EDITING BY COMPUTER

A thesis submitted for the degree of

MASTER OF SCIENCE

at the

Australian National University
Camberra

June 1967

by

David MENDUS
B.A.(Hons), University of Oxford.
# CONTENTS

**PREFACE**

**SUMMARY**

**STATEMENT**

**CHAPTER 1. INTRODUCTION**

1.1 Introduction and definitions.  
1.2 Error checks.  
1.3 Automatic adjustment.  
1.4 Some extensions and problems.  
1.5 The logical edit.

**CHAPTER 2. LOGICAL EDIT SPECIFICATION**

2.1 Logical edit design.  
2.2 The extent of the language.  
2.3 Existing techniques.  
2.4 Evaluation of existing methods.  
2.5 The suggested approach.

**CHAPTER 3. THE LANGUAGE**

3.1 Introduction.  
3.2 The structure of FRED.  
3.3 Executable statements.  
3.4 Declarative statements.  
3.5 Examples.

**CHAPTER 4. THE IMPLEMENTATION**

4.1 Pre-processor form.  
4.2 Compiling techniques.  
4.3 The recogniser.  
4.4 Processing routines.  
4.5 Implementation limitations.
CHAPTER 5. USING RED

5.1 Example 1
5.2 Example 2
5.3 Possible improvements.

APPENDIX A SYNTAX NOTATION

APPENDIX B JOB CONTROL LANGUAGE

REFERENCES

APPENDIX C A LISTING OF THE PREPROCESSOR

APPENDIX D THE EXAMPLES

attached

attached
PREFACE

I wish to thank Mr B. W. Smith for supervising this thesis, and for his many valuable criticisms and suggestions.

I wish also to thank the Commonwealth Statistician for permission to include work done as a member of the Bureau of Census and Statistics, Canberra.
SUMMARY

Editing and correction of records by computer is very important when large volumes of data are being processed. As well as improving the quality of the results, a well designed edit can also simplify the processing of the data by detecting and correcting records which could cause the processing routine to malfunction.

Chapter 1 considers this task, and justifies its division into two sections. File handling and input/output is dependent on the machine and peripherals used, but requires no knowledge of the logical content of the record. On the other hand, the checking of the items and generation of error messages is less machine dependent, but needs a detailed knowledge of the logical content of the record. This latter task, called the "logical edit" is considered in this thesis.

The creation of a logical edit involves three phases; analysis, programming and testing. Commonly a different language is used for the analysis and the programming, and this can lead to errors and difficulty with the testing. Using decision tables the first two phases can be combined, as decision tables can be used as a programming language in conjunction with a suitable compiler.

However, they are not ideal as a means of specifying a logical edit. In particular, they are cumbersome and time consuming. A need exists for a better language that can be used both for analysis and as a computer program. Chapter 2 considers the properties of such a language.

"FRED", a language devised by the author to fill this need, is described in Chapter 3.
The statements of FRED represent the most common logical tasks required in editing, and include such features as look-up tables, error flag setting, adjustments and so on, not normally contained as single statements in a programming language. To deal with less common tasks, arbitrary blocks of PL/I statements may be included in an edit.

FRED also has a structure to represent the branches of the edit in an intuitive way. This structure exists in two forms, one of which is used for the program itself, and hence appears on the program listings. This makes a program written in FRED particularly easy to read and debug.

FRED is written to be compatible with PL/I, and a FRED edit will generally form one procedure of a PL/I program to perform all editing tasks.

Chapter 4 describes the way in which a FRED pre-processor has been written and tested. The pre-processor is written in PL/I and converts an edit written in FRED into a logically equivalent PL/I procedure. Any PL/I statements read by the pre-processor are copied unchanged so that a program may be written of mixed PL/I and FRED blocks, and the whole submitted first to the pre-processor, then to the PL/I compiler to produce an edit.

Chapter 5 contains sample edits written in FRED, and mentions improvements suggested by the early use of FRED.

The pre-processor is listed as an appendix.
STATEMENT

The material reported in this thesis is the author's own work except where references to other sources are explicitly stated in the text.

[Signature]

Donald Mendes.
1. INTRODUCTION
1.1.  **INTRODUCTION AND DEFINITIONS.**

Editing is a process to check the accuracy of a set of items of information. These items are usually organised into logical records (frequently referred to as records). A logical record is a collection of related items; for example, if a set of items is the information collected for an area in a population census, then those items relating to one dwelling may form a logical record.

The items may also be grouped because of the properties of the medium which stores them. For example, only a few items may be contained on one punched card or one length of paper tape. Such a group is called a physical record. In many applications there is a simple relation between logical and physical records.

A logical record is represented by a sequence of data characters on some medium together with an implied structure which enables these characters to be interpreted as the items of information. The data normally consist of alphanumeric characters which represent the information, and separators, which may also be alphanumeric, which do not represent information, but which, in conjunction with the implied structure, enable the items of information of the record to be determined. Thus, the character string "£142/10/4" contains three separators "£", "/" and "/", and these enable the characters "142104" to be interpreted as one hundred and forty two pounds, ten shillings and fourpence.

Although edits are sometimes designed to attempt to detect and remove all errors from the given file of information, it is more common for the edit to be used to
improve the quality of the information and remove these errors which will impede later operations on the file. Thus for example, in an edit attached to the job of computing pay, an attempt will be made to remove all errors by including redundant information such as checksums to allow the checking of all items. In a typical statistical application on the other hand, such an edit would be impractical with the available resources and the edit will be designed to remove all errors which would seriously affect the statistics and merely reduce other errors to an acceptable level. The edit may also be used to point out exceptional, but correct records. For example, the import of a warship costing several millions of dollars disturbs the detailed import statistics so much that special mention may have to be made. Such events are easily highlighted at the editing stage.

A further use of the edit is to simplify later processing tasks. Difficult situations such as zero divisors can often be detected at the edit stage and appropriate action taken quite easily, thus simplifying the later processing runs. Missing and absurd information such as illegal code values can also be detected and corrected during the editing. Also at this stage codes and serial numbers can be inserted to be of use in the later tasks, such as a serial to be used as a subscript in later tabulation routines, and batch totals can be computed to check on the completeness of the data.

The errors arise in two ways. The first is due to incorrect, incomplete or inconsistent information at the earliest stage, for example, an error of observation, or the wrong stock number written on a package or an incomplete census return. These errors will be referred to as "source
errors. The second type of error is that caused by a mistake in the extraction, transcription or storage of the information. For example an error caused by punching an incorrect character into paper tape, or by including some records twice in the file, or an error while reading the data from magnetic tape. These errors will be referred to as "transcription errors".

The value of editing is such that it is common to include redundant information just for the purpose of providing an editing check. Thus, the total of a group of items may be included in the record, or the rate at which duty is levied may be included in an excise record even though this should be determinable from the nature of the item, and a check digit may be applied to a code number.

1.1.2 There are many advantages in using a computer for editing. The types of logical and arithmetic checks that are most useful can be performed quickly and cheaply on a computer. Further, the computer does not get bored by applying the same checks over and over again, and continues to operate with very high reliability.

The value of computer editing is increased if the edit is used to query unlikely events as possible errors and also to make corrections where possible to the errors found.

The unlikely events queried in this way can then be checked manually for two important conditions. First, reference to source documents or beyond may show that the record is in error. For example this will usually be the case when hospital records report the birth of quintuplets, though it may sometimes be correct. Secondly, the
information may be correct, but so significant as to need special action, such as the case of the import of a warship mentioned earlier. Records queried in this way form a class which must usually be distinct from the class of records found to be in error, since many programs are written that do not allow further processing to be done on a file that contains errors, whereas, as noted above, many of these queried records will be correct and have to be processed in the usual way. The attention that can be given to these exceptional cases when singled out in this way is much greater than is possible when this type of record has to be searched for by hand from the whole set of information.

Making corrections by computer is a very valuable technique, though it is very difficult to program. Indeed some corrections that can be recognised fairly easily manually seem too complex to program using existing techniques. An example is recognising that a punched card has been read upside down. However, many simple transcriptions and source errors or omissions can be properly corrected by program, particularly when the records contain numeric data. Corrections are particularly valuable in work of a statistical nature, where complete accuracy is not expected, and limited resources make it impractical to investigate all source-errors. Here, it is a common procedure to apply an "arbitrary adjustment", that would make the information an acceptable record with at the worst, an insignificant effect on the accuracy of the final statistics. These adjustment rules can be programmed very effectively, and the ability of the computer to make complicated decisions very rapidly allows these adjustment routines to take account of many more relations, and hence become more accurate. Further
the reliability and predictability of the computer allows the effect of such rules to be estimated with a high degree of accuracy, and steps can be taken to reduce bias in a way that is not possible with clerical amendment.

Using the computer for editing has a further important advantage in the very common case of the further processing also being done by computer. This is that the incidence of transcription errors between editing and processing is reduced to a minimum. If the processing occurs immediately after editing, then no transcription occurs at all, and the data will only be altered by a malfunction of the computer - the likelihood of which can truly be regarded as negligible.

If the processing occurs at another time, even on another computer, then the data can be stored on one of the magnetic mass-storage media such as magnetic tapes, discs or drums. The reliability of these devices is far greater than that of mechanical readers and writers involving for example paper tape or punched cards. The error rate will generally be less than one error in 100,000,000 characters, and since an error is generally detected by the reading device or by program, using such devices as checksums and parity bits, the chances of an undetected error will generally be even less.

There is, however, one severe problem in editing by computer. All types of error must be anticipated and included in the program before editing commences. This is naturally a much more difficult task than that of recognising errors as they occur.

Computers do not have, nor will they have in the foreseeable future, the ability to store many items of
information over a large period of time and access them by a rapid process of association. Thus, if the edit of a personnel file specifies no check on date of birth and if due to a mechanical fault a large number of records are transcribed with dates of birth in, say, the seventeenth century, these will not be detected by a computer edit and may remain unnoticed until a report quotes the average age of employee as 342 with embarrassing consequences! Using clerical editing however, there is a fair chance that such errors, unforeseen initially, may be discovered and corrected before much damage is done.

The original analysis of the information for computer editing needs to be very complete. Though this analysis can be tested by test runs on actual data, the results of which are closely examined, much work still needs to be done on this problem.

1.1.3 Once the computer edit has been written it can be used rapidly and frequently. Thus, the data can be edited in conveniently sized batches soon after these have been received. There are several advantages to such a scheme.

Any source errors that are detected can be queried soon after the information is collected. Memories will still be fresh, documents still accessible and respondents found easily. The quality of the corrections to these errors will then be much higher than if the query was made later.

Transcription errors which are detected may lead to repairs to the transcription devices to correct a recurring error, or further explanation to an operator or coder who is misinterpreting the instructions. In this way the build up of a large number of systematic transcription errors may be avoided.

Analysis of the errors recorded may lead to improvements to the editing itself, or may highlight
deficiencies in the information being provided. Thus the system may be improved at the earliest possible moment.

Because of these considerations, editing is usually carried out at an early stage of computer operations on the file, and is often associated with the transcription from an external medium such as punched cards, paper tape or documents for an optical reader, to a computer oriented medium such as magnetic tape. This procedure has the further advantage that the other computer runs can be written with the assumption that the data will contain no destructive errors such as zero divisors or out of range subscripts, since these can be excluded during the edit.

In some cases the edit will form a computer run on its own, in others it will merely be a part of a run to perform several steps. If the information arrives in batches over a period of time, but cannot be processed until all has been edited, then a batch will normally be edited as soon after its receipt as possible by a separate edit routine. If, on the other hand, the volume of data is small and the edits are not too complicated, it may be desirable to edit and process the data (or at least the acceptable section) at once. Another factor in this decision is whether or not the combined editing and processing programs can be conveniently fitted into computer memory directly or by a suitable overlay process.

Though there are many schemes for performing edits, the two represented by the attached flow charts are typical of a large number of edit runs.

The first flow chart represents an application where there is only a small volume of data, and the error rates will be low, for example a scientific computation. Here the edit would be the first part of the processing program.
The second flow chart represents a larger application where the edit is a separate computer program. The output of this program will be a file of records now arranged in the form best suited for later processing. Since the record size is frequently large, even error records are included on the file and a second edit program is written to correct the file by altering records, removing records or inserting records. Records altered or inserted in this program will generally be edited with the same edits as the original records.

References (2) and (3) discuss the importance of editing and its relation to other computer operations.
(i) First edit run, repeated as required.

(ii) Amendment run, repeated as required.
11.1.4 There are two kinds of output from an edit. The first is the file of logical records in their edited form, and the second is a file of the messages about the records that were generated by the edit. Although the form, and even the number of output files of the above types vary from application to application the general terms "edited file" and "error listings" will be used for the two types.

In those programs where the editing and processing are combined, the term edited file can be applied to the one logical record currently being processed, but where the editing is separated from the processing then all edited records must be stored on an external file. Since this is created by, and used exclusively by the computer it is usually written on one of the magnetic mass storage media, tape, discs or drums. This file will frequently be very different in format to the original data file.

The logical record will frequently have a more rigid format. It is common, especially when using paper tape as an input medium, to economise on characters by devices such as excluding leading zeros, non-significant-blanks and omitting sections of the record if possible. This is valuable in that it reduces significantly the number of characters to be punched, often the most expensive operation of the whole processing, but it does mean that the layout of the records will vary considerably on the file and a special routine is needed to interpret the varying forms of structures. As this routine may be quite extensive it is desirable to use it once only and after the first reading the file kept in a rigid format which may include a large number of non-significant characters but which is easy to interpret.

The logical record may also contain items which were not present on the input record. These items may include.
values that were implicit in the input form of the record and which are needed explicitly for processing. These values may not have been present in the source data or they may have been omitted during transcription. For example, the results for weekly turnover for a retail store may include the figures for "values of goods sold including sales tax", and "value of sales tax", from which "value of goods sold exclusive of sales tax" can be input for processing.

Other added items may be special items that are attached for sorting, classification or tabulation. The derivation of these extra items may be very complicated, and it is advisable to perform this on a once-and-for-all basis. The logical edit is a suitable time for such computation. An example might be to add a "size code" of 1 if the stores turnover is less than 10,000 annually, 2 if it is 10,000 but less than 50,000 or 3 if it exceeds 50,000. If figures were later required for medium sized stores only it is easier to consider those shops of size code 2 only, than to test the turnover of every record, particularly as each record may be edited once only, but processed on several different runs.

A unique identifier or serial number may be attached to each record on the edited file. This will enable amendments to be specified to a particular record without fear of ambiguity. A check digit may be attached to this number to reduce the chances of a misquoted or badly transcribed number causing an adjustment to the wrong record. There may also be a flag attached to each record specifying the condition on the record - whether good, queried or in error. This may be referred to so that bad records may be excluded from processing runs.
In very large jobs the edited file is sometimes split into two. One file will contain only those records that have passed the edit, or possibly only those batches for which all records have passed the edit. This file should contain a large proportion of the records and may be processed at once. The other file contains all bad or doubtful records. The main advantage is that to process the amendments it is necessary to read only the shorter file, not the whole set of data. The running time of the amendments program should be correspondingly reduced.

The error listings in some cases may contain details of every record, but more commonly list only those records that were found to be in error, were queried or had adjustments made to them. The important features are that the items of the record are clearly laid out in a way that facilitates comparison with the source information, that clear but concise messages give details of the errors found, and that the serial number is included ready for the amendment processing. In some cases it is possible to design the listing so that corrections may be made on the listing which is then used as a source document for the data preparation of the amendment.

A separate error listing may be needed for those records so bad that they cannot be interpreted as items of information. These records are particularly common on paper tape. Since the items, which are the most important feature of the above listing cannot be distinguished, a completely new layout is required to show the record as a string of characters. Again error messages will be given; these will generally refer to unintelligible separators. Since the record is so bad, it is rarely worthwhile to include it on
the edited file, so unless the sequence of records is so important that a space must be reserved for the corrected version of this record, no serial number will be allotted to it. Since the layout of these listings is so dependent on the source information, little more will be said about them. Reference (4) discusses the uses of these listings in relation to the needs of one organisation.
1.2 ERROR CHECKS.

1.2.1 An a priori knowledge of the nature of the items of information and the relationships between them is needed to define the error checks. Nordbotten (5) classifies this knowledge into two general classes, theoretical knowledge and empirical knowledge.

Theoretical knowledge is the knowledge given by the definitions of the items and relations. Thus, if an item is defined to be numeric it cannot contain an alphabetic character, and value for sales tax multiplied by rate of sales tax must equal sales tax collected. If an item, or a relation, fails a check based on theoretical knowledge, then there must be an error. In most applications, the record cannot be accepted without some adjustment so that the record now satisfies the check. Theoretical checks are easier to define than empirical checks, and are fundamental to the editing process. However, the value of the edit will be much increased if empirical checks are added to the core of theoretical checks.

Empirical knowledge is knowledge about the items of information that is true as a matter of fact and not of necessity. This will be gathered from experience of previous records of this type or by estimation of the values that may occur from models, samples or similar edits. Thus, the yield of oats should not exceed 40 bushels per acre, or the temperature during an experiment should not exceed 120 degrees.

In general, it is not possible to say that an item or group of items which fail an empirical check are in error; all that can be said is that there is only a low probability
that the items are correct. However, it may be desirable, though not rigorous, to say that an item failing on empirical check is in error.

The limit on such a check may be chosen to satisfy one of two criteria. Firstly, the probability that a good record is rejected as in error may be kept to some limit or, secondly, the probability that a bad record may be accepted as correct can be kept to some limit. As there are more good records than errors, and as the good records are "better behaved", it is more convenient in practice to satisfy the first criterion, and this is usually done.

Empirical checks may be applied on a two-level basis, for example "if the age at first marriage exceeds 70 then query the record; if it exceeds 100 then the record is in error". Other extensions of the empirical checks are to reset the limits during editing either to refine the limits, or to follow trends, or to use historical or other external information particular to that record.

No further distinction is made between theoretical and empirical checks in this section.

1.2.2. The most common type of check is that in which one item is checked against values or rules written into the edit program. In its simplest form this is the "validity check" that is applied to most items of information to ensure that the data characters are of an acceptable form. Thus a numeric item should contain only numeric characters and a name must contain only alphabetical characters, spaces, hyphens and apostrophes and the first and last characters must be alphabetic. The validity check may be extended
to take special action on encountering certain characters; for example, as well as checking that there are numeric characters only in an item, special action may be taken on reading "?", which is often used to denote a transcription error while converting data from one medium to another by computer.

Other checks that can be applied to one item of information alone use more knowledge of what the information represents. For example, the code for sex must be M or F, or the amount of income tax paid annually must exceed one dollar. Large look-up tables may be required to establish validity in some cases; for example, country of origin codes on overseas trade statistics number over 100. The item may even be split into sub-items and a relation between these examined, for example a number with a check digit.

1.2.3. A second class of checks consists of those carried out by applying logical or arithmetic checks to two or more items within one record. These checks are often referred to as "cross-checks". They apply to the information represented by the items and usually assume that the data characters are valid. For this reason these checks commonly follow a series of validity checks and are often omitted when one or more of the items involved has failed a previous validity check.

Logical relations of this type are widely applied to numeric, coded or alphabetic information. Very common are simple existence relations, if A then B; for example, if number of hours of overtime is nonzero then extra pay must be nonzero. More complicated logical relations would generally involve the meanings of various items and codes;
for example if relation to head of household on a population census form is D for daughter, then sex must be F, female.

For arithmetic information, a large class of functional checks is available. In the most general form such a check requires that the value of some function $f$ of the items being checked must lie between the values of two other functions $a$ and $b$ of items of the record; that is

$$a(x_1, \ldots, x_n) \leq f(x_1, \ldots, x_i) \leq b(x_1, \ldots, x_n)$$

where generally $a$ and $b$ are simple functions such as constants or values obtained from a look-up table on one item. The principal function $f$ may however be quite complicated.

Some special cases of this check are very important. One is a check where $f$ is the ratio of two items and $a$ and $b$ are constants or simple functions of a third item. For example, the ratio of the amount of wheat produced to the number of acres planted must lie between two limits, each dependant on the state in which the farm is situated.

Another important case is that where $f$ is an algebraic sum of several items and $a$ and $b$ are either both zero or some small value, negative and positive respectively. For example, in a report on the composition of a chemical mixture: $-a \leq (\text{sum of weights of component items} - \text{total weight}) \leq a$ where the value $a$ has been chosen to allow for any reasonable weighing inaccuracies, but is still small enough to detect most transcription errors.
Further checks are possible when more than one record is edited at a time. These checks are similar in structure to those described above but involve items from more than one record. Again, these checks usually assume the validity of the data characters has been checked previously, and may be omitted when items fail a validity check. There are three ways in which checks of this kind can be contructed.

Firstly, the file of records may have a heirachal structure. The records in such files refer to information on two or more levels. For example, records at the first level might be a summary of a group of records at the second level. This may occur, for instance, in the records of a blood bank where the first level record would contain details of the blood required by one hospital served by the bank. This would be followed by a set of second level records listing the blood required for each surgeon in the hospital. An obvious check is to confirm that the total quantity of blood of each type required by the surgeons is equal to the quantity of that type required by the hospital. There may even be third level records listing the blood required for each patient, a set of these records being grouped behind the record for the appropriate surgeon.

A second common type of heirachal file is that where the high order records contain the information always present about one entity, and are followed by variable numbers of lower order records containing variable information. For example, a personnel file may contain a high level record for each employee containing personnel number, date of birth, present position and so on. Each would be followed by a group of lower level records giving information about each position held previously. These might be edited to query
any record for which salary for a previous position exceeds present salary.

A practical problem with this kind of editing that is sometimes very severe is that it is not always possible to retain the high level record while all low level records relating to it are edited without destroying the file structure. If it is important not to write the high level record until all records of the group have been edited special devices such as scratch files may be needed.

Secondly, a record may give information about an identifiable unit, and another record for this unit exists on some other, already edited, file. Once the identifying information has been edited the two files can be matched and edits applied between the records. Common examples of this type occur when a similar file exists for an earlier period of time; here items can be compared for "start and finish" matches, for example stock held at the end of the earlier period should equal stock held at the beginning of the later period.

Thirdly, the file may be homogeneous, but checks can be made between each record and its neighbours. The nature of the information may suggest checks, for example if the file contains records representing the same items over a period of time many checks can be devised to allow only reasonable variations. Even in general cases, statistical checks can be applied to short strings of records, often 3, to report unlikely variations in the items (see Nordbotten (5)).
1.3. AUTOMATIC ADJUSTMENT.

1.3.1. The correction of erroneous items, or the insertion of missing items by computer is a difficult, but worthwhile operation. It is less common than editing by computer, and even in those applications where it is used, it is often done on a trial basis and limited to a few items. However, its use does lead to a very large saving in time and labour, and hence cost, and it may lead to a considerable improvement in accuracy.

The alternative, correction by clerks, is sometimes regarded as exact, but in fact it is often a very unreliable process. Even in cases where it is possible for the adjuster to find the correct value, by reference to source documents or beyond, there can be no guarantee that this has been done, or if it has, that the amendment has been processed without transcription error. Where it is not possible to recreate the correct value, such as in statistical applications where an error exists in the source data, then it is common practice for the adjuster to create some new value by means of rules provided by the systems analyst or by using his experience. The former kind of adjustment will clearly be done better by a computer which will follow faithfully the instructions of the analyst, and which should make possible an elimination of bias. It is the experience of the writer that as many as 10% of corrections made clerically are incorrect after transcription, in the sense that the record when adjusted still fails the editing checks.

There are two severe problems in automatic adjustment by computer, deciding which item is incorrect, and choosing the correct value. These problems are exaggerated by the
fact that the checks and corrections have to be written before editing commences, and so the analysis has to be very thorough indeed to be effective in all cases. Once put into operation it is difficult to change any of the instructions, and if there is an error, in extreme cases, several hundred thousand records may be edited before it can be corrected.

The procedure then is difficult, but the results so worthwhile that automatic adjustments should be included wherever possible. Most of the methods discussed below are particularly applicable to numeric data. Correcting alphabetic data such as names and addresses in much more difficult.

1.3.2. The difficulty of deciding which item is in error occurs only when functional or logical checks between two or more items fail. This kind of check unfortunately is a very common detector of source error. The preferable way of resolving the problem is to apply further checks to each of the items until the erroneous item can be determined, but this is not always possible.

In the case of editing a pay slip, for example, if the relation

\[
\text{net pay} + \text{net tax} + \text{deductions} + \text{superannuation} - \text{gross pay} = 0
\]

is not satisfied, then we could introduce other checks such as "does gross pay = annual salary / 26?" or "does the value for deductions equal the sum of the various deduction items?" If one of these checks fails and if the other value for the item satisfies the original equation then we can be confident not only that we have found the item in error, but that we know its correct value.
In other cases a check as to the reasonableness of each item may be of value. Thus, if in a chemical mixture, the sum of the component weights does not equal the total weight, then a check on each weight might reveal that one was of unlikely magnitude, and the error could be attributed to that item with some degree of confidence.

A method that is very effective when the record is of a suitable type is to evaluate all possible checks before making any adjustment, then making all necessary adjustments from the information then available about every relation of interest. After such an adjustment the record is frequently re-edited and possibly further adjustments are made until either the record is correct, or adjustment is halted after a specified number of iterations.

Frequently however, there is not enough information in the record to allow items to be checked in several ways so that the erroneous one of a set can be determined. Usually the data preparation resources available do not allow sufficient redundant items to be included for this purpose. A common method in such cases is to evaluate a priori the reliability of the various items, and where an adjustment has to be made, to make it to the item considered least reliable. For example it is frequently found that values or prices will be more reliable than quantities, especially in respect of source errors, since it is always in the interests of one party or another to ensure the accuracy of such an item.

A further method sometimes may be justified on grounds of expediency. In this method the item which is least important, or else least significant, will be adjusted. No attempt is now made at absolute accuracy, and this method
is only adopted where absolute accuracy is either impossible or too expensive.

The technique used wherever possible by the author is a combination of the first and third methods above. This is to take a small group of items classed as most reliable and to edit these with all possible rigour. The validity of these items having been established (as it usually is), the scope of the edit is gradually extended by considering new items and editing these individually and against the group of items already edited. If necessary these new items are adjusted to agree with the previously edited items. In this way the whole record can often be edited so that at any failure it is obvious which item is in error, and how it should be corrected. If an error is detected in the starting group, then as these items are generally the most significant in the record, the record is in an unacceptable condition and must be corrected after careful reference to the source documents or beyond. In this case the edit of the rest of the record will be confined to signalling error, and no corrections will be made.

One of the first edits using this technique was tested against clerical correction. A batch of data was edited using the program, and the original form of those records that were automatically adjusted were shown to a clerk, experienced in editing such data. He was asked to correct these records, and when he had done so, his corrections and the computer corrections were compared carefully with the source documents. It was found that the computer adjustments were of a significantly higher quality.
1.3.3 The value to which an item should be adjusted is in some cases obvious. Where some arithmetic equality or near-equality has to be satisfied, or where further relations suggest some value which satisfies the check, then the adjustment can be made in some confidence.

It often happens however than no such method is available. One possible approach is to use a knowledge of the likely forms of error to the item during extraction of transcription to find an acceptable value. For example, if a numeric value read from a punched card is found to contain an alphabetic character, then a knowledge of the punching machine's characteristics could suggest a value such as than numeral whose punching forms part of the code for the digit found. Thus, if the cards were punched on an I.B.M. 026, an A would be adjusted to 1, B to 2, C to 3 and so on.

Similarly, if a set of characters from paper tape is found to be in error, then the characters which have the same punching but with different case shift would be tried. On a less complicated level, if an item is found to be too large, then it might be worthwhile to truncate the leading digit and accept the reduced value.

If none of the above methods is suitable then the method of "safest choice" can be used. In this method, the item is adjusted to a value so that either the maximum possible error of the adjustment is least, for example replacing an alphabetic character in a numeric item by 5, or else it is adjusted so that the effect on the output is least, for example if sex is not stated then set this to male.

Three very interesting refinements of the "safest choice" method are discussed by Nordbotten (5). They are
particularly applicable to statistical data, and the reference includes some discussion of the effect of these methods on the final statistics.

Each method depends upon the construction of a cross-classification on a group of items so that the differences between records belonging to different classes is great, while there is little variation between records in the same class. When an item is found to be in error, or missing, the record is classified by the other items and the value for the bad item is chosen from this classification in one of three ways.

In the "cold deck method", prior to editing a set of values is stored in the computer, at least one for each classification. The bad item is adjusted to the value, or one of the values, stored for this item in the classification which agrees with the record in all the other items of the group. If there are several values, then one is selected by a systematic or random selection process.

The "hot deck method" is similar except that when a record is found to be valid it is used to update the deck by replacing the old values of its classifications by its own values. The correction procedure will therefore adjust an item to the value of that item in a recent valid record of the same classification.

Suppose, for example, that the set of values contains one element only and that the items in the group were sex and hours worked last week, and the preset values for hours worked were 41 for males, 25 for females. If the cold check method is used, whenever the figure for hours worked is unacceptable then it will be set to 41 if the sex is male and 25 if the sex is female. In the hot check method these values are
continually updated. If a record is read and accepted where a female works 37 hours, then the stored value is changed to 37 and if the next record has sex female but no value for hours worked, this item will be set to 37.

It can be seen readily that the hot deck method has less tendency to concentrate records around a few sets of values, and is also better able to follow any trends that may be present in the data.

The third method, the "Monte Carlo method" is more complicated, but except where there are trends in the data which favour the hot deck method, it may be more acceptable statistically. In this method two passes are made over the data. On the first pass, no corrections are made, but the distribution of values in each of the classifications is built up by assuming that the accepted values are true values. Using these sets of values and a theoretical model of their distribution - usually the Normal distribution, the parameters of the distribution are computed. On the second pass, whenever a correction has to be made a random number generator is used to sample from.
1.4 SOME EXTENSIONS AND PROBLEMS.

1.4.1 Most of the preceding ideas are especially relevant to numeric data, where a range of arithmetic checks is available for the edit, or to coded classifications, where sufficient logical relations exist to produce useful checks. The problem of writing a satisfactory edit for large alphabetic items such as names and addresses or descriptive items is much more difficult.

This is because of the very wide range of acceptable names, and the small number of rules in their formation. The rules are full of ambiguities, for example JOHN is acceptable both as a given name and as a surname, and in some cases, they are deliberately broken, such as KOSY KAFE. The author, when studying a computer produced list of names and addresses of companies with statisticians experienced in the field, found that about 5% of all names could not be classed as right or wrong except by reference to source documents. There seem to be no rules available for such names as WESTERN GOLF FIELDS EXPLORATION CO. or THE KWICK CLEANERS PTY. LTD.

However, in view of the large amounts of this sort of data that are processed, and the huge amount of work involved in checking these lists clerically, it seems desirable to edit as far as possible. For example, it is possible to restrict the characters used in names to alphabetic, hyphens and apostrophes, to detect such errors as SM2TH. Other checks can be determined by the nature of the data being processed. Thus, in a motor vehicle registration application, the number of manufacturers is small and any name given as such can be tested against each. A correction process can even be devised where the illegal character is masked and another
search made down the list to find the correction, thus F4RD could be corrected to FORD, but FROD would still give trouble. Again, if names are being held, then it may also be convenient to hold the initials for addressing say, and an edit can be written to check given names against initials.

1.4.2 A similar lack of possible checks exists for a code number, or reference, which distinguishes a record from others on the file. This is a very important item and it may be used frequently for updating and retrieval purposes, but generally it has no logical relation to any other information in the record. Similar considerations apply to items such as account numbers which may not form part of the processing system, but which are vitally important.

To overcome this problem the techniques of check digits have been devised (see Beckley (5)). Here an extra character or characters are added to the number. These characters carry no extra information and do not increase the scope of the number, but they are calculated from a relation between the other digits of the number. The method of derivation is carefully chosen so that the common types of error that can occur while transcribing the number, that is misreading one character or transposing two adjacent characters, will cause the relation to be broken. Thus each time that the code is read by the computer, the calculation is repeated and if the computed check digit agrees with that given then there is a very strong probability (often as high as .999) that the number has been correctly transcribed.

The most commonly used check digits are based on the "modulo 11" technique, where a product is formed of the digits and powers of 2, and the product divided by 11.
The remainder is used as the check digit. Where the remainder is 10 either the digit X may be allocated, or else the code number may not be allocated. Sometimes 11 alphabetic characters are used.

For example, if the number is 1234, we form the sum

\[ 1 \times 2^3 + 2 \times 2^2 + 3 \times 2 + 4 = 26 \]

and dividing by 11 we have remainder 4 so the number with check digit is 12344.

1.4.3 It has been mentioned before that it is often valuable to use the data currently being edited to adjust and improve the edit checks themselves. This technique is frequently applied to the upper and lower limits of a functional check where the function is a ratio.

This adjustment can serve two purposes. Firstly it can be used where there is little or no knowledge on which the limits can be set initially. At the start of editing the limits can be set very wide, and use made of this technique to improve the limits as the editing progresses. Secondly, where the data are not homogeneous, for example temperatures recorded which will vary seasonally, this adjustment can be used to follow the trend of the data and keep the edit effective.

The resetting of the limits is usually carried out at set intervals in the editing, not necessarily after each record. Thus, the limits may be reset after each batch of processing, or during the batch after say, every 1000 records. Normally the amount of data that is processed between recalculation of the limits is too small for statistical
values derived from it to represent the values of the whole of the data. For this reason some historical aggregates are usually used too. That is some totals will be kept of the records to date and these will be merged with the recent values before the calculations and resetting are performed. The weights given to the recent records will be dependent on the nature of the data; if the purpose of the adjustment is to follow the trend of the data then more weight will be given to the recent value than if the data were homogeneous and the adjustment used to improve the limits.

A similar approach, with less statistical justification is to count, during the editing run, the number of records submitted to this check, the number that were too high and the number that were too low. At the end, compute the ratio of those rejected as too high to the number submitted, and if the ratio is higher than a predetermined value, raise the upper limit by a preset amount. If the ratio is lower than a second predetermined value, then lower the upper limit. Treat the lower limit similarly.

1.4.4 It is desirable to produce some statistics on the editing of each batch of data. In its simplest form this can be merely a count of the number of records in error, the number queried and the number accepted. These figures give an instant, but rough, picture of the quality of the data, and may be of use in deciding that a batch of data is so bad that it cannot be accepted. Such a decision may enable the printing of the error listings to be cancelled, as these may be held on magnetic tape and printed only when the statistics as above have been examined. As printing is a relatively expensive operation, this may represent a large saving.
More useful is a count of the number of failures at each editing check. These give a very clear picture of the type of error occurring so that attention can be given to any error occurring frequently. This may be prevented by clearer instructions to those preparing the data, or by modification of the edit program if this is at fault. These figures can be made available very rapidly and give a better picture of trends in the results than the long process of clerical inspection of the complete error listings.

Particularly in those cases where adjustments are made on a statistical, or even arbitrary, basis. it may be useful to extend the statistics given with each run by aggregating the changes made to each item throughout the batch. These provide a control not upon the data quality but on the editing process, and may provide an estimate of any bias that is introduced into the file by these adjustments.

A further output of the editing run may be a list, or a summary of a key item of each record, set out so that the list can be scanned rapidly to detect omissions or duplications in the data. Sometimes both are produced, and the summary used to detect the existence of a discrepancy and the appropriate section of the list used to find the actual errors. Such a listing is often called a "balancing listing".
1.5 THE LOGICAL EDIT.

1.5.1 The editing process, as it is usually programmed, can be divided into four operations, which can be described in general terms as:

(1). The data characters are read from some input device and interpreted according to some predetermined structural rules.

(2). The record, as produced by (1) is subjected to editing checks and adjustments.

(3). Messages are produced to signal any errors, queries or adjustments.

(4). The edited record is stored in some form suitable for later processing.

The file update and correction phase follows a very similar pattern, step (1) now being extended to include the matching of the amendment information to the appropriate information on the file to be updated. The remarks made below will apply in general to both processes.

The purpose of this section is to define a subset of this whole process, called the logical edit, and to justify its consideration as a separate entity.

Operations (1) and (4) do not require any knowledge of the logical content of the record. All that is required is a knowledge of the physical structure of the record, and of the characteristics of the particular input/output device being used. The connection of these sections with the rest of the program is simple, usually being only a block of data representing one logical record. However, this linkage
is complicated if an error is detected in the input operation.

It may happen that the input record is impossible to interpret. This sometimes occurs in a free format medium such as paper tape. Under these circumstances, the record must be excluded from the edit, but some message must be given to enable the record or records to be identified and resubmitted for later processing. This message will usually consist only of the character string and some diagnostic, as not enough information will be available to write the record in accordance with the principles discussed earlier. These messages will thus be quite different from those written by operation (3) and may be regarded as distinct from them. They may either be written on a separate file or else on the same file relying on the sophistication of the software to accept messages from two independent sources for the same file.

On the other hand, the input record may be interpretable, but may contain errors in some items that are detected at the input stage; for example a parity error may be signalled. In such cases, it is often desirable to note this fact during the editing of operation (2), possibly modifying the checks because of it and possibly commenting on the error in the normal listings of operation (3). For these purposes a flag or set of flags may be needed to communicate between (1) and (2).

It does seem, therefore, that the first and last operations can be separated from operation (2). Some communication between them is necessary, but this is not very involved. Further such a separation seems logically justified because, in general terms, operations (1) and (4) can be specified without knowledge of the logical content
of the record but need a knowledge of the particular machine
environment. On the other hand, operation (2) can be
specified without regard to the machine environment, but is
dependent on the logical content of the record. Furthermore,
there is some justification for regarding the problem of
efficient specification and implementation of (1) and (4) as
"solved". Even apart from the input/output facilities of
most high level languages which are usually adequate for this
task unless there is a very large volume of data, most machines
have available an input/output program to deal very rapidly
with logical records in the very manner needed by the opera-
tions. In the few cases, such as paper tape, where the
conventions of each installation differ to such an extent
that little can be done on an inter-installation basis, it is
quite easy to provide a package to read logical records
according to the installation's own standards.

These operations will not be considered further
in this paper.

Operation (3) above is also to some extent machine
dependent, as again an output device is used requiring
programs particular to the machine. However it cannot be
isolated from operation (2) as clearly as can the others.
This is because some knowledge of the logical content of
the record is needed to provide efficient diagnostics. For
example "illegal character" is not as useful as "illegal
character in account number".

A reasonable compromise is to require operation (2)
to set up the diagnostic messages in an explicit form, and
hand these over to a generally coded output routine to write
them in the necessary form. Any editing statistics needed
can be treated in the same way.
1.5.2 In accordance with the above, the author has written a generalised edit routine for the Control Data 3600. This routine must be used in conjunction with a program performing operation (2) for the particular application.

Control cards are read which specify the structure required for the input file, the edited file and error listings. The routine then reads one logical record for the input file and interprets this according to the specified structure. Any errors that are noted at this stage are signalled by a set of flags.

The provided routine is given this record and must edit it, storing the edited record and any diagnostic messages to be written in arrays accessible to the generalised routine. The edited record is then written and the error listings written if necessary. The next input record is then read and so on.

This routine has been used by several people for different applications and the division into provided and generalised routines seems quite satisfactory.

1.5.3 The rest of this paper will therefore consider that subset of the editing process which requires a knowledge of the logical content of the records. A generalised technique will be introduced to enable this section to be written with a minimum of time and effort, yet with the capability to write efficient programs and to simplify debugging as much as possible.

This subset will be referred to as the "logical edit", and is defined as follows:

"a logical edit is a procedure which accepts a record as a list of items, held internally and coded and sorted as
necessary, and evaluates its condition by applying checks on or between the items and external values which may be part of the edit program or may be other records of some kind. The edit may adjust items if appropriate, and add new items, and it will generate the appropriate messages to report on the conditions found."
2. LOGICAL EDIT SPECIFICATION

The objective of this phase will coincide with the analysis of the inputs and outputs and the relationships between them, and will result in a program design that meets the required degree of confidence in its accuracy and completeness and ready for production. This process may involve several people in a typical application, and to ensure coherence in these processes.

(1) The analysis of the received and the possible

2. LOGICAL EDIT SPECIFICATION

The objective of this phase will coincide with the analysis of the inputs and outputs and the relationships between them, and will result in a program design that meets the required degree of confidence in its accuracy and completeness and ready for production. This process may involve several people in a typical application, and to ensure coherence in these processes.

(1) The analysis of the received and the possible

2. LOGICAL EDIT SPECIFICATION

The objective of this phase will coincide with the analysis of the inputs and outputs and the relationships between them, and will result in a program design that meets the required degree of confidence in its accuracy and completeness and ready for production. This process may involve several people in a typical application, and to ensure coherence in these processes.

(1) The analysis of the received and the possible

2. LOGICAL EDIT SPECIFICATION

The objective of this phase will coincide with the analysis of the inputs and outputs and the relationships between them, and will result in a program design that meets the required degree of confidence in its accuracy and completeness and ready for production. This process may involve several people in a typical application, and to ensure coherence in these processes.

(1) The analysis of the received and the possible

2. LOGICAL EDIT SPECIFICATION

The objective of this phase will coincide with the analysis of the inputs and outputs and the relationships between them, and will result in a program design that meets the required degree of confidence in its accuracy and completeness and ready for production. This process may involve several people in a typical application, and to ensure coherence in these processes.

(1) The analysis of the received and the possible

2. LOGICAL EDIT SPECIFICATION

The objective of this phase will coincide with the analysis of the inputs and outputs and the relationships between them, and will result in a program design that meets the required degree of confidence in its accuracy and completeness and ready for production. This process may involve several people in a typical application, and to ensure coherence in these processes.

(1) The analysis of the received and the possible

2. LOGICAL EDIT SPECIFICATION

The objective of this phase will coincide with the analysis of the inputs and outputs and the relationships between them, and will result in a program design that meets the required degree of confidence in its accuracy and completeness and ready for production. This process may involve several people in a typical application, and to ensure coherence in these processes.

(1) The analysis of the received and the possible

2. LOGICAL EDIT SPECIFICATION

The objective of this phase will coincide with the analysis of the inputs and outputs and the relationships between them, and will result in a program design that meets the required degree of confidence in its accuracy and completeness and ready for production. This process may involve several people in a typical application, and to ensure coherence in these processes.
2.1 LOGICAL EDIT DESIGN.

2.1.1 The creation of the logical edit starts with the analysis of the items of information and the relationships between them, and ends with a program tested to the required degree of confidence in its correctness and completeness, and ready for production. This process may involve several people in a typical application, and is commonly separable into three phases.

(1) The analysis of the record and the possible edits. The object of this phase is to devise and specify a set of edit checks which will detect at least a predetermined percentage of all errors. There will often be a class of errors which must be detected with 100% certainty. Any adjustments or imputation of values must also be specified.

It is uncommon to consider programming problems in this phase, indeed; it is often performed by someone who has little or no knowledge of programming. The usual outcome is a list of checks and adjustments written in some manner such as flow charts or conditional English sentences. This list may or may not give an indication of the sequence in which the checks are to be applied.

(2) The programming. The object of this phase is to convert the specifications from phase (1) into a computer program in a manner which is efficient in some sense. The usual measures of efficiency are minimum execution time or minimum core storage requirements, but
particular projects may require other properties. Associated problems such as file design and run structure may also be relevant to this phase.

During the programming omissions or contradictions may be noticed in the specifications and will have to be resolved. Conversely however, the interpretation of the specifications may not be the intended one, and the program may not represent the desired set of checks.

The programme will normally be coded in some high level language such as COBOL, FORTRAN or PL/I, with consequent saving of labour, and improvement in clarity. The conversion of this code by the compiler into equivalent machine code can normally be regarded as error free.

(3) Testing. The program must now be tested until it can be asserted that the edit meets the required standard of error detection. This involves the testing of both phases that have gone before, the design and the programming.

Testing of the programming is assisted by the diagnostics that form a part of most high-level languages, and by special debugging aids such as snap dumps and traces. On the other hand, there is no analogue of these devices for the analysis. The testing of this is often complicated by the fact that the analyst cannot be sure that his specifications have been implemented correctly, and he will sometimes
be unfamiliar with, and confused by, the nature of the output that he gets from the computer runs. The testing of the analysis can therefore be a tedious process and it is often neglected for this reason, sometimes with unfortunate results.

2.1.2. The preceding section suggests that it is both inefficient and troublesome to keep the analysis and programming phases separate. A technique that allows the analyst to communicate directly with the computer should lead to considerable improvement in the efficiency of the logical edit creation. Even for those applications that are too large, or otherwise unsuitable for such direct communication, a better job can be done if the analyst can fully understand the input to and output from the computer.

It seems unreasonable that such a technique can be developed by extending the analysis only. That is, by developing an analytic technique capable of considering the most complex record structures which produces specifications in some existing programming language. A more feasible approach is to extend the role of the computer also, by creating a new language this is sufficiently powerful to be used as an analytic tool and that can also be used as input to a specially written compiler.

Specifications written in such a language would need to be more precise than many now used to describe the results of an analysis. This need not be too restrictive however if the language is carefully designed to be suitable for its main purpose - that of describing logical edits. The disadvantages of this rigour will in any case be offset if the use of the language allows the production of
diagnostics and debugging aids relevant to the analysis as well as to the coding.

The technique of using decision tables for the analysis goes some way towards solving this problem. However, editing is only one of the uses of decision tables, and there are many difficulties that could be removed in a more specialised technique.
2.2 **THE EXTENT OF THE LANGUAGE.**

2.2.1 As a preliminary to the design of a language and a compiler in accordance with the preceding ideas, the scope of the compiler must be determined. That is the process of designing an edit must be divided into two phases, the first to be carried out manually and the second automatically.

It seems possible that a compiler could be written to perform the whole of the logical edit design, leaving only the problem definition to be done manually. In such a scheme the compiler would be given a list of all items. The nature of each item and all permissible values would be specified together with a list of all relations between the items. In addition the probability of an error in each item should be specified. From these lists, the compiler would produce an edit which checked every relation specified, and when some relations were unsatisfactory would use the supplied probabilities and any available cross-checks to detect, and if possible correct, the incorrect item.

Some extensions to this basic scheme seem desirable. The compiler could evaluate the completeness of the information that it has been given and comment on any inconsistencies or ambiguities that it found, it could also indicate any items of the record for which no useful edits were given. The compiler could examine the efficiency of its object code and, by reordering the checks for example, could improve or even optimise the efficiency of the edit by some pre-determined structure such as least execution time (see, for example Reinwald and Solange (9)). The function of the compiler could be extended to include the design of input and output files, or even to control the whole system of information processing, given knowledge of the aims of this system.
Research on two different levels into compilers of this nature is known to the author.

The first level is research into Abstract Information Systems, (see Pridmore (11)). This research is concerned with the more general ideas outlined above, and the logical edit is included as one step in the general processes considered. The aim of this research is to develop a computer program to design all phases of an information system, given knowledge of the source of information and of the requirements of the system. The program will design the input files, the editing program, the error listings, the amendments program, the processing of the edited data, and the output files. The associated language must be capable of describing all the sources of information, including the timing at these sources, the sinks of information, that is the output required, and the resources available including file types and capacities and the properties and availability of the computer itself.

Such a program, when written, will be of considerable value. However, the research is not very far advanced, and it does not seem likely that any useful results will emerge in the near future. Some languages have been proposed and a few simple examples implemented, but much more work is needed before a useful system emerges. The work on optimisation of such designs is even less advanced.

The second level is a much more practical one. Several systems are known to the author which will edit any record within a fairly restricted class. Typically, such a system is sufficiently general to deal with all foreseeable variations in one particular application, and thus the need for reprogramming with each change of data record is eliminated. Each system includes a language to describe the record, the items and the relations between the items, and
a master program to read this language and produce an edit program for the record as described.

Some of these systems have been working successfully for some time, during which major revisions of the record have been made without further programming being needed. However, none of the systems seems capable of extension to deal with the general editing problem, and the languages are also specialised to the particular application, and usually rather cryptic.

2.2.2 The work discussed above, particularly that in Abstract Information Systems, suggests that a general program to write logical edits given the properties of the items to be edited is feasible. However, two severe problems need solution, and it is the opinion of the author that the complexity of these problems will prevent a useful solution in the near future.

The first problem is that of minimising the execution time of the edit. A most important factor is the relative frequency of the possible kinds of record. The edit should be written so as to deal with the more common cases as rapidly as possible, even if this may be at the expense of some loss of time in less common records. For example, a program editing import warrants may have to check that the country of origin code is valid. Suppose the code is compared with a list of acceptable codes until a match is made. If the code list is ordered so that the codes for the large exporters, say the United Kingdom, the United States, and Japan come first, then execution time will be less for an average batch of data, than if the list were ordered say alphabetically, so that Aden and Afghanistan were recognised first. Execution time
might be improved even further if a special fast test was made for the principal exporters before entering the slower large look-up process.

Devices such as this are readily invented when writing a specific logical edit, and their implementation will lead to useful savings in execution time at little expense in extra labour. The time of execution will not be optimal, this being an intuitive rather than an algorithmic approach, but it will be sufficiently close to it for the vast majority of cases.

This saving is so valuable that no scheme for producing a logical edit can be considered practical unless some optimisation along these lines is possible. There are two ways in which this could be implemented but both seem very difficult.

The dynamic approach would produce initially an edit program with arbitrary ordering. As editing progresses, the program produces tables of the frequency of values of specific items and orders itself to take advantage of these results. Such a program, in the general case, would be very complex, and in any case the extra time spent in updating the tables and revising the program may offset all but the most obvious changes in ordering.

The static approach would require frequency distributions and probably joint frequency distributions for particularly significant sets of items to be included in the specifications. The works of Reinwald and Soland (9), who produce optimal programs from limited entry decision tables using this technique, shows that this technique makes considerable enlargements to the amount of specifications required, and also increases compilation time. These features,
particularly, the former, would decrease the chances of a ready acceptance of such a language.

The second problem is that of processing automatic adjustments. The difficulty is in selecting the item to adjust and the new value for this item.

It is possible to select the item for adjustment on a probability basis, either by specifying an ordered list of items such that each item is considered to have less chance of error than any following item, or by assigning a probability to the chance of error in each item. However, as explained before, this is the least satisfactory of the methods, and an error in the item designated to be the most reliable can cause a long chain of spurious adjustments.

Similarly the new value could be chosen by one of the statistical methods, "hot-deck", "cold-deck" or "Monte-Carlo", but again these are the least satisfactory.

The more accurate solutions to these problems are dependent on the analyst's skill and experience in determining the error in each set of circumstances. There are, at the moment, no rules that can be applied to formulate these to such an extent that the program-produced adjustments would be more effective than the manual set.

2.2.3 Because of these difficulties it seems likely that manual analysis of the editing will remain the most effective technique for some time. Thus, a compiler to read directly the results of an analysis and produce an edit program from these seems to be the most useful approach for the present. It would be assumed by this compiler that the path of the edit has been specified and useful optimisations already made. The methods of automatic adjustment will also have been determined before calling the compiler.
This will be an improvement on existing techniques because it will reduce the time involved in programming; it will remove a possible source of error; and it will make testing much easier.
2.3 **EXISTING TECHNIQUES.**

Three methods of specifying the results of a logical edit analysis are in widespread use. In each case, their use for logical edits is only one application of a general technique.

In this section each method is outlined, and an example is given. The example is the editing of a pay slip. The items to be edited are annual salary, gross pay (monthly), net pay, tax, total other deductions, deduction 1, deduction 2, deduction 3. It is assumed that all values are numeric and positive.

2.3.1 The traditional method of specifying an edit is by the use of conditional sentences, for example:–

"If temperature at start exceeds temperature at end then accept record, otherwise there is an error in temperature".

"Code number must be "AB", "CD" or "EF" followed by three numeric digits. If it is any other value, then set it to "XY999" and continue".

There are no generally accepted conventions in this method, ambiguities are hard to detect, and the clarity of the result is dependant upon the ability of the writer. The method is not satisfactory except in the simplest cases, and becomes quite unmanageable in complex cases. Despite this it is in widespread use, probably because no rules have to be learnt before the method is used.

An important problem is that in most edits the checks are not independent and the editing for one record will take one of many paths through the checks. It is common, for example, to submit a record which gives a particular result
at one check to supplementary checks which are not applied otherwise. In more complex edits, there may be several levels of checks, and the number of paths will be very large. In such circumstances it is difficult to specify all the paths required without ambiguities or contradictions. Methods that give more assistance with this difficulty are much to be preferred.

This method also tends to be cumbersome. Precision can often only be obtained by using a high degree of repetition.

Finally, the edit as specified in this manner is rarely in a suitable form for programming, and much intermediate work is often necessary.

The example illustrates some of these points.

(1) If \((\text{net pay} + \text{tax} + \text{total other deductions}) = \text{gross pay}\), and if \((\text{gross pay} \times 12) = \text{annual salary}\), and if 
\((\text{deduction 1} + \text{deduction 2} + \text{deduction 3}) = \text{total other deductions}\) then accept the record.

(2) If \((\text{net pay} + \text{tax} + \text{total other deductions}) = \text{gross pay}\), and if \((\text{gross pay} \times 12) \neq \text{annual salary}\) then adjust annual salary to \((\text{gross pay} \times 12)\).

(3) If \((\text{net pay} + \text{tax} + \text{total other deductions}) = \text{gross pay}\), and if \((\text{deduction 1} + \text{deduction 2} + \text{deduction 3}) \neq \text{total other deductions}\) then error in deductions.

(4) If \((\text{net pay} + \text{tax} + \text{total other deductions}) \neq \text{gross pay}\), but if \((\text{net pay} + \text{tax} + \text{total other deductions}) = (\text{annual salary}/12)\) then adjust gross pay to \((\text{annual salary}/12)\).

(5) If \((\text{net pay} + \text{tax} + \text{total other deductions}) \neq \text{gross pay}\), but if \((\text{net pay} + \text{tax} + \text{deduction 1} + \text{deduction 2} + \text{deduction 3}) = \text{gross pay}\) then adjust total other deduc-
tions to (deduction 1 + deduction 2 + deduction 3).

(6) If any other, then error in record.

This example is meant to be typical rather than optimal. It uses a personal mixture of mathematical abbreviations. It is also highly repetitious, though this could have been reduced by writing, for example:

"(1) If (net pay + tax + total other deductions) = gross pay then go to check (2), else go to check (5)"

and making the obvious changes to the other checks. This is however a dangerous practice in general as there is no way of telling from say, the revised check (6) alone that this check is performed only if the equality of (1) does not hold. This problem can cause errors both in the design and the programming.

It is also unclear as to whether check (3) should be performed after check (2) if the latter has made an adjustment. Different programmers might put different interpretations on this point.

Finally, the rather vague (6) "If any other..." is so easy to write. It would be preferable to do a thorough analysis of the possible happenings here, but there is nothing in this method to encourage it.

2.3.2 An improvement on the conditional sentence method is the use of flow charts to specify an edit. A structure is imposed on the edit by splitting the conditional sentences into phrases, and displaying these and the connections between them pictorially. This method is much more precise, and less tedious to write. It has been used successfully on several occasions by the author, and though it has
several rules and conventions to be learnt, it has been used with little difficulty by statisticians who had never previously seen a flow chart.

(i)

(ii)

(iii)
Note that at (1) it has been decided to perform the check on deductions not only on those records with adjusted annual salary, but also on those with adjusted gross pay. The omission of this link was not at all obvious in the earlier method. Also at (2) a decision must be made as to whether to check annual salary with gross pay here, again the earlier method hid the need for a decision.

The flow chart also gives a much clearer picture of what cases constitute "Error in record" than was given by statement (6) above.

The flow chart method can cope adequately with the most complex edits, and the relations between the checks are clearly brought out. Most valuable is the fact that the number of lines leaving each box can be checked rapidly, and a line which peters out is very distinctive. This method does force a higher degree of consideration of the path of the edit than the previous method, although contradictions are not always obvious. The flow chart is also quite close to the program, and fewer errors will be made in the programming.

It is difficult, however, to write multiple decisions neatly. For example:

"If code is AB set flag to 1, if code is CD set flag to 2, if it is EF set flag to 3, otherwise set flag to 9".

needs the tedious set of boxes given in illustration (ii) of the previous page. This disadvantage can be circumvented at some increase in the risk of errors by loosely using one large box, as illustration (iii), but this is not so for multiple branch statements for example.
2.3.3 The third method is the use of decision tables. This method is of comparatively recent origin, and has many advantages claimed for it. These will be discussed in the next section. The most real advantage is that no further programming is required, as several languages exist which include the means of reading a decision table. Decision tables are discussed in general terms by Dixon (7), and Smith (10), and their implementation as a programming language is discussed by the preceding references and by Press (8), Reimwald and Soland (9), and King (12).

A decision table is an array in which the rows represent edit checks or actions (analogous to the boxes of a flow chart), and the columns represent the paths. All checks specified in one table are applied in one step, instead of a sequence of steps. That is, the answers to all questions are evaluated and then the first column with similar answers marked along it is the path of the edit. The actions specified on that column are performed.

It is usual to consider the decision table as being in four parts, as in the diagram,

<table>
<thead>
<tr>
<th>CONDITION STATEMENT</th>
<th>CONDITION ENTRY</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACTION STATEMENT</td>
<td>ACTION ENTRY</td>
</tr>
</tbody>
</table>

The edit checks are listed, one to a row, in the condition statement, and all required actions are listed in the action statement. These parts, and the others are conventionally separated by double lines.

Each column is called a rule, and may be numbered. It consists of entries in both the condition entry and the action entry. In the most common type of decision table,
called a limited entry decision table, the possible entries in the condition entry are Y, N or blank. The Y indicates that the condition on that row of the condition statement is true, the N that it is false, and the blank that it is irrelevant. The possible entries in the action entry are X or blank. The X indicates that the action on that row of the action statement is to be performed, the blank that it is to be omitted.

It is customary to have as the last column an "else rule", for which all entries in the condition entry are blank. This rule is applied if none of the previous rules are applicable.

The sample edit is now:

| Net pay + tax + total other deductions = gross pay? | Y | Y | Y | Y | N | N | N |
| Gross pay × 12 = annual salary? | Y | N | Y | N | | | |
| Deductions + deduction 2 + deduction 3 - total other deductions? | Y | Y | N | N | Y | N | N |
| Net pay + tax + total other deductions - annual salary / 12? | | | | | | | |
| Tax pay + tax + total other deductions - annual salary / 12 = gross pay? | | | | | | Y |
| Record grand | X | | | | | |
| Set annual salary = gross pay × 12 | X | X | | | | | |
| Error in deductions | X | X | X | | | |
| Set gross pay = annual salary / 12 | | | | X | X | |
| Set total other deductions + deduction 2 + deduction 3 + deductions | | | | | X | |
| Error in record | | | | | | X |
| Continue editing | X | X | X | X | X | X | X |
An extension of the limited entry decision table is the extended entry decision table, where the entries of the condition entry and action entry may be more general. For example, a row could be:

\[
\begin{array}{c|c|c}
\text{Net pay + tax + total other deductions} & \text{\textbf{= gross pay}} & \text{\textbf{= annual salary/12}} \\
\text{or} & \text{\textbf{\neq gross pay}} \\
\text{Set annual salary} & \text{\textbf{\neq}} & \text{\textbf{= gross pay x 12}}
\end{array}
\]

Every extended entry decision table can be converted to limited entry form.
2.4 EVALUATION OF EXISTING METHODS.

2.4.1 Of the existing methods of specifying a logical edit, only decision tables are capable of being used both as an analytic tool and as a programming language over a wide range of editing application.

The method of writing an edit as a string of conditional sentences would need considerable formalisation before it could be used as a programming language. Furthermore, it is doubtful whether this would be a useful language for, however it was finalised, it would be difficult to show the path of the edit in a clear manner.

The flow charts used to define an edit could also be formalised to form a programming language, and in such a scheme, the path of the edit would be obvious. However, the problem of transcribing an edit written in this way onto some computer input medium is very severe. Unless this can be done by unskilled staff with a small chance of error, many of the advantages of the flow chart as a programming language will be lost. Conversion seems to depend upon some device such as labelling every connecting line on the flow chart, and then transcribing each box as one unit, preceded by the input connector label and followed by the one or two output connector labels. This approach leads to very cumbersome results, and a large amount of work must be done to prepare the edit for computer input as distinct from the work done in designing the edit. For this reason it does not seem worth working on this approach while alternative methods exist.

More direct input of a flow chart by means of graphical input such as a cathode ray tube and lightpen may be feasible in the future. Even for this case, however, a large amount of formalisation of the syntax of flow charts
will be needed, together with some complicated software to transform the structure to a programming language.

There are several implementations of decision tables as a programming language. These have all been designed with general applications in mind, but are all suitable for editing. They generally use as condition and action entries a subset of statements from an existing general purpose language. For example, the Rand Corporation's FORTAB uses FORTRAN, DETAB-65 uses COBOL, and a recent implementation at the A.N.U. (Smith (10)) uses PL/I.

When viewed from the specialised context of edit writing, the limitations of these languages produce limitations of the decision table technique. Since, if it were shown to be worthwhile, a decision table language with a more suitable set of statements could be easily devised, the comments of this section will be directed at such a hypothetical decision table language, and not at any existing implementation.

2.4.2 Dixon, (7), summarises the advantages of decision tables as a programming language of general application in the following eleven points.

(i) (the decision table technique) forces a clear problem statement and shows where information is missing.

(ii) It forces a complete logical description of the problem.

(iii) It completely defines, at system level, decisions to be implemented.

(iv) It leads to low cost translation of a defined system into a working computer program.

(v) It permits development and orderly presentation of systems too complex for effective flowcharting.

(vi) It allows extensive use of subroutines through the segmentation of the over all system into logically manageable tables.
(vii) It is a superior form of documentation for communication among system analysts, programmers and management.

(viii) It is easy to update and revise and shows more clearly than flowcharts the effects system changes will have upon the decision logic.

(ix) It permits system definition and description without imposing a premature sequence of problem solving operations.

(x) It is a technique that is easily learned.

(xi) Decision table language is suitable for direct translation into machine language; i.e. it lends itself to direct compiling.

When evaluated as a language to solve the problem of section 2.1, points (iv), (x) and (xi) become essential conditions rather than advantages, and will not be discussed further.

Point (v) is contentious, and some people assert that it is in fact the converse that it true. It is their claim that for very large systems either the individual tables become too large to handle, or else the segmentation is so complicated that the overall picture of the system is lost. The author has attempted to define some large scale edits using decision tables and found that the amount of work involved in the analysis of a large edit is very large indeed, and it was necessary to return the flow chart technique to complete the projects in the allowed time.

The most important points, as far as edit writing is concerned are (i), (ii) and (iii). Here, Dixon shows that decision tables are a useful tool for editing, as they assist the analyst to complete the edit. However, the present author feels that these points are overstated, and though decision tables are good in this context, they
are by no means perfect. As King (12) points out, a complete logical description is not always desirable. Because of this, completed decision tables sometimes contain apparent ambiguities, and more commonly an "ELSE-column" which is defined to be all cases not included in a previous column. For example, if one condition statement is \( \text{AGE} < 18 \), and another is \( \text{AGE} > 65 \) then an apparent ambiguity would arise between any two rules which were similar for all other conditions and for which the first contained a \( Y \) for \( \text{AGE} < 18 \), and a blank for \( \text{AGE} > 65 \) and the second contained a blank and a \( Y \) respectively. It would appear that, such a table would be ambiguous for a record with \( Y \) to both lines.

Decision tables in practice therefore do no more than assist a complete logical description, and this should be possible by other methods also.

Point (viii) is also open to question. Minor changes to the logic are easy to make with decision tables, but larger changes involving new items, or new checks seem more difficult. This is because additional rows may cause extensive and time-consuming redrafting of one table, and because the highly segmented approach may make it difficult to ensure that the modification applies to the necessary and sufficient set of records. That is, it may be difficult in a complex system to determine the conditions under which a particular table is entered.

Provided that excessive use is not made of the feature, the ease of segmentation (point (vi)) is a useful facility of decision tables. Similarly the ability to describe a system without imposing a premature sequence of operations (point (ix)) is useful. The best feature of decision tables is however point (vii). They are a very clear form of documentation indeed, and this is probably
the reason for the widespread interest in this method. However, when used as a programming language, the need to confine the statements to conform with the chosen syntax is a slight handicap to the simultaneous use of decision tables for documentation.

A further advantage of decision tables that has been implemented since the paper by Dixon was written, is that the table can be converted into an optimal computer program, where optimal means that program that minimises a nondecreasing function of expected processing time and total storage requirement (Reinwald and Soland (9)). This is obviously an advantage for jobs with a large expected processing time, so that the extra analysis and compilation costs can be offset by reduced execution costs.

2.4.3 The advantages outlined in the preceding section do not seem to be unique properties of decision tables. Other methods can be devised which are also capable of use in edit analysis and as a programming language which would have these features and others besides. Such a language would not be an improvement on decision tables in every respect. Indeed the ability of decision tables to provide clear documentation of the edit would be hard to improve. The principal improvement should be in the time that it takes to design an edit. This time is very high when using the decision table technique for two reasons. Firstly, the technique needs a very complete analysis of the whole problem, which must take a lot of time and produce much redundant information. Though a complete analysis is sometimes desirable, it is usually unnecessary and often impossible to complete in the time available. In particular, a well designed editing language will suggest to the analyst
any considerations that have not been included, will facilitate testing, and will make a critical review of the edit as written very easy. Given such a language and some experience and ability on the part of the analyst there seems to be no reason why the edits written should not as good as those following a complete decision table analysis.

Secondly, an amount of time must be spent on the drafting of decision tables, particularly as this often involves writing a large number of redundant columns which are afterwards removed. This time, though not large when compared with flow charts, contribute little to the analysis, and could be reduced in another method.

Although decision tables express complex relationships between several items very clearly and compactly, they are not convenient for simple checks and table lookups. These simple checks form a large proportion of the typical edit program. Techniques exist for getting around this problem, such as Smith (10), where the bulk of the edit is written in PL/I, and decision tables are embedded in the edit to deal with the complicated checks. However, it would be preferable to have just one language capable of expressing all kinds of checks.
2.5 **THE SUGGESTED APPROACH.**

A language suitable for describing a logical edit must be able to describe the statements and the structure of the edit. The statements define each edit test, edit action or supplementary action (such as imputing a value for an item not present in the source data). The structure defines the order in which the statements are executed so that it is possible to determine unambiguously the sequence of tests and actions for each record.

Thus, in the case of a flow chart representing a logical edit, the words written in each box represent the statements, and the pattern of boxes and connections represents the structure. In a decision table the condition statement and the action statement represent the statements and the condition entry and the action entry represent the structure, since the condition entry defines a unique column for a record and the action entry for this column defines the series of actions to be performed.

This section discusses some of the features that the statements and structure of an editing language should have.

2.5.1 Little consideration has been given in the past to the design of a set of statements suitable for a logical edit and no published work on this subject is known to the author. A general purpose, or a business oriented language must, of course be capable of including a logical edit, and languages such as COBOL, PL/I and FACT do contain statements sufficient for this purpose. However, the design of a language to fulfil such a variety of purposes must inevitably depend on a high degree of compromise. Because of this these languages, though they contain the means to write a logical edit, are by no means ideal for this purpose.
This compromise is most obvious in the complicated subroutines that are often necessary to control the error listings, and to set the record condition flag. It is also noticeable in the number of times that one edit action has to be expressed by a group of statements of the language, which is sometimes so involved as to make the connections with the specifications of the edit not at all obvious.

For example, the imputations of a code that may be defined:

"Code 'VALUE into classes less than 100, less than 500, ..., less than 5000,5000 or over"

would require a set of PL/I statements such as:

```pli
DECLARE LOOKUP (10) INITIAL (100,500,...,5000);

DOLOOP: DO I = 1 TO 10;
  IF VALUE LOOKUP (I) THEN DO;
  VCODE = I; GO TO NEXT; END;
END DOLOOP;
VCODE = 11;
NEXT: . . .
```

and this code is cumbersome and has led to a considerable loss in clarity, particularly if the values of INITIAL are declared some distance away from DOLOOP.

There are three important considerations in the design of a set of edit statements. These are not independent, nor even compatible, so that even for a specially designed edit language, some degree of compromise is necessary.

The first consideration is that the set of statements must be complete. That is, it must be capable of expressing a sufficiently large set of edit tests and activities. It is desirable that these tests and actions may be written simply
and naturally, for example look up tables should be written in the required place and not have to depend on declarative statements written some distance away as in the example given earlier.

Although it will not be possible to accept all conceivable tests and actions in such a natural way, it is possible to devise a language which will accept all the common types of edit statements in this way. The most necessary extensions to existing languages are statements to handle error messages, changing the record condition flag, tests and actions involving look up tables as above, more powerful if-statements, and trial adjustments, that is adjustments which are to be made only if this one adjustment is sufficient to correct the record.

To cope with more general demands, the edit language must contain sufficient general statements, such as replacement, logical and control statements, to enable the more involved tests and actions to be established in several steps, involving subroutines if necessary. A good way to do this seems to be to allow sections of code of a general purpose language, such as those mentioned above, to be embedded in an edit.

The second consideration is to keep the syntactic rules to a minimum. This should reduce the time needed to write an edit, but more importantly it should reduce the time needed to learn the language. This should encourage more people to use the language and should increase its appeal especially to non-programmers.

The clarity of the edit should also be improved by this, making it easier for non-specialists to read and understand the edit, and making the debugging phase easier for the programmer.
The third consideration is that the statements should be unambiguous, and acceptable as input to a compiler. This point is of especial importance in view of the two considerations above, as it represents a real limitation on the extent to which they can be implemented.

It may however be of much value during the analysis, by assisting the statement of the edit to be complete and unambiguous at this stage. For example, it is not uncommon for an analysis to include such vague statements as "if tax is too high", without giving any criterion for this decision. By making such statements impossible to write, the language may force an improvement to the analysis.

Another common error is to ignore one or more alternatives of a multiple choice. For example, if the record included marital status and checks this for feasibility against age, the first version of the edit may consider "never married", "married" and "widowed", but omit "divorced" with results that may not be noticed during testing. It would be advantageous if the statements of the language were such as to make omissions like this as noticeable as possible. (However, as mentioned earlier, these omissions may be logically correct so that the language cannot insist on completeness.)

A language of this nature should also prevent ambiguities created by its own syntax. A well known case of this type is the "dangling-else" problem of ALGOL, where IF...THEN clauses may or may not have a subordinate ELSE clause. If a number of IF...THEN clauses is grouped with a lesser number of ELSE clauses, then the problem is to attach the ELSE clauses to the proper IF...THEN clauses. Although rules exist (see Abrahams (13)) whereby this can be resolved by the compiler, some confusion must often
exist in the mind of the programmer, and in the present context such a situation is best avoided.

Finally, the number of statement types must be sufficiently small for the compiler to be of a reasonable size and work with reasonable efficiency. This may be a very real limitation on the number of actions which are expressed by a special statement.

2.5.2 More attention has been given to the structure of an editing language. Both flow charts and decision tables are usually treated as a structure added to some existing language. Thus, in a typical flow chart the entries in the boxes are English phrases, and in a typical decision table the statements are in COBOL. However, none of the structures used previously seem ideal, and there is ample justification for discarding them and starting afresh.

The points to be considered are generally those of the preceding section, but the major object is to devise a scheme with as little unnecessary complication as possible. This is important both to reduce the amount of time wasted on writing down the edit (as opposed to devising it and checking it) and also in keeping it clear. Flow charts in particular have many disadvantages in this area, since they require some time to draw, and either several attempts, or a knack, to make complicated cases clear. Decision tables have the same failing to a lesser extent, as their peculiar form means that a whole set of checks and actions has to be considered and determined before the table can be set out. Even when this stage has been reached, a transposition of the columns can lead to a much clearer picture.

However, both these methods have won wide acceptance because of the structure, while cumbersome, does determine
the edit path in a clear, explicit manner. Also the structure assists the analysis by making omissions and contradictions quite clear, thus satisfying one of the most important of the above considerations. Any attempt to devise a simpler structure will have to guard against any loss of these advantages.

As with statements, it is unlikely that any structure can cover every desired case and so provision will have to be made for breaking down sections into a series of subsections, and the calling of subroutines and so on. The most common cause of this will be when so many statements are involved that all available space is used, and some analogue of the cross-page connectors of flow charts and table numbers of decision tables will be necessary.

The structure should be such that the edit can easily be changed. This consideration also points to keeping the structure as simple as possible.

To make the structure recognisable to a compiler is another difficult problem, and one which has been avoided in the method of flow charting. Since lines and arrows are very difficult to input to a computer it seems that forms close to flow charts will not prove feasible. Other devices such as level number or indentations could be used to display the position of statements, and of the two indentations could be used to display the position of statements, and of the two indentation seems the clearest.

2.5.3 The author thinks that a language should be written specifically for logical editing. It would be useful for analysis and also acceptable as input to a compiler, thereby resolving the problem of section 2.1. The statements would be designed only for editing, and should be understandable
to an analyst with no programming experience. On the other hand they should give the experienced programmer some idea of the way in which they will be compiled so that he can select the most efficient options for his purpose.

Typical of the extensions over a general purpose language would be more powerful "IF" branching statements, simple table lookup handling and trial adjustments which would only be executed if the result satisfied some condition. Statements would also be included to control the flag reporting on the overall condition of the record, and write the error listings. These statements would be as clear and concise as possible and no elaborate subroutines should be needed for these purposes.

The language should feature a simple structure designed for the purpose. It should not be similar to decision tables as these seem to be inadequate for the whole editing process, and to involve too much wasted time in writing and compiling.

The implementation would also feature debugging aids at the logical level such as cross-reference tables, compile time logical diagnostics and so on.

It is hoped that such an implementation would throw some light onto the question as to how necessary the complete logical evaluation required by decision tables is. At present, no practical testing can be given to this question as no language other than those based on decision tables is capable of direct computer input.

Whatever the answer to this question, the author thinks that such a language would be of definite value in three cases:

(i) for simple edits where the logical structure is not complicated enough to warrant decision tables;
(ii) as an easy to learn approach for a part-time programmer such as a scientist, who wishes to write an edit, and who should find the simple structure and few extra statements easier to learn than the decision table technique;

(iii) for the experienced programmer, or other person, who has to write and test an edit in a hurry.

In addition, the author hopes that such a language will be of general use, and be a rival to the decision table technique over most of the classes of edits that have to be written.
3. THE LANGUAGE.

This chapter describes a high-level editing language written to incorporate the ideas of Chapter 1. A pre-processor has been written in ANSI Fortran for the IBM 7040 computer at the A.E.R.E. Harwell Centre, to translate an edit written in this editing language into an optimised

W71 procedure. This pre-processor is described in Section

1. Since it is better document and demonstrates the simpler

purposes, to identify the language by an appendix, it will be

referred to as P500 Edit.

In accordance with the ideas of Section 1, the

statements of P500 have evolved from the start to modify

the need of a language to be as simple as possible in order to
be welcomed by the programmer as a tool for use without

frequent use. In particular, it may not be necessary to learn

and remember all the keywords of the language to use it

effectively. It is possible to have a complete set of

keywords from the other language.

Similarly, it is not necessarily true that it is

not possible to use an edit language to contain all possible

editing checks in addition to direct statements. There will

always be new unusual and complicated choices that have to

be broken or inter-cancelled above. These above will

periodically involve each another in solenoid and so on.
3.1 INTRODUCTION

3.1.1 This chapter describes a logical editing language written to implement the ideas of Chapter 2. A pre-processor has been written in PL/I and tested on the I.B.M. 360 model 50 computer at the A.N.U. Computer Centre, to translate an edit written in this editing language into an equivalent PL/I procedure. This pre-processor is described in Chapter 4. Since it is both fashionable, and convenient for reference purposes, to identify the language by an acronym, it will be referred to as FRED (For EDiting).

In accordance with the ideas of Chapter 2, the statements of FRED were devised from the start to satisfy the needs of a logical edit, and are not merely an adaptation or an extension to an existing language. However, it did not seem wise to consider FRED in complete isolation from existing languages for the following reasons.

The logical edit is usually only one part of a job, and the rest of the program must be written in some other language. Because of this, it is advisable to ensure that the principal conventions of this other language are not contradicted by FRED. In particular it must be possible to load and execute jobs containing both languages and it should be as easy as possible for a programmer to change from the other language to FRED and vice versa.

Secondly, as has been pointed out before, it is not possible for an edit language to contain all possible editing checks and actions in direct statements. There will always be some unusual and complicated checks that have to be broken up into several steps. These steps will frequently involve such actions as computation or data conversion,
and may include actions such as input/output or sorting which require a complicated compiler. Facilities for these actions are already present in all general purpose languages, and to add these actions to FRED would involve much duplicated effort, and would complicate the language with rarely used facilities. A much neater alternative is to allow statements from a general purpose language to be embedded in the exit in some simple way.

For these reasons, the detailed design stages of FRED were carried out so that FRED can be contained in, and contain, blocks of PL/I (I.B.M. (14)). In particular, the data definition conventions of PL/I are copied into FRED as far as possible.

PL/I was chosen as it is the most suitable for editing purposes of the languages in common use at the A.N.U. Computer Centre, and because it seems likely to become widely used, machine independent language in the near future. One of its principal advantages is that it is a very clear language to read and write, its syntax being nicely balanced between the cryptic and the verbose. It has useful, device independent input/output instructions, and its facilities for handling data-structures, characters and strings are especially valuable for editing and allied data processing tasks. Although, at the time of writing, its use is restricted to the I.B.M. 360 series and the computers of some of the smaller manufacturers, it seems likely that the language will spread, in a standard form, to other computers soon, becoming widely accepted.

3.1.2 The primary purpose of the edit is to determine the "condition" of the record, that is to determine which of the conditions given below applied to the current values of the record being edited.
good: all edit tests have been passed.
adjusted: one or more items have been adjusted during the edit, but the final values are such that all edit tests have been passed.
queried: one or more items have been queried (i.e. have possible errors and should be checked by some means not available to this edit, such as reference to source documents). Also, one or more items may have been adjusted, i.e., this condition includes the previous one.
in error: one or more edit tests have failed. There may also be items that have been queried and adjustments may have been attempted (but see below).

This edit must include the definition of a "condition flag", which is set during the editing of each record to one of four values, depending upon the condition of the record.

0 good.
1 adjusted.
2 queried.
3 in error.

The purpose of this flag is to inform the containing PL/I procedure of the condition of the record as simply as possible. Within the FRED edit there is no need to set the flag by explicit statements since all necessary updating is supplied by the compiler.

Each edit must also include a precise definition of those items that form the record to be edited. If, during the edit, adjustments are made to some of these items,
but at the end the record is still in error, then it is probable that these adjustments were spurious, possibly adjusting good values to agree with bad ones. Because of this, at the start of each edit, a copy is made of the record, and if at the end, the record is in error, then this copy which is (in general) unaltered during the editing, is used to restore the original values to the record. This action is taken automatically by the implemented form of the edit, and need not be specified by explicit statements in the FRED form of the edit.
3.2 THE STRUCTURE OF FRED.

3.2.1. The statements of FRED, like those of many other high level languages, can be split naturally into two classes, declarative and executable. There is no structure for combining the declarative statements of FRED other than a few ordering rules which will be mentioned as the statements themselves are being discussed. All declarative statements (except START SUBEDIT, END SUBEDIT and END EDIT) must precede all executable statements.

The structure of the executable statements exists in two equivalent forms. The "matrix" form is two-dimensional and affords the clearest picture of the logic of the edit (note that flow charts and decision tables also use two dimensions to clarify the logic). This is the form intended for use in the analysis. In practice, the number of columns of the matrix will be quite small, say five or six.

For program punching the matrix form must be converted by a very simple process, to the "indented" form, which is a representation of the edit in a form suitable for card punching and program listing. Each statement is punched on one, or more, cards, and the sequence of cards corresponds to reading the matrix row by row, as in the diagram.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
</tr>
</tbody>
</table>
|   | 11| 12| 13|...|etc.

Blank elements need not be included. The system of indentation improves the clarity of the logic. Though
not as suitable as the matrix form for analysis, edits of reasonable size can be designed and written as one step in indented form.

3.2.2. The matrix form of the edit depicts the edit as a rectangular matrix, the elements of which are either blank or FRED statements. In practice, the format is extremely free, the column width being determined to suit the size of the paper and the expected number of columns. Most edits will have at least three columns, but very few will have more than six. There will usually be only one FRED statement in a row.

Some of the larger statements may be too long to write as one line of the column, and in such cases should be written on as many lines as necessary, though conceptually these lines will form a part of one row. Comments may be written anywhere, including elements already containing statements, and may be distinguished by any means that the writer chooses, though the PL/I convention of enclosing a comment with the symbols "/\*" and "\*="/ is recommended, and is necessary in the indented form.

Provision should be made for attaching a label to any row of the matrix. The label should be of five or less alphanumeric characters, and the first character must be alphabetic.

The first statement of the edit to be performed is the one in the top left-hand corner of the matrix. The path of the edit then runs down the column until an IF, statement, or jump statement is encountered.
The IF-statement is the principal branching statement of the language. Its syntax is described in detail later, but a brief description is needed here to explain its effect on the path of the edit.

An IF-statement may contain several scalar expressions (defined as in a PL/I IF-statement, these will frequently be relational expressions such as "X = 15" to which the value "1" is assigned if the comparison is true and "0" if not true, or they may be scalar variables such as "TESTVAL", or expressions such as "TESTVAL == 2-4".) which may be split up into several groups, and which must be followed by an "ELSE". Each group and the "ELSE" will be written in the same column, though not necessarily in consecutive rows.

An IF-block will exist to the right of each group of scalar expressions and the "ELSE", commencing in the same row and occupying one or more rows. Exactly one IF-blank will be performed whenever the original IF-statement is performed. An IF-block may be just one statement, or it may be a string of statements. It may even contain other IF-statements and hence nested IF-blocks occupying other columns to the right. The IF-block is terminated by the presence of another statement (i.e. any entry except a blank or a comment) in the same column as the original IF-statement or to the left of this. This may be another group of scalar expressions, or the ELSE, or the next statement of
the edit. Note that by definition an IF-block can never include an entry in the column of the original IF-statement or to the left of this.

When an IF-statement is performed, the scalar expressions of the first group are evaluated until the first non-zero expression is found. If a non-zero expression is found then the next statement to be performed is the statement in the same row but next column, i.e. the first statement of the associated IF-block. The statements of the IF-block and any nested IF-blocks are performed according to the usual rules until the end of the block is reached, i.e. the next statement in this column is outside the IF-block. The next statement to be performed is the next statement in the same column as the IF-statement which initiated this block but following the ELSE. (In the case of nested IF-blocks this may be outside the higher level IF-block, in which case this rule is applied at the higher level and so on as necessary).

If none of the scalar expressions of the first group are non-zero then those of the second group are tested. If one of these is non-zero then the associated IF-block is performed. Otherwise the next group is tested, and so on.

If none of the scalar expressions of any group are non-zero, then the IF-block associated with the ELSE-statement is performed.

<table>
<thead>
<tr>
<th>ROW</th>
<th>COL</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>IF</td>
<td>IF</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>A</td>
<td>B</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>C</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td>ELSE</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Thus, in the above diagram where the letters represent scalar expressions and the numbered boxes represent blocks, suppose that A and B are 1. Then, since A is 1 we perform the IF-block starting in row 1, column 2. Since B is 1, we perform the IF-block starting in row 1, column 3. This, and the next statement in row 2, column 3 are performed. The next statement by the usual rule would be row 3, column 3, but since row 3 column 2 is non-blank, the IF-block is concluded before this statement. We therefore scan down the column of the last IF, column 2, to find the first non-blank statement after the ELSE. This is row 4, column 2, but, row 4 column 1 is non-blanks and so this IF-block is concluded before this statement. The next statement is therefore row 8, column 1.

Similarly, we can derive the following table giving the order of execution of the blocks for the value of each scalar expression:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4,5</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>3,5</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>4,5</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>3,5</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2,5</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1,5</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1,5</td>
</tr>
</tbody>
</table>

Other jump statements in the language cause jumps to be made to a labelled row. When such a jump is made, the next statement to be performed is the leftmost non-blank element of the row. There are no restrictions on the starting or finishing columns of a jump, but if a jump is made to a statement within an IF-block, execution continues as if the IF-block had been entered in the usual way.
A simple subroutine option also exists. A block of code forms a subedit provided that it begins with a "START SUBEDIT" statement in column 1 of the first row and ends with an "END SUBEDIT" statement in column 1 of the last row. This may be performed at any time by calling it with a "PERFORM" statement giving the name of the subedit. No redeclaration or declaration of new variables is allowed, and there is no provision for parameters.

3.2.3 The indented form of the edit may be regarded as a translation of the matrix form into a language suitable for punching on 80-column cards. This form is reproduced on the program listings and was chosen as the most suitable way of representing the edit within the confines of the printed page.

For the indented form, the statements are written on 80-character lines, such as those found on coding forms, consecutively down the page. In general, one statement is written to a line, though by using the continuation code one statement may be written over several lines. Each line may be punched onto a card for program input exactly as written.

The first five characters of each line may be used for labelling the row. For conformity with the matrix form only that line which corresponds to the leftmost non-blank column of a matrix should be labelled. The sixth character is a continuation character. If this is non-blank then this line is treated as a continuation of the previous line. A continuation line must not be labelled.

Characters 7 through 72 may be used for a FRED-statement, part of a statement or a comment. A comment must be prefixed by the character pair "/*" and ended by
the pair "*/*". A comment may be included on the same line as a statement. If a comment extends over more than one line, the continuation character should be used in the normal way and only one set of "*/*" and "*/*" symbols is needed, (i.e. they need not be repeated one each card).

Characters 73 through 80 are reserved for program identification and serial numbering in accordance with accepted practice.

The structural rules stated for the matrix form, in particular those concerned with IF-blocks, apply also to the indented form. The concept of columns of a matrix is however replaced by the indentation of a statement. To represent a statement in the first column of a matrix, the statement is written in indented form with its first non-blank character in character position 7. To represent the second matrix column, the first non-blank character must be in position 9, that is indented two positions. To represent column 3 the indentation is four characters, for column 4, six positions and so on.

Using this indentation to represent the matrix columns, there is no need to include blank elements in the indented form, as the structural rules permit determination of the row without ambiguity.

When a continuation card has to be written, there should be a non-blank character in character position 6. This is not significant other than as a continuation flag. The continued statements should have the same indentation throughout.

The indented form can be considered a structure in its own right by recording section 3.2.2 along the following lines:
"The first statement of the edit to be performed must have its first non-blank character in position 7. The succeeding statements have their first non-blank character in the same position until an If-statement or jump statement is encountered...

and so on mutatis mutandis.

3.2.4 As a simple example of the structure of FRED, consider the editing of figures extracted from an income tax return. Assume that two of the values given are NOCLAIMS, the number of children for which education expenses are being claimed and EDUCEXPS, the total amount claimed for education expenses. Assume further that both values are known to be numeric and non-negative. Since the amount claimed per child must be at least one dollar, and may not exceed 300 dollars, we have the following checks, using statements of FRED to be defined later - the intuitive meaning should suffice for the moment:

```
IF EDUCEXPS > 0 THEN
  IF EDUCEXPS > NOCLAIMS * 300 THEN ERROR 'EDUCEXPS TOO HIGH'
    EDUCEXPS < NOCLAIMS THEN ERROR 'EDUCEXPS TOO LOW'
    ELSE CONTINUE /* CHECK GOOD */
  ELSE IF NOCLAIMS = 0 THEN ERROR 'EDUCEXPS ZERO'
    ELSE CONTINUE /* CHECK O.K */
/* NEXT CHECK */
```

In indented form, this would be written:
IF EDUCEXPS>0 THEN
  IF EDUCEXPS>NOCLAIMS*300, THEN
    ERROR, 'EDUCEXPS TOO HIGH'
  EDUCEXPS<NOCLAIMS THEN
    ERROR, 'EDUCEXPS TOO LOW'
  ELSE
    CONTINUE; /* CHECK O.K. */
  ELSE
    IF NOCLAIMS>=0 THEN
      ERROR, 'EDUCEXPS ZERO'
    ELSE
      CONTINUE; /* CHECK O.K. */
/* NEXT CHECK */
3.3 **EXECUTABLE STATEMENTS.**

The statements of FRED are divided for convenience into four classes, called A-, B-, C-, and D- statements respectively. Within each class, the statements are numbered, so that for example, a CONTINUE statement is also referred to as a D-7 statement. A- and B- statements are declarative, and C- and D- statements are executable. The distinction between C- and D- statements is not well defined, but generally speaking C- statements are editing actions such as signalling errors, making adjustments, coding new items and so on, while D- statements keep control of the edit and are used to lead up to the proper editing action.

3.3.1 Before defining the executable statements of FRED some preliminary definitions are required.

**variable-name.** This is a name which represents a variable. It is a string of alphanumeric and break characters and must not be composed of more than 31 characters. The initial character must always be alphabetic. Since these names may also be used in the PL/I translation of the edit, all PL/I rules must be obeyed.

The variable will usually be a scalar variable, that is, a data item which may take on more than one value during the execution of the program, provided that these values are restricted to one data type (e.g. arithmetic or character), and if arithmetic to one base, scale, mode and precision.

The name may also represent one item of an array or structure, i.e. it may be a subscripted
name such as I(3,4). The name may even represent an array or structure, provided that the statement which includes it is well formed according to the obvious extensions of the PL/I rules for expressions involving arrays or structures. Since these rules are so complicated, no attempt is made to define them here, but an experienced PL/I programmer should find little difficulty in extending the PL/I rules for any required FRED statement.

Examples of variable names are
A AB A3 A-B THIS-IS-A-VARIABLE-NAME

expression. An expression is an algorithm used for computing a value. Expressions are of three types, scalar array, and structure; as for PL/I. For definitions of array and structure-expressions the PL/I specifications should be consulted. Scalar-expressions are discussed below.

Examples of expressions are
A+B A=B A+B**(C/D)

scalar-expression. Syntactically a scalar-expression consists of a constant, a scalar variable, a built-in function reference (see PL/I specifications), a scalar expression enclosed in parentheses, a scalar expression preceded by a prefix operator, or two scalar expressions connected by an infix operator.

Operators will usually be arithmetic, logical or comparison. A comparison operation results in the value 1 if the relationship is true, or 0 if it is false.

Examples are PAY.TAX * 2 , A<8*VAL and SIN (X)
comparison-operator. A comparison operator is one of

<  =  >  <=  >=

and has the expected meaning.

label. A label is a string of up to five alphanumeric characters, of which the first character must be alphabetic. It is used to identify a particular row of the matrix.

'message'
message-name
*flag*

When an error condition is found, it is signalled by a message on the error listings, or by a special flag. Such a signal can be given in three ways. The first is to give an explicit message to be written on the error listings. This may be of up to 30 characters. It is to be written as a character string surrounded by apostrophes, for example

'ERROR IN TAX' or 'PAY CODE ADJUSTED TO 3'
this is shown as 'message'.

The second way is by giving a character string variable name. The contents of the variable, which should not exceed 39 characters, at the time of performing the relevant statement, will be written on the error listings. This is shown by message-name.

The third way is by setting a flag. Here some variable, which must be defined in the external PL/I procedures, is set to some value by an assignment statement. This must be surrounded by asterisks, such as

*FLAG7 = 1* or *ERRORCT = ERRORCT + 1*
This is shown by *flag*
A further symbol required by the metalanguage is:

```
//
```

This is a special symbol which appears only on the statement definitions, and not in the language itself. It indicates that the following syntactic elements occur in the same column, but in a lower row than the preceding elements.

3.3.2 This section describes the C-statements. An important point is that the values of any item declared to be in the record being edited can only be altered by C-statements. Any attempt to alter them by a D-statement may cause serious errors at execution time.

C1 ERROR ('message'/message-name/* flag *)

This statement is used to define the condition of the record to be 'in error', and to write a message on the error listings, or set a flag.

Examples are:

```
ERROR DIAGNOSTIC (5)
ERROR * CODE(7) = 1*
```

C2 QUERY ('message'/message-name/* flag *)

This statement is used to define the condition of the record to be "queried", unless the condition has previously been set to "in error", and to write a message on the error listings or set a flag.

Examples are:

```
QUERY 'AGE TOLOW'
QUERY MESSAGE(1)
```

C3 ADJUST variable-name=expression, ('message'/message-name/* flag *)

Note that the expression must be followed by a comma, as both ' and * are valid characters in an expression.
This is the third of the basic error statements of FRED. The value of the expression, which may contain items of the record and external values, is calculated and assigned to the item of the record identified by the variable-name. All the usual PL/I conventions with regard to mixed mode expressions and consignments apply. If the condition of the record was "good" previously, then it is changed to "adjusted".

An adjustment made by this statement applies only to the copy of the record, so that if, at the end of the edit, the condition of the record is "in error", this adjustment will be reversed.

Examples are:

ADJUST SEX = 1, 'SEX CODE CHANGED TO MALE'
ADJUST ITEM(5) = 'NA', 'ITEM 5 NOT STATED'
ADJUST X = 5 * Y, * FLAG (7) = 1*

C4 TRY (variable-name = expressions, ('message'/message-name/ * flag *),) ... FAIL ('message'/message-name/ * flag *)

This is an extension of the ADJUST statement which is only valid when it forms a complete IF-block. When this statement is performed for the first time for a particular record, the first variable is adjusted to the current value of the first expression exactly as for an ADJUST statement. The IF-statement preceding this statement is then re-entered, and if a different path is taken then this adjustment remains. If however this statement is re-entered, then the adjustment is reversed and the error message is deleted from the listings, or the flag cleared, and the next adjustment of
the string is made. The IF-statement is re-entered, and so on.

If all adjustments are unsuccessful, then the record condition is set of "in error" and the error message specified beyond the "FAIL" is written.

Example:

IF TONS/COST > LIMIT THEN TRY TONS = TONS/20, 'WEIGHT GIVEN IN CMTS', FAIL 'WEIGHT TOO HIGH' ELSE CONTINUE

If the ratio of weight to cost is too high, then the weight is adjusted as if it has been expressed in hundredweights instead of tons. If this is successful, the adjustment remains. If this fails, the weight is reset to its original value and the record is in error.

C5 LOOK UP variable-name TO variable-name,
(comparison-operator expression, expression,)
OTHER expression, ('message'/message-name/ flag )

This statement is a further extension to the ADJUST statement which allows the adjustment to be made on a table look-up basis. The second variable-name (which must represent an item of the record) is to be assigned a value dependant upon the value of the first variable. A scalar-expression is formed from the first variable-name the first comparison-operator and the first expression. If this is true (non-zero), then the value of the second expression is assigned to the second variable-name. If it is not true, then the operation is repeated with the first variable-name and the next set of (comparison-operator expression, expression,) and so on until either a true value
is obtained or until OTHER is reached, when the value of the last expression is assigned to the second variable-name.

If the record condition was previously "good", then it is set to "adjusted" and an error message written in the usual way.

Note that the scalar-expressions need not be exclusive nor complete. The first one that is true determines the value, and if none are true, a value is set by the "OTHER" expression.

Example:

```
LOOK UP NUM TO INT, < 100, 0, < 200, 100,
< 300, 200, < 400, 300, < 500, 400, OTHER 500,
'SET INT TO WHOLE NUMBER OF HUNDREDS'
```

This adjusts INT to the greatest whole number of hundreds less than NUM and not exceeding 500.

C6 CHANGE variable-name = expression

C7 DEFINE variable-name = expression

There are occasions in an edit when it is necessary to adjust or set a value without writing a diagnostic, or changing the condition of the record. For example, a particular error may be very common and easy to adjust, and because of this it is not wanted to slow down the error checking process by including such adjustment on the error listing. Also, it may be desirable to introduce during the edit some values not present in the record read, such as the average of several items, or a code value derived from some relationships between items.

These two statements can be used for this purpose. Their effect is identical, the two forms
exist merely to convey the cause of the change to the source listing. The former implies a trivial adjustment, the latter the definition of a new item. In both cases, the item name is set to the value of the expression. Any values changed by these statements are not restored if the record condition is "in error".

3.3.3 In general terms, the D-statements are used to control the path of the edit and perform arithmetic or other operations on variables that are not part of the record.

\[
\text{D1} \quad \text{IF scalar-expression [ , scalar-expression]...THEN [// scalar-expression [ , scalar-expression]...THEN] }\quad \ldots \quad // \text{ELSE}
\]

The scalar-expressions, which will frequently be comparison operations such as "X = 20", but may be more general expressions, are evaluated in turn until the first non-zero value is encountered. The program then performs the IF-block which has its first statement in the same row but next column to the non-zero scalar expression. If no scalar expression is non-zero then the IF-block beside the ELSE is performed.

Exactly one IF-block is performed, and after that, the next statement to be performed will be in the row following the end of the IF-block attached to the ELSE and in the same column as the ELSE unless this row terminates a higher-level IF-block (i.e., one containing the IF-statement under discussion) in which case the next statement to be performed will be that given by applying this rule to the higher level IF-block (and so on if necessary).
A convention that allows for easier coding is that if all the scalar-expression are comparison operators with the same operand as the left hand side, then this operand need be written only in the first expression. For example

IF \( x > 40, x < -40, x = 0 \) THEN

is equivalent to

IF \( x > 40, < -40, = 0 \) THEN

This rule may apply through all the sets of scalar- expressions in one IF-statement.

Example:

To check the validity of a date

\[
\begin{array}{c|c|c}
\text{IF MONTH} = \text{'FEB'} & \text{IF DAY} > 28 & \text{ERROR 'DATE BAD'} \\
\text{THEN} & \text{THEN} & \text{CONTINUE} \\
\text{ELSE} & \text{ELSE} & \text{CONTINUE} \\
\text{ELSE} & \text{ELSE} & \text{CONTINUE} \\
\end{array}
\]

D2 variable-name = expression

Any variable which is not declared as an item of the record may be altered or defined by an assignment statement of this type. Any assignment that is acceptable to PL/I is acceptable to FRED.

D3 DO < PL/I statements > END

More complicated PL/I including, for example, input/output, data conversion, internal procedures
and so on, can be embedded in this statement. The "DO" and "END" are keywords recognised by the FRED compiler. Whatever is between those is accepted as PL/I statements. These statements may include "DO" or "END" and must include the proper PL/I punctuation. The whole of the PL/I group is one statement to FRED, and hence must be written on one card, or as many continuation cards as necessary.

D4 GO TO label

A jump may be made to any labelled row of the matrix by this unconditional jump statement. The next statement that will be performed is the left-most statement of the row with the given label.

If a jump is made to a statement within an IF-block, then on concluding the IF-block, the path of the edit will be the same as if the IF-block were entered as usual by finding a non-zero scalar-expression.

D5 LOOK UP variable-name FOR variable-name,
    (comparison-operator expression, expression,)
    OTHER expression

This statement is similar to the "LOOK UP...TO" statement C5. Whereas C5 was used to adjust items declared as part of the record, this statement is used to look up values for any variable which is not part of the record. Again sets of (variable-name -1 comparison operator expression) are built up, and for the first one that is true, the second variable-name is set to the value of the corresponding second expression.

Example:
LOOK UP X FOR Y, = 'MALE', 1, = 'FEMALE', 2,
    = 'NOT STATED', 3, OTHER 9
D6 LOOK UP variable-name JUMP, (comparison-operator expression, label, ...) OTHER label.

This is a jump depending on a table lookup. A scalar-expression is formed from the variable-name, first comparison-operator and first expression. If this is non-zero, the program jumps to the first label. If it is a zero, a second scalar-expression is evaluated in the same way and so on. If all scalar-expressions are zero, the jump is made to the label following the "OTHER".

Example:
LOOK UP POINTER JUMP, = 1, LABL2, OTHER LABL3
If POINTER = 1, the next statement will be in row LABL1, if POINTER = 2 it will be in row LABL2 and otherwise it will be in row LABL3.

D7 (CONTINUE/**)

This is a "do-nothing" statement, useful for IF-blocks that are needed only for completeness. The two forms are equivalent, the latter being a shorter alternative to the former.

Example:
IF REMAINDER = 0 THEN **
ELSE ERROR 'NON ZERO REMAINDER'

D8 PERFORM subedit-name

This statement calls a subedit. The subedit with the given name will be performed, and the next statement will be the one beneath the PERFORM-statement unless an IF-block is concluded at this point when the usual rules apply. Note that only subedits that are internal to the edit may be called, hence, a subedit cannot be used by
more than one edit. Note also that no parameters are allowed.

D9 /* some comment */

Comments can be embedded freely anywhere in the edit in the above form.
3.4 DECLARATIVE STATEMENTS.

There are two classes of declarative statements. The A-statements are primarily concerned with the definition of variables, and the B-statements with the error listings.

3.4.1 The A-statements are very similar to corresponding PL/I statements. This is necessary to allow FRED to be embedded in PL/I and vice versa. A knowledge of PL/I, particularly in respect of the definition of variables, is assumed in this section.

An edit is translated into a block of code which is equivalent to a PL/I procedure, which may be internal or external, and generally is the same as for an ordinary procedure in the same situation.

It is useful to consider at this point the variable items (i.e. all values except those written as literal constants) used in a typical edit. There are, generally speaking, four classes.

The first class consists of those items that form a part of the record being edited, say "record variables". In the typical scheme, these will be read and interpreted by a suitable PL/I procedure before being edited. These variables will therefore be defined by the calling procedure and will not need redefinition (except in the EDIT-statement q.v) if the edit is written as an internal procedure. If the edit is written as an external procedure these will need definition with the EXTERNAL attribute, or as parameters.

There are those items that are used in checking all records; look-up tables, limits, lists of acceptable codes, and so on. Although these are not always strictly constant, as in some edits the limits may move to follow
trends in the data, these will be called "reference constants". These are rarely used outside the edit, so will normally be defined in the edit itself, usually with the STATIC attribute to preserve the values between calls.

There are those items that are used to check one record, or a group of records only, such as a previous record for the same entity or a different member of an hierarchic structure. These will be called "reference variables". They may be obtained before entering the edit, in which case they will be defined in the calling procedure and the remarks for record variables apply. Alternatively, the appropriate set may be found during the edit, in a DO-statement, in which case the necessary declarations must be made at the start of the edit.

Finally, there are values that are not part of the record, but that are computed from it for use in the editing — sums, differences, quotients of items and so on. These, called "computed variables" will usually be local to the edit, and any necessary declarations must be made at the start of the edit. (All default options of PL/I are available in FRED.)

Thus, most edits require some definition of names, particularly of reference constants and computed variables. In addition, when the record is written as an external procedure, all variables in all classes will need definition, and most of this must be done explicitly.

3.4.2 A-statements.

A1 START EDIT edit-name [(parameter...)]

This is the statement that defines the start of an edit. A name, of not more than seven
characters, must be supplied, and the edit can be called by this name from any point in the program, exactly as a PL/I procedure.

"START EDIT" is a reserved character string (except in comments) in any PL/I program that contains a FRED edit. The parameter string, if there is one, follows the PL/I rules.

A2  END EDIT [edit-name]

This statement ends on edit and signals that the following code (if any) is PL/I.

A3  DECLARE [level] name [attribute]...

[,[level] name [attribute]...]

The declaration of variable names discussed in 3.4.1 above can be done using one or more DECLARE statements. These are written exactly as for PL/I including factoring of attributes. The only difference is that this statement is terminated by reaching the end of the card, or last continuation card, instead of by a semi-colon.

The programmer must ensure that, as in PL/I all variable names are sufficiently defined by means of these DECLARE statements. The only variables that do not have to be defined are those that hold the copy of the record made by FRED and mentioned in 3.1.2.

This statement is optional, but will usually occur.

A4  EDIT [level] name [attribute]...

[, [level] name [attribute]...]

A further declaration is required to define the record to be edited. The items which are to be
considered as part of the record must be listed in this statement together with all attributes other than storage class. These attributes may be factored as in the general case.

If the edit is coded as an external procedure this statement will duplicate some of a DECLARE-statement, but in the more normal internal procedure, this will rarely happen.

This statement allows the copy of the record to be made, assists in producing the cross-reference table, and defines the scope of the C-statements.

A5 FLAG name

One variable, not in the record, must be chosen as the condition flag and specified by this statement. The variable should be fixed binary and must be defined in both the edit and the calling procedure. Its value at the start of an edit may be arbitrary. At the conclusion of editing each record, the flag has one of the following values:

0 if the record is "good"
1 if the record is "adjusted"
2 if the record is "queried"
3 if the record is "in error"

A6 MAXADJ constant

If many adjustments are made during the editing of one record, then it is unlikely that the record is being adjusted to its correct values. More probably, many items are being adjusted to agree with a spurious item, which itself is not being corrected. Hence, the number of adjustments that are made is counted, and when this count exceeds
a certain number the condition of the record is set to "in error".

This number is usually two, but can be set to any constant by using this statement.

A7 START SUBEDIT subedit-name
A8 END SUBEDIT [subedit-name]

A simple subroutine capability is included in FRED. A subedit is a block of code to which a "return jump" can be made from any point of the edit using a PERFORM-statement. The subedit does not allow redefinition of variables or explicit parameters, but is merely a block of code of the edit.

These two statements define the start and finish of a subedit. The subedit-name should not exceed seven characters, the first of which must be alphabetic. Nested subedits are not allowed.

3.4.3 B-statements control the content and format of the error listings. The order of appearance of the B-statements is arbitrary provided that all B-statements precede all C- and D-statements. All B-statements are optional, but one of B1 and B2 must appear.

The standard (default) procedure, which is followed when only a B1-statement is present is as follows. The error listings are written on the standard output medium; each page is numbered. All records which are not "good" are printed. If the record has been adjusted (and possibly queried), then both the original and the final versions are listed. If the record has been queried and not adjusted, then only the final version is listed; if the record is in error then only the original version is listed. Diagnostic messages are printed underneath the record.
B1 LIST variable-name [, variable-name]...

The LIST-statement defines the items to be printed on the error listings in the order in which they are to appear. No attributes need be specified. Array and structure names may be used where appropriate.

B2 PRINTNONE

This statement is an alternative to the LIST-statement, and specifies that no listing is to be written. Each error condition in this case must be signalled using the flag setting options as defined for C-statements.

B3 FORMAT remote-format

The format of the record on the error listing is by default that given by a "PUT LIST" statement. Any suitable format statement may be supplied by this statement. The remote-format is written as in PL/I and should include parentheses.

B4 FILE file-name

The listings are normally written on SYSPRINT, the standard system output file, but by using the statement, any file may be nominated for the output.

B5 HEADING 'character-string'

If a heading is required for the error listings, in addition to the page numbering, then one of up to 110 characters can be supplied by this statement. These characters will form the left end of the top line of each page.

B6 PRINTALL

This statement specifies that every record is to be listed, regardless of the error condition.
3.5 **EXAMPLES.**

3.5.1 Suppose that a record contains "date of birth" in day, month, year as "D1", "M1", "Y1", and "date of marriage" in similar form as "D2", "M2", "Y2" and "age at marriage" in years as "AGE". Then a check between these items could be coded as:

```
IF Y2 - Y1 = AGE THEN IF M2 > M1 THEN IF D2 > D1 THEN **
                   = M1  THEN                  ELSE ERROR 'AGE TOO HIGH'
                   = AGE-1 THEN ELSE ERROR 'AGE TOO HIGH'
                    > AGE THEN ERROR 'AGE TOO LOW'
                    ELSE ERROR 'AGE TOO HIGH'
```

3.5.2 Suppose that the record gives details of agricultural production, and contains acreage under wheat "AREA", and production of wheat "PROD", and state code "STATE". To test that the yield per acre is reasonable, a table look-up may be combined with an IF-statement as follows.

```
LOOK UP STATE FOR RATIO,
  = 'NSW', 10, = 'VICT', 20, = 'QLD', 20, = 'WA', 15,
  = 'TAS', 25, = 'ACT', 20, OTHER 10
YIELD = PROD / AREA
IF YIELD > 1.7 x RATIO THEN ERROR 'YIELD TOO HIGH'
     > 1.5 x RATIO THEN QUERY 'YIELD HIGH'
     > 0.5 x RATIO THEN QUERY 'YIELD LOW'
     ELSE * *
```
3.5.3 Suppose that at a computer installation, statistics are kept of machine utilisation. Suppose further that special statistics are kept for a string of "compile only" jobs, and that the data punched are starting time "START", finishing time "FINISH", language code "L", number of cards "CARDS", estimated time "EST", and user code "USER". START and FINISH are punched in minutes in two decimal places, and EST is in minutes.

The additional assumption is made that for each language, compilation time will be approximately proportional to the number of cards... An edit for these cards could be written as an internal FRED block as follows:

```plaintext
/* DECLARATIVE STATEMENTS */
START EDIT JOBTIME
EDIT (START, END) FIXED DECIMAL (7,2),
     (CARDS, EST) FIXED DECIMAL (3,0),
     L CHARACTER (1), USER CHARACTER (5)
LIST USER,START,END,ELAPSED,EST,L,CARDS
FORMAT (SKIP, A(7), X(2), 3 F(7.2), F(4), A(1), F(4))
HEADING 'COMPILCE ONLY JOB STATISTICS'
DECLARE (ELAPSED, FACTOR) FIXED DECIMAL (7,4)
```
/* EXECUTABLE STATEMENTS */

ELAPSED = END-START

IF ELAPSED < 0 THEN ERROR 'TIME NEGATIVE'
    ELAPSED = 0
    \( \geq 20 \) THEN ERROR 'TIME TOO LONG'
    ELSE **

LOOK UP L FOR FACTOR,
    \( = P, 0.01 \),
    \( = F, 0.003 \),
    \( = C, 0.012 \),
    \( = A, 0.008 \),
    OTHER 0

IF FACTOR = 0 THEN ERROR 'ILLEGAL LANGUAGE'
    ELSE FACTOR = FACTOR \times CARDS
        IF ELAPSED \geq 3 \times FACTOR THEN QUERY 'TIME HIGH'
            \( < 3 \times FACTOR \) THEN QUERY 'TIME LOW'
        ELSE **

IF EST > 5 \times ELAPSED THEN QUERY 'ESTIMATE HIGH'
    \( < 2 \times ELAPSED \) THEN QUERY 'ESTIMATE LOW'
    ELSE **

END EDIT JOSTIME
4. THE IMPLEMENTATION

Note that a complete listing of the pre-processor is attached as appendix C.
4.1 PRE-PROCESSOR FORM.

In this section the following definitions are used to distinguish between a "pre-processor" and a "compiler".

A **translator** is a program or routine which reads sets of statements in one programming language and processes them to form sets of statements or instructions in another programming language in such a way that the two sets of statements are logically equivalent. The set of statements processed is referred to as the "source code", and the set of statements or instructions produced as the "object code".

An **assembler** is a translator, whose source code is written in a symbolic assembly language, possibly with macroinstruction facilities, and whose object code is a machine language routine that may be loaded and executed on a specific machine after any necessary linkages and relocation.

A **compiler** is a translator, whose source code is written in a procedure-orientated or problem-oriented language, and whose object code is either a machine language routine or a symbolic assembly language program. In either case the object code is capable of being loaded and executed without further compilation, although assembly, linkage and relocation may be required.

A **pre-processor** is a translator, whose source code is written in a procedure-orientated or problem-oriented language, and whose object code is in some other procedure-orientated or problem-oriented language, so that further compilation is required before a machine language or assembly language form is available for loading and execution.
FRED, as outlined in the previous chapter, is closely related to PL/I, and allows arbitrary blocks of PL/I to be embedded in a FRED routine. If FRED were translated by a compiler, in the above sense, it would therefore be necessary to include PL/I itself as a subset of the source language. Further, a FRED edit will generally be only one of a set of procedures, the others being written in PL/I, such that the set forms a complete editing and processing routine. A FRED compiler would therefore have to establish the expected PL/I linkages of external names, entries, and so on, and conform to the dynamic allocation concept of PL/I. These tasks are immense, and beyond the scope of an experimental language, the specifications of which may change as experience is gained in its use.

The alternative approach, that of a pre-processor, was therefore chosen. A FRED edit is translated into an equivalent PL/I procedure. Any embedded PL/I statement, when recognised as such, can be copied without regard to either syntax or semantics. In this way a large amount of work is saved.

Furthermore, the pre-processor will also read and copy any PL/I statements which surround the FRED edit. These statements will usually form the remainder of the editing and processing program and hence the entire program may be pre-processed and compiled as one unit. The FRED edit may then be treated as an internal or an external procedure by locating it analogously to a PL/I procedure. In this way the scope of names and linkage rules of PL/I have their exact equivalent in FRED.

A further advantage of this approach, is that the pre-processor is machine-independent and should be able to
run on any machine with a PL/I compiler. Though there are very few PL/I compilers at the moment, an increasing number should become available in future.

The code produced by a FRED pre-processor run is written on a disc, and can either be saved for a later PL/I compilation or else compiled and executed at once. In most cases where a FRED diagnostic is written during pre-processing, the object code will not be valid PL/I, and the object code should not be submitted to the PL/I compiler.

The "Job Control Language" of the Operating System/360 allows a procedure of this kind to be effected very neatly. Although several control statements are necessary to load the pre-processor, establish the disc storage and load the PL/I compiler, these can be generated by one "catalogued" statement. Using this technique, one control statement, for example.

```
// STEPL EXEC FREDCLG
```

would be needed to perform all required steps. If there were no pre-processor diagnostics then the PL/I compiler would be called in and the program would be executed. If however, there were some pre-processor diagnostics, the PL/I compiler would not be called, and the code stored in the disc would be scratched.

Other control statements would specify the pre-processor step only, or pre-processor and compile only.
4.2 COMPILING TECHNIQUES.

4.2.1 In this section the term "compiler" will be used in its accepted sense to describe any procedure to translate one programming language to another, irrespective of the nature of the second language.

A review was made of several papers describing compiling techniques, to find one suitable for FRED. In the early stages of this review, it seemed that a table driven compiler using a syntax for FRED written in Backus Normal Form would be very suitable. A modified form of the five phase compiler described by Marshall and Shapiro (15) was envisaged. These phases included a syntactic analyser, a macro-writer, an optimiser, a code selector and an assembler. The optimising phase could be omitted in the first implementation and added at a later stage, after consideration as to how much of this phase would be duplicated by the PL/I compiler in its treatment of the FRED object code.

The principal advantage of this scheme was that the bulk of the work was in the syntactic analyser, and this would be unaffected by any changes in the language that would seem desirable after operational experience of FRED. An adaptation of FRED to produce object code in say, FORTRAN IV (used extensively at the A.N.U.) would require modification only to phases 4 and 5, and could be effected fairly easily.

It also seemed likely that a direct adaptation of some existing and proved syntactic analyser could be made, for example "ANALYZE" of Cheatham and Sattley (17), with consequent saving of debugging time.

A tentative syntax table was drawn up for FRED in Backus Normal Form.
<edit> ::= <start><declarative block><executable block><end>
<executable block> ::= <executable statement> |
                  <execuable statement> |
                  <executable block><executable block>
                  <subedit>
<subedit> ::= <start subedit><subedit block><end subedit>
<subedit block> ::= <executable statement> | <executable statement>
                  <subedit block>
<start subedit> ::= START SUBEDIT <subedit name>
<end subedit> ::= END SUBEDIT | END SUBEDIT <subedit name>
<subedit name> ::= <name>
<declarative block> ::= <declarative statement> |
                      <declarative statement>
                      <declarative block>
<start> ::= START EDIT <edit name> | START EDIT <edit name>
                  (<parameter string>)
<parameter string> ::= <parameter> | <parameter>,
                      <parameter string>
<end> ::= END EDIT | END EDIT <edit name>
<edit name> ::= <name>
<executable statement> ::= <if statement> | <error statement> |
                      <query statement> |
                      <adjust statement> |
                      <else statement> |
                      <continue statement> |
                      <look statement> | <do statement> |
                      <go to statement> | <try statement> |
                      <change statement> |
                      <define statement> |
                      <perform statement> |
                      <assignment statement>
<if statement> ::= IF<if kernel> THEN
<if kernel> ::= <if kernel A> | <if kernel> <name><operator><expression>
<if kernel A> ::= <name><operator><expression> | <if kernel A>, <operator><expression>
<error statement> ::= ERROR <diagnostic>
<query statement> ::= QUERY <diagnostic>
<adjust statement> ::= ADJUST <name> = <expression>, <diagnostic>
<diagnostic> ::= ' <message>' | <message-name>
<else statement> ::= ELSE
<continue statement> ::= CONTINUE
<look statement> ::= <look to statement> | <look for statement> |
<look jump statement>
<look to statement> ::= LOOK UP <name> TO <name>, <look set>
OTHER <expression>, <diagnostic>
<look for statement> ::= LOOK UP <name> FOR <name>, <look set>
OTHER <expression>
<look jump statement> ::= LOOK UP <name> JUMP, <jump set>
OTHER <label>
<look set> ::= <comparison operator><expression>, <expression> |
<look set><comparison operator><expression>, <expression>,
<jump set> ::= <comparison operator><expression>, <label> |
<jump set><comparison operator><expression>, <label>,
<do statement> ::= DO< PL/I statements> END
<go to statement> ::= GO TO <label>
<try statement> ::= TRY<try set> FAIL <diagnostic>
<try set> ::= <name> = <expression>, <diagnostic> |
<try set><name> = <expression>, <diagnostic>,

For this syntax, the following are assumed terminal symbols, defined in PL/I.

\[
\begin{align*}
\text{parameter} & \mid \text{name} \mid \text{operator} \mid \text{expression} \mid \text{message} \\
\text{message name} & \mid \text{label} \mid \text{comparison operator} \mid \text{PL/I statements} \\
\text{declaration} & \mid \text{digit} \mid \text{remote format} \mid \text{character string}
\end{align*}
\]
4.2.2 The investigation of these techniques led to the conclusion that general syntax directed compiling techniques are not suited to a FRED pre-processor.

The structure of FRED is difficult to bring out in the syntax tables. This structure is most important in the semantics of FRED, and is of relevance in the syntax too. This is particularly noticeable in the IF-blocking rules, which we inadequately described in the above table. Also, there are other structural rules, for example, the declarative section must contain exactly one edit statement.

Furthermore, like many other languages, FRED has many declarative statements, examples of delayed coding, a complicated set of terminal characters, and a need for helpful error messages. These properties are difficult to implement in a phrase structure grammar, and hence it was decided not to continue with this approach.

The other papers that were concerned with generalised techniques also seemed inappropriate for a first version of FRED.

Floyd (21) described the use of precedence grammar, a specialisation of the more general phrase structure grammar with many advantages in compiling techniques, but, the first specialisation required by Floyd is that the language must be an "operator language". An operator language is a phrase structure grammar in which no production takes the form \( U \rightarrow x U_1 U_2 y \) where \( U_1 \) and \( U_2 \) are non-terminal characters. This condition is violated by the first line of the syntax table. Floyd's techniques are therefore not directly applicable to FRED.

Graham (18) discusses bounded context translation. An important condition of his paper is that at each step in
the scan of an expression, the decision as to what action to take next is a function of the symbol currently under scan and of $N$-symbols on either side, where $N$ is fixed for the particular language.

This condition does not hold for FRED as can be seen by considering the case of a statement such as

"$X = Y + Z + \ldots \ldots$"

where an arbitrary number of symbols may be read before the end of the statement. Only at that point is it possible to decide whether this is an assignment statement or a secondary IF-statement (denoted by read a "THEN"). Yet the initial action is dependent on this.

4.2.3 In view of the above discussion, the method chosen for the implementation of FRED was a straightforward context-dependent translator, i.e. of the traditional type used for nearly all production FORTRAN compilers. This approach avoided most of the difficulties mentioned above. In particular it allowed free use to be made of "terminal characters" on different levels. Thus, when compiling an EDIT-statement, each parenthesised expression must be broken down into its elements and these considered individually. However, when compiling an assignment statement a parenthesised expression is treated as a terminal character, and its content ignored.

An unfortunate consequence of this decision is that this approach is the least flexible of the methods discussed. To try and alleviate this, the translator was written in a strictly modular form. No attempt was made to save space by linking the coding for two or more statement types if this meant that the format of one statement became dependent on the others. Because of this, it is hoped that
any changes to FRED suggested by experience in its use will be easily accommodated in the coding. Further, because the translator is a pre-processor written in its own object language, and not for example in a specific assembly language, the compiler coding should be intelligible to any competent programmer reading it.
4.3 **THE RECOGNISER.**

4.3.1. The recogniser is constructed on two levels. The first is essentially a character recogniser which also reads and lists the cards, and the second level is a pair of routines which assemble the characters into words.

The character recogniser "NEXTONE", listed in appendix C, is the only routine which reads and lists a card. It returns one character on each call, together with a flag to distinguish between alphanumeric, special and blank characters and another flag to signal the end of a statement.

The routine is context dependent in the sense that the nature of the last character signalled influences the present signal. The most important example of this is that if the last character signalled was a blank, then all blank characters on this call will be ignored, and the character returned will be the next non-blank character. Thus blanks (except in character string constants) are squeezed out. Use is also made of this feature to simplify the building up of words. If the last character read was alphanumeric and the present character is special, then a blank is signalled so that the processing of the name or the numeric constant can be completed without the complications of a trailing special character. The special character is returned on the next call. Finally, this feature is used at the end of the statement. If the last character signalled was non-blank, then a blank is signalled and the end of statement signal is deferred until the next call.

NEXTONE also lists cards. A continuation card is listed as soon as it is read, but the listing of a non-continuation is deferred until the call after the one in which the end of the preceding statement was signalled.
At this time all diagnostic messages referring to the preceding statement should have been written.

Continuation cards are checked by this routine to see that the first non-blank character is in the correct column. If this is not so, then a diagnostic is written and the contents of the card are ignored.

Comments are detected and squeezed out by this routine. They are regarded as blanks, and all the rules concerning these apply. Character string constants are signalled as alphanumeric throughout, regardless of the individual characters within the string.

4.3.2 "ENTITY", one of the routines in the second level of the recogniser, is used to assemble one syntactic unit—a name, numeric constant, character string constant, or special character pair. ENTITY makes calls to NEXTONE. If it receives first an alphanumeric character, it continues to call NEXTONE and assemble the characters until the first blank is signalled. The name is returned as a character string together with a count of the number of characters in it. If the number of characters exceeds 32, a diagnostic is written and the excess characters are truncated.

For all statements following an edit statement which defines the item names or the variables being edited, a name which has been assembled is checked against the list of item names. If the name is found in the list, the flag corresponding to that name is set. These flags are used to update the logical cross-reference table for all executable statements.

When a special character is read, it is returned as one syntactic unit unless it is the first character of one
of the pairs ":="", ":="", ":="", ":="", "="", "="", "="", "="", when the pair is returned as one unit.

The other routine in this second level of the recogniser, "_KEYWORD"", is called at the start of each statement. It assembles the first syntactic unit, and when complete, compares it with a list of all possible first words, for example "DO", "ERROR", "GO", "DECLARE". If a match is made, the statement type flag is set accordingly, and a flag is set to indicate the column containing the first non-blank character. This flag is used to check the IF-block structure.

If the unit is alphanumeric, but is not a keyword, then the statement is tentatively identified as an assignment statement or a subsidiary IF-statement. A later consideration of the history of statements starting in this column will resolve the type, as it is not possible to have an assignment statement between an IF and an ELSE, and conversely a subsidiary IF-statement must lie within these limits.

If the unit is special, unless it is possible as the start of a subsidiary IF-statement (":="", ":="", ":="", ":="", ":="", ":="", "="", "="", "="), the statement type is invalid. A diagnostic is written and this card, and any continuation cards are ignored.

In the case of an assignment or subsidiary IF-statement, the word read by KEYWORD is compared with the list of item names so that the cross-reference table can be updated in the usual way.

4.3.3 One further routine, "BRACKET", may be considered as part of the recogniser. This is called whenever the start of a parenthesised expression is encountered and
when this expression may be treated as a terminal character. This routine reads units from ENTITY and writes them as object code, keeping count of the number of right and left parentheses. When the number is equal, a return is made to the calling routine. Diagnostic messages are written if the number of right parentheses exceeds the number of left parentheses, or if the end of statement is read without completing the expression.

The units are read through calls to ENTITY so that the mechanism in ENTITY to update the logical cross reference table is effective in this case too.
4.4 PROCESSING ROUTINES

4.4.1 The processing routines consist of a set of routines to process each statement type and a set of control routines to establish the linkages needed between statements. The structure of these routines is shown on the next page. The control routines are "FRED", "COLCHK", "STARTER", "COMPLT", and "EXECON".

FRED is the master control routine which initiates and completes the processing, and retains control throughout. It first initialises flags and opens all necessary files. STARTER is then called to read and print any PL/I statement that may precede the first FRED edit block. On return a flag signals either the start of an edit block, or an end of file mark detected. In the latter case the job is terminated. In the former case, other flags are initialised, and the edit name is processed.

KEYWORD is called to determine the next statement type. On return, a flag is tested, and the appropriate processing routine is called. These steps are repeated until the first executable statement is read.

At this point, FRED tests that a list or printfone, an edit and a flag statement have been read, and if necessary, prints a diagnostic. EXECONE is then called to write some object code that is necessary at this point, principally the code to make a copy of the record.

Each executable statement is processed by the appropriate routine. The cross-reference flags for this statement are then tested, and if any are set, the cross-reference table is updated accordingly. KEYWORD is then called for the next statement, and so on. If a declarative statement is read at this stage, a diagnostic message is written and the statement treated as illegal.
Schematic layout of compiler routines.
For each statement, COLCHK is called to check that the starting column of the statement is consistent with the apparent IF-block structure. Thus, if a statement starts to the right of the previous one, the previous statement must be an IF or a ELSE, and the present statement starts an IF-block. If a statement starts to the left of the previous one, this ends an IF-block and an IF or ELSE must be the previous statement of the column currently under consideration.

When an end-edit statement is read, COMPLT is called to write the necessary code, including the section to write the diagnostic messages. COMPLT also writes the cross-reference table. Then STARTER is called again to read any following PL/I statements until the next FRED edit block or end of file. When an end of file is read, a message is written stating either that the edit was satisfactory or that it contained errors. In the former case the object code can be used as input to the PL/I compiler. In the latter case it should be terminated as it is most unlikely that the object code will be error free.

STARTER reads card images and writes them unaltered to the object code file. No check is made on the validity of the PL/I statements being read. When a card with "START EDIT" is read, the next name is extracted and stored as the name of the edit. A procedure statement is written using this name and copying any parameters to the name. A declare statement is then written to define the internal error flags that will be used by the edit.

EXECON E is called on reading the first executable statement. By this time the set of items to be edited should have been defined, and EXECON E writes the code which will initialise the error flags and make a copy of the items at the start of each edit.
COLCHK contains an array of labels, one corresponding to each of ten columns of the matrix form of the edit. When an IF-block is terminated, COLCHK determines the column corresponding to the IF-statement, and writes a "GO TO" statement using the label stored for that column. When the end of an IF-sequence is detected, COLCHK determines how many sequences are ended at this point (there will frequently be more than one) and writes the label stored for each of these. The labels just written are then modified to preserve uniqueness and stored. When an error in the IF-block sequence is detected, an appropriate diagnostic is written and COLCHK clears all its flags to try and reduce the number of later statements that will seem wrong because of the error in the present statement.

COMPLT writes code to interrogate the internal edit flags, set the external flag and, if necessary, restore the record to its original values. The various error listing options that were requested are scanned and code written to produce the listings in the form specified. This may involve the standard output or a nominated file, a given format or no format, all records or only error records, or no listings at all. Finally COMPLT writes the logical cross-reference table on the standard output to follow the program listings.

4.4.2 Most declarative statements do not produce object code at this stage, but merely involve the storing of names and flags which control the later writing of code, particularly during COMPLT. Many declarative statements can occur once only, and a diagnostic is written if there are two occurrences. Similarly a diagnostic is written if there are two mutually contradictory statements such as printall and printnone.
The edit statement is analysed by "EDST". In this statement the items to be edited and their attributes are set out exactly as in a PL/I declare statement, but without storage class attributes. This statement is written on the object file as a declare statement, but with all item names prefixed by $Q#$ to prevent double definition of any of the names. The variables defined by this object code statement are used to store the copy of the record and must therefore have the same attributes as the original variables.

The names declared are also stored in an array used for three purposes. Firstly by EXECONE and COMPLT to write object code to store, and if appropriate restore, the original values of the record. Secondly by ENTITY and KEYWORD to recognise item names and update the cross-reference table accordingly. Thirdly to check that names given as items of the record in some of the editing statements do in fact form an item of the record.

The other declarative statements that cause immediate output of object code are the heading, format, and declare statements. "DCST" reads the declare statement and writes the equivalent PL/I statement on the object file. "HEST" reads the heading statement and writes an "ON ENDPAGE" statement to ensure that the heading will be written. Since the file name must be given in this statement, if there is file statement, then it should precede the heading statement. Otherwise "SYSPRINT" is assumed. The format statement, interpreted by "FOST" causes the given format to be written prefixed by the label "$Q#FORM". Flags are set so that this will be given as a remote format by the code written by COMPLT.
4.4.3 Many executable statements have a straightforward PL/I translation. For example, assignment statements are copied without change by "AIST", which finally adds a semicolon. Some of the routines, however, use special devices as mentioned below.

When the translation of an IF-statement is started, the first name that is read is stored, and a flag is set. The first relation is then coded in parentheses. If this is followed by a comma, indicating that a further relation follows, an or-sign is written. The next unit is then read, and if this is not an item name but a comparison operator, then the stored name is written before the operator. A whole string can be treated in this way, the stored variable being added to each. Thus

"IF X = 5, = 6, = 7 THEN"

is translated as

"IF (X = 5) | (X = 6) | (X = 7) THEN DO;"

If a name is given then the flag is cleared and each following relation must include a name on the left hand side.

Any subsidiary IF-statements are treated similarly, a separate store and flag for each matrix column preventing confusion when several IF-groups are being processed. The code for each statement is preceded by "END;", thus concluding the do-group of the previous block. A further "END;" is written for the else statement, but the IF-block associated with this is not coded as a do-group. Thus, if several IF-blocks are terminated by the one statement there is never need to compute the number of "END;" 's needed. This structure is in addition to the go-to statements and labels supplied by COLCHK.

The diagnostic supplied with each error, query or adjust statement is translated by a common routine, "ADCON", 
part of "ERST". It is first tested to see that it is of the correct form. That is, if there is a printnone statement, the code must specify the setting of a flag, otherwise there must be a diagnostic message given either as a character string constant or the name of a character string variable. The first option is processed by writing the assignment as given. For the second option, the object code declares an array of 40 character string variables, each 32 characters long, and a counter. The object code produced increments the counter, unless its value is 40, and then equates the corresponding variable of the array to the character string constant or variable. If during the course of one edit, more than 40 diagnostics are required for one record then since the counter is not incremented above 40 the first 39 and the last diagnostic will be written. The others will be lost.

The three kinds of look up statement are processed by a single routine, "LOST". The first variable name given is read and stored, then the type of look up statement is determined. If it is "to" or "for", then the second variable name is next read and stored, if it is "jump" this step is omitted.

A set of "operator expression, expression" is then read. For a "to" or "for" statement this set is coded as:
"IF first-variable-name operator expression THEN DO;
second-variable-name = expression; GO TO label; END;"
where the label is stored in LOST. For a "jump" statement, the set is coded as:
"IF first-variable-name = expression THEN GO TO
label;"
where this label is the one read from the source code.
On reading the "OTHER" and following expression, or label, this is coded unconditionally as
"second-variable-name = expression;"
or
"GO TO label;"
The processing is now complete for a "jump" statement. For the others the stored label is now written and the stored version is modified to preserve uniqueness. For a "to" statement a jump is now made to the common diagnostic processing routine to process this in the usual way.

4.4.4 With the introduction of a time-sharing system on the A.N.U.'s IBM 360/50 the amount of core storage available to a program will be reduced, and it seems likely that the complete pre-processor will then be too large to be loaded as one unit. For this reason a simple overlay structure has been tested. The main segment contains the recogniser and the control routines. The first overlay contains those routines needed to process the declarative statements and the second overlay these needed to process the executable statements.
4.5 IMPLEMENTATION LIMITATIONS.

Some limitations on the source code exist in this first implementation of the language which, it is hoped, will be removed in later versions. The most common reasons for these are a desire to keep the compiler tables to a reasonable size without introducing push down stacks or other forms of dynamic storage, and an attempt to avoid as many of the complexities of PL/I as possible by keeping the terminal characters at a high level.

These limitations are summarised below.

1. The try statement has not been implemented. The principal difficulty was in writing code which would establish, during each execution of the edit, the number of times that the try statement had been entered. It seems necessary to initialise such a counter some distance before the actual try statement, and this necessitates either a lot of rarely used initialisation written in every edit in case it will be needed, or else the introduction of a second pass by the compiler. Neither of these seemed justified for a non-essential statement.

2. The conditions of an IF-statement must be of the simple format

   "item operator expression".

   Forms involving parenthesised and concatenated operators such as

   "IF (A = B) (C = D)"

   are not allowed.

3. If the record has been adjusted, the error listings will include only the final values of the items.

4. If the repeated left hand side of an IF-statement is to be omitted, then it must be a simple variable name. A subscripted name cannot be accepted.
5. Similarly the names in a look up statement must be simple.

6. Character string constants may not exceed 30 characters (except for heading statements) and may not contain embedded apostrophes.

7. Diagnostic messages may not exceed 32 characters.

8. Only 10 columns are allowed for the matrix form of the edit.

9. Blank cards may not be inserted between continuation cards. If a comment is continued, the usual rules about first non-blank character apply.

10. No account can be taken of subscripts in the logical cross reference table. Arrays are treated as one item.

11. All parameters to an edit must be contained on one card.

12. No check is made to see if an assignment statement is altering a variable declared to be part of the record.

13. The names supplied as items of the record are prefixed by "Q-#" before definition as a copy of the record. All internal flags written by the edit also begin with "Q-#", and this could lead to duplication of names. This could be avoided if the copies were prefixed by, say, "Q-@".
5. **USING FRED.**
5.1 **EXAMPLE 1.**

The first example, listed in Appendix D, shows a FRED edit included in a simple processing program. "WEATHER" can be used to read any number of cards containing climatic information, and produce averages and totals for the set of records provided. The edit was written on the assumption that all records would apply to the same area, so that it was possible to set a realistic upper limit to, say, maximum temperature in February, which would be applicable to all cards. This edit was written in a few hours and required one run to remove some punching errors in the source code, and a second run to improve the formats of the error listing. The edit was then ready for use, and appears to work in all respects.

Later, some deliberate errors were caused and the program resubmitted to the FRED pre-processor to demonstrate some of the diagnostics and use of the cross-reference table.

The edits applied were devised by searching published records for the Canberra area (22). From these, checks could be made on the upper limit of maximum temperature, the lower limit of maximum, and typical values of these to insert into incomplete records. Checks on the lower limit of maximum and the upper of minimum also seemed desirable, but as no values were available, *a priori*, the only check applied was to ensure that maximum exceeded minimum.

Limits that seemed reasonable were used to test rainfall and hours of sunshine.

The date was tested, but as year is not included in the data, every occurrence of February 29th was accepted.

A particular feature of this edit is the use made of in-line look-up tables. Although these are not
necessarily the shortest, or the fastest (measured in execution time) way of implementing these checks, they are very clear. This clarity was an important feature in reducing the time taken to test an edit.

As there are very few items in the record, the number of adjustments that can be allowed was reduced to 1. The external flag, "FLAGn" was used to exclude any incorrect records from the averaging steps.

The FRED diagnostics shown in the first listing attached show that in two cases, no allowance has been made for an invalid month code. In the first case, the IF-statements used to check the number of days in a month are not followed by an ELSE. Unfortunately this error was detected within the next IF-block, and as a result the pre-processor, in this early implementation, becomes uncertain of the present IF-block structure and writes two more spurious messages. These will disappear when the first error is corrected.

The second error occurred in a look-up statement, where the "OTHER" option was omitted. This was signalled in a straightforward way.

Examination of the logical cross-reference table for this compilation shows that there is no entry in the row of the table corresponding to the item "HRS". This implies that this item is unedited, and it would be advisable to devise a suitable check. The entries for "FALL" and "DAY" show that these items are present in the edit, but are not cross-checked. In the circumstances this seems inevitable for FALL. DAY is in fact checked against MONTH, but as this is performed using the intermediate variable DAYLIM, this cross check does not appear in the table.
The second run listed was an execution. The PL/I code produced by the pre-processor is listed, though this is not normal practice. The error listings and program results may be seen on later pages. The data used was needless to say, a special test-pack and has a very high error rate.
5.2 **EXAMPLE 2.**

The second example, also listed in Appendix D, shows a FRED edit that may be used as one part of a processing run, or as part of a file creation run. The program as listed, includes only the edit and a single card reading routine.

"INVOICE" is used to check the items on an invoice, a classical problem for editing. This was chosen as it is a very different type of edit from the preceding example. A large number of arithmetic cross checks can be applied, and the limits are the same for all records.

The edit was designed and coded in a few hours, but required three runs to debug. A reason for this is that the author has had little experience in the use of PL/I arithmetic expressions. The early versions therefore contained statements such as

"IF DISCOUNT = COST * DISCRATE/100 THEN" which were rarely satisfied. This is because DISCOUNT has been rounded before being punched as a data item, whereas the right hand side is evaluated and the comparison made with all significant figures present. The listed version of the edit allows a small tolerance to such tests to overcome the problem.

As before, a second run was made with deliberate errors to demonstrate some FRED diagnostic messages.

The items of the record were a customer code "CUSTOMER" and an item code "ITEM", both numeric, which were simply checked to lie within a given range. The quantity of the item purchased "UNITQTY", and the number of items purchased "UNITVAL" can be multiplied together to
give the price, "COST". There may be a discount "DISCOUNT", at one of several rates, and the rate is included as "DISCRATE". Similarly, there may be insurance "INSURANCE" at "INSRATE". A further charge may be made for delivery, "FREIGHT", and the ratio of FREIGHT to COST is imputed during the edit and included in the listings. The last item is the amount payable, "BILLED" which is

\[
\text{COST} + \text{INSURANCE} + \text{FREIGHT} - \text{DISCOUNT}
\]

An adjustment is made during the edit if BILLED does not equal the algebraic sum of the components as above and if either INSURANCE or DISCOUNT has been miscalculated. If the value obtained for INSURANCE by multiplying COST and INSRATE is such as to satisfy the check of BILLED, then INSURANCE is adjusted to this value. A similar adjustment may be made to DISCOUNT.

The listings are printed using a "PUT LIST" option, so that a format need not be given. A heading is given however, and this will be repeated on each page of the error listings.

As for the previous program, the PL/I object code is listed for demonstration purposes.
5.3 POSSIBLE IMPROVEMENTS.

The specifications and implementations of FRED, as set out in this thesis, seem to have fulfilled most of the aims listed in Chapter 2. In particular, FRED offers a very quick and reliable method of writing and testing small and medium sized logical edits. However, experience gained in the use of FRED, particularly by other programmers, will undoubtedly suggest improvements, and it is hoped to revise the language at a suitable time. As well as removing the implementation limitations, this revision is likely to include the following.

In some areas, the present specifications seem too restrictive. To allow the programmer more freedom, without increasing the chance of logical errors, a class of informative pre-processor messages could be useful. These would not inhibit execution, but would signal unusual actions.

An example of this is the adjust statement, where for a particular adjustment the programmer may wish to use the flag setting option instead of the message writing option, or even suppress the error signal altogether.

An extension of this scheme would be to insert a control so that the informative messages may be included or omitted for any particular run. There is a very similar option in the PL/I compiler.

The cross-reference table could be improved by indicating not only which items are checked with any given items, but also the groups in which these checks are made. Thus, if three checks include X,Y,Z;X,Z,W; X,Y respectively, the present entry for X is

\[ X \quad V \quad W \quad X \quad Y \quad Z \]
In the improved version, the entry would be

\[ X \quad Y \quad Z \quad W \quad V \]

with an additional code to signify that the item appeared, but without cross checks.

The efficiency of many of the instructions should be improved, particularly the look-up statements.

A version, compatible with FORTRAN IV would be especially useful to scientific users.

Two new instructions would be useful. These are

**LIMIT expression, limit-1, limit-2, ('message'/message-name/flag*)**

This would evaluate the expression, and unless its value exceeded limit-1, and was less than limit-2, the record would be queried and the error signalled in the usual way.

**COMPARE expression-1, expression-2, (value/value PERCENT), ('message'/message-name/flag*)**

This would evaluate the two expressions, and if the difference exceeded the value given in magnitude, or the percentage of expression-1 given, the record would be queried and the error signalled in the usual way.
APPENDIX A.

SYNTAX NOTATION.

In the description of the statements of FRED, the syntax notation is adapted from that used by I.B.M. in the PL/I Language Specifications (14). The relevant portion of the definition of this notation (pages 11 to 13 of the specifications) are copied below.

...(This notation) does not describe the meaning of the language elements, merely their structure; that is, it indicates the order in which the elements may (or must) appear, punctuation that is required, and options that are allowed.

1. A notation variable is the name of a general class of elements in the language. A notation variable must consist of:
   a) lower-case letters, decimal digits, and hyphens and must begin with a letter.
   b) A combination of lower-case and upper-case letters. There must be one portion in all lower case letters and one portion in all upper-case letters, and the two portions must be separated by a hyphen.
   All such variables used are defined either formally using this notation, or are defined in prose.
   Examples:
   a) edit-name-
   b) IF-block

2. A notation-constant denotes the literal occurrence of the characters represented. A notation constant consists either of all upper-case letters or of a special character.
Example:

\texttt{HEADING 'character-string'}

This denotes the literal occurrence of the word \texttt{HEADING}, followed by the literal occurrence of ', followed by the variable "character-string" defined elsewhere, followed by the literal occurrence of '.

3. \textbf{Parentheses} ( ) are used to denote grouping, and the oblique stroke / indicates that a choice is to be made.

Example:

\texttt{(CONTINUE / \textordmasculine)}

This denotes that a choice has to be made, and there must be either the literal occurrences of \texttt{CONTINUE} or the literal occurrence of \textordmasculine.

4. \textbf{Square brackets} [ ] indicate options. Anything enclosed in brackets may appear one time or may not appear at all.

Example:

\texttt{END EDIT [edit-name]}

This denotes the literal occurrence of the words \texttt{END EDIT} optionally followed by the variable "edit-name".

5. \textbf{Three dots...} denote the occurrence of the immediately preceding syntactical unit (i.e. a single variable or constant, or any collections of these surrounded by parentheses or square brackets) one or more times in succession.

Example:

\texttt{LIST variable-name [, variable-name]...}

The literal occurrence of \texttt{LIST} is followed by variable-name which may be optionally followed by one or more sets of the literal occurrence of , followed by a variable-name.
APPENDIX B.

JOB CONTROL LANGUAGE

The deck structures required to run FRED programs in three typical cases are as follows.

B1 Compile only.

a: Catalogued.

//JOB card
//STEP EXEC FRED
FRED program
/*

b: Uncatalogued.

//JOB card
//J OBLIB DD D S NAME = FREDLIB, UNIT = 2311,
VOLUME = REF D PACK8, DISP = (OLD,PASS)
//G EXEC PGM = FRED
//G OBJECT DD D S NAME = OBJ 001,SPACE = (80,
1000,50),RLSE), UNIT = 2311,DCB = (RECFM = F,
LRECL = 80), VOLUME = REF = PACK8, DISP = (NEW,
DELETE)
//G SYSIN DD *
FRED program
/*

B2 Compile and save PL/I code.

a: Catalogued

//JOB card
//STEP1 EXEC FRED
FRED program
/*
a: Catalogued.
//JOB card
//STEP1 EXEC FREDCLG
  FRED program
/*
//G. SYSIN DD *
  Data
/*

b: Uncatalogued.
//JOB card
//JOBLIB DD DSNM= FREDLIB, UNIT = 2311
  VOLUME = REF = PACK8, DISP = (OLD, PASS)
//G. EXEC. PGM = FRED
//G. OBJECT DD DSNM = OBJOOL, SPACE = (80, 100, 50), RlSE), UNIT = 2311, DCB = (RECFM=F, LRECl=80), VOLUME = REF = PACK8, DISP = (NEW, KEEP)
//G. SYSIN DD *
  FRED program
/*
//S2 EXEC CLEAN
//S3 EXEC PLIFCLG, PARM. C= 'SM= (1,80)', PARM.L = LET
//C. SYSIN DD DSNM = OBJOOL, DISP = (OLD, DELETE), UNIT = 2311, VOLUME = REF = PACK8
//G. SYSIN DD *
REFERENCES


