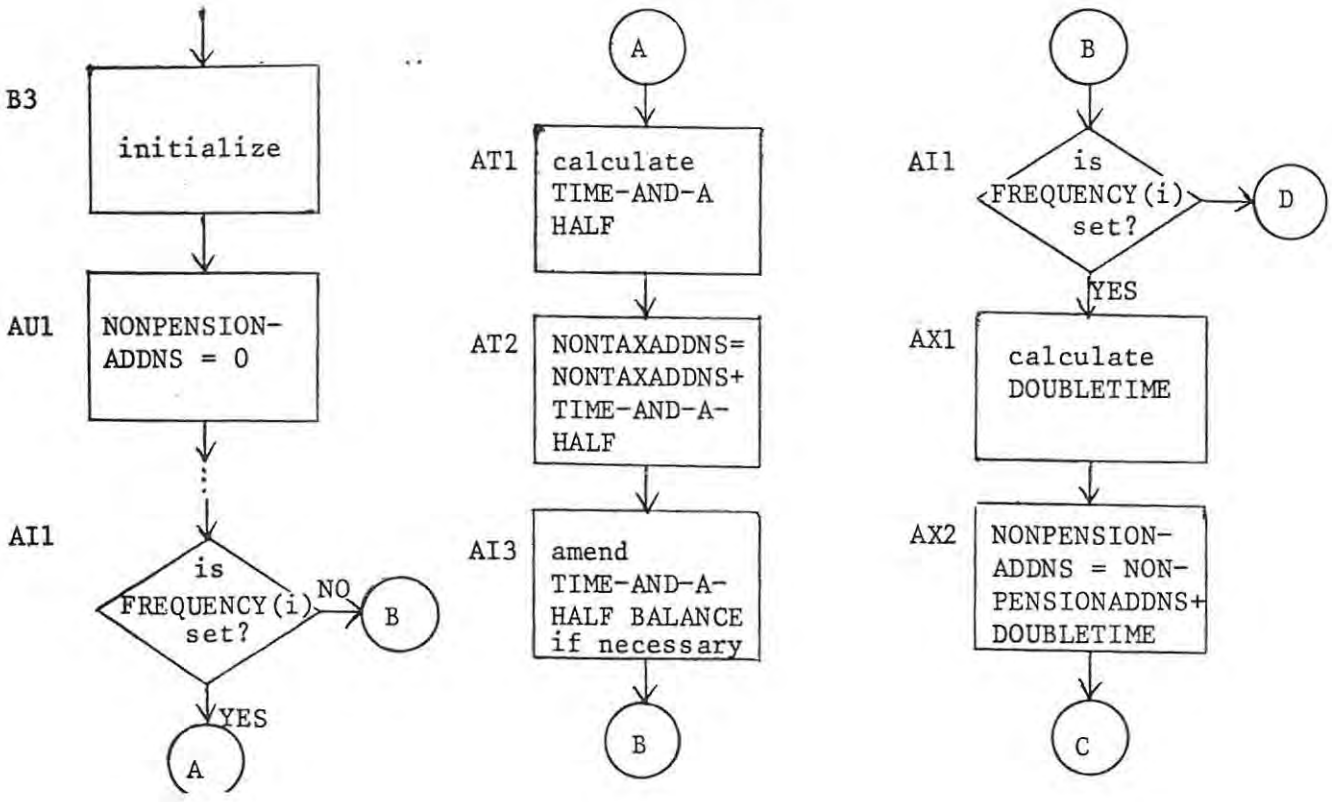


Figure 4.14

Figure 4.15



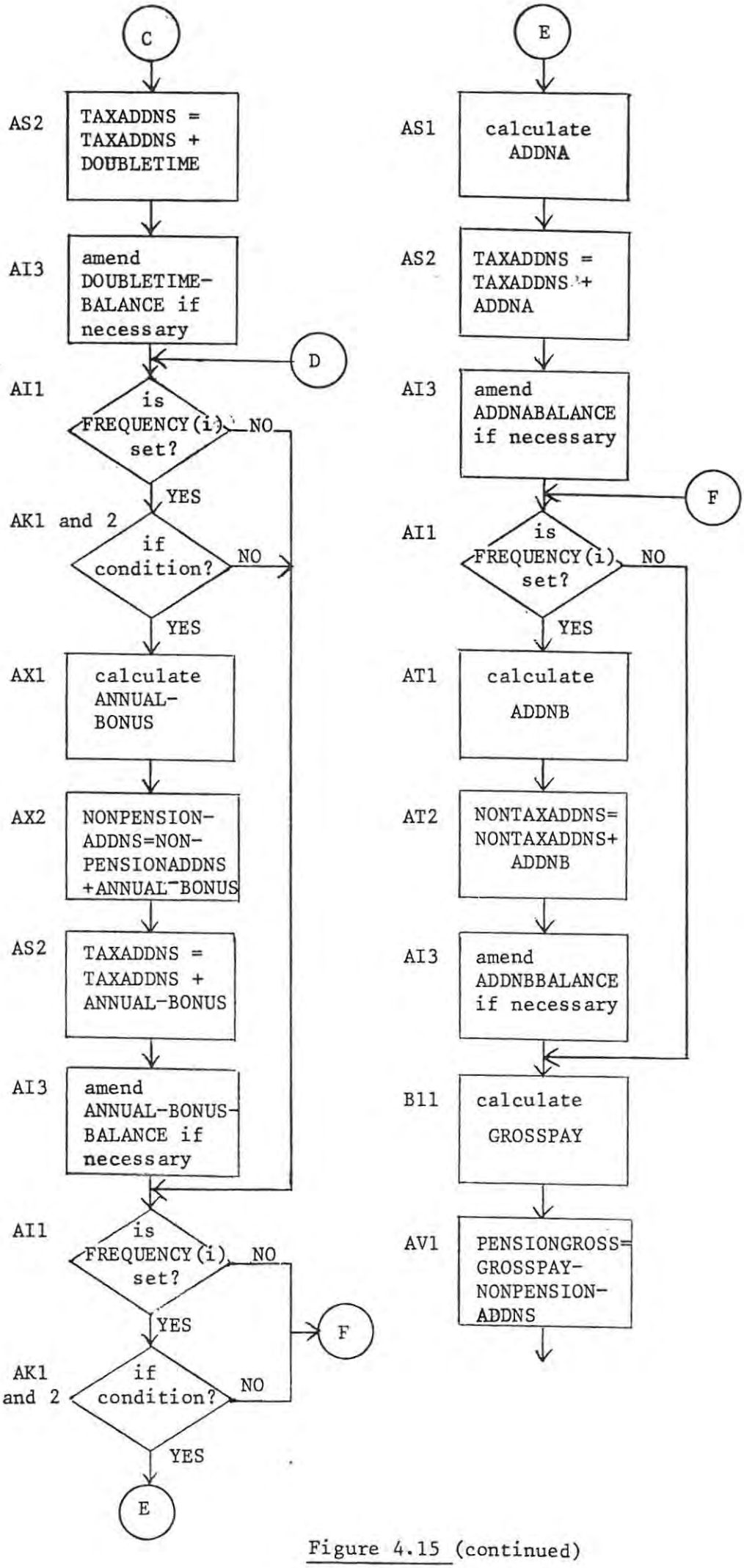


Figure 4.15 (continued)

Further assume that TIME-AND-A-HALF and ADDNB are non-taxable. Then, if DOUBLE-TIME and ANNUAL-BONUS were specified to be non-pensionable in the answer to the question under consideration, the flowchart would take on the form shown in Figure 4.15 after executing this action component.

Examination of the flowchart and structures pictured in Figures 4.13, 4.14 and 4.15 reveals that all taxable additions and all non-taxable additions are accumulated separately into two totals and non-pensionable additions are added into another total. This is in order to compute a taxable gross total by adding the taxable additions only to basic pay; a gross pay total by adding the taxable additions and the non-taxable additions to basic pay and a pensionable gross total by subtracting the non-pensionable additions from the gross pay total.

Question 27, considered very briefly next, displays another form of the action ← .

Q27 UPPER LIMITS TO CALCULATED ADDITION CLAIMS

THE ADDITIONS FOR WHICH CLAIMS ARE CALCULATED ARE LISTED BELOW. TYPE TYPE NEXT TO EACH THE VALUE OF THE UPPER LIMIT IN RANDS AND CENTS (IF THERE IS ONE). IF THERE IS NONE, TYPE A NEGATIVE VALUE. THE VALUE MUST BE IN RANDS AND CENTS.I.E. IN THE FORM D.DD, AND MUST HAVE AT LEAST ONE DIGIT BEFORE THE DECIMAL POINT.

EXPANS

$\mathcal{O}$ (G16, r ⇐ readno, if r < 0.0 then N ↓ anstack else P ↓ anstack fi;
charstring(r) ↓ G17; outpointer ↓ G18)

ACTION

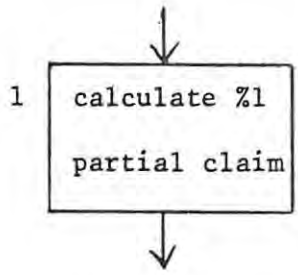
$\forall$(BB, if ans = P then box ← BM(G18, G17, G18, G17) fi)

$\mathcal{b}$ (G17)

$\mathcal{b}$ (G18)

Onto all the flowchart boxes concerned with calculating additions which are

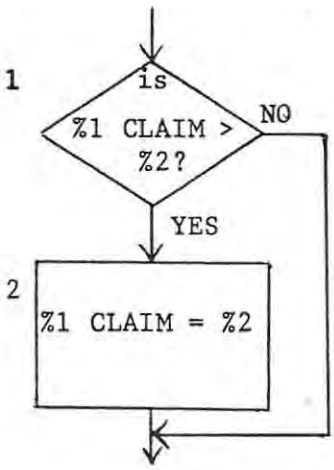
partially reclaimable, will have been added the structure BB. It has the form shown in Figure 4.16 below. Global stack 16 has pointers to the names



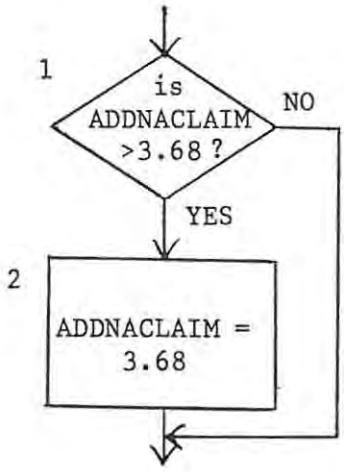
Structure BB

Figure 4.16

of these reclaimable additions, and corresponding to each element of global stack 16, the expected answer component stacks a value of P or of N onto the local stack, anstack, depending on the values typed in by the user.
The action component then works as follows. It locates the first box of type BB in the flowchart. If the value of the first element of the anstack is P then the box is expanded using the structure BM with the first elements of global stacks 17 and 18 as parameter values. Structure BM has the form shown in Figure 4.17 below.



Structure BM



An example of the use of structure BM with parameter values substituted.

Figure 4.17

If the element of anstack has value N, then box BB is left unchanged and a search is made for the next occurrence of boxtype BB. The procedure is then repeated using the second elements of anstack, global stack 17 and global stack 18, and so on until every box of type BB in the flowchart skeleton has been located and expanded if necessary.

The final question presented as an example shows another form of the → action.

Q31 HOW MANY GROUPS

EACH ADDITION FOR WHICH THE PARTIAL RECLAIM IS NON-UNIFORMLY CALCULATED, IS LISTED BELOW. TYPE NEXT TO EACH THE NUMBER OF GROUPS INTO WHICH THE EMPLOYEES ARE DIVIDED FOR CALCULATING THE RECLAIM OF THAT ADDITION - I.E. IN HOW MANY DIFFERENT WAYS THE ADDITION CLAIM IS CALCULATED.

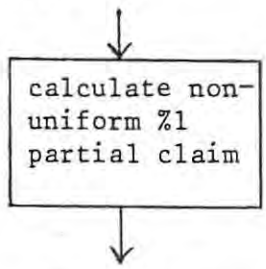
EXPANS

$\mathcal{O}$ (G18, y ⇐ readn$_0$; y → G15)

ACTION

@(G15, G15 ↑ n, $\mathcal{F}$(BD,box →* n BE(%1))

All partially reclaimable additions for which the claim is non-uniformly calculated will have had the structure BD inserted after the box responsible for calculating the addition in the flowchart. The structure BD may be depicted as shown in Figure 4.18.



Structure BD

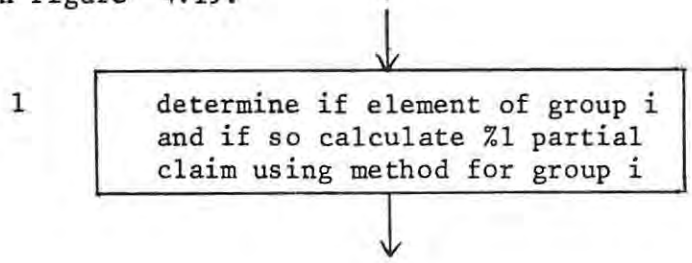
Figure 4.18

After the expected answer component of question 31 has been performed, global stack 15 will contain an element for each non-uniformly calculated

partial addition claim.

The action component causes an element of global stack 15 to be accessed and a box of type BD to be located and expanded into as many structures as the value of the accessed element specifies. This is repeated for each of the elements of stack 15.

Thus each box of type BD becomes n boxes of type BE which has the form shown below in Figure 4.19.



Structure BE

Figure 4.19

As the examples considered should be sufficient to have presented an idea of the use of the various forms of the flowchart operations and a picture of the whole strategy of flowchart expansion, no further examples will be discussed here. The method of formulating the actions and the top down approach to flowchart development proved to be extremely powerful.

4.4 ERROR RECOVERY

A method of error recovery has been implemented, which enables a user to rectify any error which he may have made at some stage during the answering of the questions. The way in which this is done does not necessitate the system being restarted and the user re-answering questions which he answered correctly before making a mistake. Instead, once an error is indicated by the user, the system automatically recreates the situation which existed before the incorrect answer was given.

Error-recovery has been allowed for by having two modes in which the Question-Answering Program runs viz. normal mode and error-recovery mode. In normal mode, when the Question Answering Program is started, an initialising

procedure is called. Ths procedure reads in data which remains unchanged throughout the processing, as well as the initial flowchart skeleton. Since this fixed data and the initial program skeleton are all the data required for error recovery, as the program skeleton is read, it is also written to a disc file, which will be referred to as the answerfile. In addition, in normal mode, as each question presented to the user is answered, the input procedures accept user responses from the console, and also write these answers to the answerfile.

Then, if the user realizes that he has made an error at some stage in answering a question, he can correct his mistake by typing the character "#" the next time that he is invited to type in the answer to a question. This then causes the error-recovery procedure to be entered. This procedure firstly requests the user to type in the number of the last question which he is satisfied he answered correctly, and calls a reinitialising procedure. The reinitialising procedure sets the mode to error-recovery mode, resets the answerfile to the beginning and reads in the initial flowchart skeleton once more. Thereafter, the data for the appropriate questions is read as question after question is reprocessed. In error-recovery mode, the question is not presented to the user, but the input procedures obtain the answers from the answers previously given and stored on the answerfile. This continues until the question specified as being the last correctly answered question has been processed. Then the mode is reset to normal and questions are once more presented to the user to be answered, and answers are written to the answerfile.

In addition to the error recovery described above, some checking of answers is also done by the input procedures which implement read operations, and may result in the user being requested to retype his reply. If an incorrect answer is detected, or an answer is out of the range of a list of choices that has been provided, then a wronganswer flag is set. The action $\mathcal{J}$ examines this flag and if it is found to be set, causes the program to invite the user to resubmit his answer.

4.5 GENERAL DESCRIPTION OF THE QUESTIONS

At this point the format of the question data, the notation in which the expected answer and action components are formulated, the method of presenting the questions to the user and the processing of his replies, as well as the possible inputs and outputs of the generated program have all been discussed. There remains only the question of what the questions themselves are actually concerned with.

The point that had to be borne in mind during the designing of the questions, was that their purpose is not only to extract from the user information about his requirements, but also to obtain the information necessary to generate a payroll program. This is evident from the second of the problems requiring solution in order to produce a program-generating system, which were listed in section 2.4. Inherent, therefore, in the questions chosen, is the result of determining both what information is actually necessary in order to generate a payroll program, as well as the needs of such a program.

What information has to be collected and the order in which the collecting has to be done is best shown by looking briefly at the questions themselves. A preview of the type of information that has to be obtained about inputs and outputs has been provided by section 3.3.

The first group of questions deal with whether the program to be generated is a special-purpose program (e.g. a program to set up a file of data) or a periodic payroll program. Special-purpose programs that may be generated are described in section 4.7. The questions then determine the type of special-purpose program to be generated or, alternatively, if a payroll program is to be produced, what additional facilities such as reconciliation, are required. Additional facilities that may be generated are also to be found in section 4.7.

Depending on what was determined concerning information to be input to the generated program, the next questions concern processing of information - for example whether information is to be sorted prior to being used or not.

After this, there follows a long range of questions concerning payroll calculations. These questions establish what additions there are and what deductions. The frequency with which additions and deductions are to be calculated and the employees to whom each addition or deduction is applicable are the subject of several questions. There are questions to discover which additions and deductions are taxable and which are not, which are pensionable and which are not. Questions also establish which additions may be reclaimed from some other source by the employer, and the source names. For deductions it is slightly more complicated as it has to be established not only which deductions are subsidized by the employer, but also which deduction subsidies are reclaimable from some other source by the employer, and these source names.

After the names of all the additions, deductions, addition claims, deduction subsidies and deduction subsidy claims have been determined, the questions concern how these quantities are calculated. For every quantity it is asked whether the quantity is fixed or calculated. Fixed values for quantities are extracted by questions requesting these values. Questions also exist to determine for each quantity that is calculated whether it is calculated uniformly i.e. in the same way for all employees to whom it applies, or whether it is calculated non-uniformly i.e. in different ways for different groups of employees. If a quantity to be calculated is calculated non-uniformly, the groups over which it is uniformly calculated are dealt with. In the case of addition claims, deduction subsidies and deduction subsidy claims the questions determine whether the quantities are fractions of the additions, deductions or deduction subsidies respectively and extract these fractions as well as upper and lower limit values.

The calculation of quantities, such as PAYE possibly, by using a table can be allowed by providing a structure responsible for table look-up and by including a preset table definition. The table definition will be part of the fixed data for the final program, which is described in section 6.4.
A question is present to determine whether or not a coinage breakdown must be

calculated. There are also questions to establish what quantities must be added into cumulative totals and what balances (e.g. savings balance, loan balance) must be maintained.

After all the questions responsible for obtaining information regarding payroll calculations, there follow questions which are concerned with the inputs and outputs i.e. information, files, and reports, of the program to be generated.

The contents of the employee payroll records are obtained and names of further fields which must exist are requested from the user. Thereafter, questions establishing contents of remaining files and sort keys for files which have to be sorted, occur. Next, the contents of information input are requested. Once the names of contents of information have been obtained, questions detailing which of these fields must be verified can follow. Verification may be a check as to whether a field is alphabetic or numeric and, possibly, if it should be numeric, the range in which its value should lie. The match fields for information can also be obtained by questions, once the names of contents of information are known. A match field is a field that is used to determine with which record of a file each information record is associated. An example of this occurs in, say, cancelled information. There will probably be a field in each record containing a cheque number, and this number is used to find the record of the reconciliation file with the same value in its cheque number field. Similarly, questions use the information content names to determine sort keys for information that has been specified as having to be sorted.

Questions responsible for obtaining all information required to generate reports firstly request the report names. Next they establish the frequency of reports and thereafter are responsible for determining the type of each report - i.e. whether a report is a full list report or a partial list report or a summarized report. The contents of full and partial lists and the names of all quantities involved in each summarized list are the aim of the next few questions. For partial lists, questions determine what categories of employee must be present, and the order of full and partial lists is obtained. The final questions

concerning reports establish sort keys for full and partial lists that must be sorted; categories for subdivision of full and partial lists, categories over which summarized lists give breakdowns and what quantities must be subtotalled and totalled in each list of each type.

Lastly, the questions are concerned with obtaining edit codes to represent edit operations; determining what field changes may occur during an edit of the payroll master file; determining quantities requiring cheques and what types of cheques are to be produced and collecting information regarding the printing of reminder notices.

As a matter of interest, cheque types may be

(a) individual - i.e. one cheque is printed for each employee, or

(b) single-implying that one cheque is produced as all payments are accumulated into a single cheque, or

(c) mixed - in which case some of the employees fall into categories and one cheque is printed for each category and also one cheque is printed for each employee who does not belong to a category.

As has been mentioned previously, the path followed through the questions depends on the user's answers to questions at each stage .

4.6 INFORMATION STORED AND OUTPUT BY THE QUESTION-ANSWERING PROGRAM

Two types of information are required in order to generate a payroll program, as was stated in section 2.5.1. The information required is, firstly, information related to the structure of the program to be produced, and, secondly, information concerned with inputs to and outputs from the program.

However, the information concerning the program structure is not independent of the information concerning program inputs and outputs, as the structure is really a representation of the processing of the inputs and outputs or of fields which are contents of inputs and or outputs.

During the course of the questions, every time the existence of a field is detected, its name is stacked onto global stack 1. These field names are

used to fill in parameter values of structures when the flowchart is expanded - the parameters are set to point to the appropriate field names. The fields are also used in the questions determining contents of information, files and reports, and so the field names are common to both information pertaining to the structure of the final program and to the information concerning inputs and outputs of this program.

Information which gives the structure of the final program is stored entirely in the form of the flowchart skeleton together with the character area containing character strings to which the flowchart parameters point. At the end of the question-answering process the character area and the flowchart skeleton are output to a disc file. This file then acts as input to the Program Generator which constructs COBOL PROCEDURE DIVISION statements from this information. Additional output for the Program Generator consists of the values of several flags, which may have been set during the course of the questioning process. These are written to a file at the end of the flowchart skeleton and they indicate whether processes such as updating of the payroll master file, printing of cheques or reminders etc. are to be performed. The flags that exist and their use will be described in Section 6.4, and their function is to control the listing of fixed data during the generation of the final program, depending on what options have been selected by the user.

All remaining information obtained from question answers and required for further use is stored in the global stacks and also the character area. It is best shown what information is actually extracted from question answers and stored by considering what the various global stacks are used to store.

Global stack 1 : This stack is used to stack up pointers to the names of all the fields that have been established. The field names themselves are stored in the character area, and the pointers point to their positions in this area. All field names are stored irrespective of whether a field is only used to hold temporary intermediate results in calculations or whether the field is a field of a file or report or of information.

Global stack 2 : In this stack is stored pointers to the names of indicators or flags. The names are again stored in the character area.

Global stack 3 : This stack contains pointers to the names of information input. The input information names are strings in the character area.

Global stack 4 : This is used to store pointers to all file names in addition to the payroll master file. As is the case for information the names are stored as strings in the character area.

Global stack 5 : This contains pointers to the names of all reports. Again, the names are stored in the character area as strings.

Global stack 6 : The contents of this stack consist of information processing codes (cf. section 3.3.1). There will be the same number of elements in this stack as in stack 3 and the information stored in corresponding elements in these stacks is associated.

Global stack 7 : This is used to store file version codes (cf section 3.3.2). The elements of this stack contain information concerning corresponding elements of stack 4.

Global stack 8 : Contains pointers to the names of files that have to be sorted.

Global stack 9 : Contains pointers to the names of information that have to be sorted.

Global stacks 10 - 29 : Contain different information at different stages of the questioning process or information that is not required at the end of the Question-Answering Program.

Global stack 30 : This stack is used to store pointers to the names of fields that constitute the contents of each employee record in the payroll master file. As is always the case the names are stored in the character area.

Global stack 31 : In this stack is stored the contents of employee records as field numbers. By this is meant that the element number of stack 1 that corresponds to each contentname is stored in this stack.

Similarly,

Global stack 36 : contains contents of information as pointers to the names in the character area and

Global stack 34 : contains the contents of information in the form of field numbers . In stacks 36 and 34 each element consists of a pointer to a stack whose elements are pointers to the content-names or are the actual field numbers. Information stored in each element of stack 36 pertains to the corresponding element of stack 3 as does the information of stack 34. Exactly the same method is followed in storing the contents of files as pointers to the content names and as field numbers, and for storing the contents of all full list reports, partial list reports and summarized reports. In addition the identity numbers of reports are stored in three stacks whose elements correspond to those of the three stacks which contain the names of the full list reports, partial list reports and the summarized reports.

This description should suffice to give an idea of how information is stored in the various global stacks and so no further enumeration of global stacks and explanation of their contents will be done.

The output to the disc file which is required by the Format Editor is obtained from these global stacks and consists of, among other information : the names of all fields as strings of characters; the names of all information, files and reports, also as character strings and the contents of these as field numbers; the processing codes for information and the version codes for files, and so on. Exactly what information is received by the Formal Editor and how it is utilized is the subject of Chapter 5.

4.7 PROPERTIES OF THE FINAL PRODUCT

The payroll functions for an organization are very often performed, not

by a single program, but by a suite of programs. It was therefore decided to formulate the questions in such a way that the user could decide whether to generate a single program which provides all the facilities he requires, or whether to generate a number of programs to perform all the payroll functions.

The very first of the questions determines whether a periodic payroll system or a special-purpose program is required. The functions which can be performed by each of these will now be described.

4.7.1 SPECIAL-PURPOSE PROGRAMS

The special-purpose programs which the user may generate can perform any single one of the following functions :

(i) create a disc file - i.e. read data and store it on a file, which may then be sorted.

(ii) effect changes to a payroll master file - i.e. perform editing functions.

(iii) produce reports from quantities that have been calculated previously and stored on the payroll master file.

(iv) reconcile cancelled cheques or payslips.

(v) perform some calculation(s) using payroll master file fields.

A special-purpose program which performs any selection of functions (ii), (iii), (iv) and (v) may also be generated.

If the user has generated special-purpose programs responsible for some aspects of the payroll system, when he generates the payroll program itself, he will obviously specify as "not required" those functions for which programs have already been provided.

4.7.2 PAYROLL PROGRAM FACILITIES

The payroll program which is generated may optionally provide a facility for altering or editing the payroll master file and may be able to perform the reconciliation of cancelled cheques or payslips, depending on the user's specifications. Periodic information such as hours worked may be read by the

program in order to perform payroll calculations. The user has to specify whether or not this is necessary as it probably depends on whether the payroll system is a wages system or a salary system.

In order to give an idea of the structure of any payroll program that may be generated, a general flowchart is pictured in Figure 4.20. (This is structure B).

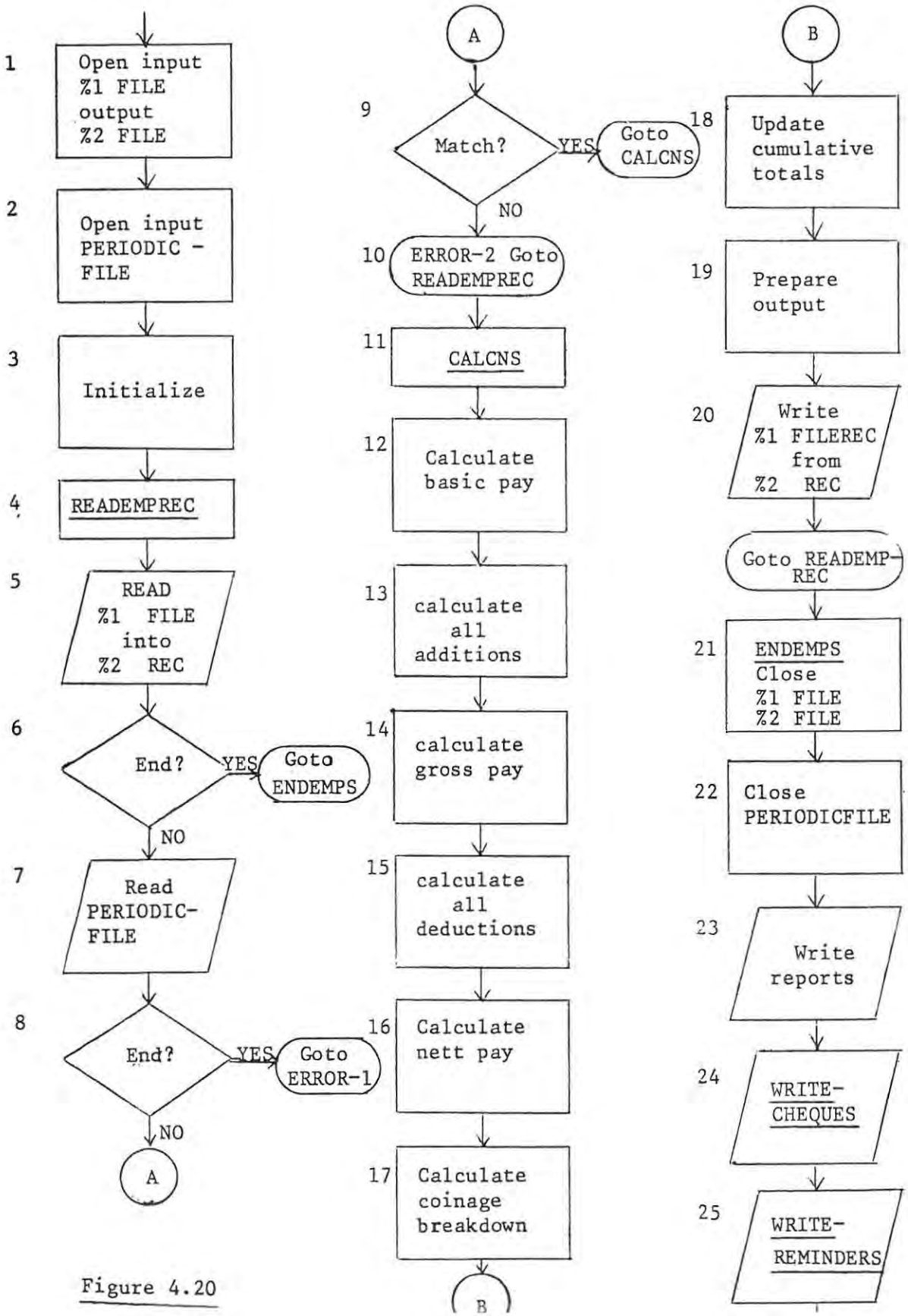


Figure 4.20

4.8 ERRORS AND TRACING IN THE QUESTION-ANSWERING PROGRAM

In setting up the program-generating system, an enormous problem lies in obtaining error-free question data. Two types of errors are possible - errors in the form or syntax of the actions of the expected answer and action components, and errors in the logic or content of the formulation of these components. In fact, the only way of eliminating errors is by repeatedly running the Question-Answering Program and fixing errors as they are detected until the data seems to be error free. It will be appreciated that this is a very laborious task.

To facilitate the eradication of syntax errors, the question listing program does enable the questions to be listed and provides some means of checking that correct codes have been used in converting the data. However, not all errors will be evident on inspection of this listing. The Question-Answering Program checks that codes are correct as it interprets them. When an error is encountered by this program, an error message containing an error number is printed and the program halts. A complete list of these error numbers and their meanings can be found in Appendix 3.

An additional aid to discovering format errors in the data, as well as the means for checking the logic errors, is the tracing facility. Before answering each question, the user is asked whether he wants a monitor or trace of the question. If he specifies one as the value of the monitor flag, a trace of the path followed through the various procedures during the interpretation of the expected answer and action components is given. The form of the trace is a listing of the names of the procedures entered. In addition, the contents of the character area, the stacks and the flowchart are provided at the end of the processing of the question concerned. This enables the effects of the actions to be checked and the procedure which was last entered to be determined.

5 THE FORMAT EDITOR

5.1 INTRODUCTION

The task of the Format Editor is to obtain information about input and output formats, as well as field descriptions, from the user, and to use this information to generate COBOL statements for the FILE-CONTROL, DATA DIVISION and PROCEDURE DIVISION sections of the COBOL program to be generated. The Format Editor receives information as to what fields and inputs and outputs exist, as well as related data, from the Question-Answering Program, and then uses an interactive method to obtain the user's format specifications.

The output of the Format Editor consists of four disc files, containing FILE-CONTROL, FILE SECTION, WORKING-STORAGE SECTION and PROCEDURE DIVISION code respectively.

In the description of the Format Editor that follows, the data structures that it uses will be described as they are encountered in the course of the explanation.

5.2 INPUT AND STORAGE OF THE INFORMATION FROM THE QUESTION-ANSWERING PROGRAM

5.2.1 DATA STRUCTURES

The data structures involved in the setting up of the data from the Question-Answering System are depicted in Figure 5.1.

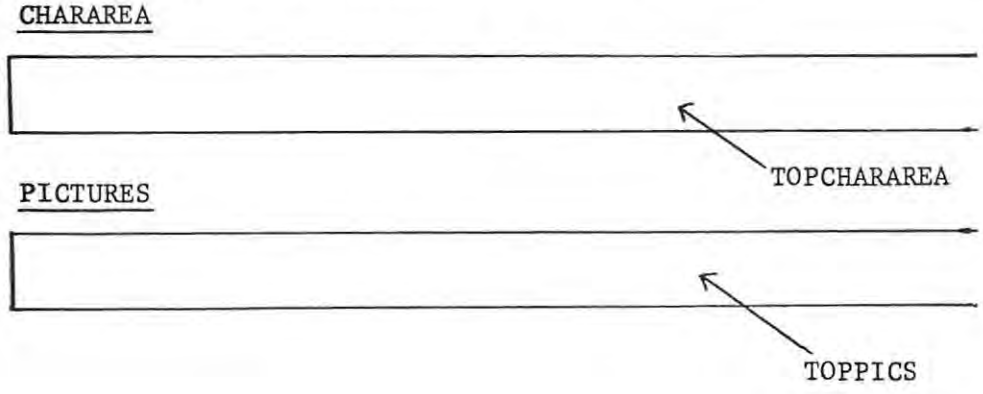


Figure 5.1

FIELDS

name (pointer to CHARAREA)	format (pointer to PICTURES)	picture size

INDICATORS

name (pointer to CHARAREA)

NOIO

number of iofiles
number of information
number of reports

CONTENTS

name (element number of FIELDS)	original or not 0 = original 1 = not original

EDITFIELDS

name (element number of FIELDS)

INFORMATION

name (pointer to CHARAREA)	processing code	number of contents	start of contents (pointer to CONTENTS)

IOFILES

name (pointer to CHARAREA)	version code	number of contents	start of contents (pointer to CONTENTS)

REPORTS

name (pointer to CHARAREA)	type (1 = full list 2 = partial list 3 = summarized)	sorted or not (1 = to be sorted 2 = not to be sorted)	number of contents	start of contents (pointer to CONTENTS)

Figure 5.1 (continued)

5.2.2 INFORMATION RECEIVED

The information received from the Question-Answering Program consists of :

(i) The names of all the fields

(ii) The names of all indicators or flags

(iii) The names of all information input, their processing codes and the contents of each of these inputs, if any

(iv) The names of all files, their version codes and the contents of each file

(v) The names of all reports, the type of each report, an indication as to whether each report must be sorted or not, and the contents of each report, if there are any reports

(vi) The names of all fields that may be altered during an edit of the payroll master file, if any.

All of the data structures represented in Figure 5.1 are set up as shown in the diagram using the data from the Question-Answering Program, with the exception of the second two columns of array FIELDS and the array PICTURES. These were put in for completeness but are used during the processing of the Format Editor.

As the possible values of file version codes and information processing codes, and the implications of these have been discussed in sections 3.3.2 and 3.3.1 respectively, all that remains to be explained is what an "original" contentname is, or, alternatively, what is the use of the second column of array CONTENTS.

A problem which arises in determining the contents of files and information input is that the same field may be chosen as being part of the contents of several inputs and outputs. The problem lies in knowing which occurrence or use of the field is "original" and which occurrences are subsidiary. An example to clarify what is meant is as follows. The field, employeeno, may appear both on the payroll master file and as a field of periodic information. The use in the payroll master file record may be deemed "original", whilst its use in the

periodic information is to associate a record of the information with a master file record and so it is simply another copy of the original field. Different data names must be given to different occurrences of a field and when reports are prepared, the "original" occurrences of fields should be used to obtain field values.

To overcome this problem, it was decided to mark one occurrence of a field as being original and all other occurrences as being non-original. This was done in the following manner. A value which was initially set to zero, indicating originality was associated with each of the fields. Then, to determine the contents of an input or an output, the Question Answering Program listed the fields for the user to select contents. A contents list was compiled for the input or output using the selected items and, at the same time a corresponding list indicating which contents were original and which were not, was drawn up. Then, in the list associated with the list of fields, all the fields which were selected were marked as no longer original. The order of determining contents of inputs and outputs is therefore important and was as follows. First the contents of the payroll master file, then the contents of other files, and, after that the contents of information were determined.

Thus it is these "original" values that are stored in array CONTENTS.

5.3 ORDER OF GENERATION OF INPUT AND OUTPUT FORMATS

The order in which the user is requested to provide formats is as follows :

(i) File formats are the first to be obtained. When the format of a field which is marked as original is provided, it is stored in array PICTURES and columns 2 and 3 of the appropriate row of FIELDS are set. The generation of file formats is the subject of section 5.4.

(ii) Information formats are the next to be generated. Once again the formats of "original" fields, once obtained are stored in array PICTURES and array FIELDS is set as necessary. At the end of this stage, the formats of every

field used either in a file or in information will have been stored. This stage is described in section 5.5.

(iii) Report formats are requested from the user next. This process is fully described in section 5.6.

(iv) Remaining fields are presented to the user in order for him to provide descriptions. The way in which formats are generated for these items is described in section 5.7.

The remaining generation of code is done automatically without the user having to provide any further information. This consists of:

(i) generating a reportfile description for the file to which all report records for full and partial list reports are written (cf. section 3.3.3)

(ii) generating report sortfile descriptions for any reports which have to be sorted before being written

(iii) generating a description enabling edit information to be read

(iv) generating descriptions of all indicators

(v) generating I-O-CONTROL code

The above processes are all described in section 5.7.

The different generation processes listed are each only performed if it is necessary i.e. only if there are reports are report formats generated and only if there are full list or partial list reports is a reportfile description produced etc.

The fixed codestrings used in generating COBOL code are identified by number and are accessed by using the code number to obtain a pointer to the string. The following structures are used.

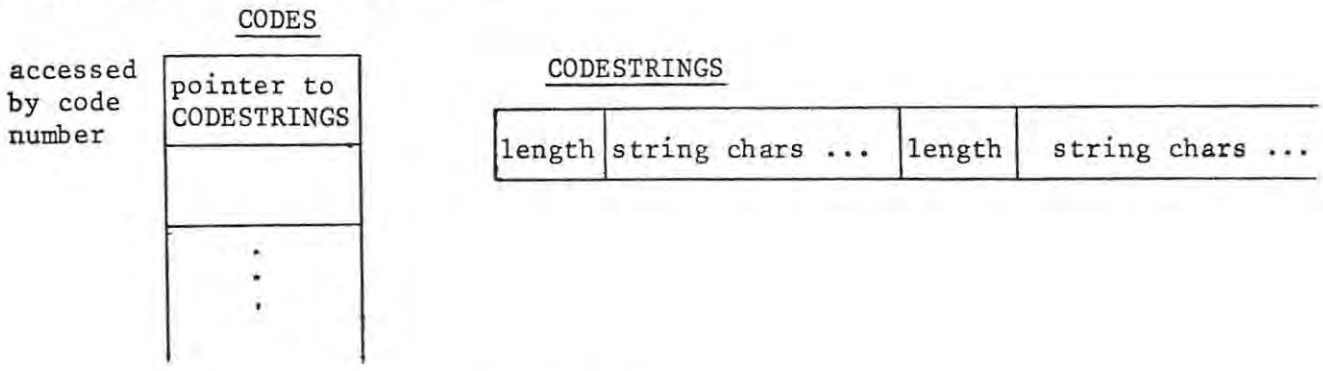


Figure 5.2

5.4 GENERATION OF FILE FORMATS

5.4.1 USER VIEW

In being asked to provide the format specification for a file, the user will be presented with something along the following lines :

FORMAT FOR PAYROLLFILE. THE CONTENTS ARE :

1) EMPNAME

2) EMPNO

3) .
.
.
.

PLEASE INPUT THE FORMATS FOR THESE ITEMS :

whereupon he should type his specification. A sample reply might take the form:

▽▽▽%1"@@@@@@@@@@"▽▽▽▽%2"#### #".. ↑ .

Spaces are significant and corresponding FILLERs will be generated. The example indicates that a file record starts with a FILLER of three spaces followed by the EMPNAME field which consists of ten alphanumeric characters, then another FILLER of four spaces and then the EMPNO field of five digits and so on. The ↑ marks the end of the description.

The different options available in format specifications may be found in Appendix 4.

5.4.2 NAME GENERATION AND STRATEGY OF GENERATING FILE DESCRIPTIONS

The policy adopted in generating descriptions for files is to generate one detailed WORKING-STORAGE record description for each file. While this is being done, the record length is counted and then as many FILE SECTION descriptions as necessary are generated, with the record description of each consisting of a single field of size equal to the record length. A FILE SECTION description for a sort file must also be detailed and will be generated at the same time as the WORKING-STORAGE description, with data names for contents being of the form **filename**content**name**SORT.

If the file is of type setup, the WORKING-STORAGE record description is subdivided into group items each containing up to 80 characters.

The datanames used for contentnames in the WORKING-STORAGE record description are of the form

a) filename - contentname if the content item is not original

b) contentname if it is original

To recap, the different file version codes and their meanings are :

1 - No other version

2 - Old version

3 - New version

4 - Old and New Versions

5 - Temp and Sort versions

6 - File is of type setup

7 - File is of type setup and sort version is required.

The FILE SECTION descriptions that will be generated for each of these cases are :

1)
```
FD  filenameFILE
      .
      .
      .
01  filenameFILEREC  PIC X(reclength).
```

2)
```
As for version code = 1 and :
FD  OLDfilenameFILE
      .
      .
      .
01  OLDfilenameFILEREC  PIC X(reclength).
```

3)
```
As for version code = 1 and
FD  NEWfilenameFILE
      .
      .
      .
01  NEWfilenameFILEREC  PIC X(reclength).
```

4)
```
As for version code = 1 and :
FD  OLDfilenameFILE
      .
      .
      .
01  OLDfilenameFILEREC  PIC X(reclength).
```

and

```
        FD  NEWfilenameFILE
              .
              .
              .
        01  NEWfilenameFILEREC  PIC X(reclength).
```

5) As for version code = 1 and

```
        FD  filenameTEMPFILE
              .
              .
              .
        01  filenameTEMPFILEREC  PIC X(reclength).
```

and

```
        SD  filenameSORTFILE
              .
              .
              .
        01  filenameSORTFILEREC
```

detailed description using contentnames of the form filename**content**nameSORT

6)

```
        FD  filenameFILE
              .
              .
              .
        01  filenameFILEREC  PIC X(reclength).
```

7) As for version code = 1 and :

```
        FD  filenameTEMPFILE
              .
              .
              .
        01  filenameTEMPFILEREC  PIC X(reclength).
```

and

```
        SD  filenameSORTFILE
              .
              .
              .
        01  filenameSORTFILEREC
```

detailed description using contentnames of the form filenamecontentnameSORT

5.4.3 END RESULTS OF FILE FORMAT GENERATION

At the end of generating fileformats the following will have been produced :

(i) All the COBOL statements concerning the files involved, which are necessary for the FILE-CONTROL section of the COBOL program.

The convention used here is to assign all files with TEMP as part of their names (i.e. temporary files) to EDS 1, all SORT files to EDS 2 and all remaining files to an EDS followed by a unique number from 3 upwards.

(ii) All the FILE SECTION statements i.e. FD's and SD's and WORKING-STORAGE record descriptions required for the files concerned.

The FD's for these disc files and the SD's will all have the forms

```
FD  filename
RECORDING MODE IS F
BLOCK CONTAINS $ CHARACTERS
LABEL RECORDS ARE STANDARD
VALUE OF ID IS "            ".
01     etc.

SD  filename
RECORDING MODE IS F
BLOCK CONTAINS $ CHARACTERS
LABEL RECORDS ARE STANDARD.
01     etc.
```

The record descriptions in the FILE SECTION and WORKING-STORAGE SECTION will be as specified in section 5.4.2.

While the detailed record description for each file is being generated, the record length is calculated. In addition, a value for the blocklength for each file is computed and is stored in array BLOCKINFO. At the end of the Format Editor the contents of BLOCKINFO are written at the end of the disc file containing the FILE-CONTROL code, to be used by the Program Generator to calculate blocksizes to replace the $ characters appearing in the FD's and SD's.

(iii) Every time a new sortfile name is produced, a pointer to it is stored in array SFNAMES, as well as the value 2, to indicate that the filename consists of the name pointed to by the pointer followed by the string SORTFILE. Array SFNAMES therefore has the format shown in Figure 5.3

SFNAMES

pointer to name	name code
⋮	⋮

Figure 5.3

and it is used in a later part of the Format Editor to generate I-O-CONTROL code. This is described in section 5.7.

5.5 GENERATION OF INFORMATION FORMATS

5.5.1 USER VIEW

The user is requested to provide an information format in precisely the same manner as he is requested to provide a file specification. An example of a request might be :

FORMAT FOR PERIODICINFO THE CONTENTS ARE :

1) EMPNO

2) HOURS-WORKED

PLEASE INPUT THE FORMATS FOR THESE ITEMS.

Once again, the manner in which a reply should be presented resembles the form of a file format specification. Spaces are again significant and Appendix 4 should be consulted for a description of the different symbols which may be used in a specification.

5.5.2 FILE- AND DATANAME GENERATION

The different values of the information processing codes and their implications have been described previously (cf. section 3.3.2), but will be reproduced here for convenience :

Code value	meaning
1	information is not processed further. (It is left)
2	information must be filed
3	information must be sorted
4	information is used for setup

All record descriptions associated with information will be part of the FILE SECTION code. No WORKING-STORAGE SECTION code is generated unlike the case of files.

The FILE SECTION descriptions that will be generated for each of the different processing code values are :

1) FD infonameINFO.

01 infonameINFOREC.

02 dataname PIC ....

02 dataname PIC ....

⋮

where the dataname generated will be either :

infoname - contentname if the contents item is not original

or contentname if it is original

2) FD infonameINFO.

01 infonameINFOREC.

02 dataname PIC ....

02 dataname PIC ....

⋮

where the datanames generated will be of the form infonamecontentname.

and,

FD infonameFILE.

.

.

01 infonameFILEREC.

02 dataname PIC ....

02 dataname PIC ....

⋮

where the datanames generated will be either :

infoname - contentname if the contents item is not original

or contentname if it is original.

3) FD infonameINFO.

01 infonameINFOREC.

02 dataname PIC ....

02 dataname PIC ....

⋮

where the datanames generated will have the form : infonamecontentname

and,

```
FD  infonameTEMP.
      :
      :
01  infonameTEMPREC.
    02  dataname  PIC ....
    02  dataname  PIC ....
          :
          :
```

where the datanames have the form : infonamecontentname

and,

```
SD  infonameSORT
      :
      :
01  infonameSORTREC.
    02  dataname  PIC ...
    02  dataname  PIC ...
          :
          :
```

where the datanames again have the form : infonamecontentname

and,

```
FD  infonameFILE
      :
      :
01  infonameFILEREC.
    02  dataname  PIC ...
    02  dataname  PIC ...
          :
          :
```

where the datanames generated have the forms :

infoname - contentname if the contents item is not original

contentname if it is original

4)

```
FD  infonameINFO.
01  infonameINFOREC  PIC X(80) .
```

Where more than one detailed record description must be produced, i.e. in cases (2) and (3), as the first description is generated, it is also stored so that subsequent generations can be done automatically. The description is stored in the arrays INFDESCR and PICARRAY which are depicted in Figure 5.4 and show the manner in which storage is carried out.

INFDESCR

name (pointer to CHARAREA or -1 (indicating FILLER)	pointer to PICARRAY or FILLER size	original or not or -

PICARRAY

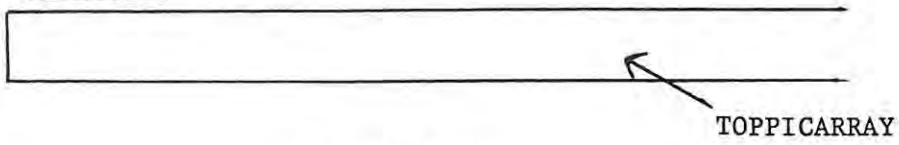


Figure 5.4

5.5.3 END RESULTS OF INFORMATION FORMAT GENERATION

The results of information format generation are as follows :

(i) FILE-CONTROL section code will have been produced for all INFO and associated files. All information is assigned to CARD-READER 0, and the convention of assigning temporary files to EDS 1, sort files to EDS 2 and other files to unique EDS numbers greater than 2, is again followed.

(ii) All the necessary FILE SECTION FD's and SD's for information will have been generated. Again, the form of these for disc and sort files will be

```
FD  filename
RECORDING  MODE  IS  F
BLOCK  CONTAINS  $  CHARACTERS
LABEL  RECORDS  ARE  STANDARD
VALUE  OF  ID  IS "          ".
01 .. etc.
```

and,

```
SD  filename
RECORDING MODE IS  F
BLOCK CONTAINS  $  CHARACTER
LABEL  RECORDS  ARE  STANDARD.
01   etc.                        respectively,
```

and, for card files :

FD filename

01 etc.

The record descriptions will be as described in section 5.5.2. As is the case during file format generation, blocklengths for disc and sort files will have been calculated and stored in array BLOCKINFO, for use by the Program Generator.

(iii) Pointers to sortfile names are added to array SFNAMES with the name code values being set to 1 to indicate that the name consists of the name pointed to followed by the string SORT .

5.6 GENERATION OF REPORT FORMATS

5.6.1 BACKGROUND

In generating report formats, the user has to be made to describe the physical appearance of the report. The way in which the appearance may be described is based on the ideas of the reportwriter feature which is provided by some COBOL compilers, but not, unfortunately, by the COBOL compiler used at Rhodes University.[4] A report is considered to consist of groups of lines, namely :

(i) overall heading - this is a group of lines produced only once at the beginning of a report.

(ii) page heading - this is a group of lines produced at the beginning of each new page.

(iii) subsection heading - this is a group of lines which is printed at the beginning of a new subsection of a report. Full and partial lists may be subdivided and subsection headings may head these subdivisions.

(iv) detail lines - this constitutes the main contents of the report.

(v) subsection footing - this is a group of lines which is printed at the end of a subsection of a report.

(vi) page footing - this is a group of lines that is printed at the end of each page.

(vii) overall footing - this is a group of lines that is printed only once at the end of a report.

Any of these groups may be present in a report, but they do not all have to be present.

5.6.2 USER VIEW

The way in which a user is requested to specify the format of a report is therefore as follows :

FORMAT FOR PAYROLL-REGISTER REPORT. THE FOLLOWING QUANTITIES ARE TO BE PRINTED OUT

1) EMPNO

2) EMPNAME

3) BASICPAY

etc.

PLEASE INPUT THE FORMAT TO BE USED FOR EACH OF THE FOLLOWING OUTPUT CATEGORIES:
(IF A CATEGORY IS NOT TO BE USED, TYPE "NONE↑" - EG. PAGE HEADING AND SUB-SECTION HEADING MAY BE THE SAME THEREFORE TYPE NONE↑ UNDER SUBSECTION HEADING)

(The Question-Answering Program will have determined what running totals are to be printed, and the names of these will be listed as part of the contents).

OVERALL HEADING :

At this point the user should type his specification for his overall heading. An example of this may be :

∇∇∇ ..... ∇∇∇PAYROLL-REGISTER∇∇∇ .....∇∇∇↑

∇∇∇ ..... ∇∇∇----------∇∇∇ .....∇∇∇↑

↑

This specifies that the first line of the overall heading consists of the heading PAYROLL-REGISTER.

The second line is to underline the heading. Note that ↑ is used to terminate the description of each line, and a ↑ in the first character position denotes the end of information concerning a category, in this case, the OVERALL HEADING.

The next thing to be printed will be :

PAGE HEADING :

After which the page heading should be supplied or NONE↑ typed.

Then any subsection heading will be requested after the printing of

SUBSECTION HEADING :

After the subsection heading has been supplied, the detail lines will be requested:

DETAIL LINES :

A sample reply might be

```
▽▽▽%1"####"▽▽▽▽▽%2"@@@@@@@@@ "▽▽▽%3"####.##" .......↑
↑
```

The report contents can also be used in specifying headings and may well be used, particularly in subsection headings and in footings. Totals will usually be required to be printed in footings. For a full description of what special characters exist and their effects in specifying formats, consult Appendix 4. Among these special characters is one which denotes that the date should be printed.

Finally the headings

SUBSECTION FOOTING :

PAGE FOOTING :

OVERALL FOOTING :

will be printed in turn and the user's replies accepted for each category. After the formats for all the categories have been specified the following is printed:

GIVE THE NUMBER OF THE LAST LINE ON THE PAGE IN WHICH A DETAIL LINE CAN START :

and this value must be supplied by the user. In effect, the value given serves to determine the page size, or number of lines per page, for the particular report. During the actual writing of the report, a count is kept of the number of lines printed, and as soon as it is equal to or exceeds this value, any footing lines are printed and a new page is started.

5.6.3 END RESULTS OF REPORT FORMAT GENERATION

As the specifications for each output category for each report are obtained, they are analysed and descriptions of these are generated. These descriptions will form part of the WORKING-STORAGE SECTION of the final COBOL program. In addition, COBOL statements which will appear in the PROCEDURE DIVISION of the final program are produced.

The clearest manner in which to explain the strategy followed in producing the COBOL code necessary for writing a report is to analyse an example of this process. This will be done after the procedure which is followed has been described briefly in the form of an algorithm.

Firstly, it should be noted that the information about each report is stored in the array REPORTS in order of ascending identity number. The identity numbers range from one to however many reports there are, and may be used to access REPORTS. Identity numbers were mentioned in section 3.3.3. and are used here in order to generate unique record and datanames for each report, as will be shown. They are referred to as idnos. Secondly, each of the output categories: overall heading, page heading, subsection heading, detail lines, subsection footing, page footing and overall footing has associated with it two characters. There are OH, PH, SH, DL, SF, PF and OF respectively, and are also used in generating datanames. They will be referred to as the groupcodes. Also, a line number counter, lineno and a field number counter, fieldno are used :

The procedure for analysing each category specification is as follows : (Brackets are present only to improve readability)

(a) A user's specification is read into a buffer, until the character ↑ is encountered.

(b) A COBOL paragraph name is produced for the PROCEDURE DIVISION. It has the form :

PRIN [groupcode][idno]. e.g. PRINOH1 .

The counter lineno is initialized at zero.

(c) The user's specification is then examined and if it is found to consist of NONE↑, then the end of analysing the category will have been reached.

(d) Otherwise, the record description to form part of the WORKING-STORAGE SECTION is begun. The first line of code generated will have the form

$01_{\nabla\nabla}$[groupcode][idno] .

(e) As analysis of the buffer contents begins, the counter lineno is incremented by one and the counter fieldno is initialized at zero. Another line of code is generated to appear in the WORKING-STORAGE SECTION, and it has the form

02_{∇}[groupcode][idno]L[lineno].

(f) The buffer contents are scanned until '↑' is encountered. Fields are determined and, depending on their types, one of the following occurs :

(i) if the field is a sequence of more than 2 spaces, the line of code 03_{∇} FILLER PIC X(n). where n = the number of spaces, is produced for the WORKING-STORAGE SECTION.

(ii) if the field is a string of characters other than the character % and contains fewer than 2 spaces consecutively, then fieldno is incremented by 1 and the line of WORKING-STORAGE code generated has the form :

03_{∇} [groupcode][idno]L[lineno] F[fieldno] PIC X(n) VALUE "string of characters". where n = the length of the string of characters.

(iii) if an item which comprises one of the contents of the report is referred to, i.e. %i "format specification" is found, then fieldno

is incremented by one and a line of code of the form :

03_{∇}[groupcode][idno] L[lineno]F[fieldno] PIC ....

result of analysing the format specification enclosed in " ... "

is produced for the WORKING-STORAGE SECTION, and, the statement

MOVE Rcontentname[idno] TO [groupcode][idno]L[lineno]F[fieldno] .

(if the report was not one that had to be sorted), OR

MOVE RScontentname[idno] TO [groupcode][idno]L[lineno]F[fieldno].

(if the report did have to be sorted)

is produced for the PROCEDURE DIVISION.

(g) After step (f), the code : MOVE [groupcode][idno]L[lineno] TO PRINTLINE.
WRITE PRINTLINE AFTER ADVANCING 1 LINES.
ADD 1 TO LINECOUNT.

is added to the PROCEDURE DIVISION code. The next specification provided by the user is read into the buffer, and if it is found to consist of one character, '↑' the end of analysing a category has been reached. Otherwise, the process is resumed at step (e).

Steps (a) to (g) above are executed for each of the 7 categories for each report.

(h) When input for all categories has been requested and analysed, the paragraph name EOR[idno] . is added to the PROCEDURE DIVISION code, before the next report is dealt with.

Example : The code generated for a report with identity number 1, with contents

1) EMPNO
2) EMPNAME
3) NETTPAY,

for which the user had specified :

```
OVERALL HEADING :       ▽▽▽▽▽▽▽WAGES REPORT▽▽▽▽▽▽▽↑
                        ▽▽▽▽▽▽▽-----------▽▽▽▽▽▽▽↑
                        ↑
PAGE HEADING :          NONE↑
SUBSECTION HEADING :    NONE↑
DETAIL LINES :          ▽▽▽▽▽%1"#####"▽▽▽▽▽%1"@@@@@@@@@@"  ▽▽▽▽▽%3"####.##"▽▽▽▽▽↑
                        ↑
SUBSECTION FOOTING:     NONE↑
PAGE FOOTING :          NONE↑
OVERALL FOOTING :       NONE↑
```

would be : (assume the report does not have to be sorted before being written)

WORKING-STORAGE SECTION code

```
01  OH1.
    02 OH1L1.
        03 FILLER PIC X(7) VALUE SPACES.
        03 OH1L1F1 PIC X(12) VALUE "WAGES REPORT".
        03 FILLER PIC X(7) VALUE SPACES.
    02 OH1L2.
        03 FILLER PIC X(7) VALUE SPACES.
        03 OH1L2F1 PIC X(12) VALUE "------------".
        03 FILLER PIC X(7) VALUE SPACES.
01  DL1.
    02 DL1L1.
        03 FILLER PIC X(5) VALUE SPACES.
        03 DL1L1F1 PIC 99999.
        03 FILLER PIC X(5) VALUE SPACES.
        03 DL1L1F2 PIC XXXXXXXXXX.
        03 FILLER PIC X(5) VALUE SPACES.
        03 DL1L1F3 PIC  9999V.99.
        03 FILLER PIC X(5) VALUE SPACES.
```

PROCEDURE DIVISION code

```
PRINOH1.
    MOVE OH1L1 TO PRINTLINE.
    WRITE PRINTLINE AFTER ADVANCING 1 LINES.
    ADD 1 TO LINECOUNT.
    MOVE OH1L2 TO PRINTLINE.
    WRITE PRINTLINE AFTER ADVANCING 1 LINES.
    ADD 1 TO LINECOUNT.
PRINPH1.
PRINSH1.
PRINDL1.
    MOVE REMPNO1 TO DL1L1F1.
    MOVE REMPNAME1 TO DL1L1F2.
    MOVE RNETTPAY1 TO DL1L1F3.
    MOVE DL1L1 TO PRINTLINE.
    WRITE PRINTLINE AFTER ADVANCING 1 LINES.
    ADD 1 TO LINECOUNT.
PRINSF1.
PRINPF1.
PRINOF1.
EOR1.
```

These paragraphs simply have to be executed using the PERFORM statement in order to write the lines of the report belonging to the different categories.

5.7 REMAINING GENERATION OF FORMATS

Once the formats of files, information and reports have been produced, array FIELDS is scanned, and each field which has not got a pointer to PICTURES (i.e. a format) associated with it, is presented to the user for him to describe it, as follows :

GIVE THE FORMAT OF fieldname ENCLOSED IN QUOTES AND FOLLOWED BY ↑.

WORKING-STORAGE SECTION code is generated producing a 77 level number item for each field, and the format given is also stored in array PICTURES and a pointer set up to it in array FIELDS.

All remaining code generation is done automatically without any further information being required from the user. This consists of

(i) Generating a description of the file to which the information for all full and partial list reports is written when these reports are prepared. This code forms part of the FILE SECTION and has the form :

```
FD  REPORTFILE
      .
      .
      .
01  REPORTFILEREC[idno].
    02 IDNO[idno] PIC 99.
    02 R[contentname][idno]  PIC ...

    02 R[contentname][idno]  PIC ...
01   etc.
```

There will be one 01 level record description for each full and partial list report and PICTURE values are obtained from array PICTURES by using array FIELDS to access the appropriate values.

(ii) Generating a description of the sort file and sorted file for each full or partial list report that has to be sorted before being written. For each of these reports, the following is produced.

```
SD  RSORT[idno].
      .
      .
      .
01  RSORTREC[idno].
    02 SORT[contentname][idno] PIC ...
       .
       .
       .
    02 SORT[contentname][idno] PIC ...
```

and

```
FD  RSFILE[idno].
      :
      :
01  RSFILEREC[idno].
    02 RS[contentname][idno] PIC ...
         :
    02 RS[contentname][idno] PIC ...
```

Again, PICTURE values are obtained from array PICTURES via array FIELDS.

(iii) A description of the form of information required for editing is generated if editing is to be performed. This information will always consist of a card indicating :

(a) the edit operation - this may be deleting an employee record or, inserting a new record or altering a field of a record.

(b) the match field value - e.g. the employee number may be used to find the correct record on the master file.

(c) which field is to be altered - if the operation is that of altering a field.

In the case of insertion of a new record, this card is followed by the new values, or if a field is to be changed, a card containing its new value follows.

This description has the form :

```
FD  EDITINFO.
01  ERECA.
    02 EDITCODE  PIC X.
    02 EDITMATCH PIC X(20).   } description of first card
    02 EDITFIELD PIC X.
01  ERECB   PIC X(80).        used in insertion for reading
01  Efieldname   PIC ...      } there will be one of these lines
01  Efieldname   PIC ...      } for each field that may be altered
      :
```

(iv) Each indicator has a COBOL 77 level item of the form

77 indicatorname PIC 9. generated for the WORKING-STORAGE SECTION.

(v) Finally the array SFNAMES is used to generate the I-O-CONTROL paragraph for the FILE-CONTROL section. It has the form

I-O-CONTROL.

APPLY SORTINFO TO SORT-* ON sortfilename ... sortfilename.

SORTINFO is fixed and the description of its form and contents is produced by the Program Generator. It supplies information for the sort routines for the ICL COBOL compiler used at Rhodes University.

5.8 FORM OF FORMAT EDITOR OUTPUT

The code generated by the Format Editor is written away to four disc files as it is produced. It is produced in the order in which it will appear in the final program and consists of COBOL source statements with the necessary spaces for indentation (i.e. 7 or 11 spaces at the beginning of each statement) and the end of each line being marked by the character ↑ .

Four files are set aside to contain FILE-CONTROL, FILE SECTION, WORKING-STORAGE SECTION and PROCEDURE DIVISION code respectively.

One of the listing programs mentioned in section 2.6 enables the contents of these files to be listed with each statement on a new line.

5.9 ERROR MESSAGES

If the user forgets the terminator of '↑' in typing a specification, or the line exceeds 500 characters, the message "TERMINATOR MISSING OR LINE TOO LONG :- RETYPE LINE" will be printed on the console and he can retype his answer.

Errors occurring during the running of the Format Editor cause an error message containing an error number to be printed and the program to halt. A list of these error numbers and their meanings can be found in Appendix 5.

At the beginning of the Format Editor the user is asked to provide a value for a monitor flag (0 or 1) and the width of a lineprinter line in

characters. If he supplies a value of 1 for the monitor flag, a trace of all procedures entered in the course of the processing of the Format Editor will be listed.

6 THE PROGRAM GENERATOR

6.1 INTRODUCTION

The Program Generator is the program which is responsible for converting the flowchart skeleton produced by the Question-Answering Program into COBOL source statements and combining these with the formats produced by the Format Editor to generate the COBOL program which meets the requirements specified by the user. In addition to the information produced by the Question-Answering Program and the Format Editor, the Program Generator reads data which is used to generate certain fixed portions of the COBOL program depending on the values of flags passed across from the Question-Answering Program.

6.2 CONVERSION OF THE FLOWCHART SKELETON

The final program skeleton generated by the Question-Answering Program is written to a disc file, when all the relevant questions have been presented to the user and answered by him. The form in which this skeleton is output is the same as that in which it is stored during use in the Question-Answering Program i.e. each box of the skeleton is stored in one row of an array as shown below in Figure 6.1 (cf. section 4.2.3.)

box type	box number	parameter1 or -1	parameter2 or -1	parameter3 or -1	parameter4 or -1	parameter5 or -1	parameter6 or -1	link (pointer to next box or negative value if final box)	usage

Figure 6.1

and this array is output, row by row to the disc file for use by the Program Generator. As the parameter values, where present, are actually pointers to character strings stored in an array referred to as the character area in the Question-Answering Program, this array is written to the disc file before the skeleton is output, to enable the parameters to be replaced by their values during the conversion of the skeleton to COBOL source statements.

The conversion of the program skeleton to COBOL statements proved to be a simple task due to the fact that all the structures which were defined were directly translatable to COBOL statements. The COBOL statement equivalents of every box of every structure were prepared and these are read by the Program Generator as data. Conversion then consists of scanning the flowchart skeleton by following along the link pointers, starting at box or row one of the skeleton and converting each box as it is reached. To convert a flowchart skeleton box, the equivalent COBOL statement or string is accessed by using the box type and box number, and this string is written out on the lineprinter character by character. The COBOL statements do contain the following special characters, with the effects described below.

[indicates that 7 spaces are to be generated

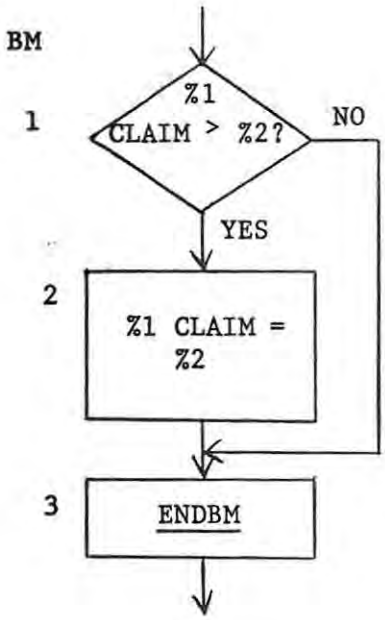
@ indicates that 11 spaces are to be generated

%i indicates that the value of parameter number i (i = positive integer) of the flowchart box being converted is to be generated. That is the string which it points to is to be accessed and written.

# indicates that the value of the usage number of the flowchart box being converted must be written.

↑ indicates that a new line is to be started.

An example of a FLOWCHART STRUCTURE and its COBOL statement equivalent is depicted below. For a complete representation of all the flowchart structures and their COBOL equivalents consult Appendix 2.



/@ IF %1 CLAIM IS NOT GREATER THAN %2 ↑@
THEN GO TO ENDBM#.↑]

/@ MOVE %2 TO %1 CLAIM.↑]

/[ENDBM#.↑]

Figure 6.2

The special character # is shown in this example. The purpose of this character and usage numbers is to ensure that paragraph names generated in the final COBOL program are unique. Each type of structure has associated with it a usage number counter, and this counter is incremented by one whenever the structure type is used in a flowchart expansion. When a structure type is incorporated into the flowchart skeleton, the current value of the usage number associated with the structure is stored in the flowchart skeleton as part of the information about each box. Thus, if the COBOL statement into which a flowchart box is to be converted contains a paragraph name, the name is extended by adding the usage number onto the end of it. This ensures that the paragraph name is unique. Whenever a structure may be used more than once for expansion of the flowchart skeleton, if its COBOL equivalent contains a paragraph name or a reference to a paragraph name (e.g. GOTO PARA) then the paragraph name will be followed by the character # signifying that the appropriate usage number must be used.

The COBOL statements which are associated with boxes that are likely to be expanded into further structures or which are likely to be used for insertion type expansion, are written in the form of COBOL comments. In expanding some of these boxes, the ← action is used, to enable the comment to

remain in the generated code to provide some documentation in the final program.

6.3 FORMAT EDITOR OUTPUT

The formats and code associated with input and output, which are produced by the Format Editor, are in the form of COBOL statements with ends of lines indicated by the character † . (cf. section 5.8). No processing of this data is necessary, other than the replacement of the character $ where it appears in FILE SECTION code by the appropriate blocksize. Otherwise this data simply has to be listed out at appropriate points during the program generation.

6.4 THE FIXED DATA

The flags passed across from the Question-Answering Program indicate whether :

a) There is periodic information

b) Editing of the payroll master file will be performed

c) Reconciliation is an option that was selected

d) Sorting is to take place

e) Verification is to be performed

f) A coinage breakdown is to be calculated

g) Cheques are to be printed

h) Reminders are to be produced.

The fixed data which is used by the Program Generator consists of COBOL statements necessary for the IDENTIFICATION DIVISION and the ENVIRONMENT DIVISION; for defining information necessary for sorting to proceed, such as workfile names etc. (i.e. SORTINFO cf. section 5.7), or defining preset tables (e.g. for PAYE calculations or coinage breakdown calculations); for defining output areas for error messages and for writing error messages.

Different parts of the fixed data are written at different points in

the generation process and the values of the flags are used to suppress the listing of fixed code concerned with options which have not been selected.

6.5 THE GENERATION PROCESS

The generation process follows the following order. Firstly the fixed data defining the IDENTIFICATION and ENVIRONMENT DIVISIONS is used to list these on the lineprinter. Then the file containing FILE-CONTROL code, produced by the Format Editor is read and listed. After this, the DATA DIVISION heading is written and the FILE SECTION code produced by the Format Editor is listed. After fixed WORKING-STORAGE SECTION code has been listed, the remainder of the WORKING-STORAGE section is generated using the file created by the Format Editor. After the fixed PROCEDURE DIVISION heading and initial paragraph have been written, the conversion of the flowchart to COBOL statements in the manner described and the listing of these statements occurs. Finally, concluding fixed paragraphs to be performed are listed and the generation process ends.

7 CONCLUSION

The most time-consuming part of this project was establishing the needs of payroll programs and designing the set of questions to obtain information for these needs and also to obtain user requirements. It is therefore unfortunate that the process involved in constructing the set of questions cannot be described formally. As all system design decisions were carefully thought out during the planning stages, the implementation of the system proceeded fairly rapidly and smoothly.

The programs comprising the system have all been thoroughly tested and found to work satisfactorily. Unfortunately, a problem arose which prevented a large program containing most of the facilities provided for by the questions, from being generated. This was due to the fact that the system was compiled using the Pascal Mark 1B ICL 1900 Compiler from the University of Glasgow, which cannot be extended to compile programs exceeding 32767 words. This was discovered at too late a stage to convert the system to be compiled using another version of the Pascal Compiler which can be extended. However, the system will be able to be converted fairly readily. This did not prevent a number of paths through the set of questions from being tested. An example of a generated program is provided in Appendix 7, and several further examples illustrating different paths and intermediate results are presented in Appendices 8 and 9. These paths were selected so that they do not require too large an array to store the final resulting flowchart, in order that the size of the Question-Answering Program remains under 32767 words.

It took approximately an hour to answer the questions and provide the format specification for the sample program provided in Appendix 7. From this it can be estimated that a full complex payroll program could be generated by a non-programmer in a matter of hours, which is a considerable improvement over supplied packages which can require years to set up parameters and hours to alter parameters if changes occur. The values of

of the mill time used to load and run the various programs in generating the sample program of Appendix 7 are as follows :

Question-Answering Program - 449.665 mill seconds

Format Editor - 41.019 mill seconds

Program Generator - 70.901 mill seconds.

The total time taken, both mill time and user time, therefore compares very favourably too with the time it would take actually to write the payroll program.

An extension that could be made to the system without too much difficulty is to add further questions concerning calculations to enable formulae to be obtained from the user and converted for use in COBOL COMPUTE statements. Otherwise, the system is very satisfactory. The question-answering interface with the user is simple to use and very effective; the system is capable of being extended to provide solutions for other classes of problems; the output of the Question-Answering Program is independent of the language of the final program as it is in the form of a flowchart skeleton. This means that programs in other languages could be produced by writing a simpler Format Editor and amending the Program Generator.

It is most encouraging that this system has reached the stage of automatically generating final programs in view of the fact that only one of three major automatic program generating systems currently in progress which are reviewed in the article by G.E. Heidorn[9] has reached the working stages, and a fourth project has been discontinued. Admittedly these projects are more ambitious in what they attempt, but it is hoped that this system will provide a small step in the direction of the goal of automatic generation of programs.

REFERENCES

1) Atkinson R.C. and Wilson H.A. "Computer-Assisted Instruction". Academic Press (1969).

2) Balzer R., Greenfield N., Kay M., Mann W., Ryder W., Wilczynski D. and Zobrist A. "Domain-independent Automatic Programming". Proceedings of IFIP Congress 71 (1974).

3) Cheatham T.E. and Wegbreit B. "A Laboratory for the Study of Automatic Programming". AFIPS Conf. Proc. 40 (1972).

4) Chai W.A. and Chai H.W. "Programming Standard Cobol" Academic Press (1976).

5) Dilloway C. "Q-Pac Payroll System." Software World. (April 1971).

6) Fuori W.M. "Introduction to the Computer The Tool of Business". Prentice Hall Inc. (1973).

7) Granholm J.W. "Parfit Payroll" reprinted from Datamation by the Agents marketing the Q-Pac Payroll System.

8) Hagamen W.D., Linden D.J., Mai K.F., Newell S.M. and Weber J.C. "A Program Generator". IBM Systems Journal 14 No. 2 (1975).

9) Heidorn G.E. "Automatic Programming Through Natural Language Dialogue: A Survey". IBM Journal Res. and Dev. 20 No. 11 (Nov. 1976).

10) Hilden J. "Guidelines for the design of Interactive Systems". The Computer Journal 19 No. 2 (May 1976).

11) Knuth D.E. "Computer Programming as an Art". Comm. ACM 17 No. 12 (1974).

12) Linder W. "Computer-Tutor : From a Student Project to a Self-Paced CAI/CMI Course". SIGCSE Bulletin 8 No. 3 (Sept. 1976).

13) Low D.W. "Programming by Questionnaire : An Effective Way to Use Decision Tables". Comm. ACM 16 No. 5 (1973).

14) McCracken D.D. and Garbassi U. "A guide to COBOL programming". Wiley Interscience. 2nd Edition. (1970).

15) Miller I.M. "Automatic System Customizer Configurator". IBM Technical Disclosure Bulletin 16 No. 7 (Dec. 1973).

16) Orilla L., Stern N.B. and Stern R.A. "Business Data Processing Systems". John Wiley and Sons Inc. (1972).

17) Peterson N.D. "Cobol Generation of Source Programs and Reports". Software-Practice and Experience 6 (1976).

18) Rin A. and Brown M. "An Overview of a System for Automatic Generation of File Conversion Programs". Software-Practice and Experience 5 (1975).

19) Sammet J.E. "The Use of English as a Programming Language". Comm. ACM 9 (March 1966).

20) Sammet J.E. "Programming Languages : History and Fundamentals". Prentice Hall (1969).

21) van Leer P. "Top-down development using a program design language". IBM Systems Journal 15 No. 2 (1976).

22) Williams M.H. "A Formal Notation for Specifying Syntax Interpretation Rules". Submitted for publication to the Computer Journal.

A P P E N D I X 1

Numeric codes used to represent actions

Code value	Function
1	represents the action ⇑
2	represents the action ⇓
3	marks the start of the action $\mathcal{M}$
4	separates the two global stacks used in the $\mathcal{M}$ action
5	marks the end of the $\mathcal{M}$ action
6	marks the start of the $\mathcal{A}$ action
7	represents the reserved variable outcount
8	used in $\mathcal{S}$ action to indicate that the item to be searched for is to follow
9	indicates the start of the action Analyse
10	indicates that the second local stack used in the action Analyse is to follow
11	indicates that the third local stack used in the action Analyse is to follow
12	preceeds the logical variable used in the action Analyse
13	marks the end of the Analyse action
14	represents the reserved variable loopcount
15	indicates that a real constant is to follow
16	indicates that the name of a local stack of type integer is to follow
20	marks the beginning of the $\mathcal{F}$ action
21	indicates that a typebox variable is to follow
22	indicates that the action part of an action is to follow
23	marks the end of the $\mathcal{F}$ action
24	marks beginning of the $\forall$ action

Code value	Function
25	marks the end of the $\forall$ action
26	marks the start of the @ action
27	indicates that a character constant is to follow
28	used as a separator of typebox variables in a box type list
29	indicates that a string constant is to follow
30	represents the → action
31	represents the ← action
32	represents the ↓ action
33	represents the reserved variable ans
34	represents the reserved variable box
35	represents and
36	represents or
38	marks the start of the access operation within the $\mathcal{G}$ or @actions
39	indicates that the name of a local stack of type character is to follow
40	marks the start of a parameter list (may be used with functions or procedures or on the right hand sides of flowchart expansion operations)
41	separates two parameters in a parameter list
42	marks the end of a parameter list
43	marks the start of the $\mathcal{I}$ action
44	marks the end of the $\mathcal{I}$ action
45	marks the start of the $\mathcal{O}$ action
46	marks the end of the $\mathcal{O}$ action
47	represents the reserved variable outpointer
48	marks the start of the $\mathcal{S}$ action
49	marks the beginning of the success action within the $\mathcal{S}$ action

Code value	Function
50	marks the beginning of the fail action within the $\mathcal{S}$ action
51	marks the end of the $\mathcal{S}$ action
52	marks the beginning of the $\mathcal{R}$ action
53	used in the $\mathcal{R}$, $\mathcal{A}$ or $\mathcal{G}$ actions to indicate that the number of the first global stack is to follow
54	used in the $\mathcal{R}$ or $\mathcal{A}$ actions to indicate that the number of the second global stack is to follow
55	marks the end of the $\mathcal{R}$ action
56	marks the end of the $\mathcal{A}$ action
57	represents the null value
59	represents *
60	indicates that a parameter indicator is to follow
61	indicates that a global stack number is to follow
62	indicates that a conversion table number is to follow
63	represents <u>if</u>
64	represents <u>then</u>
65	represents <u>else</u>
66	represents <u>fi</u>
67	marks the end of the @ action
68	represents the <u>goto</u> action
69	represents the $\Leftarrow$ action
70	represents the $\uparrow$ action
71	represents the logical value <u>true</u>
72	represents the logical value <u>false</u>
73	marks the start of the $\mathcal{B}$ action
74	represents the null action, -
75	represents the reserved variable searchpointer
76	indicates that a function or procedure name is to follow

Code value	Function
77	represents the relational operator =
78	represents the relational operator >
79	represents the relational operator <
81	represents ;
82	marks the end of the *b* action
83	represents the relational operator >=
84	represents the relational operator <=
85	represents the relational operator #
86	indicates that the name of a variable of type pointer is to follow
87	marks the start of the action *l*
88	indicates that an integer constant is to follow
90	indicates that the name of a variable of type string is to follow
91	indicates that the name of a variable of type character is to follow
92	indicates that the name of a variable of type integer is to follow
93	indicates that the name of a variable of type real is to follow
94	indicates that the name of a variable of type logical is to follow
95	marks the end of the action *l*
96	marks the start of the action *g*
97	represents the reserved variable forallcount
98	represents the reserved variable foreachcount
99	marks the end of the action *g*
100	marks the start of the action component of the data for a question
101	marks the end of the action component of the data for a question

Code value	Function
200	marks the start of the expected answer component of the data for a question
201	marks the end of the expected answer component of the data for a question

APPENDIX 2

The initial program skeleton, flowchart structures and COBOL code corresponding to them.

This appendix provides a diagrammatic representation of each flowchart structure, including the initial program skeleton, as well as the COBOL code associated with each box of each structure.

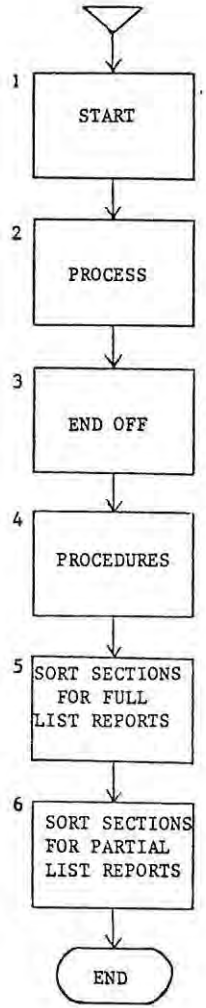
In the structure diagrams, the character % followed by a number marks the positions and numbers of parameters in each box. Also, any text which is underlined will constitute a paragraph name in the final generated program unless the box in which it occurs is eliminated during a flowchart expansion operation.

In the listing of COBOL code, the character [represents 7 spaces, the character @ represents 11 spaces and ↑ represents the end of a line. The code for a box is started with the character / and is terminated by] .

DIAGRAMMATIC REPRESENTATION

CORRESPONDING CODE

A



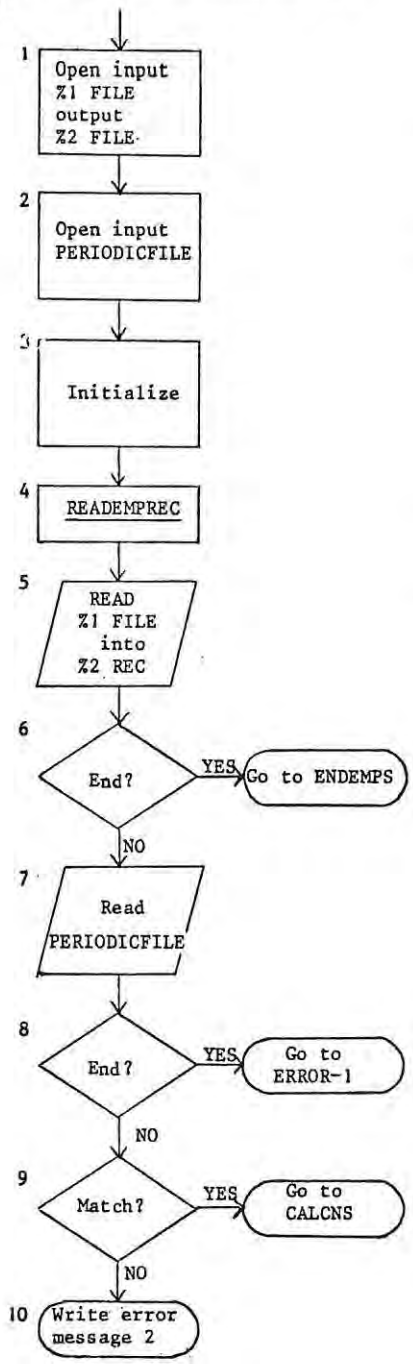

```
/[ START.↑]
/[       *** PROCESS**↑]
/[ENDOFF.↑@CLOSE PRINTFILE.↑@STOP RUN.↑]
/[ PERFORMPARS.↑]
/[       ** SORT SECTIONS FOR FULL LISTS **↑]
/[       ** SORT SECTIONS FOR PARTIAL LISTS** ↑]
```

Structure A is the initial flowchart skeleton.

DIAGRAMMATIC REPRESENTATION

CORRESPONDING CODE

B

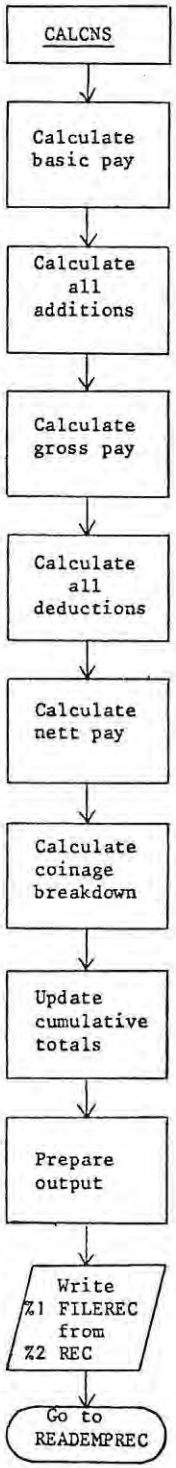



```
/@OPEN INPUT %1 FILE OUTPUT %2 FILE.↑ ]
/@OPEN INPUT PERIODICFILE.↑ ]
/[ INITIALIZE.↑ ]
/[ READEMPREC.↑ ]
/@READ %1 FILE INTO %2 REC↑ ]
/@     AT END GO TO ENDEMPS.↑]
/@READ PERIODICFILE ↑ ]
/@     AT END GO TO ERROR-1.↑ ]
/      ** MATCH **↑]
/@ THEN GO TO CALCNS.↑@PERFORM ERROR-2.↑@
 GO TO READEMPREC.↑ ]
```

DIAGRAMMATIC REPRESENTATION

CORRESPONDING CODE

B (continued)

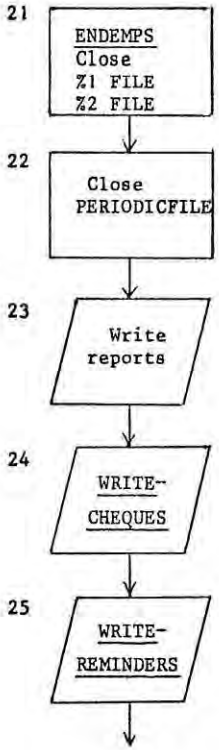



```
/[ CALCNS.↑]

/       ** CALCULATE BASIC PAY **↑]

/       ** CALCULATE ALL ADDITIONS **↑]

/       ** CALCULATE GROSS PAY **↑]

/       ** CALCULATE ALL DEDUCTIONS **↑]

/       ** CALCULATE NETT PAY **↑]

/       **CALCULATE COINAGE BREAKDOWN **↑]

/       ** UPDATE CUMULATIVE TOTALS **↑]

/       ** PREPARE OUTPUT **↑]

/@WRITE %1 FILEREC FROM %2 REC.↑@GO TO
READEMPREC. ↑]
```

DIAGRAMMATIC REPRESENTATION

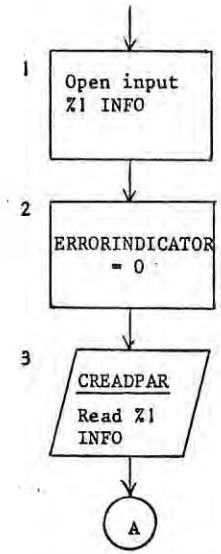
CORRESPONDING CODE

B (continued)




```
/[ENDEMPS.↑@CLOSE %1 FILE %2 FILE.↑]

/@CLOSE PERIODICFILE.↑ ]

/       ** WRITE REPORTS **↑ ]

/[WRITECHEQUES.↑ ]

/[WRITEREMINDERS.↑ ]
```

C

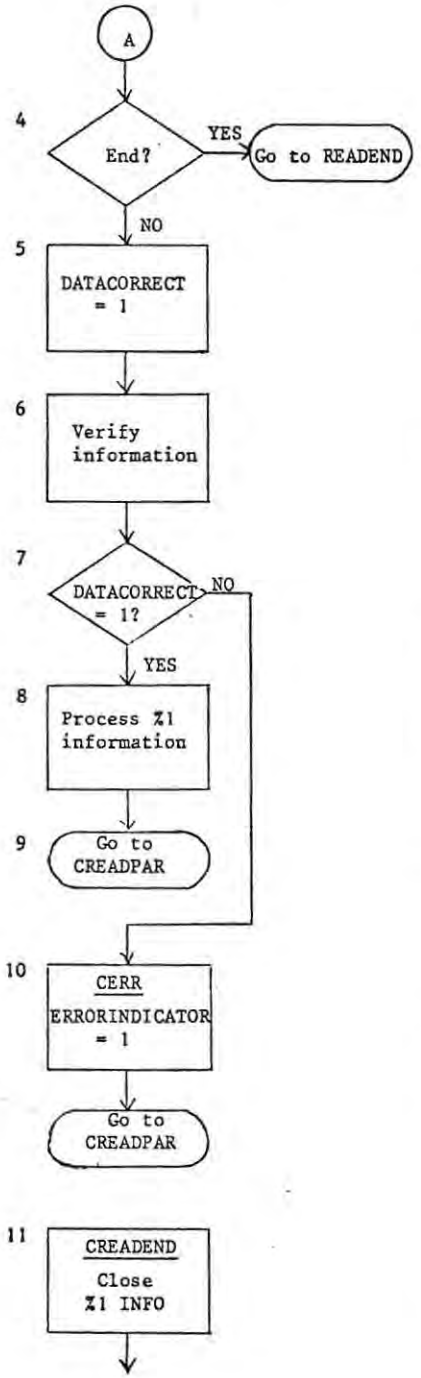



```
/@OPEN INPUT %1 INFO.↑ ]

/@MOVE ZERO TO ERRORINDICATOR.↑ ]

/[CREADPAR.↑@READ %1 INFO :↑]
```

DIAGRAMMATIC REPRESENTATION

CORRESPONDING CODE

C (continued)



/@ AT END GO TO CREADEND.↑]

/@MOVE 1 TO DATACORRECT.↑]

/ ** VERIFY INFORMATION **↑]

/@IF DATACORRECT IS NOT EQUAL TO 1 ↑@THEN GO TO CERR.↑]

/ ** PROCESS %1 INFORMATION **↑]

/@GO TO CREADPAR.↑]

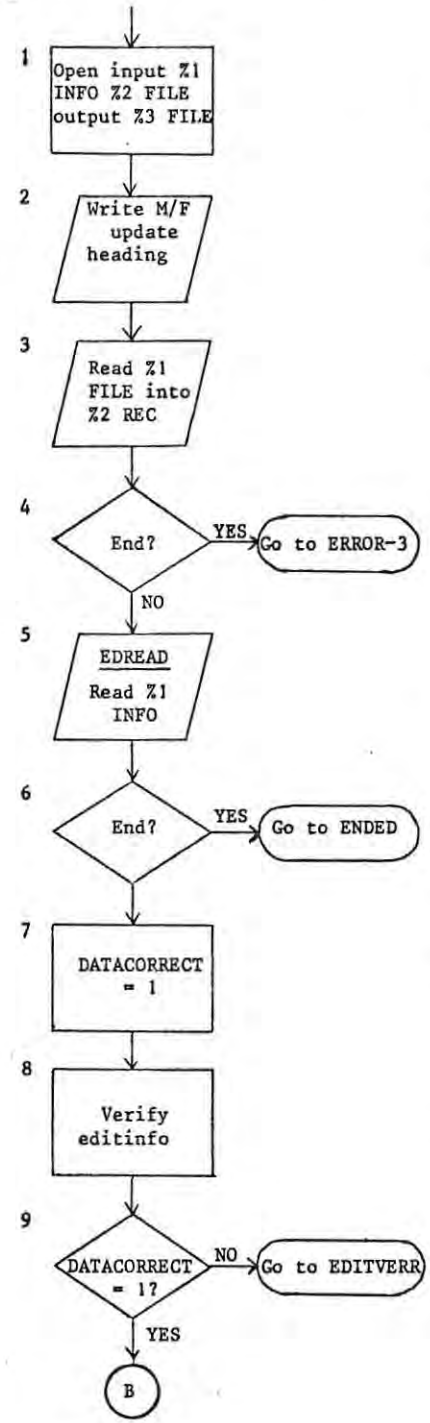
/[CERR.↑@MOVE 1 TO ERRORINDICATOR.↑@GO TO CREADPAR.↑]

/[CREADEND.↑@CLOSE %1 INFO.↑]

DIAGRAMMATIC REPRESENTATION

CORRESPONDING CODE

D



/@OPEN INPUT %1 INFO %2 FILE OUTPUT %3 FILE.↑]

/@MOVE MFUPDATEHEAD TO PRINTLINE.↑@WRITE PRINTLINE AFTER CHANNEL-1.↑@MOVE MFUPDATELINE TO PRINTLINE.↑@PERFORM WRITELINE.↑@WRITE PRINTLINE AFTER 2 LINES.]

/@READ %1 FILE INTO %2 REC ↑]

/@ AT END GO TO ERROR-3.↑]

/[EDREAD. @READ %1 INFO ↑.]

/@ AT END GO TO ENDED.↑]

/@MOVE 1 TO DATACORRECT.↑]

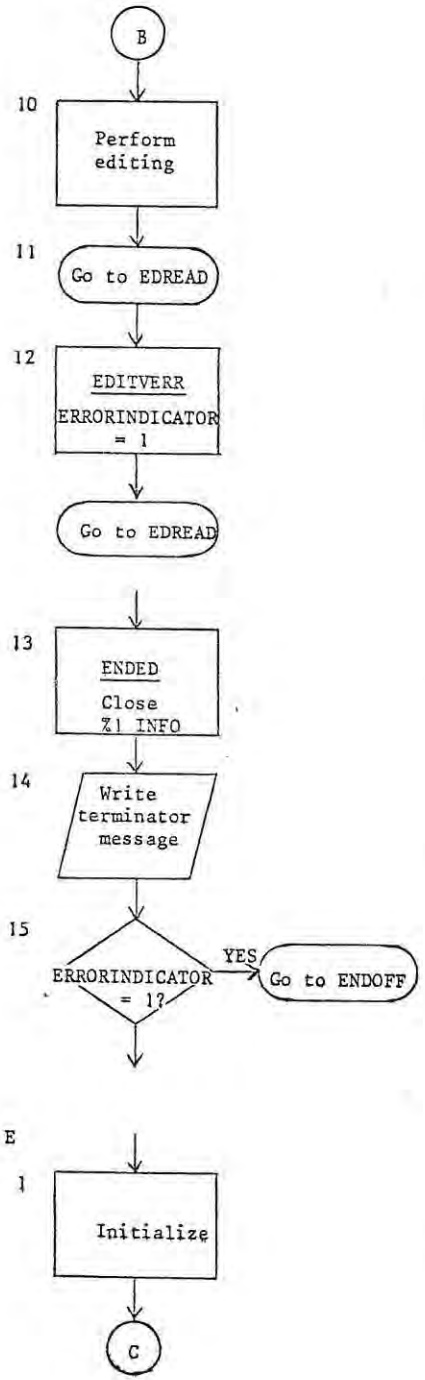
/ ** VERIFY EDITINFO **↑]

/@IF DATACORRECT IS NOT EQUAL TO 1 @↑ THEN GO TO EDITVERR.↑]

DIAGRAMMATIC REPRESENTATION

CORRESPONDING CODE

D (continued)

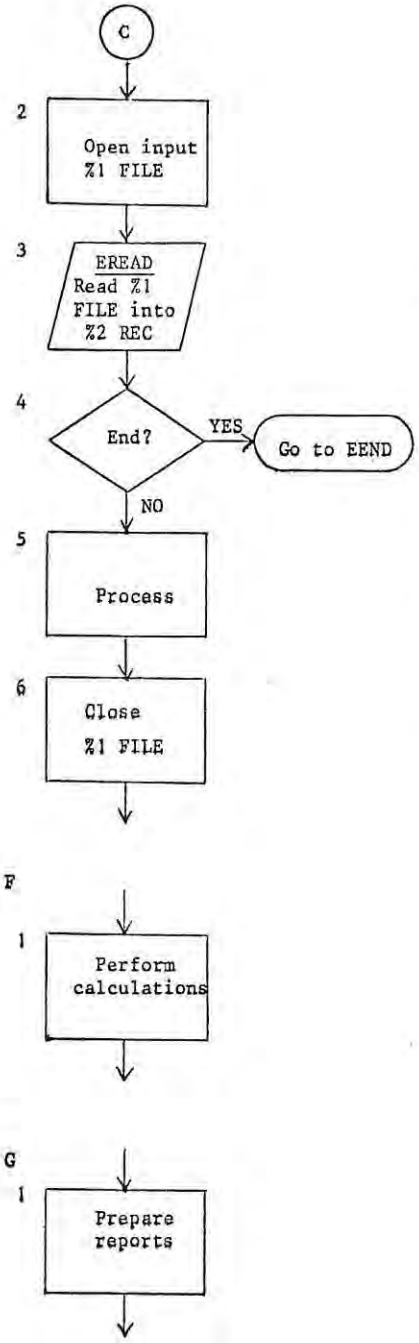



```
/       ** PERFORM EDITING**↑]

/ @GO TO EDREAD.↑]

/[ EDITVERR.↑@ MOVE 1 TO ERRORINDICATOR.↑@
  GO TO EDREAD.↑]

/[ENDED.↑@CLOSE %1 INFO.↑]

/@MOVE "END OF UPDATE" TO PRINTLINE.↑@
 PERFORM WRITELINE.↑]

/@IF ERRORINDICATOR IS EQUAL TO 1 THEN
GO TO ENDOFF.↑]

/       ** INITIALIZE **↑]
```

DIAGRAMMATIC REPRESENTATION

CORRESPONDING CODE

E (continued)




```
/@OPEN INPUT %1 FILE.↑]

/ [EREAD.↑@ READ %1 FILE INTO %2 REC ↑ ]

/@     AT END GO TO EEND.↑]

/       ** PROCESS **↑]

/[EEND.↑@CLOSE %1 FILE↑↑ ]

/       ** PERFORM CALCULATIONS **↑]

/       ** PREPARE REPORTS **↑]
```

DIAGRAMMATIC REPRESENTATION | CORRESPONDING CODE

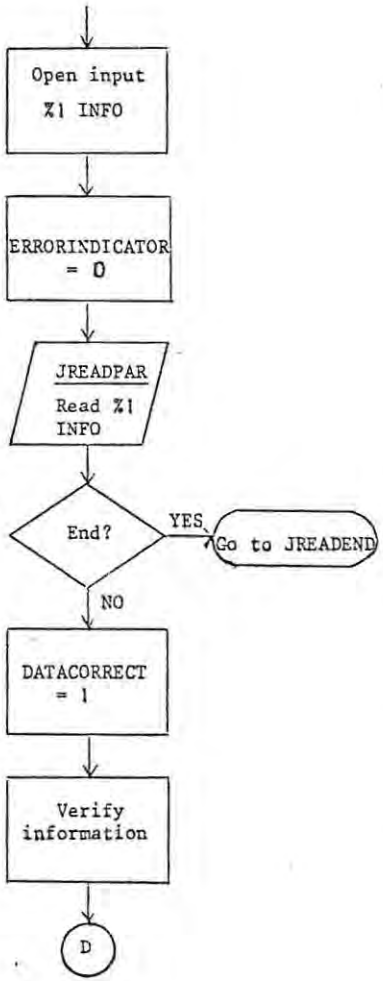
H

/ ** WRITE REPORTS **†]

I

As for structure C with CREADPAR replaced by IREADPAR, CERR by IERR and CREADEND by IREADEND.

J



/@OPEN INPUT %1 INFO.†]

/@MOVE ZERO TO ERRORINDICATOR.†]

/[JREADPAR.†@READ %1 INFO †]

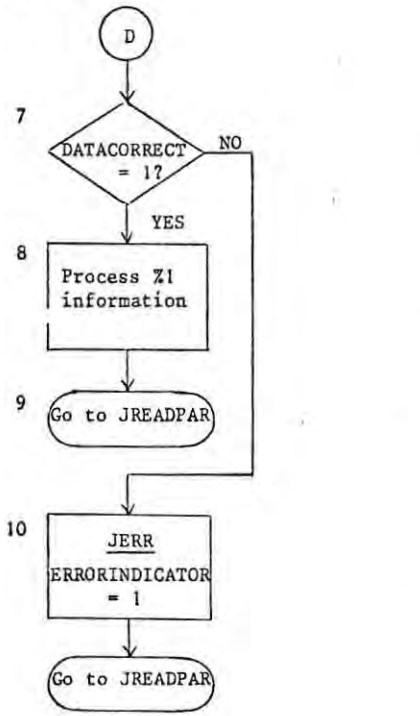
/@ AT END GO TO JREADEND.†]

/@MOVE 1 TO DATACORRECT.†]

/ ** VERIFY INFORMATION **†]

DIAGRAMMATIC REPRESENTATION | CORRESPONDING CODE

J (continued)

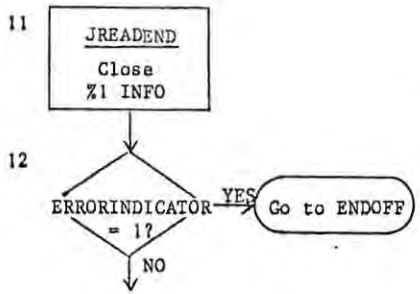


/@IF DATACORRECT IS NOT EQUAL TO 1 †@ THEN GO TO JERR.†]

/ ** PROCESS % 1 INFORMATION **†]

/@GO TO JREADPAR.†]

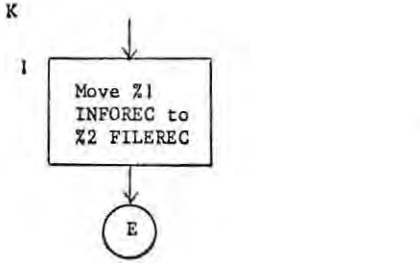
/[JERR.†@MOVE 1 TO ERRORINDICATOR.†@GO TO JREADPAR.†]



/[JREADEND.†@CLOSE %1 INFO.†]

/@IF ERRORINDICATOR IS EQUAL TO 1 THEN GO TO ENDOFF.†]

K

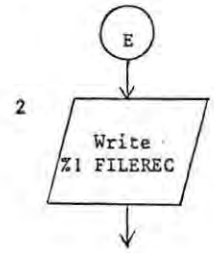


/@MOVE CORRESPONDING %1 INFOREC TO %2 FILEREC.†]

DIAGRAMMATIC REPRESENTATION

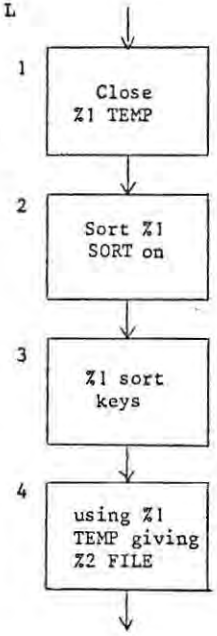
CORRESPONDING CODE

K (continued)



/@WRITE %1 FILEREC.†]

L



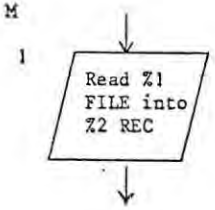
/@CLOSE %1 TEMP.†]

/@SORT %1 SORT ON †]

/ ** %1 SORT KEYS **†]

/@ USING %1 TEMP GIVING %2 FILE.†]

M

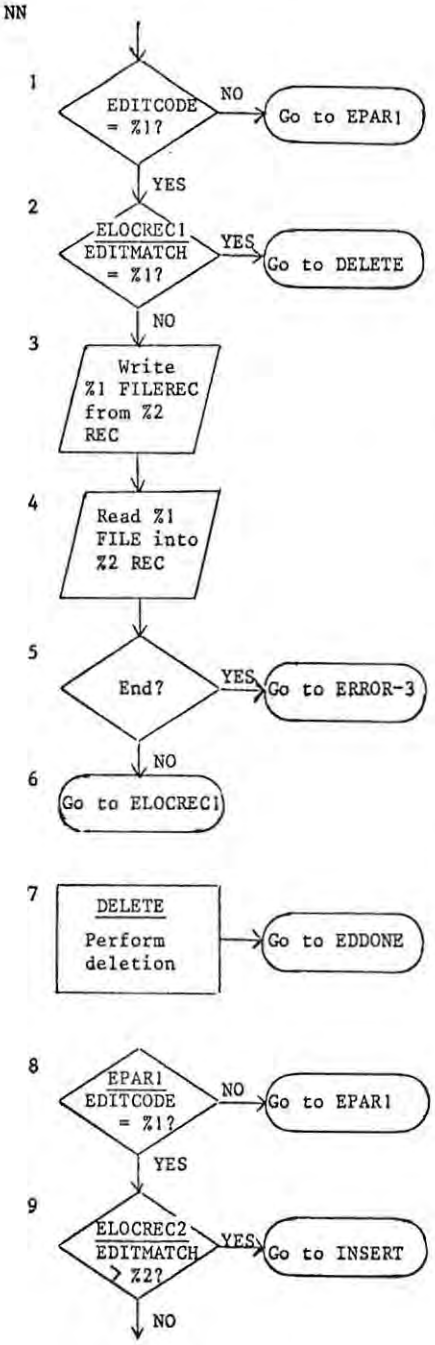


/@READ %1 FILE INTO %2 REC †]

DIAGRAMMATIC REPRESENTATION

CORRESPONDING CODE

NN



/@IF EDITCODE IS NOT EQUAL TO %1 THEN GO TO EPAR1.†]

/[ELOCREC1.†@IF EDITMATCH IS EQUAL TO %1 THEN GO TO DELETE.†]

/@WRITE %1 FILEREC FROM %2 REC.†]

/@READ %1 FILE INTO %2 REC †]

/@ AT END GO TO ERROR-3.†]

/@GO TO ELOCREC1.†]

/[DELETE.†@MOVE 1 TO DELETEDINDIC.†@MOVE EDITMATCH TO DELMATCH.†@MOVE DELETEMESS TO PRINTLINE.†@PERFORM WRITELINE.†@GO TO EDDONE.†]

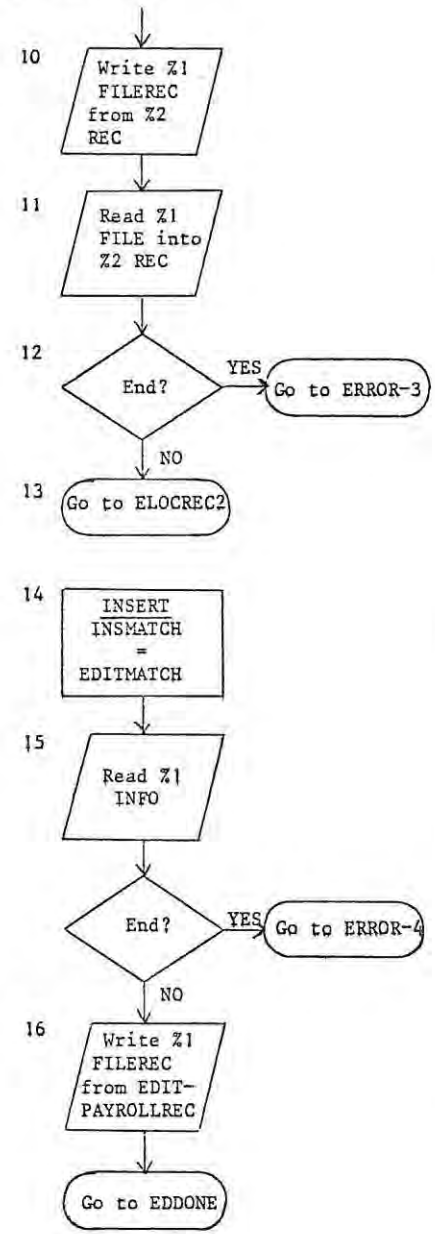
/[EPAR1.†@IF EDITCODE IS NOT EQUAL TO %1 THEN GO TO EPAR2.†]

/[ELOCREC2.†@IF EDITMATCH IS GREATER THAN %1 THEN GO TO INSERT.†]

DIAGRAMMATIC REPRESENTATION

CORRESPONDING CODE

NN (continued)

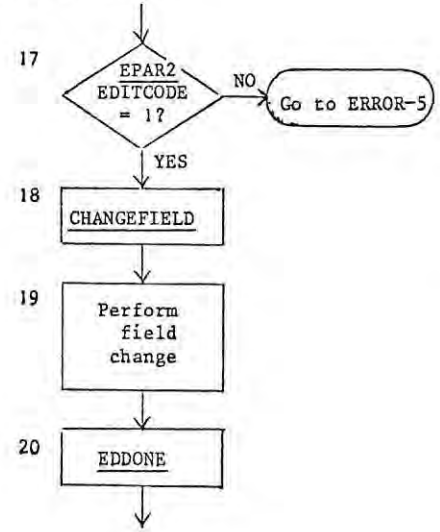



```
/@WRITE %1 FILEREC FROM %2 REC.↑]

/@READ %1 FILE INTO %2 REC ↑]

/@     AT END GO TO ERROR-3.↑]

/@GO TO ELOCREC2.↑]

/[INSERT.↑@MOVE EDITMATCH TO INSMATCH.↑]

/@READ %1 INFO AT END GO TO ERROR-4.↑]

/@WRITE %1 FILEREC FROM EDITPAYROLLREC. @↑
  MOVE INSMESS TO PRINTLINE.↑@PERFORM
WRITELINE.↑@GO TO EDDONE. ↑]
```

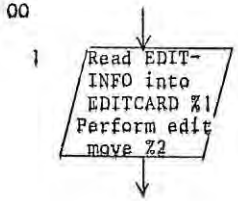
DIAGRAMMATIC REPRESENTATION

CORRESPONDING CODE

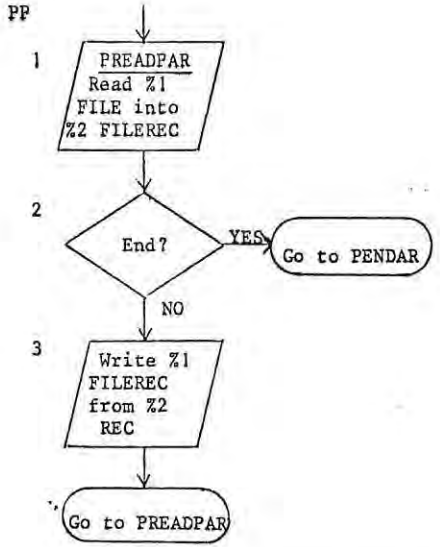
NN (continued)




```
/[EPAR 2.↑@IF EDITCODE IS NOT EQUAL TO %1
THEN GO TO ERROR-5.↑]

/[CHANGEFIELD.↑]

/      ** CHANGE FIELD **↑]

/[EDDONE.↑]
```




```
/@READ EDITINFO INTO EDITCARD %1.↑@PERFORM
EDMOVE %1.↑]
```

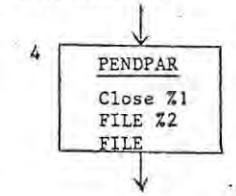



```
/[PREADPAR.↑@READ %1 FILE INTO %2 FILEREC↑]

/@     AT END GO TO PENDPAR.↑]

/@WRITE %1 FILEREC FROM %2 REC.↑@GO TO
PREADPAR.↑]
```

DIAGRAMMATIC REPRESENTATION

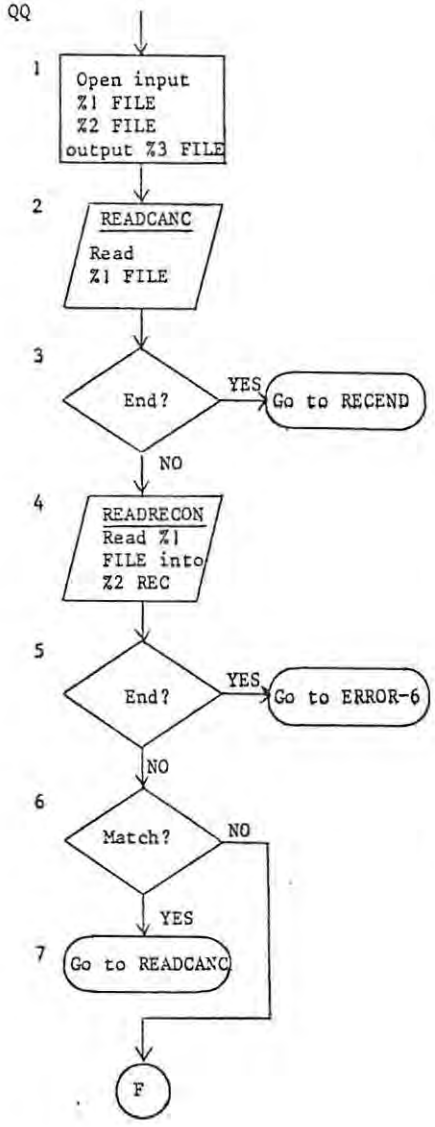
CORRESPONDING CODE

PP (continued)




```
/[PENDPAR.↑@CLOSE %1 FILE %2 FILE.↑]
```

QQ

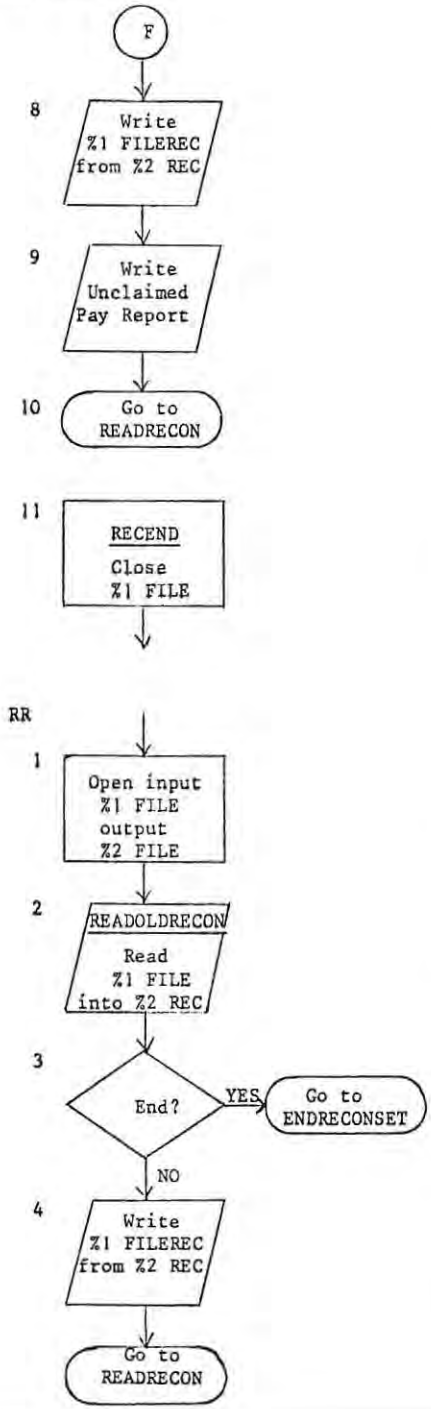



```
/@OPEN INPUT %1 FILE %2 FILE OUTPUT %3
FILE.↑[RECONCILE.↑]

/[READCANC.↑@READ %1 FILE.↑]

/@     AT END GO TO RECEND.↑]

/[READRECON.↑@READ %1 FILE INTO %2 REC.↑]

/@     AT END GO TO ERROR-6.↑]

/      ** IF MATCH **]

/@  THEN GO TO READCANC.↑]
```

DIAGRAMMATIC REPRESENTATION

CORRESPONDING CODE

QQ (continued)

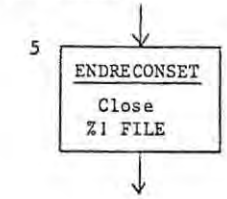



```
/@WRITE %1 FILEREC FROM %2 REC.↑]

/      ** WRITE UNCLAIMED PAY REPORT **↑]

/@GO TO READRECON.↑]

/[RECEND.↑@CLOSE %1 FILE.↑]

/@OPEN INPUT %1 FILE OUTPUT %2 FILE.↑]

/[READOLDRECON.↑@READ %1 FILE INTO %2 REC↑]

/@     AT END GO TO ENDRECONSET.↑]

/@WRITE %1 FILEREC FROM %2 REC.↑@GO TO
READRECON.↑]
```

DIAGRAMMATIC REPRESENTATION

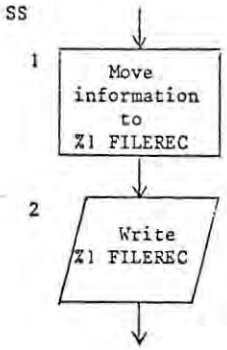
CORRESPONDING CODE

RR (continued)



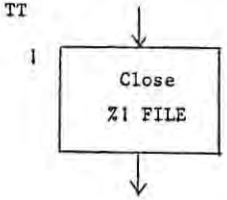

```
/[ENDRECONSET.↑@CLOSE %1 FILE.↑ ]
```

SS



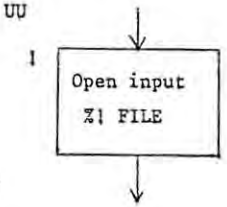

```
/      ** MOVE INFORMATION TO %1 FILEREC. **↑]

/@WRITE %1 FILEREC.↑]
```

TT



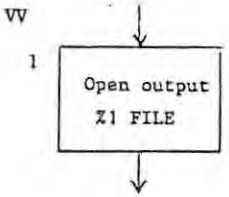

```
/@CLOSE %1 FILE.↑]
```

UU




```
/@OPEN INPUT %1 FILE.↑]
```

VV

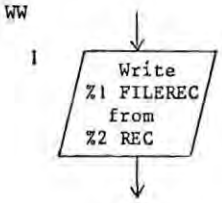



```
/@OPEN OUTPUT %1 FILE.↑]
```

DIAGRAMMATIC REPRESENTATION

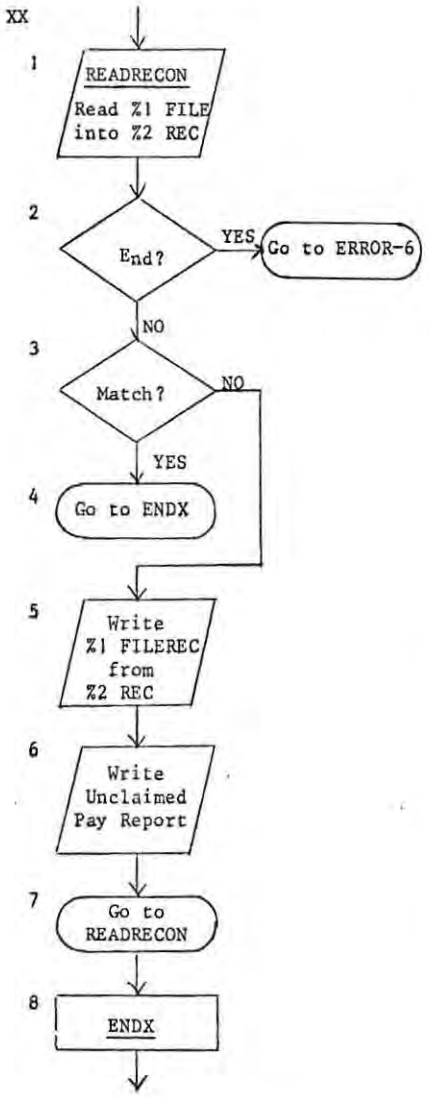
CORRESPONDING CODE

WW




```
/@WRITE %1 FILEREC FROM %2 REC.↑ ]
```

XX




```
/[READRECON.↑@READ %1 FILE INTO %2 REC ↑]

/@    AT END GO TO ERROR-6.↑]

/      ** IF MATCH **↑]

/@  THEN GO TO ENDX.↑]

/@WRITE %1 FILEREC FROM %2 REC.↑ ]

/      ** WRITE UNCLAIMED PAY REPORT **↑]

/@GO TO READRECON.↑]

/[ENDX.↑]
```

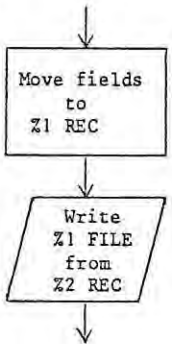
DIAGRAMMATIC REPRESENTATION

CORRESPONDING CODE

YY

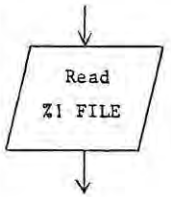

```
/      ** WRITE UNCLAIMED PAY REPORT **↑]
```

ZZ



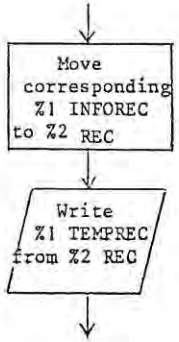

```
/      ** MOVE FIELDS TO %1 REC ** ↑]

/@WRITE %1 FILE FROM %2 REC.↑ ]
```

AA




```
/@READ %1 FILE.↑]
```

AB

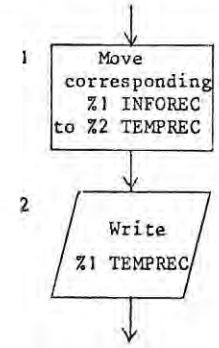



```
/@MOVE CORRESPONDING %1 INFOREC TO
%2 REC.↑]

/@WRITE %1 TEMPREC FROM %2 REC. ↑]
```

DIAGRAMMATIC REPRESENTATION

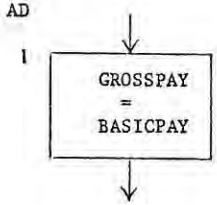
CORRESPONDING CODE

AC



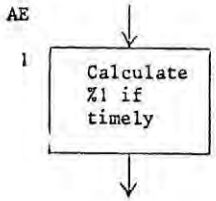

```
/@MOVE CORRESPONDING %1 INFOREC TO
%2 TEMPREC.↑]

/@WRITE %1 TEMPREC.↑ ]
```

AD



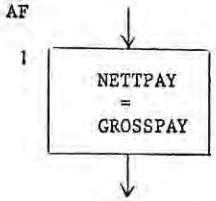

```
/@MOVE BASICPAY TO GROSSPAY.↑]
```

AE



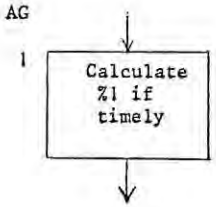

```
/      ** CALCULATE %1 IF TIMELY **↑]
```

AF




```
/@MOVE GROSSPAY TO NETTPAY.↑ ]
```

AG

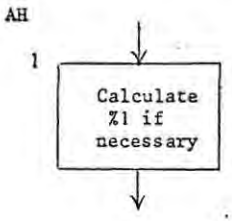



```
/      ** CALCULATE %1 IF TIMELY **↑]
```

DIAGRAMMATIC REPRESENTATION

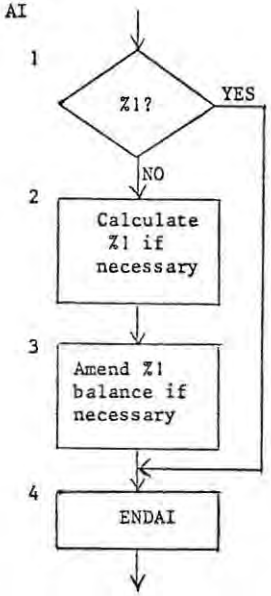
CORRESPONDING CODE

AH



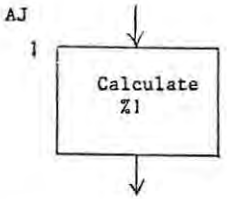

```
/      ** CALCULATE %1 IF NECESSARY **↑]
```

AI



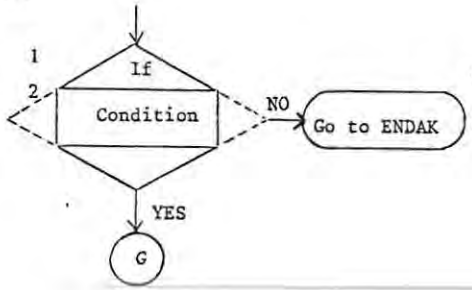

```
/@IF %1 THEN GO TO ENDAI#.↑]

/      ** CALCULATE %1 IF NECESSARY **↑]

/      ** AMEND %1 BALANCE IF NECESSARY **↑]

/[ENDAI#.↑]
```

AJ




```
/      ** CALCULATE %1  **↑]
```

AK

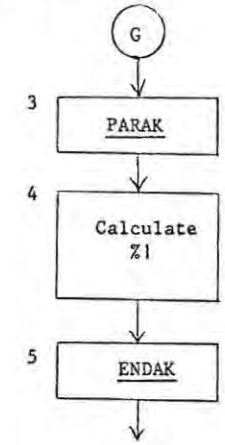



```
/@IF ↑]

/      ** CONDITION **↑]
```

DIAGRAMMATIC REPRESENTATION

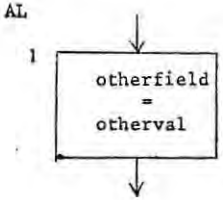
CORRESPONDING CODE

AK (continued)



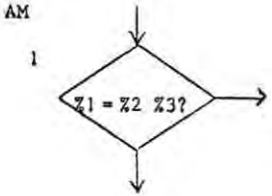

```
/@  THEN GO TO PARAK# ↑@ELSE GO TO
ENDAK#.↑[PARAK#.↑]

/      ** CALCULATE %1 ** ↑]

/[ENDAK#.↑]
```

AL



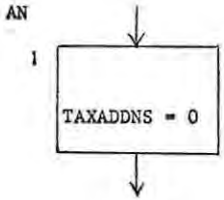

```
/      ** OTHERFIELD = OTHERVAL  **↑]
```

AM




```
/ %1 IS EQUAL TO %2 %3 ]
```

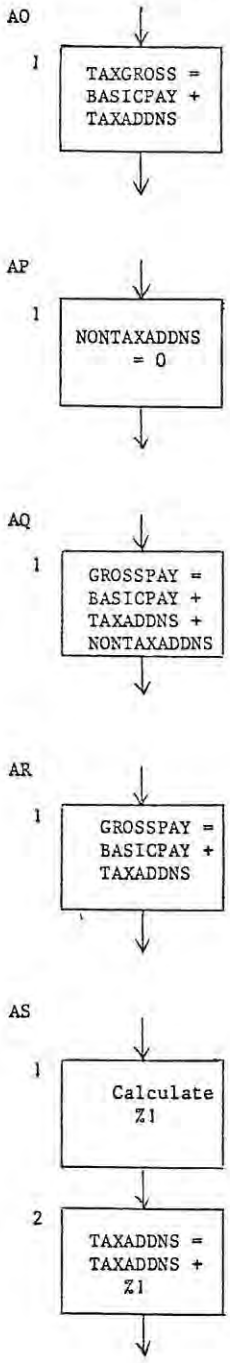
AN




```
/@MOVE ZERO TO TAXADDNS. ↑]
```

DIAGRAMMATIC REPRESENTATION

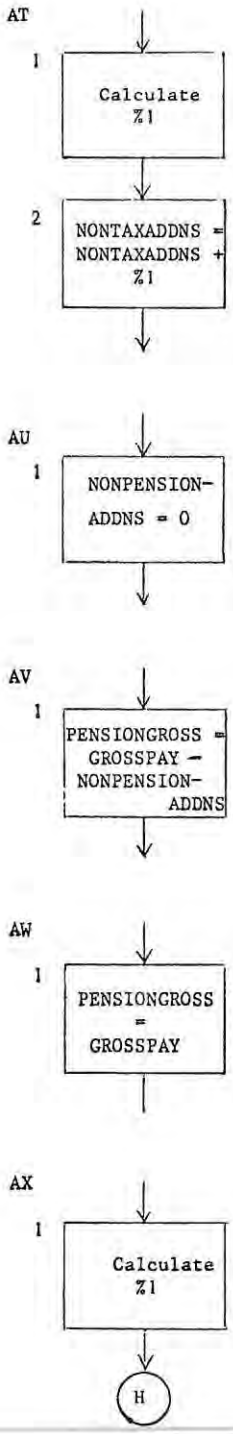
CORRESPONDING CODE




```
/@ADD TAXADDNS BASICPAY GIVING TAXGROSS.↑]

/@MOVE ZERO TO NONTAXADDNS.↑ ]

/@ADD BASICPAY TAXADDNS NONTAXADDNS
GIVING GROSSPAY.↑ ]

/@ADD BASICPAY TAXADDNS GIVING GROSSPAY.↑]

/      ** CALCULATE %1 **↑]

/@ADD %1 TO TAXADDNS.↑]
```

DIAGRAMMATIC REPRESENTATION

CORRESPONDING CODE

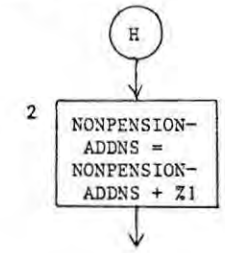



```
/      ** CALCULATE %1 **↑]

/@ADD %1 TO NONTAXADDNS.↑]

/@MOVE ZERO TO NONPENSIONADDNS.↑]

/@SUBTRACT NONPENSIONADDNS FROM GROSSPAY
GIVING PENSIONGROSS.↑]

/@MOVE GROSSPAY TO PENSIONGROSS.↑ ]

/      ** CALCULATE %1 **↑]
```

DIAGRAMMATIC REPRESENTATION

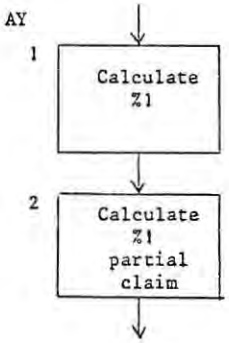
CORRESPONDING CODE

AX (continued)



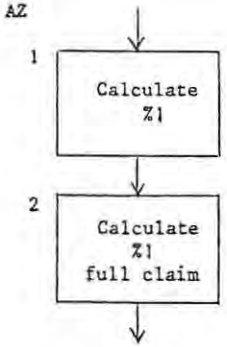

```
/@ADD %1 TO NONPENSIONADDNS.†]
```

AY



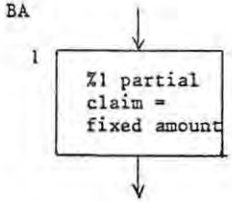

```
/      ** CALCULATE %1 **†]

/      ** CALCULATE %1 PARTIAL CLAIM  **†]
```

AZ




```
/      ** CALCULATE %1 **†]

/      ** CALCULATE %1 FULL CLAIM  **†]
```

BA

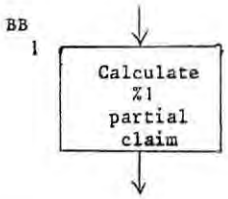



```
/      ** %1 PARTIAL CLAIM = FIXED AMOUNT **†]
```

DIAGRAMMATIC REPRESENTATION

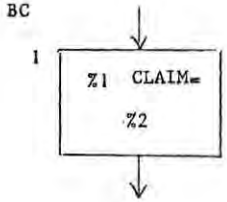
CORRESPONDING CODE

BB



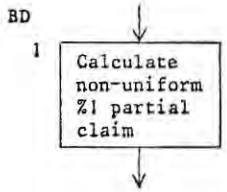

```
/      ** CALCULATE %1 PARTIAL CLAIM **†]
```

BC



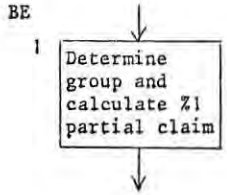

```
/@MOVE %2 TO %1 CLAIM.†]
```

BD



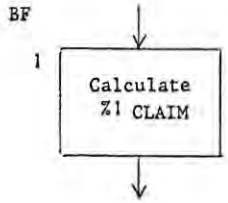

```
/      ** CALCULATE NON-UNIFORM %1
PARTIAL CLAIM **†]
```

BE




```
/      ** DETERMINE GROUP AND CALCULATE
%1 PARTIAL CLAIM USING GROUP METHOD **†]
```

BF

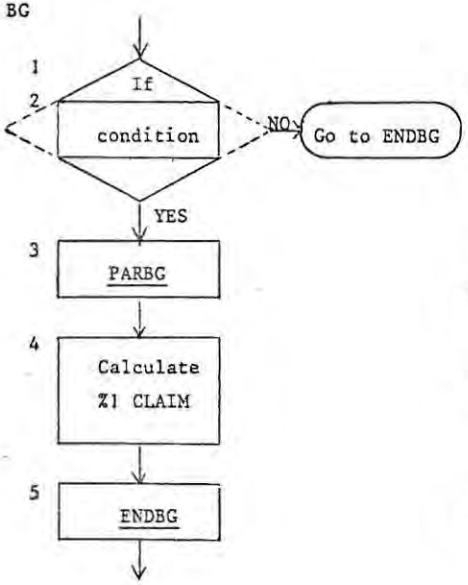



```
/      ** CALCULATE %1 CLAIM **†]
```

DIAGRAMMATIC REPRESENTATION

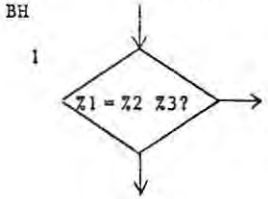
CORRESPONDING CODE

BG



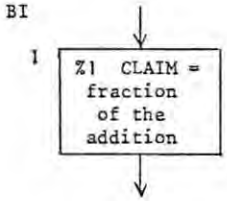

```
/@IF †]
/      ** CONDITION **†]
/@  THEN GO TO PARBG# †@ ELSE GO TO
ENDBG#.†[PARBG#.†]
/      ** CALCULATE %1 CLAIM **†]
/[ENDBG#.†]
```

BH




```
/@  %1 IS EQUAL TO %2 %3  ]
```

BI



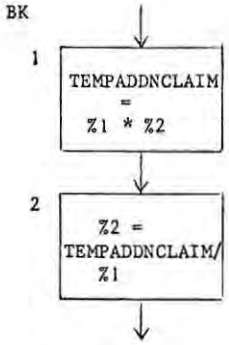

```
/      ** %1 CLAIM = FRACTION OF THE
ADDITION **†]
```

BJ : As for BI above.

DIAGRAMMATIC REPRESENTATION

CORRESPONDING CODE

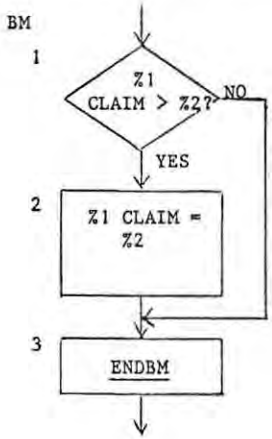
BK




```
/@MULTIPLY %1 BY %2 GIVING TEMPADDNCLAIM.†]
/@DIVIDE TEMPADDNCLAIM BY %1 GIVING
%2 CLAIM.†]
```

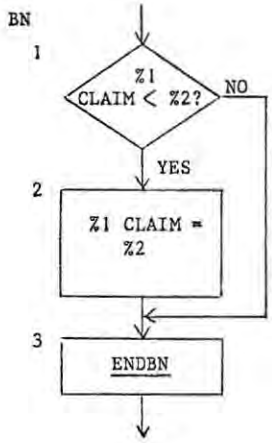
BL : As for BK above.

BM




```
/@ IF %1 CLAIM IS NOT GREATER THAN %2 †@
THEN GO TO ENDBM#.† ]
/@MOVE %2 TO %1 CLAIM.†]
/[ENDBM#.†]
```

BN




```
/@IF %1 CLAIM IS NOT LESS THAN %2 †@
THEN GO TO ENDBN#.† ]
/@MOVE %2 TO %1 CLAIM.†]
/[ENDBN#.†]
```

DIAGRAMMATIC REPRESENTATION

CORRESPONDING CODE

BO

1 %1 = fixed amount

```
/      ** %1 = FIXED AMOUNT **↑]
```

BQ

1 Calculate %1

```
/      ** CALCULATE %1 **↑]
```

BR

1 %1 = %2

```
/@MOVE %2 TO %1 . ↑]
```

BS

1 Calculate non-uniform %1

```
/      ** CALCULATE NON-UNIFORM %1 **↑]
```

BT

1 Determine group and calculate %1

```
/      ** DETERMINE GROUP AND CALCULATE
%1 USING METHOD FOR GROUP **↑]
```

DIAGRAMMATIC REPRESENTATION

CORRESPONDING CODE

BU

1 Calculate %1

```
/      ** CALCULATE %1 **↑]
```

BV

1 If

2 condition — NO → Go to ENDBV

```
/@ IF ↑]
/      ** CONDITION **↑]
```

YES

3 PARBV

```
/@ THEN GO TO PARBV# ↑@ ELSE GO TO
ENDBV#. ↑[PARBV#. ↑]
```

4 Calculate %1

```
/      ** CALCULATE %1 **↑]
```

5 ENDBV

```
/ [ENDBV#. ↑]
```

BW

1 %1 = %2 %3?

```
/@  %1 IS EQUAL TO %2 %3 ]
```

BX : As for AI with ENDAI replaced by ENDBX.

BY : As for AJ.

BZ : As for AK with PARAK replaced by PARBZ.

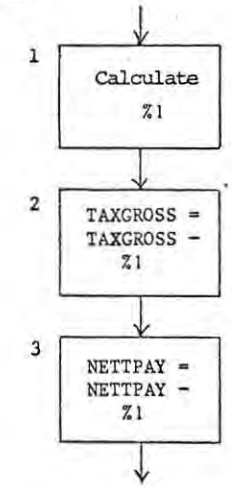
CA : As for AL.

CB : As for AM.

DIAGRAMMATIC REPRESENTATION

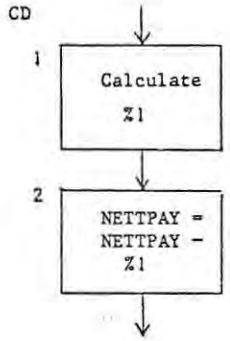
CORRESPONDING CODE

CC



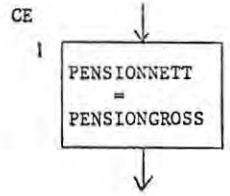

```
/      ** CALCULATE %1 **↑]

/@SUBTRACT %1 FROM TAXGROSS.↑]

/@SUBTRACT %1 FROM NETTPAY.↑]
```

CD




```
/      ** CALCULATE %1  **↑]

/@SUBTRACT %1  FROM NETTPAY.↑]
```

CE

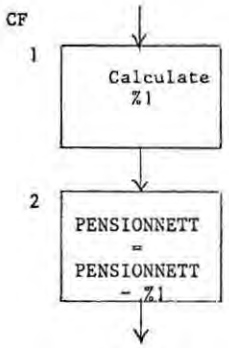



```
/@MOVE PENSIONGROSS TO PENSIONNETT.↑]
```

DIAGRAMMATIC REPRESENTATION

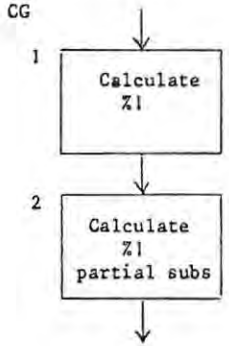
CORRESPONDING CODE

CF



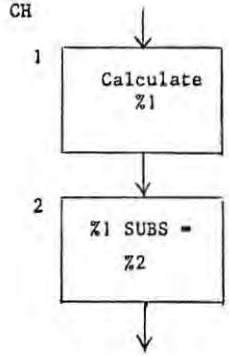

```
/      **  CALCULATE %1  **↑]

/@SUBTRACT %1  FROM PENSIONNETT.↑]
```

CG




```
/      ** CALCULATE %1  **↑]

/      ** CALCULATE %1 SUBS (PARTIAL) **↑]
```

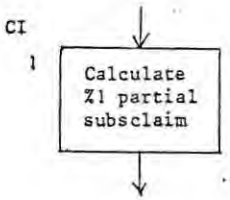
CH



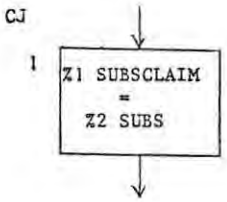

```
/      ** CALCULATE %1 **↑]

/@MOVE %2  TO %1 SUBS.↑]
```

DIAGRAMMATIC REPRESENTATION

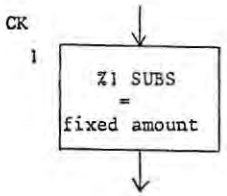
CORRESPONDING CODE



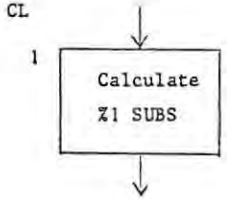

```
/      ** CALCULATE %1 SUBSCLAIM (PARTIAL) **†]
```



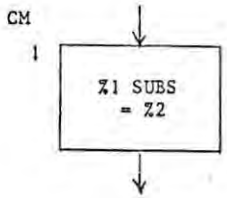

```
/@MOVE %2 SUBS TO %1 SUBSCLAIM.†]
```




```
/      ** %1 SUBS = FIXED AMOUNT **†]
```



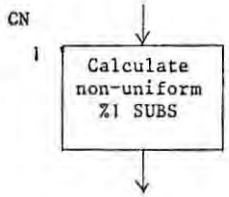

```
/      ** CALCULATE %1 SUBS **†]
```



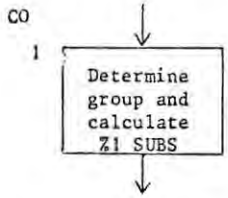

```
/@MOVE %2  TO %1 SUBS.†]
```

DIAGRAMMATIC REPRESENTATION

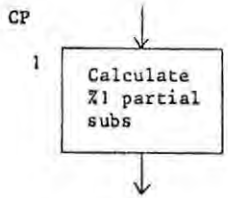
CORRESPONDING CODE



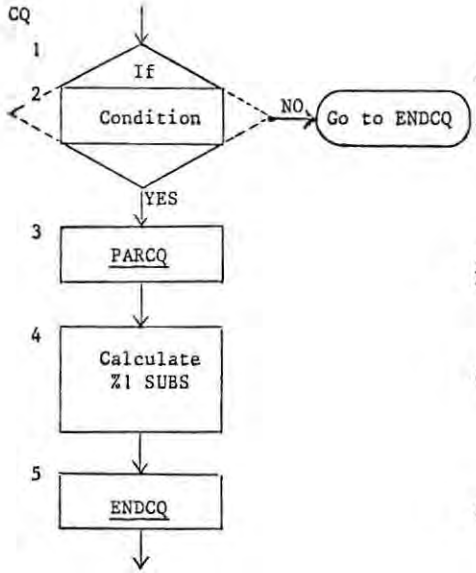

```
/      ** CALCULATE NON-UNIFORM %1 SUBS **†]
```




```
/      ** DETERMINE GROUP AND CALCULATE
%1 SUBS USING METHOD FOR GROUP **†]
```




```
/      ** CALCULATE %1 SUBS (PARTIAL) **†]
```

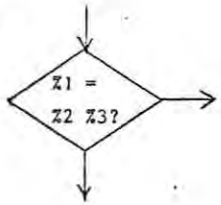



```
/@IF†]
/      ** CONDITION **†]
/@  THEN GO TO PARCQ# †@  ELSE GO TO
ENDCQ#.†[PARCQ#.†]
/      ** CALCULATE %1 SUBS **†]
/[ENDCQ#.†]
```

DIAGRAMMATIC REPRESENTATION

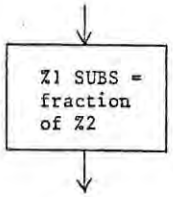
CORRESPONDING CODE

CR




```
/@  %1  IS EQUAL TO %2  %3 ]
```

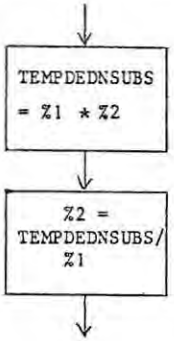
CS




```
/     ** %1 SUBS = FRACTION OF %2 **↑]
```

CT : As for CS above.

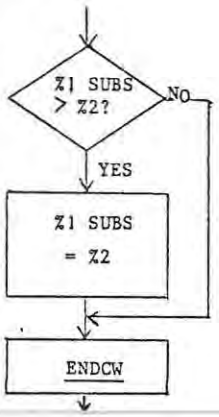
CU




```
/@MULTIPLY %1  BY %2  GIVING TEMPDEDNSUBS.↑]

/@DIVIDE TEMPDEDNSUBS BY %1  GIVING %2 SUBS.↑]
```

CV : As for CU above.

CW

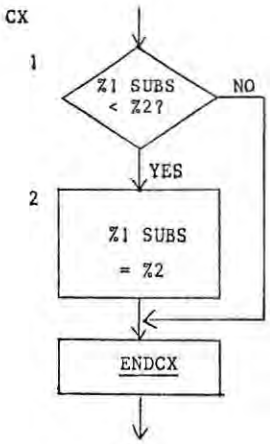



```
/@IF %1 SUBS IS NOT GREATER THAN %2
THEN GO TO ENDCW#.↑]

/@MOVE %2  TO %1 SUBS.↑[ENDCW#.↑]
```

DIAGRAMMATIC REPRESENTATION

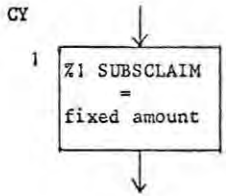
CORRESPONDING CODE

CX



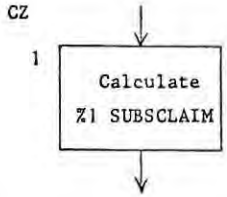

```
/@ IF %1 SUBS IS NOT LESS THAN %2  THEN GO
TO ENDCX#.↑]

/@MOVE %2  TO %1 SUBS.↑[ENDCX#.↑]
```

CY



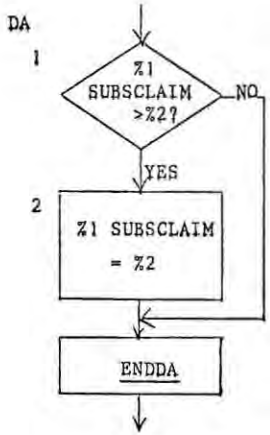

```
/     ** %1 SUBSCLAIM = FIXED AMOUNT **↑]
```

CZ




```
/     ** CALCULATE %1 SUBSCLAIM **↑]
```

DA

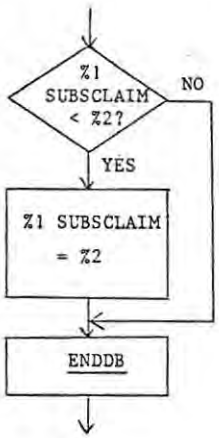



```
/@IF %1 SUBSCLAIM IS NOT GREATER THAN
%2 THEN GO TO ENDDA#.↑]

/@MOVE %2 SUBS TO %1 SUBSCLAIM.↑[ENDDA#.↑]
```

DIAGRAMMATIC REPRESENTATION

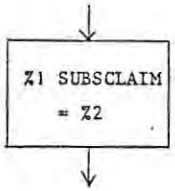
CORRESPONDING CODE

DB



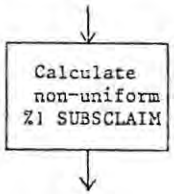

```
/@IF %1 SUBSCLAIM IS NOT LESS THAN %2
THEN GO TO ENDDB#.↑]

/@MOVE %2 SUBS TO %1 SUBSCLAIM.↑[ENDDB#.↑]
```

DE



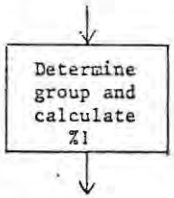

```
/@MOVE %2  TO %1 SUBSCLAIM.↑]
```

DF




```
/      ** CALCULATE NON-UNIFORM %1
SUBSCLAIM **↑]
```

DG

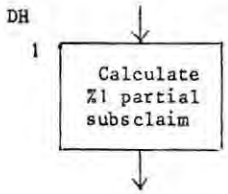



```
/      ** DETERMINE GROUP AND CALCULATE
%1  USING METHOD FOR GROUP **↑]
```

DIAGRAMMATIC REPRESENTATION

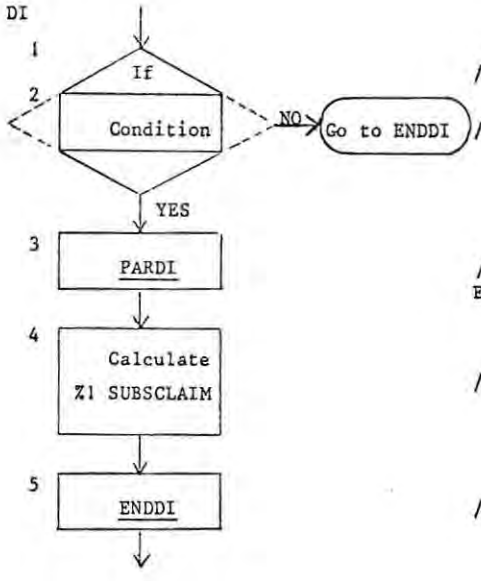
CORRESPONDING CODE

DH



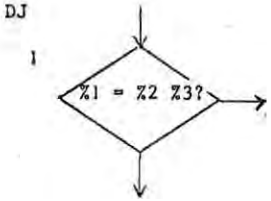

```
/      ** CALCULATE %1 SUBSCLAIM (PARTIAL) **↑]
```

DI



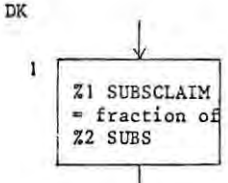

```
/@IF ↑]
/      ** CONDITION  **↑]

/@  THEN GO TO PARDI# ↑@  ELSE GO TO
ENDDI#.↑[PARDI#.↑]

/      ** CALCULATE %1 SUBSCLAIM **↑]

/[ENDDI#.↑]
```

DJ




```
/@  %1  IS EQUAL TO %2  %3  ↑]
```

DK

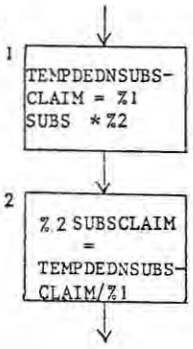



```
/      ** %1 SUBSCLAIM = FRACTION OF
%2 SUBS **↑]
```

DL : As for DK above.

DIAGRAMMATIC REPRESENTATION | CORRESPONDING CODE

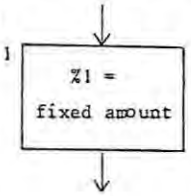
DM




```
/@MULTIPLY %1 SUBS BY %2  GIVING
TEMPDEDNSUBSCLAIM.†]

/@DIVIDE TEMPDEDNSUBSCLAIM BY %1
GIVING %2 SUBSCLAIM.†]
```

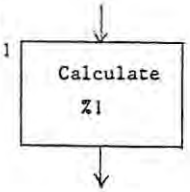
DN : As for DM above.

DO



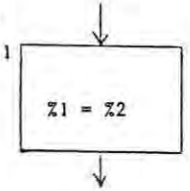

```
/      ** %1  = FIXED AMOUNT **†]
```

DQ



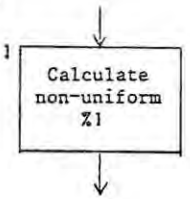

```
/      ** CALCULATE %1 **†]
```

DR




```
/@MOVE %2  TO %1  .†]
```

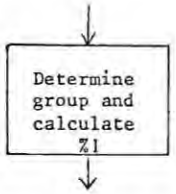
DS




```
/      ** CALCULATE NON-UNIFORM %1 **†]
```

DIAGRAMMATIC REPRESENTATION | CORRESPONDING CODE

DT



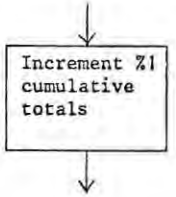

```
/     ** DETERMINE GROUP AND CALCULATE %1
USING METHOD FOR GROUP **†]
```

DU : As for DQ above.

DV : As for AK with PARAK replaced by PARDV and ENDAK by ENDDV.

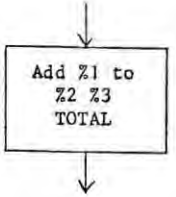
DW : As for DJ above.

DX



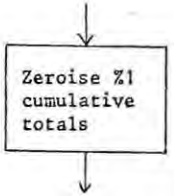

```
/     ** INCREMENT CUMULATIVE TOTALS FOR
%1  **†]
```

DY



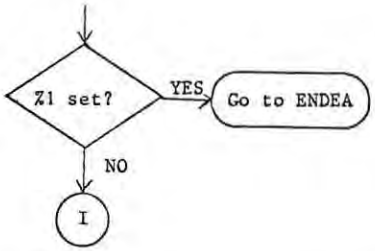

```
/@ADD %1  TO  %2 %3 TOTAL.†]
```

DZ




```
/     ** ZEROISE CUMULATIVE TOTALS
FOR %1 **†]
```

EA

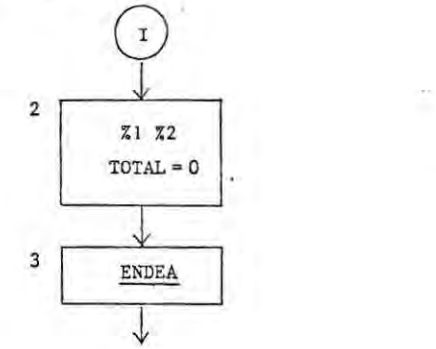



```
/@IF %1  THEN GO TO ENDEA#.†]
```

DIAGRAMMATIC REPRESENTATION

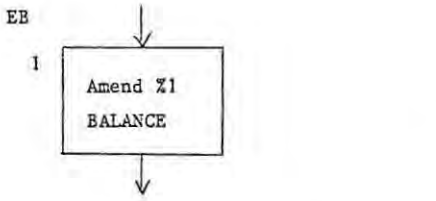
CORRESPONDING CODE

EA (continued)



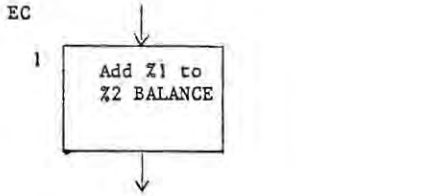

```
/@MOVE ZERO TO %1 %2 TOTAL.↑]

/[ENDEA#.↑]
```

EB



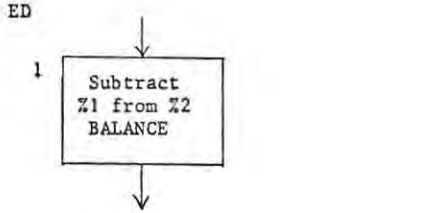

```
/       ** AMEND %1 BALANCE **↑]
```

EC



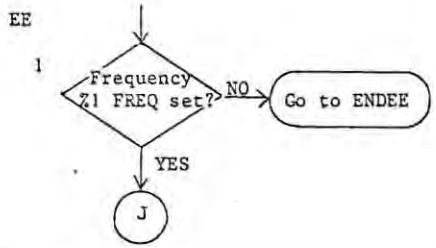

```
/@ADD %1  TO %2 BALANCE.↑]
```

ED




```
/@SUBTRACT %1  FROM %2 BALANCE.↑]
```

EE

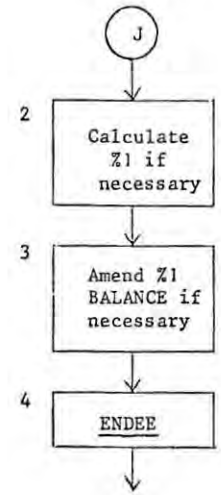



```
/@MOVE %1 FREQ TO I.↑@ IF FREQUENCY (I)
IS NOT EQUAL TO 1 THEN GO TO ENDEE#.↑]
```

DIAGRAMMATIC REPRESENTATION

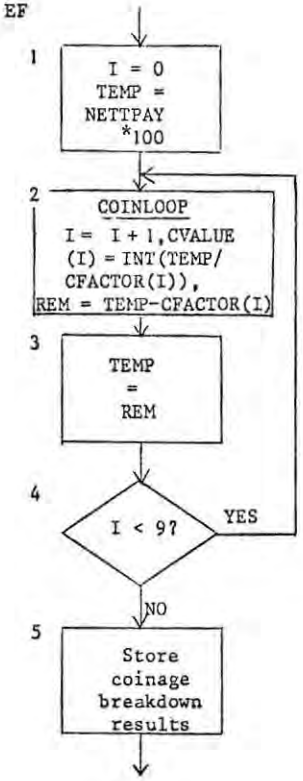
CORRESPONDING CODE

EE (continued)




```
/       ** CALCULATE %1  IF NECESSARY **↑]

/       ** AMEND %1 BALANCE IF NECESSARY **↑]

/[ENDEE#.↑]
```

EF

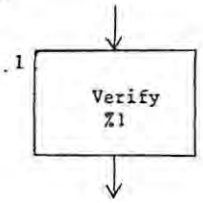



```
/@MOVE ZERO TO I.↑@MULTIPLY NETTPAY
BY 100 GIVING TEMP.↑]

/[COINLOOP.↑@ADD 1 TO I.↑@DIVIDE TEMP
BY CFACTOR (I) GIVING CVALUE (I)
REMAINDER RFM.↑]

/@MOVE REM TO TEMP.↑]

/@IF I IS LESS THAN 9 THEN GO TO
COINLOOP.↑]

/@MOVE TEMP TO ONEC.↑@MOVE CVALUE (1) TO RTEN.
↑@MOVE CVALUE (2) TO RFIVE.↑@MOVE CVALUE (3) TO
RTWO.↑@MOVE CVALUE (4) TO RONE.↑@MOVE CVALUE
(5) TO FIFTYC.↑@MOVE CVALUE (6) TO TWENTYC.
↑@MOVE CVALUE (7) TO TENC.↑@MOVE CVALUE (8) TO
FIVEC.↑@MOVE CVALUE (9) TO TWOC.↑]
```

DIAGRAMMATIC REPRESENTATION

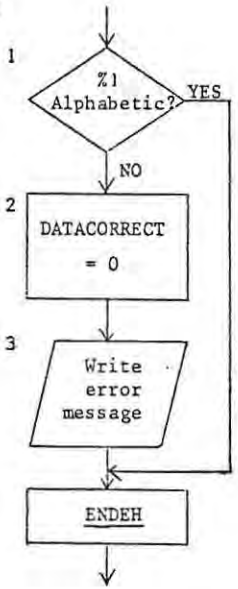
CORRESPONDING CODE

EG



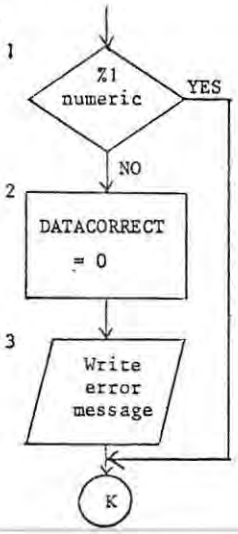

```
/     ** VERIFY %1 **↑]
```

EH




```
/@IF %1  IS ALPHABETIC THEN GO TO ENDEH#.↑]

/@MOVE 0 TO DATACORRECT.↑]

/@MOVE "%1" TO VFIELDA.↑@MOVE "IS NOT
ALPHABETIC" TO VTEXTA.↑@MOVE VERIFMESSA TO
PRINTLINE.↑@PERFORM WRITELINE.↑@WRITE PRINT-
LINE AFTER 1 LINES. [ENDEH#.↑]
```

EI

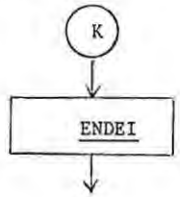



```
/@IF %1  IS NUMERIC THEN GO TO ENDEI#.↑]

/@MOVE 0 TO DATACORRECT.↑]

/@MOVE "%1 " TO VFIELDA.↑@MOVE "IS NOT
NUMERIC" TO VTEXTA.↑@MOVE VERIFMESSA TO
PRINTLINE.↑@PERFORM WRITELINE.↑@WRITE
PRINTLINE AFTER 1 LINES.↑[ENDEI#.↑]
```

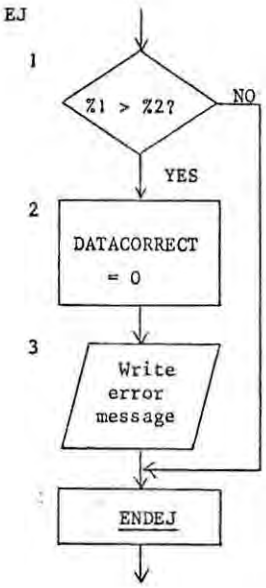
DIAGRAMMATIC REPRESENTATION

CORRESPONDING CODE

EI (continued)

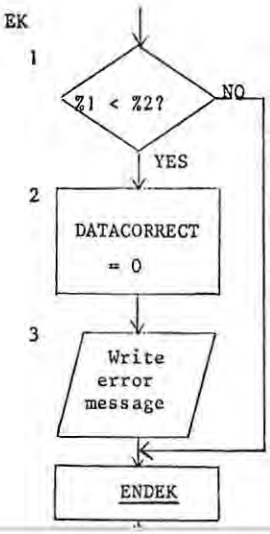


EJ




```
/@IF %1 IS NOT GREATER THAN %2  GO TO
ENDEJ#.↑]

/@MOVE 0 TO DATACORRECT.↑]

/@MOVE "%1 " TO VFIELDB1.↑@MOVE "  EXCEEDS  "
TO VTEXTB2.↑@MOVE "%2 " TO VFIELDB2.↑@MOVE
VERIFMESS TO PRINTLINE.↑@PERFORM WRITELINE.
↑@WRITE PRINTLINE AFTER 1 LINES. ↑[ENDEJ#.↑]
```

EK

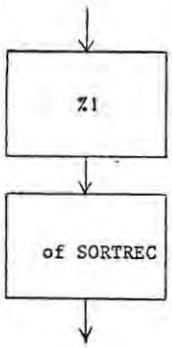



```
/@IF %1 IS NOT LESS THAN %2  THEN GO TO
ENDEK#.↑]

/@MOVE 0 TO DATACORRECT.↑]

/@MOVE "%1 " TO VFIELDB1.↑@MOVE " IS LESS THAN "
TO VTEXTB2.↑@MOVE "%2 " TO VFIELDB2.↑@MOVE
VERIFMESSB TO PRINTLINE.↑@PERFORM WRITELINE.
↑@WRITE PRINTLINE AFTER 1 LINES.↑[ENDEK#.↑]
```

DIAGRAMMATIC REPRESENTATION

CORRESPONDING CODE

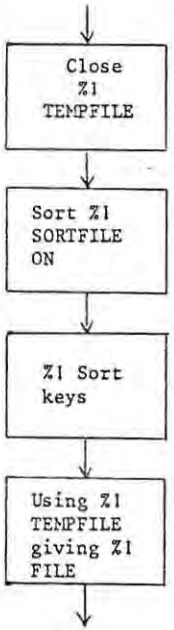
EL




```
/@   %1  †]
/@ OF %1 SORTREC †]
```

EM : As for DJ above.

EN




```
/@CLOSE %1 TEMPFILE.†]
/@SORT %1 SORTFILE ON †]
/      ** SORT KEYS OF %1 **†]
/@    USING %1 TEMPFILE GIVING %1 FILE.†]
```

EO : As for EL above.

EP


```
/      ** PREPARE REPORT %1  **†]
```

DIAGRAMMATIC REPRESENTATION

CORRESPONDING CODE

EQ


```
/     ** WRITE REPORT %1 **†]
```

ER


```
/@IF %1  THEN GO TO ENDER#.†]
/     ** PREPARE REPORT %1  **†]
/[ENDER#.†]
```

ES

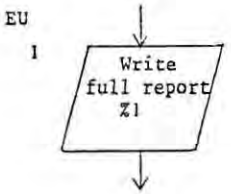

```
/@IF %1  THEN GO TO ENDES#.†]
/     ** WRITE REPORT %1  **†]
/[ENDES#.†]
```

ET

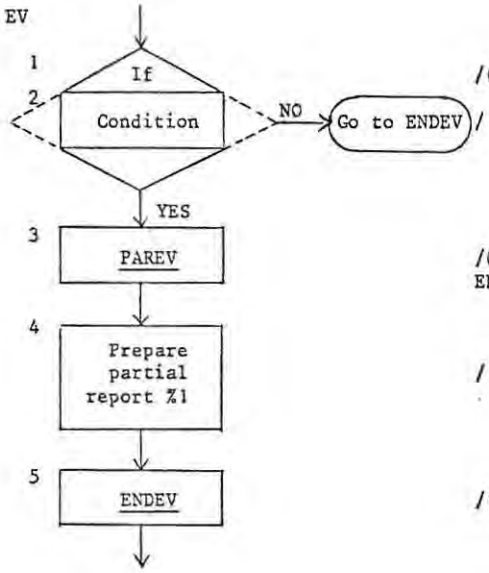

```
/     ** PREPARE FULL REPORT %1 **†]
```

DIAGRAMMATIC REPRESENTATION

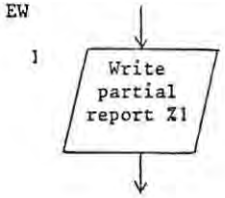
CORRESPONDING CODE



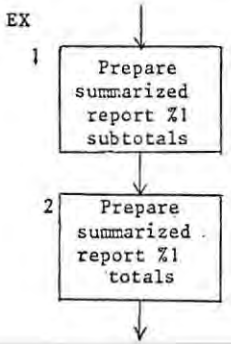

```
/      ** WRITE FULL REPORT %1 **↑]
```




```
/@IF↑]
/      ** CONDITION **↑]
/@  THEN GO TO PAREV# ↑@  ELSE GO TO
ENDEV#.↑[PAREV#.↑]
/      ** PREPARE PARTIAL REPORT %1  **↑]
/[ENDEV#.↑]
```



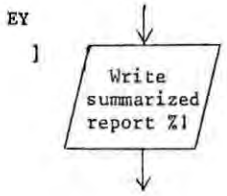

```
/      ** WRITE PARTIAL REPORT %1 **↑]
```




```
/     ** PREPARE SUMMARIZED REPORT %1
SUBTOTALS **↑]
/      ** PREPARE SUMMARIZED REPORT %1
TOTALS **↑]
```

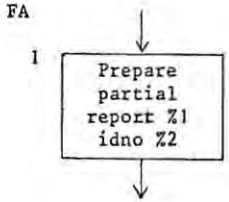
DIAGRAMMATIC REPRESENTATION

CORRESPONDING CODE

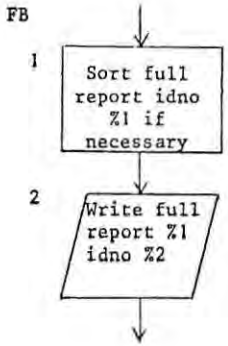



```
/    **  WRITE SUMMARIZED REPORT %1 **↑]
```

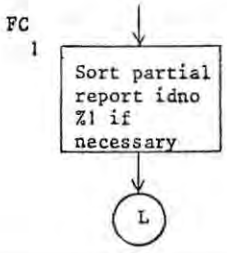

```
/     ** PREPARE FULL REPORT %1  IDNO %2 **↑]
```




```
/     ** PREPARE PARTIAL REPORT %1  IDNO %2 **↑]
```




```
/     ** SORT FULL REPORT IDNO %1  IF
NECESSARY **↑]
/     ** WRITE FULL REPORT %1  IDNO %2 **↑]
```




```
/     ** SORT PARTIAL REPORT IDNO %1
IF NECESSARY **↑]
```

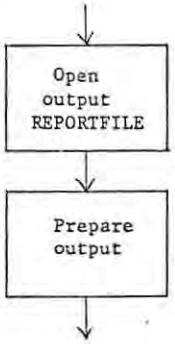
DIAGRAMMATIC REPRESENTATION

CORRESPONDING CODE

FC (continued)


```
/      ** WRITE PARTIAL REPORT %1  IDNO %2 **†]
```

FD

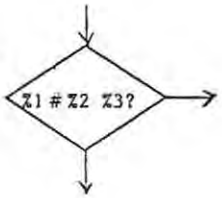



```
/@OPEN OUTPUT REPORTFILE.†]

/      ** PREPARE OUTPUT **†]
```

FE


```
/@CLOSE REPORTFILE.†]

/      ** WRITE REPORTS **†]
```

FF

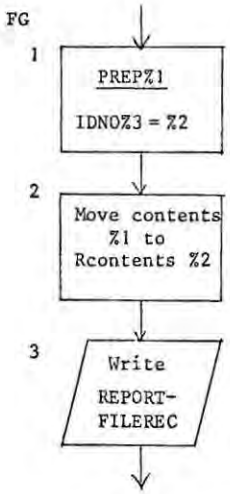



```
/@  %1  IS EQUAL TO %2  %3  †]
```

DIAGRAMMATIC REPRESENTATION

CORRESPONDING CODE

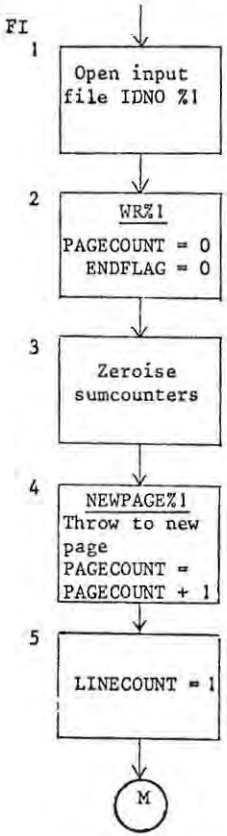
FG




```
/[PREP%1 .†@MOVE %2  TO IDNO%3 .†]

/      ** MOVE CONTENTS %1  TO RCONTENTS
%2 **†]

/@WRITE REPORTFILEREC.†]
```

FH : As for FG above.

FI

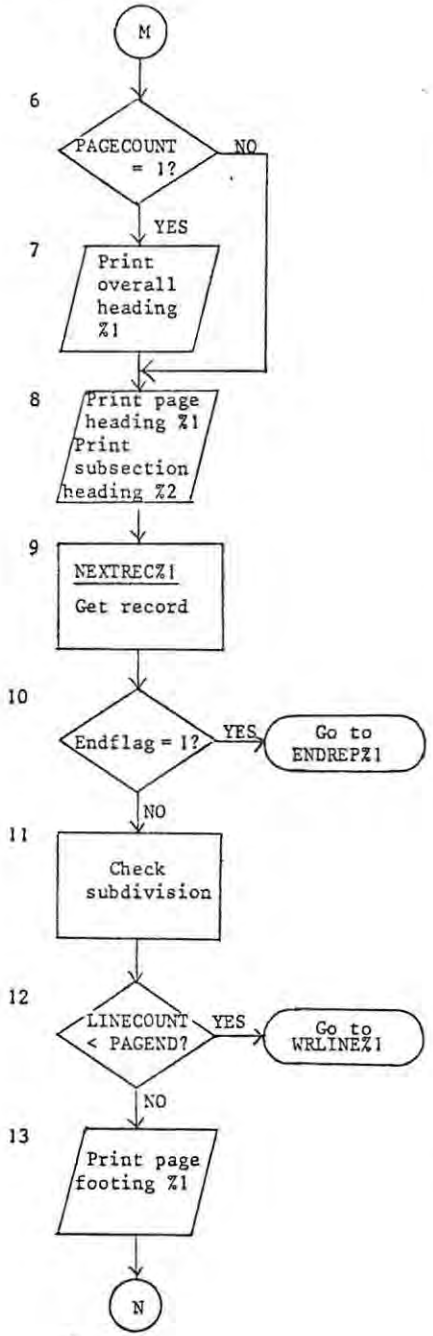



```
/      ** OPEN INPUT FILENAME IDNO%1 **†]

/[WR%1 .†@MOVE ZERO TO PAGECOUNT.†@MOVE
ZERO TO ENDFLAG.†]

/      ** ZEROISE SUMCOUNTERS **†]

/[NEWPAGE%1 .†@MOVE SPACES TO PRINTLINE.
†@WRITE PRINTLINE AFTER CHANNEL-1.†@ADD 1
TO PAGECOUNT.†]

/@MOVE 1 TO LINECOUNT.†]
```

DIAGRAMMATIC REPRESENTATION

CORRESPONDING CODE

FI (continued)

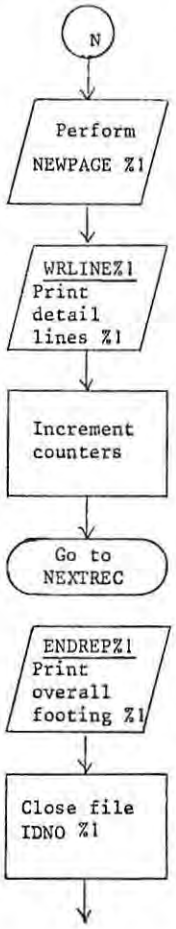



```
/@IF PAGECOUNT IS EQUAL TO 1 .†]

/@  THEN PERFORM PRINOH%1 .†]

/@PERFORM PRINPH%1 .†@PERFORM PRINSH%2 .†]

/[NEXTREC%1 .†      ** GET RECORD **†]

/@IF ENDFLAG IS EQUAL TO 1 THEN GO TO
ENDREP%1 .†]

/      ** CHECK SUBDIVISION %1  **†]

/@IF LINECOUNT IS LESS THAN PAGEND THEN
GO TO WRLINE%1 .†]

/@PERFORM PRINPF%1 .†]
```

DIAGRAMMATIC REPRESENTATION

CORRESPONDING CODE

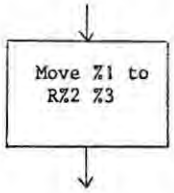
FI (continued)




```
/@PERFORM NEWPAGE%1 .†]

/[WRLINE%1 .†@PERFORM PRINDL%2 .†]

/      ** INCREMENT COUNTERS **†]

/@GO TO NEXTREC%1 .†]

/[ENDREP%1 .†@PERFORM PRINOF%2 .†]

/      ** CLOSE FILENAME IDNO %1 **†]
```

FJ : As for FI above.

FK



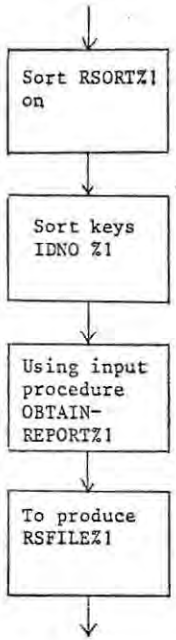

```
/@MOVE %1  TO R%2 %3 .†]
```

FL : As for FK above.

DIAGRAMMATIC REPRESENTATION

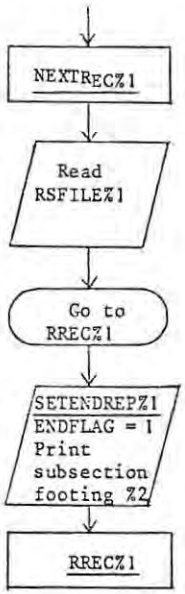
CORRESPONDING CODE

FM




```
/@SORT RSORT%1  ON †]
/      ** SORT KEYS IDNO %1  **†]
/@    INPUT PROCEDURE IS OBTAIN-REPORT%1 †]
/@    GIVING RSFILE%1 .†]
```

FN

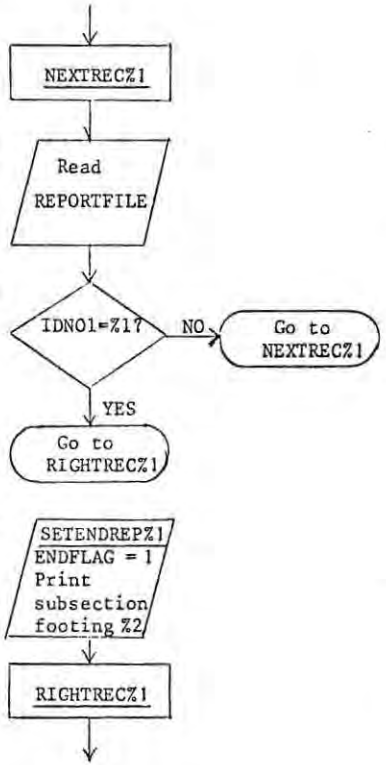



```
/[NEXTREC%1 .†]
/@READ RSFILE%1  AT END GO TO
SETENDREP%2 .†]
/@GO TO RREC%1 .†]
/[SETENDREP%1 .†@MOVE 1 TO ENDFLAG.
†@PERFORM PRINSF%2 .†]
/[RREC%1 .†]
```

DIAGRAMMATIC REPRESENTATION

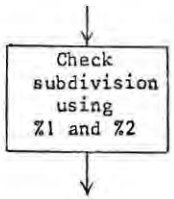
CORRESPONDING CODE

FO




```
/[NEXTREC%1 .†]
/@READ REPORTFILE†@  AT END GO TO
SETENDREP%1 .†]
/@IF IDNO1IS NOT EQUAL TO %1 THEN GO TO
NEXTREC%2  ELSE GO TO RIGHTREC%3 .†]
/[SETENDREP%1 .†@MOVE 1 TO ENDFLAG.
†@PERFORM PRINSF%2 .†]
/[RIGHTREC%1 .†]
```

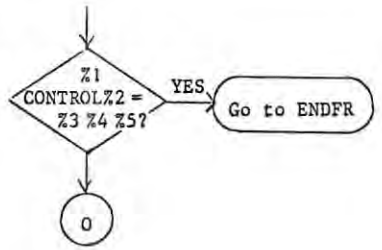
FP




```
/      ** CHECK SUBDIVISION USING %1
AND %2 **†]
```

FQ : As for FP above.

FR

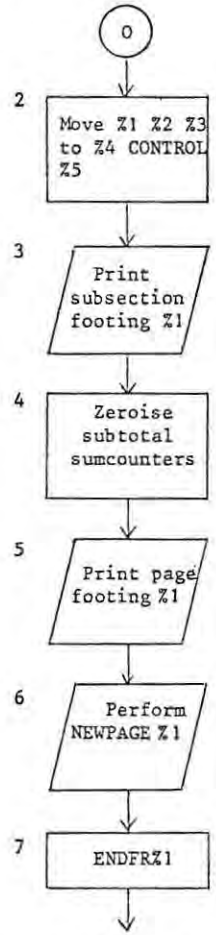



```
/@IF %1 CONTROL%2 IS EQUAL TO %3 %4 %5
THEN GO TO ENDFR%6 .†]
```

DIAGRAMMATIC REPRESENTATION

CORRESPONDING CODE

FR (continued)




```
/@MOVE %1 %2 %3  TO %4 CONTROL%5 .↑@IF
PAGECOUNT EQUAL TO 1 AND LINECOUNT EQUAL
TO 1 THEN GO TO ENDFR%6 .↑]

/@PERFORM PRINSF%1 .↑]

/      ** ZEROISE SUMCOUNTERS FOR SUBTOTS **↑]

/@PERFORM PRINPF%1 .↑]

/@PERFORM NEWPAGE%1 .↑]

/[ENDFR%1 .↑]
```

FT : As for FM above.

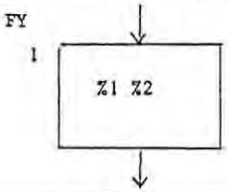
FU : As for FN above.

FV : As for FO above.

FW : As for FP above.

FX : As for FP above.

FY



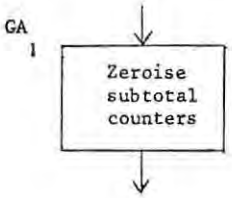

```
/@    %1 %2 ↑]
```

DIAGRAMMATIC REPRESENTATION

CORRESPONDING CODE

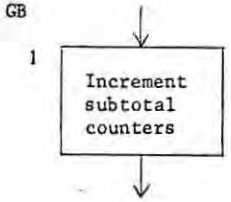
FZ : As for FY above.

GA



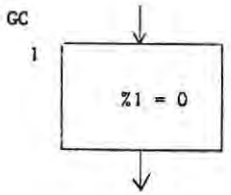

```
/      ** ZEROISE SUBTOTAL COUNTERS  ** ↑]
```

GB



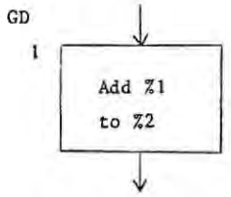

```
/      ** INCREMENT SUBTOTAL COUNTERS ** ↑ ]
```

GC




```
/@MOVE ZERO TO %1 .↑]
```

GD




```
/@ADD %1  TO %2 .↑]
```

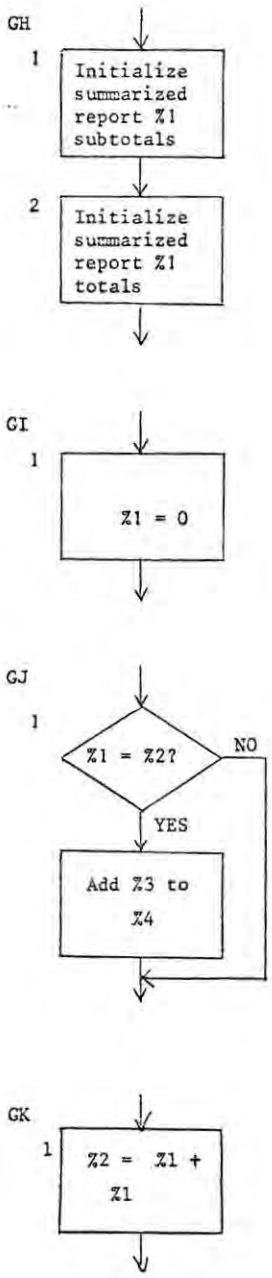
GE : As for GA above.

GF : As for GB above.

GG : As for FR above.

DIAGRAMMATIC REPRESENTATION

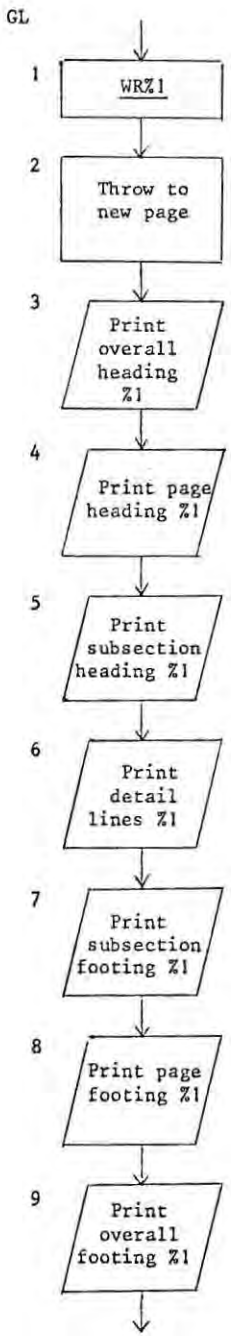
CORRESPONDING CODE




```
/      ** INITIALIZE SUMMARIZED REPORT
%1  SUBTOTALS **†]

/      ** INITIALIZE SUMMARIZED REPORT
%1 TOTALS **†]

/@MOVE ZERO TO  %1 .†]

/@IF %1  IS EQUAL TO %2  †@  THEN ADD
%3  TO  %4 .†]

/@  ADD  %1  TO   %2.†]
```

DIAGRAMMATIC REPRESENTATION

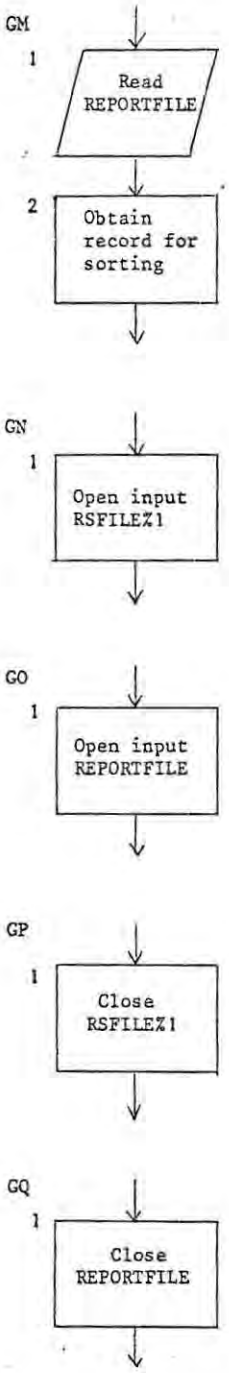
CORRESPONDING CODE




```
/[WR%1 . † ]

/@MOVE SPACES TO PRINTLINE.†@WRITE
PRINTLINE AFTER CHANNEL-1.†]

/@PERFORM PRINOH%1 .†]

/@PERFORM PRINPH%1 .†]

/@PERFORM PRINSH%1 .†]

/@PERFORM PRINDL%1 .†]

/@PERFORM PRINSF%1 .†]

/@PERFORM PRINPF%1 .†]

/@PERFORM PRINOF%1 .†]
```

DIAGRAMMATIC REPRESENTATION

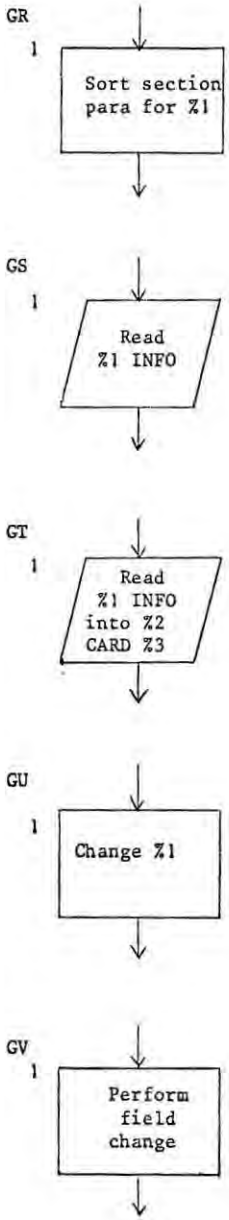
CORRESPONDING CODE




```
/ [OBTAIN-REPORT%1  SECTION.↑[OPEN-REP%2.
↑@OPEN INPUT REPORTFILE.↑[READ-SELECT%3 .
↑@READ REPORTFILE↑@  AT END GO TO CLOSE-
REP%4 .↑]

/@IF IDNO1 IS EQUAL TO %1  ↑@  THEN RELEASE
RSORTREC%2 FROM REPORTFILEREC%3 ↑@GO TO
READ-SELECT%4 .↑[CLOSE-REP%5 .↑@CLOSE
REPORTFILE.↑]

/@OPEN INPUT RSFILE%1 .↑]

/@OPEN INPUT REPORTFILE.↑]

/@CLOSE RSFILE%1 .↑]

/@CLOSE REPORTFILE.↑]
```

DIAGRAMMATIC REPRESENTATION

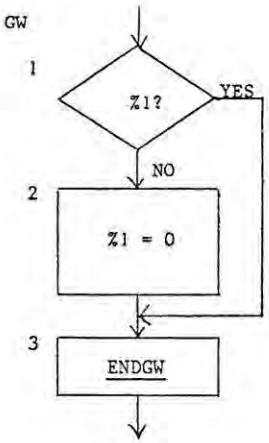
CORRESPONDING CODE



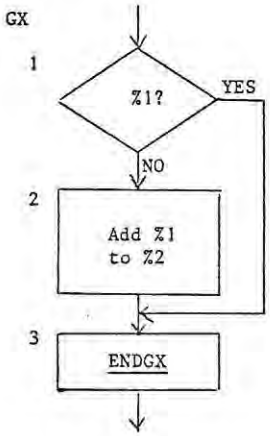

```
/      ** SORT SECTION PARA FOR %1 **↑]

/@READ %1 INFO.↑]

/@READ %1 INFO INTO %2 CARD%3 .↑]

/      ** CHANGE %1 **↑]

/@IF EDITFIELD IS EQUAL TO %1  THEN ↑@ MOVE
EDITMATCH TO CHMATCH↑@  MOVE "%2 " TO
CHFIELD↑@  MOVE CHMESSLINE1 TO PRINTLINE↑@
PERFORM WRITELINE↑@  MOVE %3 TO CHOLDFIELD-
VAL↑@  MOVE E%4 TO CHNEWFIELDVAL↑@  MOVE E%5
TO %6  ↑@  MOVE CHMESSLINE2  TO PRINTLINE↑@
PERFORM WRITELINE.↑]
```

DIAGRAMMATIC REPRESENTATION

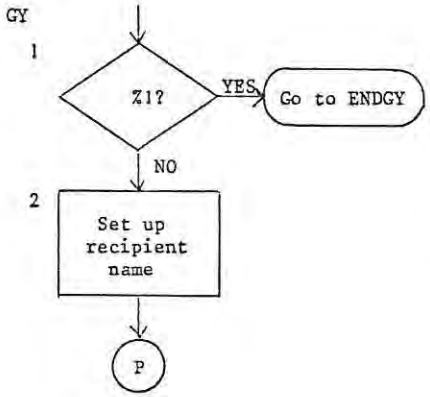
CORRESPONDING CODE




```
/@IF %1  THEN GO TO ENDGW#.↑]

/@MOVE ZERO TO %1 .↑]

/[ENDGW#.↑]
```



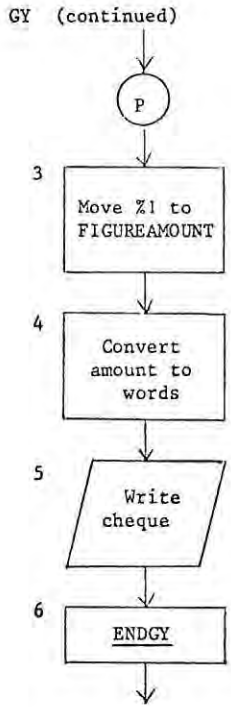

```
/@IF %1  THEN GO TO ENDGX#.↑]

/@ADD %1  TO %2 .↑]

/[ENDGX#.↑]
```



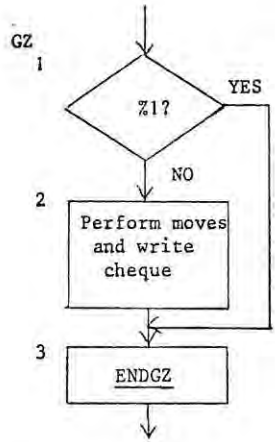

```
/@IF %1  THEN GO TO ENDGY#.↑]

/     **  SET UP RECIPIENT NAME **↑]
```

DIAGRAMMATIC REPRESENTATION

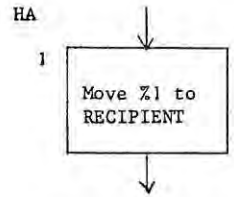
CORRESPONDING CODE




```
/@MOVE %1  TO FIGUREAMOUNT.↑]

/@PERFORM WORDS.↑]

/@PERFORM WRCHEQUE.↑]

/[ENDGY#.↑]
```



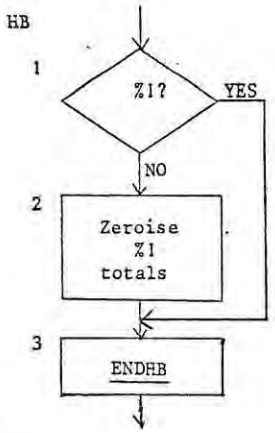

```
/@IF %1  THEN GO TO ENDGZ#.↑]

/@MOVE EMPNAME TO RECIPIENT.↑@MOVE %1  TO
FIGUREAMOUNT.↑@PERFORM WORDS.↑@PERFORM
WRCHEQUE.↑]

/[ENDGZ#.↑]
```



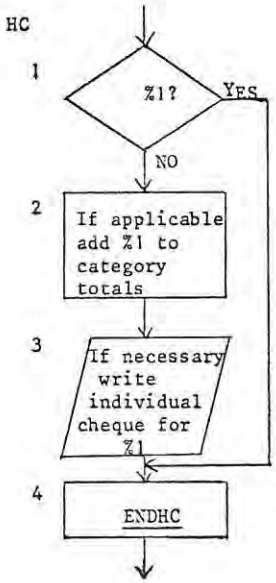

```
/@MOVE "%1 " TO RECIPIENT.↑]
```

DIAGRAMMATIC REPRESENTATION

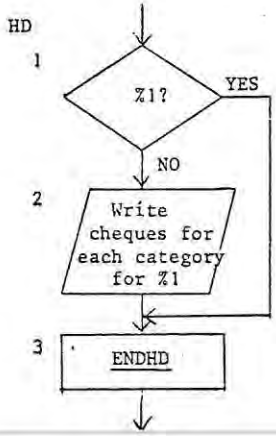
CORRESPONDING CODE




```
/@IF %1  THEN GO TO ENDHB#.↑]

/      ** ZEROISE %1  TOTALS **↑]

/[ENDHB#.↑]
```



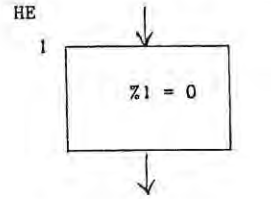

```
/@IF %1  THEN GO TO ENDHC#.↑]

/      ** IF APPLICABLE ADD %1  TO
CATEGORY TOTALS **↑]

/      ** IF NECESSARY WRITE INDIVIDUAL
CHEQUE FOR %1 **↑]

/[ENDHC#.↑]
```



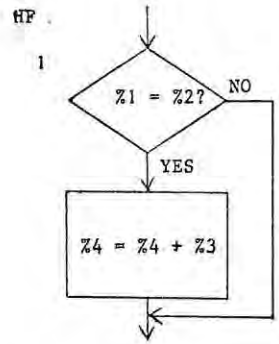

```
/@IF %1  THEN GO TO ENDHD#.↑]

/     **  WRITE CHEQUES FOR EACH
CATEGORY FOR %1 **↑]

/[ENDHD#.↑]
```

DIAGRAMMATIC REPRESENTATION

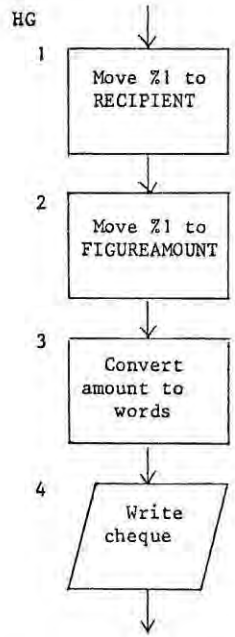
CORRESPONDING CODE



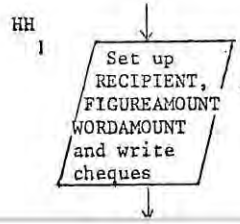

```
/@MOVE ZERO TO %1 .↑]
```




```
/@IF %1  IS EQUAL TO %2  ↑@  THEN ADD
%3  TO %4 .↑]
```



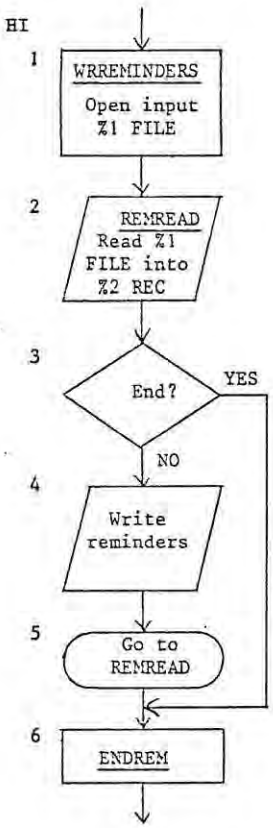

```
/@MOVE "%1 " TO RECIPIENT.↑]

/@MOVE %1 TO FIGUREAMOUNT.↑]

/@PERFORM WORDS.↑]

/@PERFORM WRCHEQUE.↑]
```



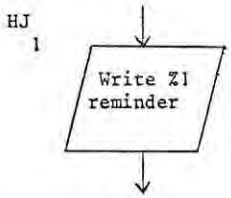

```
/@MOVE EMPNAME TO RECIPIENT.↑@MOVE %1 TO
FIGUREAMOUNT.↑@PERFORM WORDS.↑@PERFORM
WRCHEQUE.↑]
```

DIAGRAMMATIC REPRESENTATION

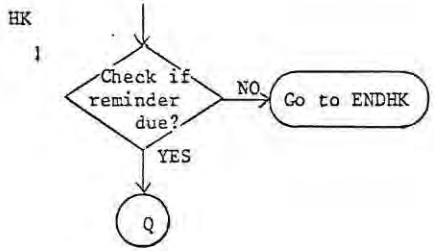
CORRESPONDING CODE




```
/[WRREMINDERS.↑@OPEN INPUT %1 FILE.↑]

/[REMREAD.↑@READ %1 FILE INTO %2 REC↑]

/@    AT END GO TO ENDREM.↑]

/      ** WRITE REMINDERS **↑]

/@GO TO REMREAD.↑]

/[ENDREM.↑]
```



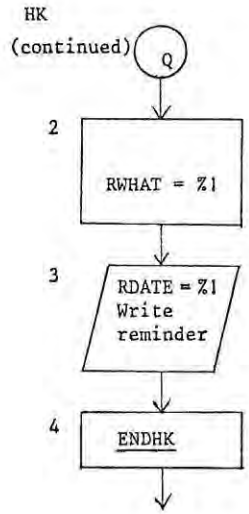

```
/      ** WRITE %1 REMINDER  **↑]
```



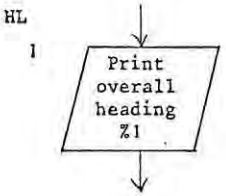

```
/@MOVE %1 TO DUEDATE.↑@PERFORM DAYS.↑@IF
DAYDIFF IS NOT LESS THAN %2  ↑@  THEN GO
TO ENDHK#.↑@MOVE "%3 " TO RIDENT.↑]
```

DIAGRAMMATIC REPRESENTATION

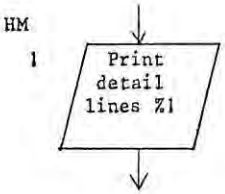
CORRESPONDING CODE



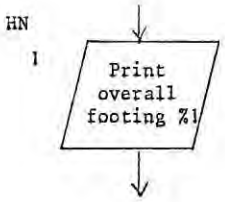

```
/@MOVE %1  TO RWHAT.↑]

/@MOVE %1  TO RDATE.↑@MOVE REMINDERMESS TO
PRINTLINE.↑@PERFORM WRITELINE.↑@WRITE
PRINTLINE AFTER ADVANCING 1 LINES.↑]

/[ENDHK#.↑]
```




```
/@PERFORM PRINOH%1 .↑]
```



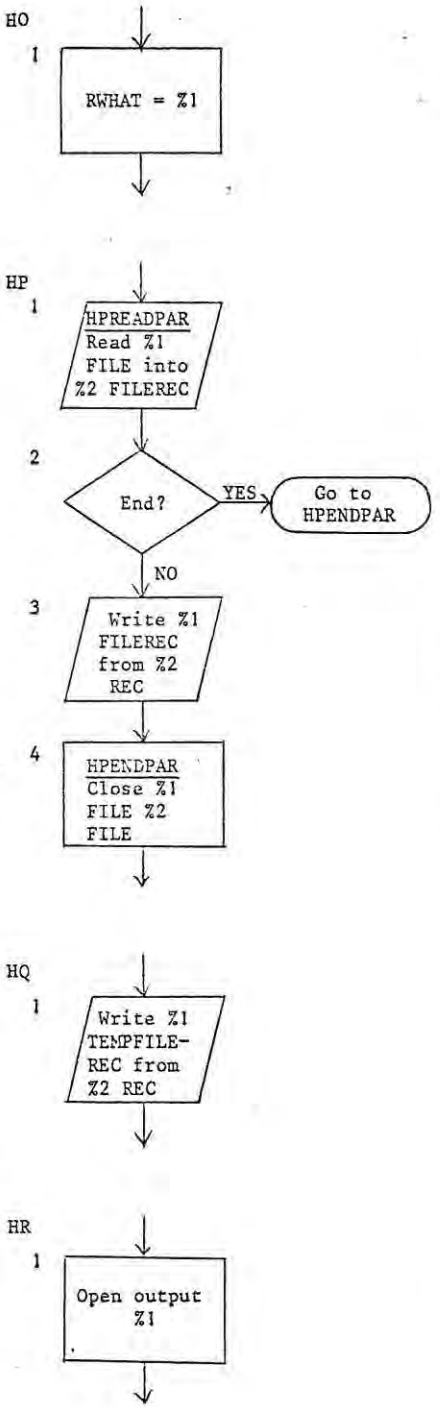

```
/@PERFORM PRINDL%1 .↑]
```




```
/@PERFORM PRINOF%1 .↑]
```

DIAGRAMMATIC REPRESENTATION

CORRESPONDING CODE




```
/@MOVE "%1 " TO RWHAT.↑]

/HPREADPAR.↑@READ %1 FILE INTO %2 FILEREC↑]

/@    AT END GO TO HPENDPAR.↑]

/@WRITE %1 FILEREC FROM %2 REC.↑@GO TO
HPREADPAR.↑]

/[HPENDPAR.↑@CLOSE %1 FILE %2 FILE.↑]

/@WRITE %1 TEMPFILEREC FROM %2 REC.↑]

/@OPEN OUTPUT %1 .↑]
```

A P P E N D I X 3

Error codes produced by the Question-Answering Program

Code value	Meaning
1	the pointer has reached the end of the buffer containing expected answer and action codes
2	the code 101 has been encountered in procedure SKIP or LCODE
3	the code 201 has been encountered in procedure SKIP or LCODE
4	the items to be compared in a condition are of different types
5	code 35 or 36 expected - i.e. an illegal code has occurred in a logical expression
6	; or the end of an action expected
7	read operation expected
8	code 22 expected
9	different types on either side of an assignment statement
10	stack name or number expected in the ↓ action
11	variable of type chstack expected after code 39
12	type chvar expected
13	stack number expected after code 61
14	type intvar expected
15	code 53 expected
16	code 54 expected
17	code 55 expected
18	parameter(s) expected (i.e. code 40 expected)
19	parameter type incorrect
20	conversion table number expected
21	parameter missing (i.e. code 41 expected)
22	item not found during execution of CONVERT
23	code 42 expected

Code value	Meaning
24	array GSTACKDIV is too small
25	code 49 expected
26	code 21 expected
27	flowchart box not found
28	pointer type expected
29	typebox type expected
30	code 10 expected
31	the link of a structure box other than the last box is zero
32	array FLOWCHART is too small
33	code 16 or 39 expected (i.e. a local stack name should follow)
34	BOXINFPTR exceeds LBOXINFO
35	the link of the last box of a structure is not zero
36	code 60, 61 or 92 expected (i.e. impermissible parameter type encountered)
37	too many parameters
38	parameters missing in a flowchart expansion operation
39	array BOXES is too small
40	code 22 or 28 expected
41	flowchart box whose link points to a box that must be eliminated, is not found
42	code 16 expected
43	code 39 expected
44	code 11 expected
45	code 12 expected
46	code 94 expected
47	= expected
48	code 13 expected
49	type intvar expected in WRITESTRING
50	name of an integer local stack expected after code 16

Code value	Meaning
51	item to be stacked onto or accessed from an integer local stack is not of type intvar
52	code 61 expected after code 73
53	code 82 expected
54	stack name or number expected in the ↑ action
55	the number of an element to be accessed exceeds the number of elements in a stack
56	global stack number expected in ⇓ action
57	global stack number expected in ⇑ action
58	attempt to access an element of a global stack that is not itself a stack, using ⇑
59	variable of type intvar expected in LOOP
60	stack name or number expected after code 26
61	code 38 expected in FOREACH
62	code 22 expected
63	variable or constant of type intvar expected in GET
64	code 53 expected in GET
65	local or global stack number expected in GET
66	GET has attempted to access an element of a stack which has a number greater than the number of elements in the stack
67	code 38 expected in GET
68	code 99 expected
69	global stack number expected after code 3
70	global stack number expected after code 4
71	code 4 expected
72	attempt to use MERGE with global stacks of different sizes
73	element values of first global stack in MERGE do not point to further stacks
74	element values of second global stack in MERGE do not point to further stacks
75	code 5 expected
76	type intvar expected in ADD

Code value	Meaning
77	code 53 expected in ADD
78	global stack number expected in ADD
79	code 54 expected in ADD
80	the number of the element specified exceeds the number of elements in the global stack to be operated on using ADD
81	the element of the global stack specified in ADD, is not itself a stack
82	code 56 expected
83	parameter expected after NOELMNTS
84	global stack number expected as parameter of NOELMNTS
85	type intvar expected in CONSTRING
86	parameter of type string expected in CONSTRING
87	parameter of type intvar or realvar expected in CHARSTRING
88	global stack number expected in CONCAT
89	a second parameter is expected in CONCAT
90	a string parameter is expected in CONCAT
91	an element of a global stack used in CONCAT is also a stack
92	typebox type expected in FIND or FORALL
93	item of type chvar or intvar expected in CONVERT
94	number of element exceeds the number of elements in a global stack in EVALPARS
95	code 41 or 42 expected in EVALPARS
96	code 16 or 39 or 61 expected in SEARCH
97	name of a local stack of type integer expected after code 16
98	code 8 expected
99	item of type intvar expected when searching an integer local stack or a global stack in SEARCH

APPENDIX 4

The character codes accepted by the Format Editor and their functions

In order to specify the format of information, files or reports, the user may use the following characters, with the effects described below :

a string of space characters	this will cause a FILLER of the appropriate size to be generated and preset to space values.
?	this character may be used in describing the format of a report and indicates a position at which the date is to be written, in the form dd/mm/yy.
+	this character is also for use in describing the format of a report, and indicates a point at which the current page number is to be written.
↑	this character must be followed by an integer, n, and indicates that a FILLER of size n must be generated and preset to space values.
↑	this character marks the end of a format specification.
%	this character indicates that the description of a field is to be given. It is followed by the number of the field (from the numbered list of contents provided) and causes the appropriate field name to be generated.
a string of characters which does not contain more than two consecutive embedded spaces, or any of the other characters listed above	this will cause a literal string to be set up (e.g. for a report heading)
"	marks the start of the description of a field
"	marks the end of the description of a field

The following characters may be used within enclosing quotation marks to describe a field and are converted as described :

@	this is converted to the COBOL character symbol X
#	this is converted to the COBOL character symbol 9
D	this indicates that a field description for the quantity has previously been described and this description must be used.

S	remains as the COBOL operational symbol S
*	this remains as the COBOL editing symbol *
R	is used as the currency symbol
.	is converted to the COBOL editing symbol V for information or files, and to V. for reports.

APPENDIX 5

Error codes produced by the Format Editor

Code Value	Meaning
1	Number of information processing codes supplied does not agree with the number of information-type inputs
2	An attempt has been made to obtain a pointer to a non-existent information, file or report name
3	An attempt has been made to obtain a pointer to one of the contents of information, a file, or a report which does not exist
4	An attempt has been made to obtain a pointer to a field name which is not one of the contents of the information, file or report being processed.
5	"%" expected in the input buffer
6	Unrecognized special code encountered
7	The number of sets of originality values provided for information contents does not tally with the number of information names
8	" expected in input buffer
9	Unrecognized character in field description
10	Digit expected after "%" in input buffer
11	The number of sets of contents of information provided differs from the number of information-type inputs specified as existing
12	The number of originality values provided for an information-type input, differs from the number of contents it has
13	The number of file version codes provided differs from the number of files specified as existing
14	The number of sets of contents of files provided differs from the number of files specified as existing
15	The number of originality values provided for a file differs from the number of contents it has

Code value	Meaning
16	A description has not been generated for a field which the user has specified must have the same description as a previously provided description
17	The number of sets of originality values provided for file contents does not agree with the number of files specified as existing

APPENDIX 5

Further Examples of Expected Answer Components of Questions

The purpose of presenting the following question was to give an example of the use of the function readcat and the action Analyse.

Q8 CATEGORIES FOR RESTRICTED ADDITIONS

EACH ADDITION WHICH IS APPLICABLE ONLY TO CERTAIN EMPLOYEES MAY BE APPLICABLE TO :

A) CERTAIN RACE GROUPS :- 1) WHITE
2) BLACK
3) COLOURED
4) ASIAN

B) PARTICULAR SEX :- 1) MALE
2) FEMALE

C) CERTAIN MARITAL STATUS : - 1) MARRIED
2) SINGLE
3) DIVORCED
4) WIDOWED

D) CERTAIN JOB CATEGORIES

E) OTHER CATEGORIES - THE NAMES OF THESE WILL BE ASKED IN FURTHER QUESTIONS.

RACE, SEX, MARITALSTATUS ETC. WILL BE FIELDS IN EACH EMPLOYEE RECORD. AS IT IS WASTEFUL OF SPACE TO STORE "WHITE" OR "FEMALE" ETC. AS FIELD VALUES IN EMPLOYEE RECORDS, NUMERIC CODE VALUES SHOULD BE USED, WHICH YOU MUST CHOOSE - EG. RACE=1 FOR WHITE, 2 FOR BLACK ETC. THE RESTRICTED ADDITIONS ARE LISTED BELOW, INDICATE NEXT TO EACH WHAT CATEGORIES OR COMBINATIONS OF CATEGORIES ARE APPLICABLE BY TYPING :

A LETTER TO CHOOSE A FIELD I.E. A,B,C,D OR E, FOLLOWED BY THE CHARACTER "=" FOLLOWED BY A NUMBER WHICH IS THE VALUE OF THE FIELD FOLLOWED BY EITHER / REPRESENTING OR, OR & REPRESENTING AND, OR ASTERISK,PERCENT IF THERE ARE NO MORE SEQUENCES TO FOLLOW.

EXPANS

θ (G13, w $\Leftarrow$ readcat; Analyse(letterstack, valstack, sepstack, rest);
@(letterstack, letterstack ↑ x, x ↓ anstack; if x = E then outpointer
↓ G14; epres $\Leftarrow$ true else convert (C5,x) ↓ G15 fi); @(valstack,
valstack ↑ y, y ↓ G16); @(sepstack, sepstack ↑ s, convert(C6,s) ↓ G17);
... ℓ(G15); ℓ(G16); ℓ(G17); ...)

Global stack 13 contains pointers to the names of all the additions. Each addition name is printed on the user's console and the user is invited to use the list provided to select categories to which the addition is applicable and to type his answers in the form requested by the question text.

The function readcat simply reads in a string of characters provided as an answer by the user until a comma, a space or the terminator, %, is encountered - much in the manner of the function readstring. The purpose of the action Analyse is to analyse the category answer into category selectors, category values and category separators and to stack these separate components onto separate stacks, ready to be used or processed. In the example above the category selectors are converted one by one into strings and the pointers to these are stacked in global stack 15. The category values are stacked as they are in global stack 16 and the category separators are converted to appropriate strings and the pointers thus produced are stored in global stack 17. The contents of global stacks 15, 16 and 17 are then used in further actions which have not been rewritten in the example, before being cleared, ready for use with the next addition.
The expected answer component of the following example illustrates the use of the functions readno and charstring.

Q26 VALUES OF FIXED ADDITION CLAIMS

EACH ADDITION WHICH IS PARTIALLY RECLAIMABLE AND FOR WHICH THE AMOUNT WHICH MAY BE CLAIMED IS FIXED AND THE SAME FOR ALL EMPLOYEES IS LISTED BELOW. TYPE THE VALUE OF THE AMOUNT WHICH MAY BE CLAIMED (IN RANDS AND CENTS, I.E. IN THE FORM D.DD), NEXT TO EACH ADDITION NAME.

EXPANS

θ (G15, r ⇐ readno; charstring(r) ↓ G17)

Global stack 15 contains pointers to the names of partially reclaimable additions. The function readno will return a result of type real as it will encounter a decimal point in the user's reply. The function charstring converts the real value to its character equivalent and stores this string in the character area, returning a pointer to the string which can be used as a parameter in a flowchart expansion.

The next example illustrates the use of θ within another θ action, as well as the usefulness of the outpointer and outcount variables.

Q115 CONTENTS OF FULL LISTS

THE NAMES OF ALL THE FULL LISTS ARE GIVEN BELOW, TOGETHER WITH THE NAMES OF ALL THE FIELDS. TYPE Y NEXT TO EACH FIELDNAME THAT APPEARS IN A PARTICULAR REPORT, AND N NEXT TO OTHER FIELDNAMES.

EXPANS

θ (G44, θ (G1, x ⇐ readchar; if x # Y and x # N then error fi;
if x = Y then outpointer ↓ G49; outcount ↓ G50 fi); G49 ⇓ G47;
G50 ⇓ G48; β (G49); β (G50))

Global stack 44 contains pointers to the names of all the full list reports, and global stack 1 contains pointers to all the field names. Each report name is listed and then each field name is listed for the user to select the contents of the particular report. As the user selects the contents of a report, the pointers to the chosen fields are stacked onto global stack 49 (by stacking the current value of outpointer onto stack 49), and the field number (i.e. the value of outcount) of the field is stacked onto global stack 50. Once all the fields from which it is possible to choose, have been listed, the global stacks 49 and 50 are stacked as single elements onto global stacks 47 and 48 respectively and then emptied for

use with the next report. At the end of executing this expected answer component, global stacks 47 and 48 will each have the same number of elements as global stack 44. The information of an element of stack 47 or 48 will pertain to the corresponding element of stack 44.

The final example has been chosen to demonstrate how a global stack with elements consisting of pointers to global stacks may be used in θ actions.

Q108 MATCH FIELDS FOR INFORMATION INPUT

THE INPUT INFORMATION FILES, FOLLOWED BY THEIR CONTENTS ARE LISTED BELOW. TYPE Y NEXT TO THE FIELD TO BE USED AS A MATCH FIELD - E.G. PERIODIC INFORMATION, THE EMPLOYEE NUMBER MAY BE MATCHED WITH THE EMPLOYEE NUMBER OF THE PAYROLL RECORD. ETC., AND N NEXT TO OTHER FIELDNAMES.

EXPANS

θ (G3, v $\Leftarrow$ outpointer; γ (outcount, G36, G36 $\Uparrow$ G38); θ (G38, x $\Leftarrow$ readchar; if x # Y and x # N then error fi; if x = Y then outpointer $\downarrow$ G40; constring (v,-,outpointer) $\downarrow$ G39 fi); β (G38))

Global stack 3 contains pointers to information names, and each element of global stack 36 consists of a pointer to an array containing pointers to field names which constitute the contents of the corresponding information names. An element of global stack 3 is listed and the corresponding element of global stack 36 is then accessed to create a temporary stack which is stored in global stack 38. The pointers in global stack 38 are used to access and list the contents of the input information one by one on the user's console. An answer is accepted each time a content name is listed. If the answer is acceptable, and happens to be the letter Y, the pointer to the content name is stored in global stack 40. This process is repeated for each element of global stack 3. At the end of executing this expected answer component, global stacks 3 and 40 will have the same number of elements. Each element of global stack 40 will contain a pointer

to the content name which is to be the match field of the information whose name is pointed to by the corresponding element of global stack 3.

APPENDIX 7

A listing of a sample generated program

```
IDENTIFICATION DIVISION.
PROGRAM-ID. PAYR50.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.  ICL-1902.
OBJECT COMPUTER.  ICL-1902.
    MEMORY SIZE 44000 WORDS.
SPECIAL-NAMES.
    *DATE IS TODAY.
    CURRENCY SIGN IS "R".
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT CARDFILE ASSIGN TO CARD-READER 0.
    SELECT PRINTFILE ASSIGN TO PRINTER 0.
    SELECT PAYROLLFILE ASSIGN TO EDS 3
           ACCESS MODE IS SEQUENTIAL.
    SELECT NEWPAYROLLFILE ASSIGN TO EDS 4
           ACCESS MODE IS SEQUENTIAL.
    SELECT PERIODICINFO ASSIGN TO CARD-READER 0.
    SELECT PERIODICFILE ASSIGN TO EDS 5
           ACCESS MODE IS SEQUENTIAL.
    SELECT PERIODICTEMP ASSIGN TO EDS 1
           ACCESS MODE IS SEQUENTIAL.
    SELECT PERIODICSORT ASSIGN TO EDS 2
           ACCESS MODE IS SEQUENTIAL.
    SELECT REPORTFILE ASSIGN TO EDS 6
           ACCESS MODE IS SEQUENTIAL.
I-O-CONTROL.
    APPLY SORTINFO TO SORT-* ON PERIODICSORT .
DATA DIVISION.
FILE SECTION.
FD  CARDFILE.
01  CARDREC PIC X(9).
FD  PRINTFILE.
01  PRINTLINE  PIC X(120).
FD  PAYROLLFILE
RECORDING MODE IS F
BLOCK CONTAINS 512 CHARACTERS
LABEL RECORDS ARE STANDARD
VALUE OF ID IS "            ".
01  PAYROLLFILEREC     PIC X(113).
FD  NEWPAYROLLFILE
RECORDING MODE IS F
BLOCK CONTAINS 512 CHARACTERS
LABEL RECORDS ARE STANDARD
VALUE OF ID IS "            ".
01  NEWPAYROLLFILEREC     PIC X(113).
FD  PERIODICINFO.
01  PERIODICINFOREC.
    02 PERIODICSAVINGS-DEPOSIT PIC 999V99.
    02 PERIODICCHARITYCONTRIB PIC 99V99.
    02 PERIODICEMPNO PIC 99999.
    02 PERIODICEMPNAME PIC XXXXXXXXXXXXXXX.
    02 PERIODICHRSWORKED PIC 99.
```

```
    02 PERIODICOTIMEHRSWORKED PIC 99.
FD  PERIODICFILE
RECORDING MODE IS F
BLOCK CONTAINS 512 CHARACTERS
LABEL RECORDS ARE STANDARD
VALUE OF ID IS "              ".
01  PERIODICFILEREC.
    02 SAVINGS-DEPOSIT PIC 999V99.
    02 CHARITYCONTRIB PIC 99V99.
    02 PERIODIC-EMPNO PIC 99999.
    02 PERIODIC-EMPNAME PIC XXXXXXXXXXXXXXX.
    02 HRSWORKED PIC 99.
    02 OTIMEHRSWORKED PIC 99.
FD  PERIODICTEMP
RECORDING MODE IS F
BLOCK CONTAINS 512 CHARACTERS
LABEL RECORDS ARE STANDARD
VALUE OF ID IS "              ".
01  PERIODICTEMPREC.
    02 PERIODICSAVINGS-DEPOSIT PIC 999V99.
    02 PERIODICCHARITYCONTRIB PIC 99V99.
    02 PERIODICEMPNO PIC 99999.
    02 PERIODICEMPNAME PIC XXXXXXXXXXXXXXX.
    02 PERIODICHRSWORKED PIC 99.
    02 PERIODICOTIMEHRSWORKED PIC 99.
SD  PERIODICSORT
RECORDING MODE IS F
BLOCK CONTAINS 512 CHARACTERS
LABEL RECORDS ARE STANDARD.
01  PERIODICSORTREC.
    02 PERIODICSAVINGS-DEPOSIT PIC 999V99.
    02 PERIODICCHARITYCONTRIB PIC 99V99.
    02 PERIODICEMPNO PIC 99999.
    02 PERIODICEMPNAME PIC XXXXXXXXXXXXXXX.
    02 PERIODICHRSWORKED PIC 99.
    02 PERIODICOTIMEHRSWORKED PIC 99.
FD  REPORTFILE
RECORDING MODE IS F
BLOCK CONTAINS 512 CHARACTERS
LABEL RECORDS ARE STANDARD
VALUE OF ID IS "              ".
01  REPORTFILEREC1.
    02 IDNO1  PIC 99.
    02 RBASICPAY1 PIC 9999V99.
    02 RGROSSPAY1 PIC 9999V99.
    02 RNETTPAY1 PIC 9999V99.
    02 RTIME-AND-A-HALF1 PIC 999V99.
    02 RRENT1 PIC 999V99.
    02 RUIF1 PIC 99V99.
    02 RMEDAID1 PIC 99V99.
    02 RPAYE1 PIC 99V99.
    02 RPENSION1 PIC 999V99.
    02 REMPNO1 PIC 99999.
    02 REMPNAME1 PIC XXXXXXXXXXXXXXX.
WORKING-STORAGE SECTION.
77  CDATE PIC X(8).
77  NDATE PIC 9(6).
01  FREQARR.
    02 FREQUENCY PIC 9 OCCURS 9 TIMES.
77  I PIC 99.
```

```
77  J PIC 99.
01  SORTINFO.
    02 LIBRFILE PIC X(12) SYNC RIGHT VALUE "PROGRAM DISC".
    02 LIBGEN PIC 1(24) SYNC RIGHT VALUE 1.
    02 LIBTYPE PIC 1(24) SYNC RIGHT VALUE 1.
    02 WKFILE1 PIC X(12) SYNC RIGHT VALUE "COB WORKFILE".
    02 WK1GEN PIC 1(24) SYNC RIGHT VALUE 1.
    02 WKFILE2 PIC X(12) SYNC RIGHT VALUE "COB WORKFILE".
    02 WK2GEN PIC 1(24) SYNC RIGHT VALUE 2.
    02 REPLY PIC 1(24) SYNC RIGHT VALUE 0.
01  ERR2MESS.
    02 FILLER PIC X(25).
    02 E2MES PIC X(60).
    02 E2EMPNO PIC X(10).
    02 FILLER PIC X(25).
01  PAYROLLREC.
    02 ENTERTAIN-ALLOWANCE PIC 99V99.
    02 RENT PIC 999V99.
    02 RENTFREQ PIC 9.
    02 SAVINGS-DEPOSITFREQ PIC 9.
    02 CHARITYCONTRIBFREQ PIC 9.
    02 PAYETAXYEARTOTAL PIC 9999V99.
    02 PENSIONTAXYEARTOTAL PIC 9999V99.
    02 SAVINGS-DEPOSITBALANCE PIC 9999V99.
    02 CHARITYCONTRIBBALANCE PIC 9999V99.
    02 SURNAME PIC XXXXXXXXXX.
    02 CHRISTIAN-NAMES PIC XXXXXXXXXXXXXXXXXXX.
    02 IDNO PIC 9999999999.
    02 DEPT PIC 999.
    02 HRLY-RATE PIC 999V99.
    02 DELETEDINDIC PIC 9.
    02 EMPNO PIC 99999.
    02 RACE PIC 9.
    02 PAYPOINTNO PIC 99.
    02 MEDAIDNO PIC 9999.
    02 RENTAGENT PIC 9.
    02 EMPNAME PIC XXXXXXXXXXXXXX.
01  OH1.
    02 OH1L1.
      03 FILLER PIC X(52) VALUE SPACES.
      03 OH1L1F1 PIC X(16) VALUE "PAYROLL-REGISTER".
      03 FILLER PIC X(52) VALUE SPACES.
    02 OH1L2.
      03 FILLER PIC X(52) VALUE SPACES.
      03 OH1L2F1 PIC X(16) VALUE "----------------".
      03 FILLER PIC X(52) VALUE SPACES.
01  DL1.
    02 DL1L1.
      03 DL1L1F1 PIC 9999V.99.
      03 DL1L1F2 PIC 9999V.99.
      03 DL1L1F3 PIC 9999V.99.
      03 DL1L1F4 PIC 999V.99.
      03 DL1L1F5 PIC 999V.99.
      03 DL1L1F6 PIC 99V.99.
      03 DL1L1F7 PIC 99V.99.
      03 DL1L1F8 PIC 99V.99.
      03 DL1L1F9 PIC 99V.99.
      03 DL1L1F10 PIC 99999.
      03 DL1L1F11 PIC XXXXXXXXXXXXXX.
01  OF1.
```

```
    02 OF1L1.
      03 FILLER PIC X(48) VALUE SPACES.
      03 OF1L1F1 PIC X(23) VALUE "END OF PAYROLL-REGISTER".
      03 FILLER PIC X(49) VALUE SPACES.
77  PAGEND1  PIC 99 VALUE 66.
77  BASICPAY PIC 9999V99.
77  GROSSPAY PIC 9999V99.
77  NETTPAY PIC 9999V99.
77  TIME-AND-A-HALF PIC 999V99.
77  SUBSIST-ALLOWANCE PIC 99V99.
77  INCENTIVE-BONUS PIC 99V99.
77  PROFITBONUS PIC 99V99.
77  TAXGROSS PIC 9999V99.
77  PENSIONGROSS PIC 9999V99.
77  PENSIONNETT PIC 9999V99.
77  UIF PIC 99V99.
77  MEDAID PIC 99V99.
77  PAYE PIC 99V99.
77  PENSION PIC 999V99.
77  TAXADDNS PIC 999V99.
77  NONTAXADDNS PIC 999V99.
77  NONPENSIONADDNS PIC 999V99.
77  LINECOUNT PIC 99.
77  PAGECOUNT PIC 999.
77  ERRORINDICATOR PIC 9.
77  DATACORRECT PIC 9.
77  ENDFLAG PIC 9.
PROCEDURE DIVISION.
MAIN SECTION.
BEGIN.
    OPEN INPUT CARDFILE OUTPUT PRINTFILE.
    ACCEPT NDATE FROM TODAY.
    ACCEPT CDATE FROM TODAY.
    READ CARDFILE INTO FREQARR.
    CLOSE CARDFILE.
START.
    OPEN INPUT PERIODICINFO.
    OPEN OUTPUT PERIODICTEMP.
JREADPAR.
    READ PERIODICINFO
        AT END GO TO JREADEND.
    MOVE CORRESPONDING PERIODICINFOREC TO PERIODICTEMPREC.
    WRITE PERIODICTEMPREC.
    GO TO JREADPAR.
JREADEND.
    CLOSE PERIODICINFO.
    CLOSE PERIODICTEMP.
    SORT PERIODICSORT ON
       ASCENDING KEY PERIODICEMPNO
     OF PERIODICSORTREC
        USING PERIODICTEMP GIVING PERIODICFILE.
    OPEN INPUT PAYROLLFILE OUTPUT NEWPAYROLLFILE.
    OPEN INPUT PERIODICFILE.
INITIALIZE.
    MOVE ZERO TO NONPENSIONADDNS.
    MOVE ZERO TO NONTAXADDNS.
    MOVE ZERO TO TAXADDNS.
READEMPREC.
    READ PAYROLLFILE INTO PAYROLLREC
        AT END GO TO ENDEMPS.
```

```
     READ PERIODICFILE.
          AT END GO TO ERROR-1.
     IF PERIODIC-EMPNO IS EQUAL TO EMPNO
     THEN GO TO CALCNS.
     PERFORM ERROR-2.
     GO TO READEMPREC.
 CALCNS.
** CALCULATE BASIC PAY **
     IF FREQUENCY (1) IS NOT EQUAL TO 1 THEN GO TO ENDAI1.
** CALCULATE TIME-AND-A-HALF **
     ADD TIME-AND-A-HALF TO TAXADDNS.
 ENDAI1.
     IF FREQUENCY (2) IS NOT EQUAL TO 1 THEN GO TO ENDAI2.
     ADD ENTERTAIN-ALLOWANCE TO TAXADDNS.
 ENDAI2.
     IF FREQUENCY (1) IS NOT EQUAL TO 1 THEN GO TO ENDAI3.
     MOVE 5.30 TO SUBSIST-ALLOWANCE .
     ADD SUBSIST-ALLOWANCE TO NONTAXADDNS.
 ENDAI3.
     IF FREQUENCY (3) IS NOT EQUAL TO 1 THEN GO TO ENDAI4.
** CALCULATE INCENTIVE-BONUS **
     ADD INCENTIVE-BONUS TO NONPENSIONADDNS.
     ADD INCENTIVE-BONUS TO TAXADDNS.
 ENDAI4.
     IF FREQUENCY (1) IS NOT EQUAL TO 1 THEN GO TO ENDAI5.
** CALCULATE PROFITBONUS **
     ADD PROFITBONUS TO NONPENSIONADDNS.
     ADD PROFITBONUS TO TAXADDNS.
 ENDAI5.
** CALCULATE GROSS PAY **
     ADD BASICPAY TAXADDNS NONTAXADDNS GIVING GROSSPAY.
     SUBTRACT NONPENSIONADDNS FROM GROSSPAY GIVING PENSIONGROSS.
     ADD TAXADDNS BASICPAY GIVING TAXGROSS.
     MOVE PENSIONGROSS TO PENSIONNETT.
     MOVE GROSSPAY TO NETTPAY.
     MOVE RENTFREQ TO I.
     IF FREQUENCY (I) IS NOT EQUAL TO 1 THEN GO TO ENDEE1.
     SUBTRACT RENT FROM NETTPAY.
 ENDEE1.
     IF FREQUENCY (1) IS NOT EQUAL TO 1 THEN GO TO ENDBX1.
     MOVE 1.87 TO UIF .
     SUBTRACT UIF FROM NETTPAY.
 ENDBX1.
     MOVE SAVINGS-DEPOSITFREQ TO I.
     IF FREQUENCY (I) IS NOT EQUAL TO 1 THEN GO TO ENDEE2.
     SUBTRACT SAVINGS-DEPOSIT FROM NETTPAY.
     ADD SAVINGS-DEPOSIT TO SAVINGS-DEPOSITBALANCE.
 ENDEE2.
     IF FREQUENCY (1) IS NOT EQUAL TO 1 THEN GO TO ENDBX2.
** CALCULATE MEDAID **
     SUBTRACT MEDAID FROM NETTPAY.
 ENDBX2.
     MOVE CHARITYCONTRIBFREQ TO I.
     IF FREQUENCY (I) IS NOT EQUAL TO 1 THEN GO TO ENDEE3.
     SUBTRACT CHARITYCONTRIB FROM NETTPAY.
     ADD CHARITYCONTRIB TO CHARITYCONTRIBBALANCE.
 ENDEE3.
     IF FREQUENCY (1) IS NOT EQUAL TO 1 THEN GO TO ENDBX3.
** CALCULATE PAYE **
 ENDBX3.
```

```
     IF FREQUENCY (1) IS NOT EQUAL TO 1 THEN GO TO ENDBX4.
** CALCULATE PENSION **
     SUBTRACT PENSION FROM TAXGROSS.
     SUBTRACT PENSION FROM NETTPAY.
 ENDBX4.
     ADD PAYE TO  PAYETAXYEARTOTAL.
     ADD PENSION TO  PENSIONTAXYEARTOTAL.
     OPEN OUTPUT REPORTFILE.
     IF FREQUENCY (1) IS NOT EQUAL TO 1 THEN GO TO ENDEP1.
 PREPPAYROLL-REGISTER.
     MOVE 1 TO IDNO1.
     MOVE BASICPAY TO RBASICPAY1.
     MOVE GROSSPAY TO RGROSSPAY1.
     MOVE NETTPAY TO RNETTPAY1.
     MOVE TIME-AND-A-HALF TO RTIME-AND-A-HALF1.
     MOVE RENT TO RRENT1.
     MOVE UIF TO RUIF1.
     MOVE MEDAID TO RMEDAID1.
     MOVE PAYE TO RPAYE1.
     MOVE PENSION TO RPENSION1.
     MOVE EMPNO TO REMPNO1.
     MOVE EMPNAME TO REMPNAME1.
     WRITE REPORTFILEREC.
 ENDEP1.
     IF FREQUENCY (6) IS NOT EQUAL TO 1 THEN GO TO ENDEA1.
     MOVE ZERO TO PAYETAXYEARTOTAL.
 ENDEA1.
     IF FREQUENCY (6) IS NOT EQUAL TO 1 THEN GO TO ENDEA2.
     MOVE ZERO TO PENSIONTAXYEARTOTAL.
 ENDEA2.
     WRITE NEWPAYROLLFILEREC FROM PAYROLLREC.
     GO TO READEMPREC.
 ENDEMPS.
     CLOSE PAYROLLFILE NEWPAYROLLFILE.
     CLOSE PERIODICFILE.
     CLOSE REPORTFILE.
     IF FREQUENCY (1) IS NOT EQUAL TO 1 THEN GO TO ENDES1.
     OPEN INPUT REPORTFILE.
 WRPAYROLL-REGISTER.
     MOVE ZERO TO PAGECOUNT.
     MOVE ZERO TO ENDFLAG.
 NEWPAGE1.
     MOVE SPACES TO PRINTLINE.
     WRITE PRINTLINE AFTER CHANNEL-1.
     ADD 1 TO PAGECOUNT.
     MOVE 1 TO LINECOUNT.
     IF PAGECOUNT IS EQUAL TO 1
       THEN PERFORM PRINOH1.
     PERFORM PRINPH1.
     PERFORM PRINSH1.
 NEXTREC1.
     READ REPORTFILE
         AT END GO TO SETENDREP1.
     IF IDNO1 IS NOT EQUAL TO 1 THEN GO TO NEXTREC1 ELSE GO TO RIGHTREC1.
 SETENDREP1.
     MOVE 1 TO ENDFLAG.
     PERFORM PRINSF1.
 RIGHTREC1.
     IF ENDFLAG IS EQUAL TO 1 THEN GO TO ENDREP1.
     IF LINECOUNT IS LESS THAN PAGEND THEN GO TO WRLINE1.
```

```
     PERFORM PRINPF1.
     PERFORM NEWPAGE1.
WRLINE1.
     PERFORM PRINDL1.
     GO TO NEXTREC1.
ENDREP1.
     PERFORM PRINOF1.
     CLOSE REPORTFILE.
ENDES1.
ENDOFF.
     CLOSE PRINTFILE.
     STOP RUN.
PERFORMPARS.
PRINOH1.
     MOVE OH1L1 TO PRINTLINE.
     WRITE PRINTLINE AFTER ADVANCING 1 LINES.
     ADD 1 TO LINECOUNT.
     MOVE OH1L2 TO PRINTLINE.
     WRITE PRINTLINE AFTER ADVANCING 1 LINES.
     ADD 1 TO LINECOUNT.
PRINPH1.
PRINSH1.
PRINDL1.
     MOVE RBASICPAY1 TO DL1L1F1.
     MOVE RGROSSPAY1 TO DL1L1F2.
     MOVE RNETTPAY1 TO DL1L1F3.
     MOVE RTIME-AND-A-HALF1 TO DL1L1F4.
     MOVE RRENT1 TO DL1L1F5.
     MOVE RUIF1 TO DL1L1F6.
     MOVE RMEDAID1 TO DL1L1F7.
     MOVE RPAYE1 TO DL1L1F8.
     MOVE RPENSION1 TO DL1L1F9.
     MOVE REMPNO1 TO DL1L1F10.
     MOVE REMPNAME1 TO DL1L1F11.
     MOVE DL1L1 TO PRINTLINE.
     WRITE PRINTLINE AFTER ADVANCING 1 LINES.
     ADD 1 TO LINECOUNT.
PRINSF1.
PRINPF1.
PRINOF1.
     MOVE OF1L1 TO PRINTLINE.
     WRITE PRINTLINE AFTER ADVANCING 1 LINES.
     ADD 1 TO LINECOUNT.
FOR1.
LAST SECTION.
NEWPAGE.
     MOVE SPACES TO PRINTLINE.
     WRITE PRINTLINE AFTER CHANNEL-1.
ENDNEWPAGE.
WRITELINE.
     WRITE PRINTLINE AFTER 1 LINES.
     MOVE SPACES TO PRINTLINE.
ENDWRITELINE.
ERROR-1.
     MOVE "ERROR:- END OF PERIODIC FILE READ" TO PRINTLINE.
     PERFORM WRITELINE.
     GO TO ENDOFF.
ERROR-2.
     MOVE "ERROR:- NO MATCHING PERIODIC INFORMATION FOUND FOR EMPLOYEE " TO E
-    RMESS.
```

```
MOVE EMPNO TO E2EMPNO.
MOVE ERR2MESS TO PRINTLINE.
PERFORM WRITELINE.
```