RECORD HANDLING

C. A. R. Hoare

A series of lectures to be delivered at
The NATO Summer School,
Villard-de-Lans
September 12 - 16, 1966.

RECORD HANDLING

C. A. R. Hoare

Introduction

Problems involving structured data arise in many areas of computer application, for example, in simulation studies, information retrieval, graph theory, compiler writing, symbol manipulation, and business oriented data processing. In each of these areas, specialised languages have been developed to define and manipulate data structures of a form suited to that particular application. This paper presents a generalised approach to data structures, which is suitable for incorporation in a general purpose programming language such as ALGOL 60, and which may extend its application to many of the areas cited above.

The principles which have been observed in the design of record handling are conceptual economy, security and efficiency.

(1) Conceptual Economy

The most valuable feature of a programming language is that it provides the programmer with a conceptual framework which enables him to think more clearly about his problems and about effective methods for their solution, and a notational technique which enables him to express his thoughts clearly. If the fundamental concepts have been correctly formulated, there is no need to introduce complex procedural facilities into the language, since the programmer has the tools with which he may construct exactly the procedures which he requires for his particular application.

(2) Security

Another most valuable feature of a programming language is that it provides a means for constructing programs which can be proved to be free from certain kinds of programming error. This feature is particularly important in programs which make use of indexing or indirect addressing, since the consequences of incorrect use of addresses are in principle unpredictable, and in many cases calamitous. We therefore ensure that a translator of a language which incorporates record handling shall be able to check against such errors at compile time, so that they can never occur in a running program.

(3) Efficiency

Many of the problems for which record handling is required are severely limited by the speed and storage capacity of the computers which are available for their solution. It is therefore most important that the object code produced by a translator shall be as fast and as compact as if it were written by straightforward machine code programmer; and that the use of store for data should be subjected only to the barest minimum of administrative overhead. These objectives should be achievable without any loss of security, without requiring complex optimising translators, and without requiring any additions to the machine code facilities of computers at present in use.

1. Basic Concepts

1.1 Objects and Records.

A fundamental feature of our understanding of the world is that we organise our experience as a number of distinct objects (tables and chairs, bank loans and algebraic expressions, polynomials and persons, transistors and triangles, etc); and our thought, language, and actions are based on the designation, description, and manipulation of these objects, either individually or in relationship with other objects. When we wish to solve a problem on a computer, we often need to construct within the computer a model of that aspect of the real or conceptual world to which the solution of the problem will be applied. In such a model, each object of interest must be represented by some computer quantity which is accessible to the program, and which can be manipulated by it. Such a quantity is known as a record.

1.2 Attributes and Fields.

Each of the objects represented in the computer by a record will possess one or more attributes which are relevant to the solution of the problem. Such attributes can generally be indicated by simple values of some appropriate type. For example, the rate of interest on a bank loan can be indicated by a real number, the date of birth of a person can be indicated by an integer, the name of a variable by a string, etc. In order to represent in the computer a group of one or more attributes which are relevant for a given object, the record which represents that object will have allocated to it a group of one or more variables. Each variable will be capable of holding a simple value indicating the nature of some attribute of the object. A variable allocated to a record is known as a field of that record; it will often be given an identifier which is suggestive of the corresponding attribute, for example, "rate of interest", "date of birth", etc.

1.3 Collective Nouns and Record Classes.

The objects of the real world are often conveniently classified into a number of mutually exclusive classes, and each class is named by some collective noun, such as "person", "bankloan", "expression", etc. In general, all the members of a given class have the same sort of attributes, whereas attributes which are appropriate for members of one class are totally inappropriate for members of other classes. Thus it is absurd to enquire about the rate of interest of a person, or conversely, the sex of a bankloan. In the computer model, records are also grouped into mutually exclusive record classes; each record of a given class has the same number of fields as every other record of that class, and the fields have the same types and identifiers. Records of another class will often have a different number of fields, the fields will often be of different types, and they will usually have different identifiers; although sometimes it is convenient to use the same field identifier for fields of records in distinct classes.

1.4 Record Class Declarations.

In order to use records in a program, the programmer must initially declare the record classes into which his records will fall. A record class declaration introduces the identifier of the class, and gives a list of declarations of the variables which will be allocated as the fields of each record of that class. These declarations for the fields take the same form as declarations of variables in a block head; they introduce an identifier for each field, and indicate that the field is of a specified type. For example:

```
record class bankloan(integer loan number, principal;
                      real rate of interest);
record class repayment(integer loan number, amount);
```

The first declaration indicates that the program will be concerned with records belonging to a class "bankloan", and that each such record will have three fields. The first field is an integer variable giving a numeric identification of the loan, and the second field (also an integer) gives the amount of the loan outstanding, expressed in some suitable units (say, pennies). The third field is a real variable which indicates the annual rate of interest to be paid on the loan. The second declaration suggests that records of the class "repayment" represent partial or complete repayments of a bankloan. Each record of the class indicates the loan number of the loan concerned, and the amount of the repayment.

1.5 The Reference Type.

In order to enable a program to refer to records currently existing in the computer, and to distinguish between different records of the same class, a new type of quantity must be introduced into the language. This is known as the reference type, and values of this type range over addresses of records. Variables and parameters may be specified to be of type reference, and will then take values which are addresses of records of some particular class. The value of a reference variable may be set or changed by assignment in the same way as values of other variables. However, the class of record to which a reference variable may refer is fixed at the time of its declaration, and it may not be used to refer to records of any other class. The translator will check that this restriction has been observed, and will also check that the reference has not been used in a manner inappropriate to records of that class. Thus the major requirement of security can be guaranteed.

Examples:

```
reference (bankloan) master; reference (repayment) transaction;
```

The first declaration introduces a variable "master" and indicates that its range of values is confined to addresses of records of the class "bankloan"; and the other declaration introduces the variable "transaction", which will be used to refer to repayments.

1.6 Field Designators.

In order to refer uniquely to a field of a particular record, the programmer must both name the field by its identifier, and also indicate the reference variable which has as its current value the address of the intended record. Such a construction is known as a field designator, and may be written thus:

rate of interest (master), amount (transaction), etc.

A field designator may be written in a program in any context in which the language permits a variable to be written. If it appears as a primary in an expression, its value is the current value of the field. If it appears on the left hand side of an assignment, its value will be altered to that of the right hand side, thus:

principal (master) := principal (master) + interest;

1.7 Example of Basic Concepts.

```
begin comment  this block accomplishes a monthly updating of a
               bankloan record referred to as master, taking into
               account (if relevant) a repayment record referred to
               as transaction.  If the loan number of the transaction
               is not the same as that of the master, the transaction
               is ignored, this month's interest is added to the
               principal, and exit is made to the label "unequals".
               If the amount of the repayment is more than that
               required to repay the loan, the principal is set to
               zero, and a refund notice is printed out.

                    The program has been adapted from an example
               presented in "APL/I Primer" IBM C28-6808-0;

    integer interest, refund;
    interest := entier (principal (master) x
                        rate of interest (master)/12);
    if loan number (master)≠ loan number (transaction)
        then begin
            principal (master):= principal (master) + interest;
            go to unequals
            end
    else if amount (transaction) ≤principal (master) + interest
            then principal (master) :=principal (master)
                        + interest
                        - amount (transaction)
        else begin
            refund := - principal (master) - interest
                        + amount (transaction);
            principal (master) := 0;
            print (loan number (master), "REFUND", refund)
            end overpaid
end update;
```

1.8 Implementation of Basic Features.

A word is defined here as the smallest conveniently addressable unit of storage; on a "character" machine, it may be a single character.

A record is implemented as a group of one or more consecutive words in a computer store. This group of words is divided up among the fields of the record in such a way that each field is allocated sufficient storage space to hold any value of its declared type. The compiler may be permitted consistently to rearrange the sequence of fields in a record so that they fit conveniently with the word boundaries of the computer. The storage used by a record is equal to the sum of the storage required by its fields. The names of the fields of a record are not normally preserved at run time, and there is no administrative overhead. The number of words preceding a given field in a record is known as the offset of the field.

A reference value is implemented as the machine address of the first word of the record it refers to. A reference variable is allocated sufficient storage space to hold any such address.

A field designator is translated as two instructions. The first loads the content of the specified reference variable into a modifier-register, and the second is a modified instruction specifying the offset of the field within the record concerned. The translator also checks that the field identifier is appropriate for the record class which was associated with the reference variable in its declaration.

2. Reference Fields and Record Creation.

2.1 Functional Relationships and References.

In addition to possessing attributes, the objects of the real world may also bear certain relationships to each other. For example, one person may be the father of another person, a town may be the destination of a certain road, a subexpression may be the left operand of a given expression, etc. The programmer will sometimes want to represent such relationships in the computer model which he is constructing and manipulating. Now, a relationship is said to be a functional relationship if it is a many-to-one or one-to-one relationship; that is, if for any object x there is at most one object y such that x bears the relationship to y. Examples of functional relationships are "father of", "youngest offspring of",etc. It happens that functional relationships are particularly easy to represent in a computer by addressing or referencing techniques identical to those already described for the identification of records.

For each functional relationship which an object of a given class may bear to another object, the programmer should declare a suitably named field of type reference. In each record of this class, he should assign to this field the address of the record with which the relationship holds. For example:

```
record class person (integer date of birth; Boolean male;
                     reference (person) father, elder sibling, youngest
                                                               offspring);
reference (person) T, J, K;
```

The record class declaration introduces the record class "person", which has records with five fields. The first indicates the date of birth, and the second the sex of the person. The last three fields will be used to hold references to other persons who are the father, elder brother or sister, and youngest son or daughter of the person concerned. For example, to indicate that the record referenced by T represents the person who is the father of the person represented by the record referenced by J, the programmer may make the assignment:

```
father(J) := T;
```

2.2 Partial Functional Relationships.

Functional relationships may be classified as either partial or total. A total functional relationship is one which satisfies the condition that for every x in the class which defines its domain, there is exactly one y in its range to which x bears the relationship. For a partial functional relationship, there may be an x for which there is no y appropriately related to it. All the relationships introduced above for persons are in fact partial, since there will be persons who have no offspring, and persons who have no elder brother or sister; and in any finite collection of persons, there must be at least one who has no father. In order to meet this problem, a special null value is provided for reference variables and fields. If a field of a record is given this value, it usually indicates that the functional relationship represented by that field is not defined (or not yet defined) for that record; and that it is therefore a partial rather than total functional relationship. The null reference value is denoted by the basic symbol null.

2. 3 Record Creation.

The only remaining feature which is required for the full use of record handling is a method for bringing records into existence in the computer. This is not accomplished by declaration, as for other quantities of the language, but can occur at any point in the execution of a program. In fact, for every record class, there is a record creating function, which calls into existence a new record of that class, and delivers as its value a reference to the newly created record. This function takes a list of actual parameters, each of which corresponds to one of the fields of the record. Each actual parameter is an expression which defines a value to be initially assigned to the corresponding field of the new record. It is convenient to use the record class identifier itself as the name of the record creating function. For example:

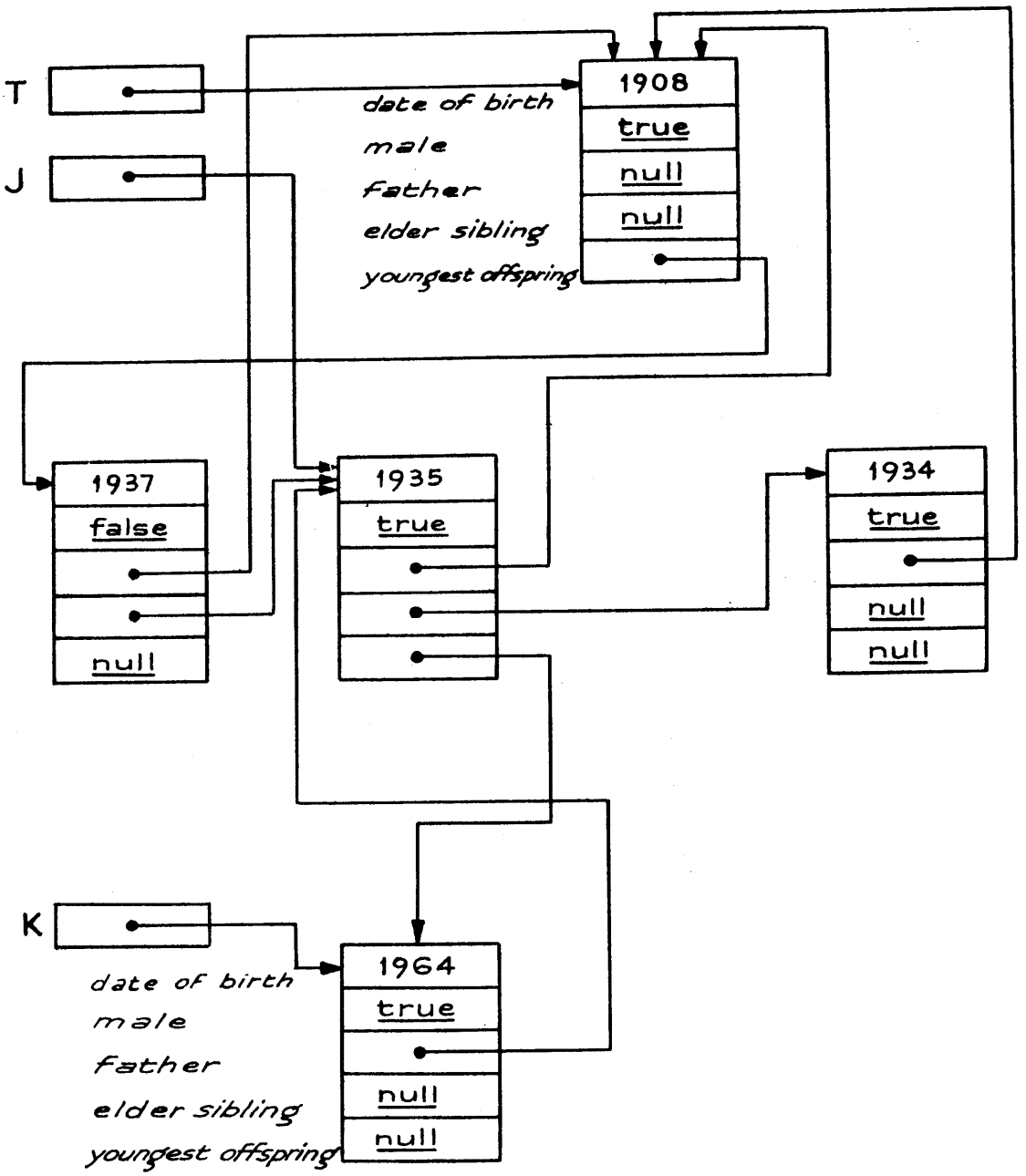
```
T:= person (1908, true, null, null, null);
youngest offspring(T) := person (1934, true, T, null, null);
youngest offspring(T) := J :=
        person (1935, true, T, youngest offspring (T), null);
youngest offspring (T) := person (1937, false, T, J, null);
K := youngest offspring (J) := person (1964, true, J, null, null);
```

After execution of the first of these statements, the variable T refers to a man born in 1908, who does not have an elder sibling or a younger offspring, and whose father is not represented by a record. The next three statements represent the birth of three children to this man; the first is a son born in 1934, who does not have an elder sibling or a youngest offspring, but whose father, of course, is T. The second is another son, born in 1935. His father is T, his elder sibling is T's previous youngest offspring, and he has (initially) no children. In 1937, the first and only daughter is born. The final statement represents the birth of a son to J, who was the second son of T. This person is also referred to as K.

2. 4 Pictorial Representation.

It is often illuminating to construct a pictorial representation of the records which may exist at some time inside the computer. A record can conveniently be drawn as a box, and each field can be drawn as a compartment of the box. For non-reference fields, the current value of the field may be written in this compartment; and for reference fields, an arrow should be drawn from the compartment to the record to which that field currently refers. If desired, the identifier for a variable or a field may be written to the left of the compartment.

In this way, the situation in a computer after execution of the five assignments of 2. 3 may be pictured thus:


T
J
1908
true
null
null
date of birth
male
father
elder sibling
youngest offspring
1937
false
null
1935
true
1934
true
null
null
K
1964
true
null
null
date of birth
male
father
elder sibling
youngest offspring

2.5 Example.

```
reference (person) procedure youngest paternal uncle (R);
    value R; reference (person) R;
begin comment this procedure yields as its result a reference to the
      youngest paternal uncle of the person referred to as R, if he
      has one: otherwise it yields null;
    reference (person) S;
    S := youngest offspring (father(father(R)));
repeat: if S ≠ null then
            begin if S=father (R)∨ ¬male (S) then
                begin S:=elder sibling (S);
                    go to repeat
                end
            end;
    youngest paternal uncle:=S;
end youngest paternal uncle;
```

Example of use of this procedure:

```
I:= date of birth (youngest paternal uncle (K));
```

2.6 Representation of the null Reference.

The null reference value should be represented in the computer by an integer value which is outside the range of addresses of the storage area allocated for holding records. It is highly desirable that, on a computer with any form of storage protection, it should be the address of the beginning of the protected area. This would mean that if a null reference is accidentally used by a field designator at run time, the error will be immediately detected by hardware, and there is no risk of unpredictable or calamitous results.

If a computer does not possess hardware storage protection of any kind, it may still be possible to choose a value outside the address range of the computer, so that an attempt to use a null reference incorrectly cannot corrupt the store, and may even given consistent results. However, on computers with cyclic store addressing and no protected area, it would be advisable to set aside a pseudo-protected area of store, to take its address as the null reference value, and to fill the entire area with null references. This means that any field designator occurring in an expression will always give uniform results, and run-time checking can be confined to the case of assignment to field designators.

2.7 Dynamic Storage Allocation

In contrast to the local variables of a block, the records created in a block generally remain in existence after exit from the block in which they were created. In general, they are only deleted on exit from the block in which the record class to which they belong was declared. However, at the time of declaration of a record class, it is not known how many records of that class are going to be called into existence during the lifetime of the class. It is not therefore possible to allocate space for records as part of the dynamic stack of local workspace which is a common feature of ALGOL implementations. Space for records must be allocated at the time when the record is created, and should therefore be taken from the other end of store to the dynamic stack.

On exit from the block in which a record class is declared, the space occupied by records of that class may be reclaimed and used for other purposes. Unfortunately, this space will not necessarily form a contiguous free area of store, but will be split into a number of holes of varying sizes, each of them separated from its neighbours by records which are still in existence. These holes should be chained together in a list, and whenever a new record is created, its space should be allocated from the smallest hole big enough to hold it. Whenever further storage space being freed happens to be contiguous with a hole, it is advisable to treat the two free areas as amalgamated into one larger hole.

If at any time it becomes impossible to satisfy a request for storage, it may still be possible to find additional free store by detecting that certain of the records in the record area have become inaccessible to the program, by virtue of the fact that there exists no accessible reference to them. This can occur both as a result of assigning new values to the reference variables which originally pointed to them, and also as a result of such variables going out of existence on block exit. The method of detecting such inaccessible records is known as "garbage collection", and the techniques can be readily adapted from those used in LISP [1]. They depend on the fact that the implementation knows the position of every possible reference value in the store.

Since the available free storage is split into a number of disjoint holes, it may happen that the total space available is sufficient to satisfy a request, but none of the holes is large enough to satisfy the requirement that the storage allocated to records must be contiguous. In such circumstances, it is still possible to retrieve the situation by copying all the records in existence to one end of the store, leaving the whole of the free area contiguous at the other end. Of course, all the references in store must also be updated to point to the new positions of the records instead of the old. This process is known as a "store collapse". It is rather time-consuming, and should probably be used only as a last resort.

3. Record Subclasses and Their Discrimination

In the real world, it is often useful to consider a class of object as being split into a number of mutually exclusive subclasses. For example, the class of algebraic expressions will include constants, variables and dyadic expressions consisting of a pair of subexpressions connected by an operator. Dyadic expressions (pairs) will themselves be subclassified as sums, differences, products and quotients.

3.1 Subclasses.

In order to represent this situation in a computer model, a programmer may include in his declaration of a record class a list of identifiers for the subclasses of which it consists. For example, the programmer may wish to introduce a record class to cover dyadic expressions. Each record will contain two reference fields to point to the two operands; and since we will later be concerned with symbolic differentiation, we introduce a field to point to the derivative of the expression represented by the record. The four subclasses into which the whole class is split are also declared, thus:

```
record class pair (reference (expression) left operand,
                                          right operand,
                                            derivative;

                   subclasses sum, difference, product, quotient);
```

A record class which is split into subclasses is known as composite; and a class or subclass which is not composite is said to be simple.

Reference variables may be declared as pointing to records of a composite class, or alternatively their range may be restricted to a particular one of the subclasses:

```
reference (pair) p, q;   reference (difference) d;
```

These declarations permit the variables p and q to point to any pair, whereas d may point only to those pairs which belong to the "difference" subclass. Any of the references may be used in field designators:

```
derivative (q), left operand (d), etc.
```

The specification of the subclass to which a record will belong is made at the time when the record is created; this is achieved by using the name of the appropriate simple subclass as the record creating function:

```
d := difference(p, q, null);
p := quotient (sum (p, q, null), d, null);
```

Identifiers for composite record classes can never be used for creation of records.

In many cases, the members of one of the particular subclasses of a composite class may possess some attribute which is not relevant for the members of the other subclasses. For example, an expression which is a pair requires a field to point to its left operand, whereas for an expression which is simply a variable or a constant, this field would be irrevelant. This situation is dealt with by allowing records of a subclass to contain fields which are peculiar to that subclass, and not shared by other subclasses. Such fields are known as private fields, and should be declared (in brackets) following the identifier of the relevant subclass; so that a record subclass declaration may take the same form as a complete record class declaration. If desired, a subclass declaration may itself be declared as a composite class, consisting of several private subclasses; in other words, the subclass mechanism may be invoked recursively to any depth.

3.2 Example.

In an application involving algebraic expressions, the class of expressions must be divided into three subclasses, constants, variables, and pairs. The constants have a single private field which indicates the value of the constant. Variables have a single private field which gives the name by which that variable will be recognised outside the computer; the nature of the "pair" subclass has been discussed above.

```
record class expression (subclasses constant (real value),
                                 variable (string printname),
pair (reference (expression) left operand, right operand, derivative;
      subclasses sum, difference, product, quotient));
```

The following declarations define variables which will be used to point to expressions and their various subclasses:

```
reference (expression) u, v, e;  reference (constant) one, zero;
      reference (variable) x, y, z;
```

As an example of the method of setting up expressions inside the computer, we take:

$$\underline{z^2 + zy - 2.7 - y^3 + 2z}$$

$$z^2 + zy - 2.7 + y^3 - 2z$$

We first note that this expression can be simplified by the introduction of auxiliary definitions for common subexpressions, thus:

$\underline{u - v}$ where $u = z^2 + zy - 2.7$

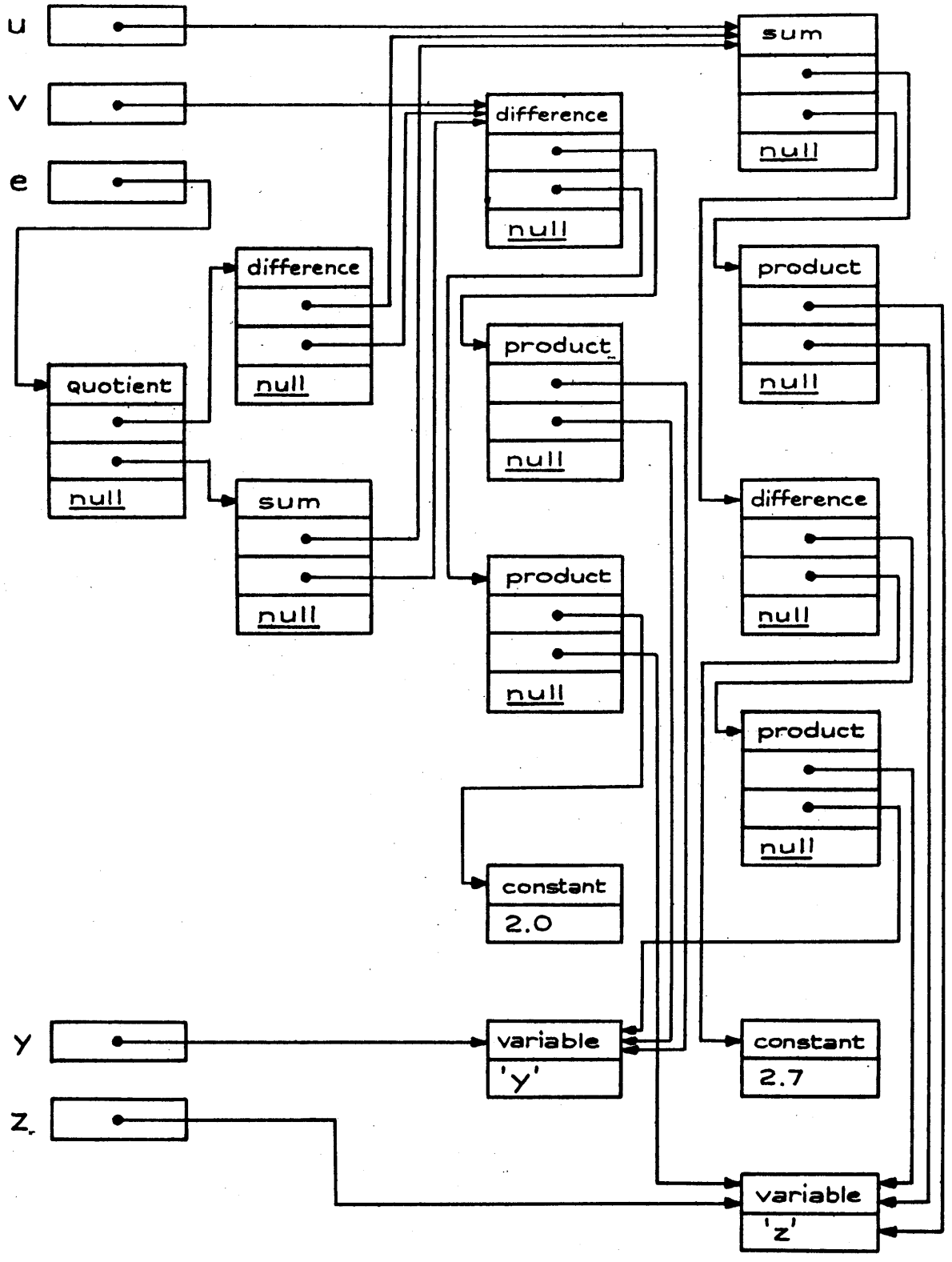
$u + v$ and $v = y^3 - 2z$.

We wish to take advantage of this analysis both to save storage space in the computer representation of the expression, and to save time in processing the expression and its derivatives. This can be achieved by the following five statements:

```
z := variable ('z');                y := variable ('y');
u := sum (product(z, z, null), difference(product(z, y, null),
                                        constant(2.7), null), null);
v := difference (product(y, y, null), product (constant(2), z, null), null);
e := quotient (difference (u, v, null), sum(u, v, null), null);
```

3. 3 Pictorial Representation.

In the pictorial representation of the result of executing these five statements, an additional compartment has been added to each record; it contains the identifier for the simple subclass to which that record belongs.



3.4 Record Class Discrimination.

When a variable has been associated with a composite class, the programmer will usually at some stage wish to determine to which of the possible subclasses the record currently referenced by that variable actually belongs. This can be achieved by a construction known as a record class discriminator, which is a statement or an expression modelled on the conditional (or case) construction. It takes the form

consider e when constant then .....
when variable then .....
when pair then .....

When this construction is executed, a test is made of the actual subclass to which the record currently referenced by e belongs, and in accordance with this test, exactly one of the statements or expressions following a then is selected for execution or evaluation. If the record belongs to the subclass "constant", the statement or expression after the first then is selected; if it belongs to the subclass "variable", then the statement or expression following the second then is selected; or if it belongs to the subclass "pair", the text following the third and last then is selected.

The result of using a construction of this form is that whichever of the alternative sections of program is selected at run time, the translator knows at compile time which subclass of record is actually referenced by the variable e at the time when execution of that section of program is initiated. Thus within each of the sections of program, the variable e may safely be used in field designators for private fields of the relevant subclass, exactly as if it had been restricted by declaration to point only to records of that subclass.

3.5 Example.

```
reference (expression) procedure assocderiv(e) with respect to:(x);
    value e, x; reference (expression) e; reference (variable) x;
begin comment this procedure assumes initially that the derivative field of
    each component pair of e either takes the value null, or contains a
    reference to the derivative of that pair. If e is a pair, then on exit
    from the procedure its derivative field contains a reference to its
    derivative, and the derivative fields of all its component pairs also
    contain references to their derivatives. The result of the procedure
    is the derivative of e. The procedure uses non-local quantities "one"
    and "zero", which are assumed to contain references to the records
    "constant (1)" and "constant (0)" respectively;
```

```
consider e when variable then assocderiv:= if e=x then one else zero
          when constant then assocderiv := zero
          when pair then begin comment compound statement for pair;
               if derivative (e) ≠ null then assocderiv:= derivative(e)
                                        else
     begin reference (expression) u, udash, v, vdash;
          u := left operand (e);
          v := right operand (e);
          udash := assocderiv (u, x);
          vdash := assocderiv (v, x);
          assocderiv := derivative (e) :=
     consider e
     when sum then (if udash = zero then vdash
               else if vdash = zero then udash
                                    else sum (udash, vdash, null))
     when difference then (if vdash = zero then udash
                           else difference (udash, vdash, null))
     when product then
          (if udash = zero then
              (if vdash = zero then zero
          else if vdash = one then u
                               else product (u, vdash, null))
          else if vdash = zero then
              (if udash = one then v
                               else product (v, udash, null))
                         else
          sum (if udash = one then v else product(v, udash, null),
               if vdash = one then u else product(u, vdash, null),
               null))
     when quotient then
               (if vdash = zero then quotient (udash, v, null)
          else  if vdash = one then
                         quotient (difference(udash, e, null), v, null)
                               else
          quotient (difference(udash, product(e, vdash, null), null),
                    v, null))
     end differentiation of pair
          end compound statement for pair
end assocderiv;
```

Example of use of this procedure on values of e, y, and z, which have been defined in 3. 2:

```
begin reference (expression) array Dy [0:15 ]; integer i;
      Dy[0 ] := e;
      for i   := 1 step 1 until 15 do
            Dy[i ] := assocderiv (Dy[i- 1 ], y);
      . . . .
```

3. 6 Implementation.

Records of a composite class must each be allocated an initial hidden field; when a record of the composite class is created, this field must be set to contain a small integer value, indicating which of the particular subclasses this record belongs to. The initial hidden field should be immediately followed by those fields which are common to all the subclasses, so that the offset of these common fields is the same for all records of the class, and field designators can correctly access the fields, no matter which subclass the record belongs to. Fields which are private to a subclass come last in the record. These rules can be applied again if any of the subclasses themselves have subclasses.

Fields of type string are often required to hold the name of the object which is represented by a record. For many applications it will be adequate if the implementation merely allocates to such fields sufficient space to hold six characters or so, depending on what fits easily within the word boundary of the machine.

A record class discriminator is translated into a section of machine code with the following effect:

1. The first instruction accesses the initial hidden field of the record, and adds its contents into the "control counter" (i. e. the register which contains the address of the instruction being executed). This achieves the effect of jumping forward to the nth following instruction of the program, where n is the value of the hidden field.

2. The "jump forward" instruction is followed by a sequence of m jump instructions, where m is the number of subclasses of the given class. The nth jump of this sequence has as its destination that section of machine code which is to be selected for execution if the actual record belongs to the nth of the possible subclasses.

3. Each section of program is translated as if the variable discriminated had been associated by declaration with the relevant subclass; the code for each section is terminated by a jump forward to the end of the record class discriminator.

4. Input and output of records.

In many applications of record handling, the amount of information which has to be dealt with is many times greater than can be accommodated in the main (core) store of a computer. Furthermore, there are circumstances in which the information embodied in records must be preserved over a period of some days or even months or years, while the main store of the computer is being used for other tasks. These considerations are particularly relevent for business-oriented data processing.

In such applications, it is common to use an external input/output medium for the output and storage of records until such occasion as it is convenient to input them. Since records which are output are intended solely for reinput to the computer, there is no need to transform the information into a notation which is intelligible to the human being, or to present it in a documentary form which he can read. Thus it is customary to use a magnetic medium for input and output, and to transfer information between the medium and the main store of the computer in exactly the same form as it is held there for normal processing, without converting it, for example, from binary to decimal.

The media which are currently in use for input and output of records are:

(1) Magnetic Tapes
(2) Magnetic Drums
(3) Magnetic Discs
(4) Magnetic Cards

Each of these devices has certain particular characteristics, but their similarities are such that it is possible to define a single concept of record input/output, and construct an appropriate set of processing facilities to cover all four cases.

4.1 Files

A sequence of records held together on an external medium is known as a file. All records of a file should belong to a single record class, and a summary of the declaration of the class should accompany the file, so that a check can be made that the records are not being used inappropriately when they are input. A file should also be accompanied by a title, by which the physical medium can be identified externally by programmers and machine operators. This title should be recorded on the medium, so that the computer can check that the expected medium has in fact been mounted. Thus a typical file will consist of a title, followed by a copy of the essential features of a record class declaration, followed by a (possibly empty) sequence of records.

There is one most important restriction on records held in a file on an external medium; that they must not have any reference fields. The reasons for this restriction are:

(1) It is impractical for the garbage collection mechanism to scan and update all reference fields of records held in files;

(2) If the file is preserved and presented as input to another program, the mainstore records will no longer be available to the new program, and the reference will be invalid.

If a program requires an existing file to be mounted on some channel for its records to be input, the file is known as an input file. If an entirely new file is to be created by successive output of records, and this file is to be preserved on completion, then the file is known as an output file. A new file which is to be used for output and reinput within the same run of a program, and does not have to be either mounted at the beginning or preserved at the end, is known as a scratch file. Finally, a file which has to be both mounted and preserved, and which is used for both input and output, is known as an update file. An update file has the property that input of a new record causes the (possibly changed) value of the previous record input to be rewritten to the file in the same position from which it came.

For any file in use, there is a file pointer which points to the position which has currently been reached in the file. This may be thought of as the current reading or writing position in the file, for example, in the case of magnetic tapes, the position of the read/write heads. Initially, this is placed at the beginning of the file, ahead of all the records. Each input or output operation moves the file pointer to just beyond the record which has been input or output. Input operations are possible only when the file pointer is not at the end of the file, and output operations are, in general, only possible when the file pointer is at the end of the file.

4.2 File Declarations.

Files are declared in the same way as other quantities of the program. A file declaration introduces an identifier for the file, and specifies its major characteristics. On entry to the block in which a file is declared, the identifier is associated with a physical medium, which in the case of an input or update file, already contains a non-null sequence of records. On exit from the block, this association is broken, and (except in the case of a scratch file) the sequence of records is preserved, and may be later reassociated with another file identifier.

Thus to indicate that a file identifier "transaction" is to be associated with an existing file of records of class "repayment", that this file has the title "REPAYMENTS 19/6/65", and that it is to be found on channel 1, the following declaration would be written:

```
input file (repayment, 'REPAYMENTS 19/6/65', 1) transaction;
```

Declarations of output and update files take the same form, with the exception of the first word:

```
output file (statement, 'STATEMENTS 21/6/65', 4)L;
update file (bankloan, 'BANKLOANS TR/1769B', 3) master;
```

4.3 File Operations

In order to specify that the next record of a file is to be input, a standard procedure is provided which takes as its parameter the name of the file:

```
inrecord (transaction);
```

For output, there is provided a standard procedure with two parameters,

```
outrecord (L, S);
```

which outputs on file L a copy of the record referenced by the expression S.

In order to refer to the fields of the record which has been most recently input on a given file, and to assign new values to fields of a record which will be rewritten to an update file, it is permitted to use the file identifier itself in a field designator as if it were a reference variable, thus:

```
principal (master), loan number (transaction), etc.
```

However, it is not permitted to assign new values to a file identifier, except by input of a new record from the file. Moreover, it is not permitted to assign the value of the file variable to other file variables or to reference variables.

If the most recent operation on a file was not an input operation, or if on input it is found that there are no more records on the file, then the file identifier takes the value null, and its use in field designators is invalid. This circumstance can be detected by a test, thus:

```
if transaction = null then ....
```

4.4 Example.

```
begin comment this program is similar to the example of 1.7;
      record class bankloan(integer loan number, principal;
                            real rate of interest);
      record class repayment (integer loan number, amount);
      record class statement (integer loan number, principal;
                              real rate of interest;
                              integer payment, balance);
      integer interest, balance, refund;
      output file (statement, 'STATEMENTS 21/6/65', 4) L;
      input file (repayment, 'REPAYMENTS 19/6/65', 1) transaction;
      update file (bankloan, 'BANKLOANS TR/17694B', 3) master;
next trans: inrecord (transaction);
next master: inrecord (master);
      if master = null then go to complete;
interest:=entier(principal(master)x rate of interest(master)/12);
if if transaction = null then true else loan number (master)
                                    < loan number(transaction)
      then begin principal(master):=principal(master)+interest;
                 goto next master
           end
else if amount (transaction)≠principal(master) + interest
      then balance:=principal(master)+ interest - amount(transaction)
      else begin balance:=0;
                 refund:=-principal(master) - interest
                                          + amount(transaction);
                 print(loan number(master), 'REFUND', refund)
           end overpaid;
      outrecord(L, statement(loan number(master), principal(master),
                             rate of interest(master),
                             amount (transaction),
                             balance));
      principal(master):=balance;
      go to next trans;
complete: end;
```

4.5 File Positioning Procedures

The operations of input and output defined for files are sufficient for the serial access of records, which, for reasons of efficiency, must be regarded as the standard technique for file processing. However, there are occasions when it is desired to reposition the file pointer, so as to go back and re-examine some earlier records, or to go on to some later point without examining the intervening records. For such purposes, certain file positioning procedures are provided.

In order to reposition the file pointer back to the beginning of the file, i. e. just ahead of the first record, we have the procedure

rewind(F);

where F is the relevant file name. Similarly, in order to position the file at the end, just beyond the last record, we have the procedure

wind on (F);

This procedure is particularly useful for resetting the file in the position where further records may be output to it. If it is desired to write new records at the current position of the file rather than at its end, it is necessary first to delete all records of the file subsequent to the current position. This may be done by the procedure:

truncate (F);

In order to delete a file, and make the medium on which its recorded available for other purposes, there is a procedure:

delete file (F);

If it is required to exercise a more precise control over the positioning of a file at some point in its middle, the procedures "position" and "reposition" may be used. The integer-valued function designator

position (F)

may be used to yield an integer representing the current position of the file pointer of the file F. The procedure statement

reposition (F, P);

will reposition the file pointer of F to the point represented by the integer value of the parameter P.

4. 6 Implementation on Magnetic Tape

The implementation of file processing on magnetic tape is fairly straightforward, except when more than one file is held on a single reel, or a single file stretches over several reels. A reel of magnetic tape has initially recorded on it the title of the file which it contains, followed by an abbreviated summary of the record class declaration, in which the actual identifiers for the fields and subclasses are omitted, but their type, number and structure are preserved. This information is written when the file is first created as an output file, and is checked whenever it is mounted as an input file. Then follows the set of records, which are stored one next to the other, without any intervening information. Finally, after the last record, there is written a specially identifiable tape mark to indicate the end of the file; this mark is not actually recorded if the last operation on the file is an output or truncation operation; but it is recorded whenever a non-output operation follows an output one, and on exit from the block in which the file was declared.

This picture is slightly complicated by the fact that records cannot be written continuously along the magnetic tape, but must be blocked together into physical blocks, each separated from its successor by an inter-block gap. For reasons of efficiency these blocks should be fairly long, capable of containing many records. Each physical input or output operation must involve transfer of a complete physical block, so that the read-write heads always come to rest in an inter-block gap. This complication can be dealt with by allocating to each file variable a buffer area in core store, which is large enough to hold the physical block, and by maintaining in the computer a simulated file pointer pointing to the next record of the block to be input, or the next position at which a record may be output. Whenever the program specifies an input or output operation, this pointer is merely stepped on within the buffer, and a physical input or output operation is initiated only when the buffer is exhausted.

Note that on the most widely used design of magnetic tape, the recording techniques make it impossible to rewrite a record to the middle of a file. Files held on such a tape may not be specified as update.

4.7 Implemenatation on Random Access Stores

The most important differences between magnetic tapes and random access store are:

(1) A single random access store can be used to hold many files.

(2) It is possible to rewrite a record in the middle of a file on random access store.

(3) The time taken to reposition a random access file is less than for magnetic tape; but it is still far from negligible.

(4) The volume of some stores is so great that files can be left almost permanently on the store, and are never physically removed.

Of these differences, numbers (1) and (4) involve the greatest differences of implementation.

The first requirement for sharing a store among several files is that the titles of each of the files shall be recorded together in a file table, which indicates for each file, the position at which the first block of records of the file is stored. Each block of records contains an indication of the position of the next block of records of that file.

A bit pattern, with one position per block, is used to indicate which blocks of the random access store are currently in use, and whenever a new block is allocated to a file, the corresponding bit is set to one. For purposes of allocation of storage on the medium, it may be convenient to take a track of store as the unit, rather than a physical block.

5. Miscellaneous Extensions

The extensions described in this section are not essential, but they widen the range of applications to which record handling can conveniently and efficiently be applied.

5.1 Array Fields.

Just as some attributes of an object can be conveniently represented by means of a simple field, there are occasions when attributes can more conveniently be represented by an array of fields of identical type. For example, a plane polygon may be considered as possessing an area and a colour, which may be represented as a simple real and a simple integer field. It may also be characterised by the lengths of its sides, and the angles which they form. These may conveniently be represented by real arrays, each possessing a number of elements corresponding to the number of sides of the polygon.

Such a record class may be declared thus:

```
record class polygon(real area; integer colour;
                     real array angle, length of side);
```

In order to refer to a particular element of an array field, the field identifier for that array should be followed by the appropriate subscript, thus:

```
x:= length of side (P) [i];
angle(Q) [j]:= 3.7 x arctan(x/2) +7;
```

Whenever a new record containing array fields is created, the structure, size, and initial values of the array fields are given by writing an actual array parameter in the corresponding position of the parameter list of the record creating function.

For example, the following section of program assigns to P a reference to a new polygon record representing a blue unit square:

```
begin real array A, S [1:4]; integer i;
      for i:=1 step 1 until 4 do
            begin A[i]:= pi/2;
                  S[i]:= 1
            end;
      P:= polygon(1, blue, A, S)
end;
```

To enable the programmer to determine the actual dimensions of an array field for a particular record, the integer standard procedures "lowerbound" and "upperbound" must be provided; they yield respectively the lower and upper subscript bounds of the array specified as their parameter, thus:

```
J:= upperbound(length of side (P));
```

As an example of the use of array fields, the following procedure computes the circumference of a polygon of any number of sides:

```
real procedure circumference (Q);
      value Q; reference(polygon) Q;
      begin integer i; real sum;
            sum:=0;
            for i:= 1 step 1 until upperbound (length of side (Q)) do
                  sum:= sum + length of side (Q) [i];
            circumference:= sum
      end;
```

5.2 Packed Fields.

In many applications of record handling, it is desired to fit very large numbers of records in a limited amount of main storage. Even for records held in files, a reduction in the space occupied by each record can lead to a great increase in the efficiency of processing them. A fruitful technique for reducing the size of a record is to pack several fields into the space which might normally be allocated to a single field. This is particularly applicable to integer fields which are known to take values in only a small part of the range which the implementation makes available for normal integers.

In order to make such packing possible, the programmer must declare the range of possible values of an integer field. For example, in an application involving playing cards, fields will be required to represent the suit and the rank of each card. The suit field may take only the values 0, 1, 2, and 3, indicating that the card is a club, a diamond, a heart or a spade. The rank field takes values 1 to 13 only, with 1 indicating an ace, and 13 a king. Such a record class may be declared thus:

```
record class card(integer [0:3] suit; integer [1:13] rank);
```

Packed fields may be used in a program in exactly the same way as other fields; it is the duty of the translator to insert the extra instructions to unpack the information when it is accessed, and to pack up the new value on assignment to it. For example, the following procedure determines whether card A is capable of taking a trick in whist or bridge, where card B is the winning card so far. It assumes that the non-local integer "trumps" holds the value of the current trump suit.

```
Boolean procedure takes (A, B);
      value A, B; reference (card) A, B;
   takes:= if suit(A) ≠ suit(B) then suit(A) = trumps
         else rank(A) > rank(B) ∨ rank(A) = 1;
```

Similar requirements for packing of information arise when a record has a field whose value is a string of characters. In previous examples, we have simply used the field declaration "string name" for declaring such fields, and it is assumed that the implementation will allocate a fixed amount of storage to such fields, sufficient, say, for six, seven, or eight characters. In practice, there are many occasions when less characters are required, and some when many more are necessary. In order to give the programmer some control over the number of character positions allocated to a string field in a record, it is necessary to permit the programmer to specify exactly the required number of characters as an integer constant, thus:

string [2] coding; string [60] name and address;

5.3 File Arrays.

In certain types of computer application, the programmer knows that he will frequently have to scan the records of a file to retrieve a record or records whose fields possess particular values or combinations of values. Since the number of records in a file is often very large, this scan will be very timeconsuming, and the programmer will wish to reduce it. This can be done by splitting the records of the file into subfiles in such a way that on each occasion when a scan is made, only the records of a single subfile need be scanned. This obviously reduces the time taken by a factor equal to the number of subfiles.

One of the most important applications of subfiles arises when a program has to select a record in accordance with the value of one of its fields, for example, the value of the loan number field of a bankloan record. The technique of searching through a whole file to find a record with given loan number would be prohibitively expensive. The solution is to split the records of the file into mutually exclusive subfiles, in such a way that one may compute, from any proposed loan number, which of the subfiles will contain a record with that loan number. A common technique is to reduce the loan number modulo the number of subfiles, and to use the result to select the particular subfile in which the record with that loan number will be placed.

To specify that a file is to be split into subfiles, the file identifier should be specified as an array, thus:

file (bankloan, 'BANKLOANS', 4) array master [0:127];

This declaration indicates that the "master" file is to consist of 128 subfiles, numbered 0 to 127.

The operations provided for array files are the same as for ordinary files, except that every occurrence of the array file name must be subscripted, to indicate which of the subfiles is intended:

```
principal (master[j])
outrecord (master [target - 128 ×(target ÷ 128) ],
          bankloan (target, a, 8.75));
rewind    (master[k]);
```

The following section of program will input a record with loan number equal to "target", if there is one; this record may subsequently be referred to as "master[k]". If there is no such record then "master[k]" will be null, and the kth subfile will be wound on to the end.

```
begin k:= target - 128 × (target ÷ 128);
      rewind (master[k]);
nextrecord: inrecord (master[k]);
      if master [k] ≠ null then
            begin if loan number(master[k]) ≠ target
                  then go to nextrecord end;
end;
```

This technique for locating records from a file at random is a very important one for certain types of business-oriented applications, particularly where on-line enquiries have to be dealt with. Another very important application of array files is in constructing algorithms for sorting records into sequence. The implementation of array file in general requires a random access backing store.

5.4 Input and Output of Groups of Records.

The facilities for input and output of records in files are based on four restrictions:

(1) Each input or output operation conceptually involves transmission of exactly one record.

(2) At most one record of the file (or subfile) is accessible to the program.

(3) No record on a file may contain a reference field.

(4) No reference variable may point to a record on a file.

These restrictions are widely observed and quite acceptable in business-oriented data processing.

In other applications, however, these restrictions would preclude the use of backing store for holding information which is too voluminous to be accommodated simultaneously in main store. For example, in the field of engineering design automation, a complete picture of an engineering product has to be built up inside the computer. The complete picture is constituted of a large number of subpictures, each of which portrays one of the components, or possible subassemblies of the final product. Each subpicture is represented by a set of records which are linked together by a very dense network of references; however, references between records of one subpicture and another are minimal, and could well be dispensed with. In general, the program only works with one or two subpictures at a time, and when work is completed on one of the subpictures, it can be relegated to backing store to make room in main store for other subpictures. Furthermore, it is usually desirable to keep such a permanent record of a completed subpicture, so that it can be referred to repeatedly on subsequent occasions.

What is therefore required is the extension of the concept of the file so that each of the restrictions listed above may be relaxed:

(1) the unit of input or output is no longer just a single record, but rather a whole group of records, for example, the group of records representing a complete subpicture.

(2) The programmer should have access at any time to a whole group of records, and must be able to create new records in a group while it is accessible.

(3) The records of the group may possess reference fields, provided that these point only to other records of the same group.

(4) Certain reference variables may be used to point to records within the currently accessible group; but such reference variables must be listed in the declaration of the file.

If a file is to be used for input and output of groups, the file declaration must be supplemented by an indication of the maximum size of the group (measured in some unit defined by the implementation), and by a list of identifiers which are to be used as reference variables to point to records of the currently accessible group. Thus, the declaration for the file might be:

```
input group [10 000] file (component, 'CRANE SUBASSEMBLIES', 7) Q (m, v, 1, r);
```

This declaration signifies that the input file "Q" is to be used for groups of information, and that the reference variables m, v, 1, and r, are going to be used to refer to records of the currently accessible group. Ten thousand locations of store are to be allocated to each group.

In order to output the currently accessible group, assuming that the file pointer is at the end of the file, one may use the procedure:

```
outgroup (Q);
```

This writes to the file all the accessible records of the group, together with an indication of which records are currently referred to by the variables m, v, l, and r. These variables are set to null, and the currently accessible group of records is made empty. A procedure:

```
delete group (Q);
```

has the same effect of emptying the group, but no physical transmission of information takes place.

In order to create a new record in the current group, the record creator should be used in conjunction with the name of the file, thus:

```
augment (Q, component (57076, m, v, 0));
```

The procedure "augment" ensures that the new record is created within the specified group. It should be used as a function designator, delivering as its value a reference to the newly created record of the group. This reference may be assigned only to one of the reference variables associated with the group file Q, or to a reference field of a record of group Q; any other assignment would violate the rule that references are not allowed to point from outside into a group. This rule can be enforced by a compile-time check.

In order to input the next group from a group file, the procedure to be used is:

```
ingroup (Q);
```

This inputs the group of records, and sets the values of the reference variables m, v, l, r to point to the same records to which they pointed when the group was output.

Apart from the facilities for input/output and creation of records in groups, all the positioning procedures for ordinary files are also available for group files.

5.5 Overloading.

The record class declaration provides a suitable technique for introduction of new types of value into a language. For example, a complex number may be defined as a record consisting of two fields, thus:

```
record class complex (real real part, imaginary part);
```

In order fully to embed a new type in the language, a facility must be provided for defining the normal arithmetical operations for quantities of the new type; so that subsequently the arithmetic operators +, -, ×, etc., may be used to operate on references to records, and deliver as their result a reference to a new record in which the fields have been appropriately initialised. For example, when the arithmetic operations have been defined for records of the class"complex",one could use them thus:

```
begin reference (complex) u, v, w;
      real x, y, z;
      . . . . .
      u:= v×w + z × complex (x, 3.7);
      . . . . .
end;
```

Extension of the definition of operators to operate on references to records of new types is known as overloading. The concept of overloading has been proposed by J. McCarthy.

The overloading of operators may be accomplished by declarations of the following form, which give a complete set of definitions for complex arithmetic:

```
reference (complex) overload (u+v); reference (complex) u, v;
        complex(real part(u) + real part (v),
                imaginary part(u) + imaginary part (v));

reference (complex) overload (u-v); reference (complex) u, v;
        complex(real part(u) - real part (v),
                imaginary part(u) - imaginary part (v));

reference (complex) overload (u×v); reference (complex) u, v;
        complex(real part (u) × real part (v)
                - imaginary part(u) × imaginary part (v),
                real part(u) × imaginary part (v)
            +   imaginary part (u) × real part (v));

reference (complex) overload (u/v); reference (complex) u, v;
    begin real denominator;
        denominator:= real part(u) ↑ 2 + imaginary part(u) ↑ 2;
        complex((real part (u) × real part (v)
                + imaginary part (u) × imaginary part (v))/
                                              denominator,
                (imaginary part(u) × real part (v)
                - real part (u) × imaginary part(v))/denominator)
    end;
reference (complex) overload (x); real x;
        complex (x, 0);
```

The last procedure defines the transfer function between real and complex quantities.

A similar technique for overloading can also be applied to the definition of vectors, matrices, etc., and of operations performed upon them.

6. A Comparison with PL/I.

PL/I is a new programming language in course of development by IBM, for the 360 series of computer. It is designed to cater for all applications of computers, including list processing, compiler writing, real time activities, and even the writing of software and operating systems. It is hoped that it will eventually supersede the use of Symbolic Assembly Language. It is an extremely large language, and we will have to deal with only the small part of it which is relevant to our subject.

All the main facilities of record handling have been incorporated into the language, but the details of the design are radically different, mainly as a result of the differences between the declared objectives of PL/I and those of record handling.

(1) PL/I does not attempt to protect a programmer from the results of his own mistakes. There is therefore no checking of the validity of field designators, either at compile time or at run time.

(2) PL/I places much less value upon conceptual economy, and provides a great many features, facilities, options, and abbreviations, so that the programmer may choose among a number of different ways of doing the same thing, in accordance with his tastes and inclinations.

(3) PL/I was designed with some bias towards the habits and practices of commercial programmers, and its notations and concepts seem to be modelled mainly on those for COBOL.

(4) PL/I was designed as a means of programming computers, rather than as a publication language for communication of algorithms, and it therefore places less value on making the notational structure of the program mirror its dynamic properties; such features as conditional expressions, and underlining of basic symbols have consequently been excluded.

The detailed study of the record handling facilities of PL/I can best be made by examining how the examples of the previous chapters can be rewritten in PL/I.

6.1 The example from 2. Reference Fields and Record Creation.

```
DECLARE     1 PERSON  CONTROLLED(X),                                    1.
              2 DATE_OF_BIRTH FIXED,                                    2.
              2 MALE              BIT(1),                               3.
             (2 FATHER, 2ELDER_SIBLING,                                 4.
               2 YOUNGEST_OFFSPRING) POINTER,                           5.
            (T, J, K) POINTER;                                          6.

NEWPERSON: PROCEDURE(D, M, F, E, Y)  POINTER;                           7.
        DECLARE D FIXED,  M BIT(1),  (F, E, Y, Z) POINTER;              8.
        ALLOCATE  PERSON  SET  (Z);                                     9.
        Z -> DATE_OF_BIRTH = D;                                        10.
        Z -> MALE = M;                                                 11.
        Z -> FATHER = F;                                               12.
        Z -> ELDER_SIBLING = E;                                        13.
        Z -> YOUNGEST_OFFSPRING = Y;                                   14.
        RETURN (Z);                                                    15.
        END NEWPERSON;                                                 16.

T = NEWPERSON (1908,  '1'B,  NULL,  NULL,  NULL);                      17.
T-> YOUNGEST_OFFSPRING = NEWPERSON(1934, '1'B, T, NULL,
                                                    NULL);             18.
T-> YOUNGEST_OFFSPRING, J = NEWPERSON (1935,  '1'B, T,                 19.
                          T -> YOUNGEST_OFFSPRING, NULL);              20.
T-> YOUNGEST_OFFSPRING = NEWPERSON(1937, '0'B, T, J, NULL);            21.
K, J-> YOUNGEST_OFFSPRING = NEWPERSON(1964, '1'B, J, NULL,
                                                    NULL);             22.
YOUNGEST_PATERNAL_UNCLE: PROCEDURE (R) POINTER;                        23.

        DECLARE (R, S) POINTER;                                        24.
        S = R -> FATHER -> FATHER -> YOUNGEST_OFFSPRING;               25.
REPEAT: IF S ¬= NULL THEN                                              26.
            IF S = R -> FATHER |¬S->MALE THEN DO;                      27.
                               S = S->ELDER_SIBLING;                   28.
                               GO TO REPEAT;                           29.
                               END;                                    30.
        RETURN(S);                                                     31.
        END YOUNGEST_PATERNAL_UNCLE;                                   32.
```

6.2 The example from 3. Record Subclasses and Their Discrimination.

```
DECLARE 1 CONSTANT CONTROLLED (W),                                    1.
          2 KIND FIXED INITIAL (1),                                   2.
          2 VALUE FLOAT,                                              3.
        1 VARIABLE CONTROLLED (W),                                    4.
          2 KIND FIXED INITIAL (2),                                   5.
          2 PRINT_NAME CHARACTER (12),                                6.
        1 PAIR CONTROLLED (W),                                        7.
          2 KIND FIXED INITIAL (3),                                   8.
          2 SUBKIND FIXED /* 1 = SUM, 2 = DIFFERENCE,                 9.
                                3 = PRODUCT, 4 = QUOTIENT */,        10.
           (2 LEFT_OPERAND,                                          11.
            2 RIGHT_OPERAND,                                         12.
            2 DERIVATIVE) POINTER,                                   13.
       (T1, T2, Z, Y, U, V, E, ONE, ZERO, X) POINTER;                14.
       ALLOCATE CONSTANT SET (ZERO);                                 15.
       ZERO -> VALUE = 0;                                            16.
       ALLOCATE CONSTANT SET (ONE);                                  17.
       ONE -> VALUE = 1;                                             18.
       ALLOCATE VARIABLE SET (Z);                                    19.
       Z -> PRINT_NAME = 'Z';                                        20.
       ALLOCATE VARIABLE SET (Y);                                    21.
       Y -> PRINT_NAME = 'Y';                                        22.
NEWPAIR: PROCEDURE (S, L, R, D) POINTER;                             23.
       DECLARE S FIXED, (L, R, D, Z) POINTER;                        24.
            ALLOCATE PAIR SET (Z);                                   25.
            Z -> SUBKIND = S;                                        26.
            Z -> LEFT_OPERAND = L;                                   27.
            Z -> RIGHT_OPERAND = R;                                  28.
            Z -> DERIVATIVE = D;                                     29.
            RETURN (Z);                                              30.
       END NEWPAIR;                                                  31.
```

```
ALLOCATE CONSTANT SET (W);                                          32.
        VALUE = 2.7;                                                33.
U = NEWPAIR (1, NEWPAIR (3, Z, Z, NULL),                            34.
                NEWPAIR (2, NEWPAIR(3, Z, Y, NULL),                 35.
                            W, NULL), NULL);                        36.
ALLOCATE CONSTANT SET (W);                                          37.
        VALUE = 2;                                                  38.
V = NEWPAIR(2, NEWPAIR(3, Y, Y, NULL),                              39.
                NEWPAIR(3, W, Z, NULL), NULL);                      40.
E = NEWPAIR(4, NEWPAIR(2, U, V, NULL),                              41.
                NEWPAIR(1, U, V, NULL), NULL);                      42.
ASSOCDERIV: PROCEDURE(E, X) POINTER RECURSIVE;                      43.
        DECLARE (E, X) POINTER,                                     44.
        (KIND_SWITCH(3), SUBKIND_SWITCH(4)) LABEL;                  45.
        GO TO KIND_SWITCH (E -> KIND);                              46.
KIND_SWITCH(1): RETURN (ZERO);                                      47.
KIND_SWITCH(2): IF E = X THEN RETURN(ONE); ELSE RETURN              48.
                                                    (ZERO);
KIND_SWITCH(3): IF E -> DERIVATIVE ¬= NULL THEN RETURN
                                                    (E -> DERIVATIVE);  49.
        ELSE BEGIN;                                                 50.
            DECLARE (U, UDASH, V, VDASH) POINTER;                   51.
            U = E -> LEFT_OPERAND;                                  52.
            V = E -> RIGHT_OPERAND;                                 53.
            UDASH = ASSOCDERIV (U, X);                              54.
            VDASH = ASSOCDERIV (V, X);                              55.
            GO TO SUBKIND_SWITCH (E -> SUBKIND);                    56.
SUBKIND_SWITCH (1)/* SUM */:                                        57.
IF UDASH = ZERO THEN E -> DERIVATIVE = VDASH;                       58.
ELSE IF VDASH = ZERO THEN E-> DERIVATIVE = UDASH;                   59.
        ELSE E->DERIVATIVE = NEWPAIR (1, UDASH, VDASH,
                                                    NULL);          60.
RETURN (E -> DERIVATIVE);                                           61.
SUBKIND_SWITCH (2) /* DIFFERENCE */:                                62.
        IF VDASH = ZERO THEN E -> DERIVATIVE = UDASH;               63.
        ELSE E-> DERIVATIVE = NEWPAIR (2, UDASH, VDASH,
                                                    NULL);          64.
```

```
RETURN (E -> DERIVATIVE);                                          65.
SUBKIND_SWITCH (3) /* PRODUCT */:                                  66.
        IF UDASH = ZERO THEN                                       67.
            IF VDASH = ZERO THEN E-> DERIVATIVE = ZERO;            68.
          ELSE IF VDASH = ONE THEN E-> DERIVATIVE = U;             69.
                                ELSE E-> DERIVATIVE = NEWPAIR
                                        (3, U, VDASH, NULL);       70.
ELSE IF VDASH = ZERO THEN                                          71.
            IF UDASH = ONE THEN E -> DERIVATIVE = V;               72.
                              ELSE E -> DERIVATIVE = NEWPAIR
                                        (3, V, UDASH, NULL);       73.
ELSE IF UDASH = ONE THEN                                           74.
            IF VDASH = ONE THEN E -> DERIVATIVE = NEWPAIR
                                        (1, V, U, NULL);           75.
                                ELSE E -> DERIVATIVE =             76.
                                NEWPAIR(1, V, NEWPAIR(3, U,
                                        VDASH, NULL), NULL);       77.
ELSE IF VDASH = ONE THEN                                           78.
            E -> DERIVATIVE = NEWPAIR(1, NEWPAIR(3, V,
                                UDASH, NULL), U, NULL);            79.
                      ELSE                                         80.
E -> DERIVATIVE = NEWPAIR(1, NEWPAIR(3, V, UDASH, NULL),           81.
                                NEWPAIR(3, U, VDASH, NULL), NULL); 82.
        RETURN (E -> DERIVATIVE);                                  83.
SUBKIND_SWITCH(4)/*QUOTIENT */:                                    84.
            IF VDASH = ZERO THEN E->DERIVATIVE=NEWPAIR
                                        (4, UDASH, V, NULL);       85.
        ELSE IF VDASH = ONE THEN E-> DERIVATIVE =                  86.
            NEWPAIR(4, NEWPAIR(2, UDASH, E, NULL), V, NULL);       87.
                  ELSE                                             88.
        E -> DERIVATIVE = NEWPAIR(4, NEWPAIR(2, UDASH,             89.
                                NEWPAIR(3, E, VDASH, NULL),        90.
                                NULL),                             91.
                          V, NULL);                                92.
        RETURN (E->DERIVATIVE);                                    93.
END ASSOCDERIV;                                                    94.
```

6. 3 Example Taken from 4. Files and Backing Stores.

```
UPDATE BANKLOANS: PROCEDURE OPTIONS (MAIN);                          1.
DECLARE   1 BANKLOAN CONTROLLED (MASTER),                            2.
            2 LOAN_# CHARACTER (7),                                  3.
            2 PRINCIPAL FIXED (8, 2),                                4.
            2 RATE_OF_INTEREST FIXED (3, 3),                         5.
          1 REPAYMENT CONTROLLED (TRANSACTION),                      6.
            2 LOAN_# CHARACTER (7),                                  7.
            2 AMOUNT FIXED (6, 2),                                   8.
          1 STATEMENT CONTROLLED (L),                                9.
            2 LOAN_INFO,                                             10.
              3 LOAN_# FIXED (7),                                    11.
              3 PRINCIPAL FIXED (8, 2),                              12.
              3 RATE_OF_INTEREST FIXED (3, 3),                       13.
            2 PAYMENT FIXED (6, 2),                                  14.
            2 BALANCE FIXED (8, 2),                                  15.
            INTEREST FIXED (5, 2),                                   16.
            BALANCE FIXED (8, 2),                                    17.
            REFUND FIXED (6, 2),                                     18.
            BANKLOANS FILE UPDATE RECORD
                        ENVIRONMENT(CHANNEL(3)),                     19.
            REPAYMENTS FILE INPUT RECORD
                        ENVIRONMENT(CHANNEL(1)),                     20.
            LISTING FILE OUTPUT RECORD ENVIRONMENT
                                    (CHANNEL(4));                    21.
            OPEN FILE (BANKLOANS) TITLE
                        ('BANKLOANS TR/1769B');                      22.
            OPEN FILE (REPAYMENTS) TITLE
                        ('REPAYMENTS 19/6/65');                      23.
            OPEN FILE (LISTING) TITLE ('STATEMENTS
                                    21/6/65');                       24.
            ON ENDFILE (BANKLOANS) STOP;                             25.
            ON ENDFILE (REPAYMENTS) GO TO NEXT_MAST;                 26.
NEXT_TRANS: READ FILE (REPAYMENTS) SET (TRANSACTION);                27.
NEXT_MAST:  READ FILE (BANKLOANS) SET (MASTER);                      28.
            INTEREST = PRINCIPAL * RATE_OF_INTEREST/12;              29.
```

```
        IF BANKLOAN.LOAN_# ¬= REPAYMENT.LOAN_#                    30.
            THEN DO;
                    PRINCIPAL = PRINCIPAL + INTEREST;             31.
                    REWRITE FILE (BANKLOANS);                     32.
                    GO TO NEXT_MAST;                              33.
                    END;                                          34.
        ELSE IF AMOUNT < = PRINCIPAL + INTEREST                   35.
            THEN BALANCE = PRINCIPAL + INTEREST - AMOUNT;         36.
            ELSE DO;                                              37.
                    BALANCE = 0;                                  38.
                    REFUND = -PRINCIPAL - INTEREST + AMOUNT;      39.
                    PUT LIST (BANKLOAN.LOAN_#, 'REFUND',
                                                    REFUND);      40.
                    END;                                          41.
        LOCATE STATEMENT FILE (LISTING) SET (L);                  42.
                    LOAN_INFO = BANKLOAN;                         43.
                    PAYMENT = AMOUNT;                             44.
                    STATEMENT.BALANCE = BALANCE;                  45.
        PRINCIPAL = BALANCE;                                      46.
        REWRITE FILE (BANKLOANS);                                 47.
        GO TO NEXT_TRANS;                                         48.
END UPDATE BANKLOANS;                                             49.
```

SHORT READING LIST

[1] J. McCARTHY

Recursive Functions of Symbolic Expressions and their Computation by Machine.

Communications of ACM, April 1960.

[2] D. T. ROSS

A Generalised Technique for Symbol Manipulation and Numerical Calculation.

Communications of ACM, March 1961.

[3] J. DAHL & K. NYGAARD

SIMULA, an ALGOL based Simulation Language

Communications of ACM, 1966.

[4] IBM Publication

IBM Operating System/360, PL/I: Language Specifications, Form C28-6571-2.

January 1966.

[5] NIKLAUS WIRTH & C. A. R. HOARE

A Contribution to the Development of ALGOL.

Communications of ACM, June 1966.