A 'Database' Subsystem for BCPL

R. A. Brooker
Computer Science Department, University of Essex, Wivenhoe Park, Colchester CO4 3SQ, UK

This paper describes a system of functions and routines for use with BCPL (or similar language) for organizing a 'database'. More precisely they enable the user to define a framework of tables, records, lists and functions within which to store, retrieve and manipulate certain primitive types of data. The system is oriented to 'structural' rather than tabular (relational) models of data. The paper is mainly concerned with the properties of the data structures, and their elementary constituents, and with illustrations of their use.

1. INTRODUCTION

This paper describes a system of functions and routines for use with BCPL[1] (or similar language) for organizing a 'database'. More precisely they enable the user to define a framework of tables, records, lists and functions within which to store, retrieve and manipulate certain primitive types of data. The data structures can be used to represent entities and (1–many) relationships, but are oriented to overtly 'structural' rather than 'relational' models of data: the connection with the latter is briefly discussed in Section 9. This paper is purely concerned with the properties of the chosen data structures, their functional specification, and with illustrations of their use. The special features of the system are as follows.

Dynamic data structuring

Every value in the database carries an explicit type identifier. Although in some respects this is grossly uneconomic, the system is free from the kind of restrictions imposed by the alternative approach of specifying the type, and possibly size, of all data at compile time, i.e. when the database is first set up. For example, lists may consist of heterogeneously typed components (including sublists); records, which are otherwise uniformly structured, can be given individual formats in special cases; names of fields, types and record subtypes are processable as strings (and thus constitute data dictionaries); and functions can be written which accept any type of value as argument. This last facility avoids the necessity of introducing separate versions of the same function for each type of argument. For example, a single output routine (OUTV) can be used to display any item of data, or data structure, in a style which will depend on the type of the data item.

Perhaps the most important advantage of dynamically structured data is that *it can be restructured dynamically*, i.e. by program means, without recourse to recompilation and recreation of the data. Such changes can be individual or systematic but they must preserve the internal consistency of the data.

Hierarchical record types

Each item of data is said to be an instance of the type referred to in its type identifier. Each type has a descriptor which is either primitive or another item of data (and hence an instance of some further type, and so on). This type 'hierarchy' is equivalent to (indeed motivated by) the record class hierarchy of SIMULA, with the additional feature that record class descriptors are processable objects. Linked to the concept of descriptor is the 'undefined value' which denotes an undefined instance of a particular type.

Functional (procedural) data

Names of functions and routines constitute one of the primitive types of data, i.e. procedural data can be included in the database. This is also a feature of SIMULA, and of LISP (where functions and data have the same internal and external representation).

LISP could be considered as a basis for the proposed system, but although it features dynamic data structures (i.e. S-expressions) these are inadequate by themselves. One also needs records accessed by field name, and tables (both in the working store and backing store) accessed by alphanumeric key. LISP lists are intended for scanning by 'cdr loop', and not for random access. Nor are they intended as backing store constructs (except incidentally via virtual memory). No doubt the necessary structures could be built up within LISP, but it would amount to using an interpreter to build an interpreter: the resulting loss of efficiency would be unacceptable. Instead, it was deemed better to use a system programming language (BCPL) as the basis of the data interpretive functions.

It is the author's view that the methods proposed here are suitable for databases which exhibit diverse structural forms, a multiplicity of special cases, and which to some degree are knowledge representation systems. Such systems will be capable of some degree of 'introspection', i.e. able to answer questions not only about their contents but about their form. For a system built from (say) Pascal structures, such questions could only be answered by the compiler—unless of course the relevant information is duplicated in the run time system.

A note on BCPL

This is a high level language control-wise and a word level language data-wise. It has no formal data types; instead a range of operators and functions is provided for combining words on the understanding that they repre-

CCC-0010-4620/82/0025-0448 $08.50

sent compatible types of value. Since there is no possibility of checking such assumptions (without losing the advantage of being 'close' to the underlying machine) such languages are generally considered to be dangerous to use, particularly for beginners. The only data structures provided are simple vectors: a local vector (size specified at compile time) and a free storage vector (size specified at run time). It is the user's responsibility to ensure that vector subscripts are within range. Certainly this is a likely source of error—far more likely, for instance, than trying to combine incompatible types of value.

Control features are: conditional expressions, implicit declaration of 'for' loop control variables, a range of iterative statements ('while', 'until', 'repeatwhile', 'repeatuntil', 'repeat'), case statements and 'escapers' ('return', 'resultis', 'loop', 'break', 'endcase'). A particularly useful feature for this application is that functions and routines can be called with a variable number of parameters (up to some limit specified in the declaration).

BCPL also has a very practical feature: it permits a program to be compiled in separate sections. The importance of this for large program development cannot be overstated.

By choosing BCPL as a host language the author is asserting that, given a suitable library of functions and routines, it can also serve as an applications language. A database application program will largely consist of control and assignment statements interspersed with calls for library functions.

2. GENERAL ORGANIZATION OF THE DATABASE

The database is organized as a set of Tables each consisting of an ordered set of key/Record pairs (or key/List pairs) (see Fig. 1). Henceforth we use TAB, REC and LIST to denote Table, Record and List. The majority of the TABs are held in the disc store: only the most frequently accessed are held in the working store. One such is the TYPE TAB and is common to all applications. It holds the type descriptors (which except for the primitive types are user defined) and the COMMAND RECs. These latter are the formal means of referring to the various query and transaction functions which constitute the end user interface. The data functions are those described in the body of this paper. They are utilized by the query and transaction functions.

The keyboard monitor module represents the outermost level of control. This interprets user messages of the form # {COMMAND key} or :{key}. In the former case the associated parameters are prompted for, checked and the COMMAND function executed. (This may entail further input/output activity at VDU/keyboard but it will be under the control of the function concerned.) In the latter case the key can be any key: the associated REC (or LIST) is displayed. ERRORS (at any level) which cannot be corrected by a keyboard dialogue cause control to revert to the keyboard monitor (which prompts the user for another message). In these situations it is essential to preserve the internal consistency of the data.

The diagram shows how the database is configured between the disc and working store when operational. (Details of how the database is first set up are given at the end of the paper.) That part of the database proper which resides in the WS is mostly invariant under user transactions, i.e. it is 'constant' to the application (the exceptional material is preserved on the disc between operational sessions). On the other hand, it is liable to be changed by the database administrator, for example the 'functional parts' may be modified or added to. This kind of activity should not affect the integrity of the database. It does mean, however, that this material has to be

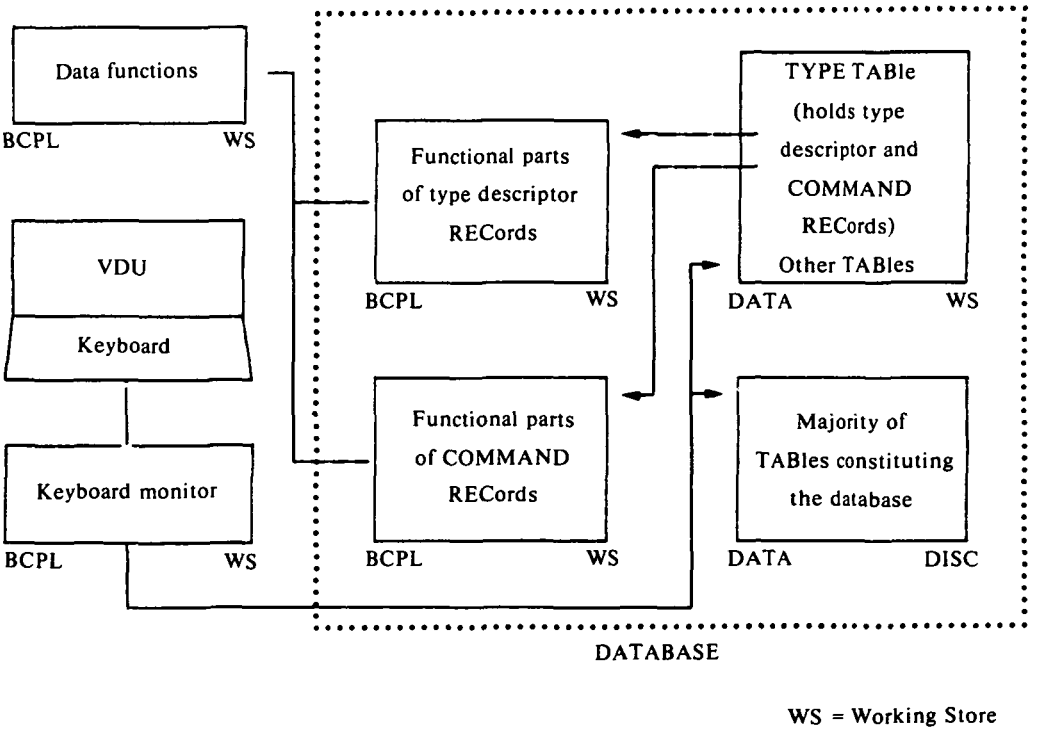

Figure 1. General organization of the database.

recompiled and the WS TABs regenerated at the start of each operational session. Thus the 'functional parts' modules are in fact active program modules which after recreating the working store TABs, can then be overwritten to leave only the passive functional parts which are referred to in the RECs of the TYPE TAB.

3. DATA TYPES AND DATA FUNCTIONS

The following types of value can be represented in the database.

(1) Five scalar types: INT, STR, BOOL, REAL, DATE.
(2) Four structured types: LIST, REC, TAB, SET (TAB = TABle, REC = RECord).
(3) Keys: the means of referencing values held in TABs. The type of a key is the TAB it refers to.
(4) Two function types: FN and FNX (the distinction is explained later).
(5) User defined types.

TABs can be either small or large (disc-based) structures. In the latter role they are the 'files' of the database. LISTs, RECs and SETs are essentially small scale structures (although as pointed out in the conclusion there is a need for a disc-based LIST). LISTs and RECs are normally held in TABs. Except for the central TYPE TAB (see below) TABs do not hold other TABs.

The type hierarchy and the TYPE TAB

The primitive types give rise to a limited hierarchy of special types, principally a system of user defined REC types and subtypes similar to the prefix record classes of SIMULA.

Also however, individual TABs (which are instances of TAB) are treated as formal subtypes, each serving as the 'type' of its own keys.

There are no sub*types* of LIST or SET, although LIST *values* name a 'base' type (the type of its components), and SET *values* refer to a 'base' TAB (whose keys serve as the base SET). The hierarchy of types can be pictured as in Fig. 2.

A central TAB, known as the TYPE TAB, holds all the type descriptors ('*templates*'). These are referred to by their keys. The templates for the primitive types are undefined, the relevant information being embodied in the system's data functions. The TYPE TAB also holds all the TABs in the system, including itself (i.e. it contains a pointer to itself). Each TAB serves as the type template for its own keys. The formal type NULL is included to top the type hierarchy: it is not to be confused with TYPE which is the type of the *keys* to the entries in the TYPE TAB.

We shall see later how RECs can serve as templates for further subtypes of REC. A feature of these templates is that they contain undefined values of some primitive types. An undefined value can be regarded as a regular value in which a flag (1 bit) is set to indicate that only the type component is meaningful. Their presence in (a field of) a REC template indicates that they should be replaced by a regular value of the same type. An undefined value of a given type is not to be confused with the key to that type. The latter is a defined value of type TYPE.

A note on the representation of data at word level

A value in the database is represented by one or more words (word = unit of storage). If more than one word is necessary the primary word contains a pointer to the remaining words which are held in the free storage. This will be the case for all the structured values and certain scalar values (e.g. strings). In the system described here the primary word also holds the type of the value, i.e. it takes the form (type + value) or (type + pointer). By assigning the primary word to a variable, a BCPL program can, knowing the representation, access the entire structure. In particular the type tag can be isolated and inspected at run time for any validating or discriminating purpose. The primary word also contains the 'undefined' bit.

Some scalar types of value in the database have a standard representation (H) in the host programming system. This is quite different from the database representation (V). Facilities are provided for converting H → V and V → H. There is no such correspondence for structured values.

We shall sometimes need to compute the *internal address* of a Value. Such quantities are introduced *temporarily* for the purposes of assignment: they have a one-word representation in the host system (distinguishable from primary words), *but no representation in the database*. Cross-referencing in the database is done by TAB keys.

Terminology

V denotes the representation of any database value, and Vint, Vstr, Vbool, Vreal, Vdate, Vlist, Vrec, Vtab, Vset the particular types of value. H denotes a standard BCPL representation, of which there are only four: Hint, Hstr,

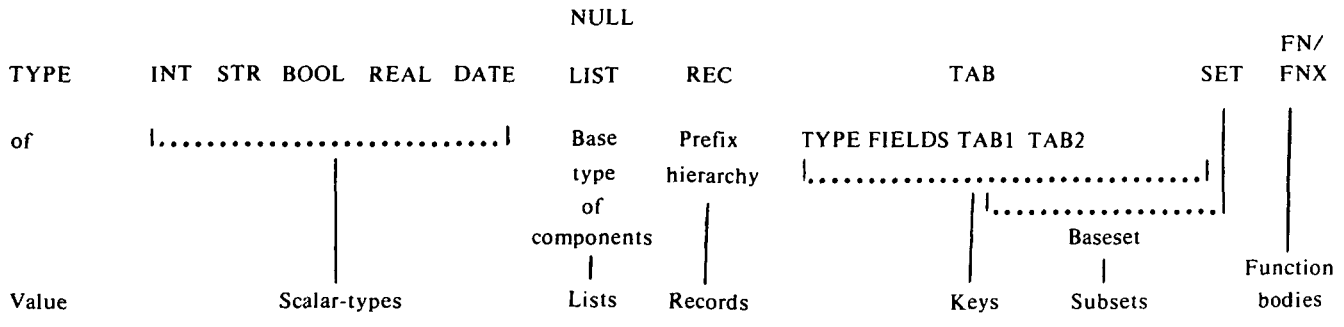

Figure 2. The hierarchy of types.

Hbool, Hreal. (Hdate ≡ Hint). Keys are denoted by K, and keys of the TYPE TAB by Ktype (or Ktab if they refer to a TAB). Different instances of values will be distinguished by numerical supplements, e.g. V1, V2, .., Vint1, Vint2, .., K1, K2. Internal addresses are denoted by AV, qualified if necessary, e.g. AVlist.

The database functions

The BCPL package contains the following functions (and routines).

M:Ktype, H, . . . → Vtype	is a function which converts appropriate H values into a V of the specified type
VTOH:V → H	is a function which converts a V value into the corresponding H representation (if there is one)
TYPEOF:V → Ktype	returns the key to the type of V (i.e. to the type template)

General functions

f:V1, V2, . . . → V	take Vs as arguments and return a V

Host predicates

p:V1, V2, . . . → Hbool	take Vs as arguments and return an Hbool as a result. These are intended for use in BCPL control constructs.
PROMPT:Ktype	prompt the VDU for a type of V
PROMPTREC:Ktype	prompt the VDU for a type of REC
OUTV:V	displays V on the VDU
STRFORM:V → Vstr	returns a string representation of V
FREEV:V	) storage
COPY:V → V	) control

Structure functions

These functions apply to a structured value specified in the first parameter. The subsequent parameters (, . . .) depend on the type of the first parameter. Details are given later in Table 1(ii).

S:V, . . . → V1	selects from V a component V1 specified by the remaining parameters
A:V, . . . → AV1	as S, but returns the internal address of V1
E:AV, . . .	extends V (at AV) with a component specified by the remaining parameters
D:AV, . . .	deletes the specified component from V
Q:V, . . . → H	returns a selector, with respect to V, of the specified component

Notes

(1) E and D are update operations, assignments in disguise. Purely functional forms could have been written but are less suitable for the expected style of programming, where the user has responsibility for storage control.

(2) A key can be (and generally is) used as the first parameter in the above functions: it is coerced to yield the associated Value (if a V is expected) or its address (if an AV is expected).

(3) Not all types of function call are relevant, e.g. no meaning can be attached to S(Vset, . . .).

Assignment

FREEV(V); !AV := V1	If V is the Value at AV, these BCPL statements first free (the extension of) V, then replace it by the primary word of V1. The same result can be achieved by
AS:AV, V	the call AS(AV, V1)

Unfortunately the internal address of a REC field refers to a byte within a word and in this case !AV is inadequate as the left part of an assignment ('selectors' are involved). For this reason it is more or less essential to use the formal AS operation when assigning to fields of records, and because this is a fairly common operation a special three parameter form AS(Krec, Hstr, V) ≡ AS(A(Krec, Hstr), V) is available for this purpose: it assigns V to the field Hstr of the REC whose key is Krec.

A complete list of the functions available (not all are mentioned here) and their specifications is given in Table 1.

4. PROPERTIES AND CONCEPTUAL REPRESENTATION OF DATA TYPES

Scalar types (and DATE)

Database values of these types can be created within the host system by means of the function 'M', for example M(INT, 15) is a Vint. Alternatively, such values can be created at the VDU via the PROMPT facility. Values can be manipulated within the formal subsystem by using such monadic, diadic, and predicate functions as are available, but it is more likely that the host language will offer a greater range of facilities and for this reason a V → H conversion function is provided. Results can be converted back using the M function (which is the H → V facility).

The type DATE is treated as primitive because it is a commonly occurring component of commercial records. A DATE value, for example 1 April 1981, is created by M(DATE, 810401). (An alternative would have been to introduce special functions to interpret an INT or a REAL as a date. However a DATE can be packed into a 16 bit word whereas an INT such as 810401 cannot.)

A DATE value can be manipulated by the functions ADDDATE, SUBDATE and UNITAGE. The first two are of the form Vdate, Vdate → Vdate, and are a composition of the separate operations of advancing (or retarding) the date by an integral number of days, months, and years. For example if V1 = 000001, V2 = 000100, V3 = 010000 and V4 = 810331, V5 = 800229 then ADDDATE(V4, V1) = 810401, ADDDATE(V4, V2) is undefined, ADDDATE(V4, V3) = 820331, SUBDATE(V5,V1) = 800228, SUBDATE(V5,V2) = 800129,SUBDATE(V5, V3) is undefined.

The function UNITAGE:Vdate1, Vdate2, Vdate3 returns a Vint, namely the largest multiple of Vdate3 which can be added to Vdate2 so that the sum ≤ Vdate1. For example UNITAGE(810930,210922,010000) = 60.

TABs (and the TYPE TAB)

TABs are the principal data structures: they are the 'files' of the database. They consist of pairs (key, value) which are ordered alphanumerically on, and accessed via, the key. A key has an external string form (e.g. "BLOGGS, J") and an internal form based on the physical address of the associated value and denoted by the unquoted string name (e.g. BLOGGS,J). The TAB itself records the string forms for I/O purposes but for reasons of efficiency it is the internal form (which also names the TAB being referred to) which is used for all internal references in the database. Internal keys (K) are thus a datatype in their right, the 'type' being the TAB referred to. The function, S(K), returns the value associated with K.

All type names, both primitive and user-defined, are held in a central TAB called TYPE. Because TAB names are also type names, the TYPE TAB has to contain its own name as a key. This is achieved by making the corresponding value (primary word) point to itself, i.e. the value S(TYPE) is the TYPE TAB itself. [*Note*: TAB and "TAB" denote the internal and external forms of the

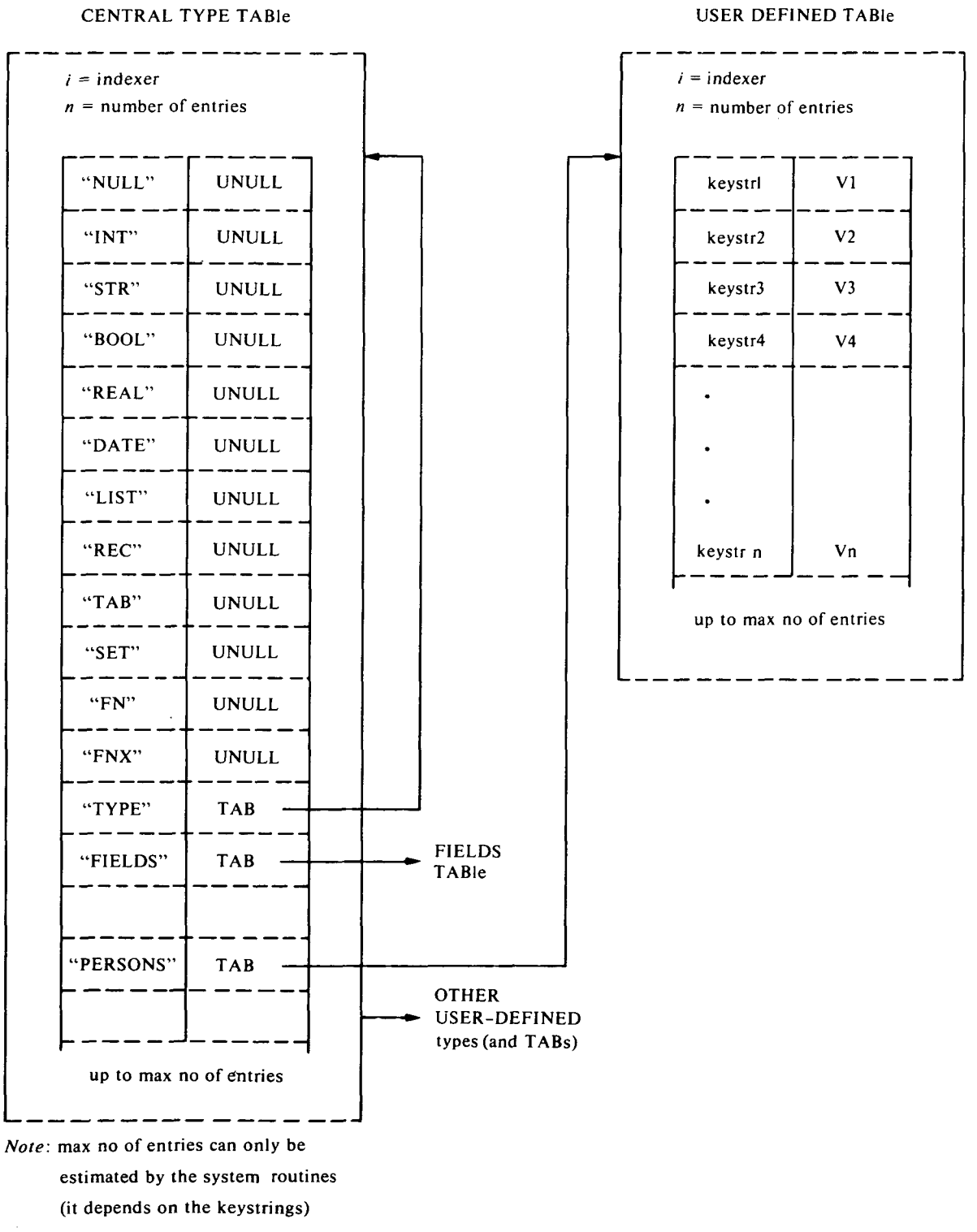

Figure 3. The central role of the TYPE TAB.

key to the 'TAB' type itself within the TYPE TAB.] The TYPE TAB has to be large enough to contain all the user-defined types, which may run to several hundred. The values held in the TYPE TAB consist of TABs, base SETs and REC 'templates'.

In addition to the TYPE TAB there is one other predefined TAB: FIELDS. This serves as a 'dictionary' holding the external string forms of all field names used within RECs. The corresponding internal keys are employed within the RECs themselves.

The TYPE and FIELDS TABs are held in the working store. The central role of the TYPE TAB is pictured in Fig. 3.

This is a conceptual picture only. The physical representation will depend heavily on its location—core or disc or some hybrid scheme. The internal key must be able to locate the associated Value with minimal computation; hence it will be a physical address of some kind, an offset in a core area or a disc block number. Once allocated it remains unchanged throughout the lifetime of the keystring. In the TYPE TAB above the system entries are listed in *order of entry*.

Tables are created by the function M(TAB, Hint, Hstr, Hbool), the last two parameters being optional. If the TAB is to be located in the working store then only the first parameter is needed: this is the approximate number of keys for which provision has to be made. (Approximate because the total storage space needed for the keystrings can only be estimated.) If adding keys to the TAB causes it to exceed capacity at some stage then it will automatically be relocated (such an event would be monitored on the VDU). If the TAB is to be located on disc (and most TABs will be) then the second parameter (Hstr) specifies the O/S name of the file which is to serve as the TAB. The third parameter (Hbool) says whether it is a new file or is to be updated. In all cases M(TAB, . . .) returns a Vtab, i.e. its primary word. This must be entered (see below) in the TYPE TAB, with the table name as key.

Once a TAB is created entries are made using E(Ktab, Hstr, V) which enters the pair Hstr, V and returns the corresponding internal key. If the keystring is already present the existing V is freed and then replaced by the new V. The function M(Ktab, Hstr) generates the internal key corresponding to a keystring assumed to be present; if it is not present an UNDEFined key is returned. This will signal an ERROR if used with the S function. Finally the function D(K) deletes the key K from the TAB it refers to, after first freeing the associated value.

Note: a call of the form E(Ktab, Hstr, M(Ktype, . . .)) can be abbreviated to EM(Ktab, Hstr, Ktype, . . .).

Thus to enter a new TAB in the TYPE TAB we call EM(TYPE, Hstr, TAB, . . .) where . . . denotes the individual table parameters, and Hstr is the name of the TAB. For example, the BCPL declaration LET PERSONS = EM(TYPE, "PERSONS", TAB, 500) creates a TAB called "PERSONS" with provision for approximately 500 entries. The internal key (with respect to the TYPE TAB) is assigned to a variable 'PERSONS'.

Small TABs with a specific list of keys (and undefined values) can be created thus M(TAB, L(Hstr1, Hstr2, . . .)). This is a most useful facility for defining base sets (see later *SETs*).

Provision is made for scanning the entries in a TAB: the functions RESET, MOREIN, NEXTIN are used for this purpose.

Denoting keys

The denotations of the internal keys of the 14 entries in the TYPE TAB, namely, NULL, INT, STR, BOOL, REAL, DATE, LIST, REC, TAB, SET, FN, FNX, TYPE, FIELDS are also the names of BCPL constants preset to the values of these internal keys.

Those data functions which expect a Ktype as a parameter will be assumed to accept the equivalent keystring as an alternative, so that for example M(INT, 15) ≡ M("INT", 15). This relies on the system being able to distinguish an Hstr from a Key by their internal representation. Otherwise it would be necessary to write, in this case, M(M(TYPE, "INT"), 15).

LISTs

These are intended to take advantage of machine indexing operations. A list is represented by a free store node of consecutively addressed words; the primary words of the items in the list. When a list is set up it is given some space to grow, and relocated (again with space to grow) if this should be exceeded. The primary word is changed accordingly, and the original node freed. The function LEN(Vlist) gives the number of items, which is needed for a 'FOR' loop scan.

There is provision for specifying the type of the components (known as the base type). A heterogeneous LIST would have base type NULL. The base type can be recovered by the function BASE(Vlist).

A primitive LIST value is created by L(V1, V2, . . . , Vn), an empty LIST by L(). These values have base type NULL. The 'L' function is supplementary to the 'M' which must be used to create more general LISTs. (The

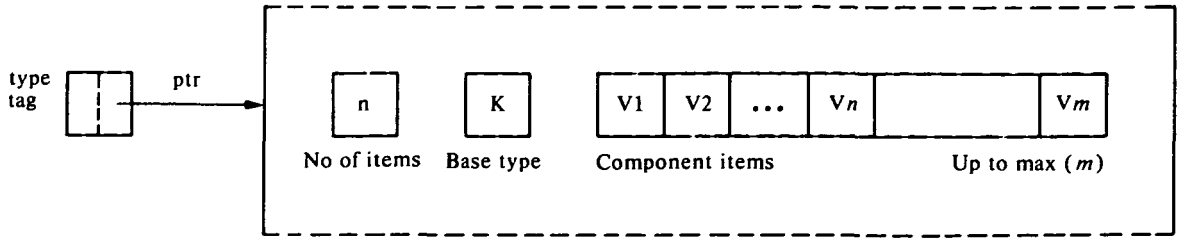

Figure 4. Conceptual picture of LIST.

'L' function—and 'R' in the next section—limits the number of parameters which 'M'—a very general function—has to take.) We have

M(LIST, NULL, L(. . .))	≡ L(. . .)
M(LIST, Ktype, L(. . .))	is a LIST with base Ktype ('M' has replaced NULL with Ktype)
M(LIST, Ktype, L())	is an empty LIST (of Ktype components)
M(LIST)	is an undefined LIST

Note: a call M(LIST, Ktype) is treated as M(LIST, Ktype, L()). There is no provision for undefined LISTs with defined base type.

In the call M(LIST, Ktype, L(. . .)) the Ktype parameter could be used to check the type of the components in L(. . .). However L(. . .) is taken as correct and instead Ktype is used constructively in the function XL defined thus.

XL(Ktype, H1, H2, . . .) = M(LIST, Ktype, L(M(Ktype, H1), M(Ktype, H2), . . .))
XL(Ktype) = M(LIST, Ktype, L())

For example XL(COMP.OPR.SCALES, "DP.ASST", "SEN.DP.ASST", "CHIEF.DP.ASST") creates a LIST of internal keys (corresponding to the listed keystrings) to the TAB named COMP.OPR.SCALES.

Once a LIST has been created items can be entered, deleted, selected and searched for by standard functions, as listed in Table 1(ii). Further functions and routines can easily be written by the user. LISTs are scanned by a BCPL 'FOR' loop in the conventional way (see Section 7).

Note: the 'L' function (and the same applies to 'R' in the next section) as implemented is permissive about the representation of its arguments. In addition to formal Vs, i.e. M(Ktype, . . .) constructs, it allows Hint and Hstr arguments which are converted to Vs as follows

Hint → M(INT, Hint)	
Hstr → M(TYPE, Hstr)	if Hstr is a keystring of TYPE TAB
Hstr → M(FIELDS, HSTR)	if Hstr is a keystring of FIELDS TAB
Hstr → M(STR, Hstr)	otherwise.

This makes specific LISTs easier both to write and read, *but again relies on the system being able to distinguish the parameters by their internal representations.*

RECords

These are represented by a list of explicit pairs (field, value), fields being represented not by their string names but by the (one word) keys of the strings in the FIELDS TABle. (Indeed only the essential information in a key needs to be represented, it being understood that this refers to the FIELDS TAB.) As with LISTs provision is made for adding new fields before the structure has to be relocated.

All RECs are 'typed' either by a template REC in the TYPE TAB (which may be typed by further templates), or by the primitive type 'REC'. The immediate type of a REC is also known as its prefix type, and the hierarchy of templates from which it has been derived is known as the prefix hierarchy.

Prefixing a REC means that it is associated with all the fields of the prefix REC type. Any one of those fields can be given a new meaning by redefining it in the new REC, along with the new fields introduced at this new (subtype) level. The value of a given field is found by searching (directly or, if the pairs are ordered by key, binarily) for the field in the pairs of the specified REC. If not found it is looked for in the prefix REC, and so on up the hierarchy until the prefix 'REC' is encountered, in which case the field is treated as invalid.

'Individual' records, i.e. instances of a REC type are normally held in separate disc based TABs because they form the bulk of the data in the database. Otherwise the only difference in the representation of a template type REC and an instance of that type is that the former (if it has more than member) will contain 'undefined' values.

Each f is an internal key to the FIELDS TABle, in which the corresponding keystring is the name of the field and the associated value is undefined: the TAB serves only as a 'dictionary'.

A REC of type Hstr' is created by M(Hstr', R(Hstr1, V1, Hstr2, V2, . . .)) where Hstr, V1, etc. are the field names and values to be entered in the REC. The component R(. . .) is itself a primitive REC, i.e. of type 'REC'. If the REC is to serve as a template it is entered in the TYPE TAB, for example EM(TYPE, "PERSON", REC, R(. . .)) creates a "PERSON" template (see Section 7 for details).

Given this "PERSON" template M("PERSON", R(. . .)) creates a "PERSON" type REC, and EM(PERSONS, "BLOGGS, J", "PERSON", R(. . .)) enters it in a TAB 'PERSONS' with key "BLOGGS, J". Alternatively we can create templates for different subtypes of "PERSON" as is done in Section 7.

Once a REC has been created it can be accessed by the function S(Krec, Hstr) which returns the value V associated with the field Hstr. The call E(Krec, Hstr, V) adds the field pair Hstr, V to the specified REC. Similarly D(Krec, Hstr) deletes the field named Hstr from D (first freeing the associated Value). In the former case the structure may be relocated if the provisional maximum number of fields is exceeded.

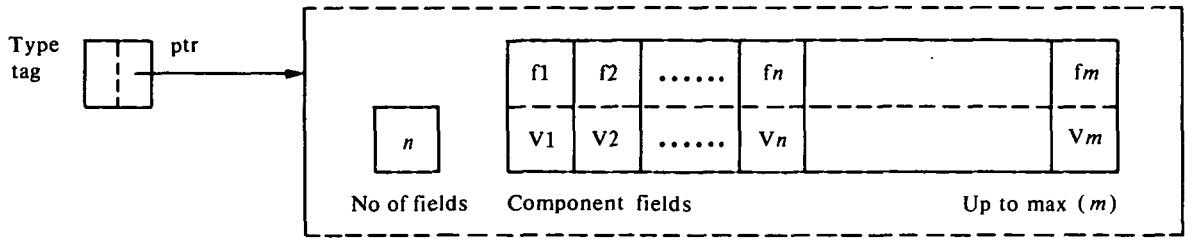

Figure 5. Conceptual picture of RECord.

Both LISTs and RECs can be freed (their storage reallocated so that they are lost to the system) and copied.

Access to the backing store (disc)

When a REC in a disc-based TAB is referred to it is first transferred to the working store (if it is not already present) and unpacked so that it appears as a regular working store REC. Its presence and location are noted in a 'transfer' list, associated with the TAB, so that further references will not invoke unnecessary transfers. It remains in the working store until (say) *N* further RECs from the same TAB have been transferred, at which point it is packed up and returned to the disc, and the 'transfer list' updated accordingly. In other words a simple paging system operates, treating the REC as a 'page'. Naturally if the entire disc store is covered by a virtual memory system there need be no distinction between disc-based and working store TABs.

SETs

A SET type can be associated with a TAB. The values of this SET type are the 2^n possible subsets of the (n) keys of the TAB. The keys of the TAB are known as the base set (and the TAB as the base TAB). It is similar to the *set* type in Pascal.

Given a TAB (Ktab) and a particular subset of its keys, Hstr1, Hstr2, . . . , the corresponding SET value is formed by M(SET, XL(Ktab, Hstr1, Hstr2, . . .)). The empty set is M(SET, XL(Ktab)), and the full set is MFULL(SET, XL(Ktab)).

The update functions E and D are available to add and delete particular keys to/from a SET value, and with these primitives the user can write functions to perform the standard set operations.

The representation of SET values is compact, essentially one bit for each possible key value. The conceptual picture is shown in Fig. 6.

The maximum number of entries can be found by reference to the base TAB. The indexer is used by the scanning operations RESET, MOREIN, NEXTIN.

A brief linguistic example[2] is included here (Section 7 includes another example relevant to the theme of that section).

```
EM(TYPE, "NOUN.FEATURES", TAB,
L("MASS", "NOUN", "NPL", "NS", "POSS",
                          "TIME", "TIM1"))
```

is the base TAB of possible syntactic features that can be associated with a noun. The set of features applicable to "fish" is

```
M(SET, XL("NOUN.FEATURES",
          "NOUN", "MASS", "NS", "NPL")).
```

Templates and undefined values

The term 'template' was introduced as meaning 'type descriptor' but it usually refers to a REC which contains undefined values. A REC template specifies the type and in some cases the default value of each field. Where a default value is inappropriate an 'undefined' value (of a default type) is given. This information is displayed when prompting (see later: PROMPTREC) the VDU operator for the actual values of the fields. The operator can 'accept' the default values, but if the undefined values are not replaced they will signal an ERROR when subsequently processed.

A function supplied with an UNDEFined argument will either yield an ERROR because there is no alternative (e.g. VTOH), return an UNDEFined value (e.g. STRFORM yields USTR) or take appropriate action (e.g. COPY returns a copy of it).

Undefined values yield their type in precisely the same way as any other value, i.e. via the functions TYPEOF and CASEOF. They are distinguished internally by a bit in the primary word (0 = defined, 1 = undefined), and of course the 'value' part is irrelevant. A predicate UNDEF(V) returns T(undefined) or F(defined).

Undefined values are created in the same way as other values by simply omitting the 'value' arguments.

Thus M(INT) is an undefined INT (similarly for STR, BOOL, REAL and DATE)
M(LIST) is an undefined LIST
M("PERSON") is an undefined "PERSON" REC
M(SET) is an undefined SET (there is no provision for specifying the base TAB)
M(PERSONS) is an undefined key to the PERSONS TAB

Note: to emphasize the nature of these special calls 'U' can be used in place of 'M', e.g. U(INT). They can also be represented by preset, implementation dependant, constants, UINT, USTR, etc. and we use such constants in Section 7.

Undefined values enter the system in three ways. They can be generated in the way described above; they can be supplied in response to a PROMPT for a scalar value (by typing ⟨RT⟩); thirdly, they can arise through incompatible H → V conversions (an important special case are undefined keys which result from 'looking up' keystrings which are not present in the TAB).

Functions

Procedural 'data' is often the most natural and economical way of describing the properties of certain entities. Thus there is provision in the system for 'function' types FN and FNX. Values of these types Vfn, Vfnx are represented

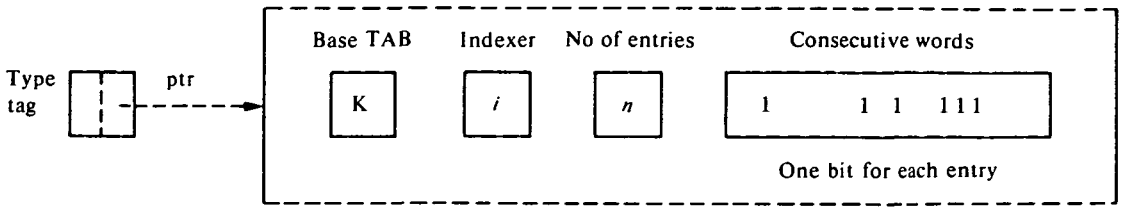

Figure 6. The conceptual picture of SET.

by primary words only: a pointer to the code body. Functions are evaluated by evaluating the expression f(a1, a2, . . .) where f is the Vfn (or Vfnx) and a1, a2, . . . the arguments. Alternatively the expression EVAL(f, a1, a2, . . .) can be used to apply f to its args.

Note: if f is *not* a fn, then EVAL(f) ≡ COPY(f).

When the OUTV facility (see below) is applied to a REC, any fields whose value is an FNX function will be listed as

{field name} FUNCTION(FNX)

i.e. simply stating that it is a function. In some cases however the function can usefully be evaluated; it must be parameterless, return a formal Value (i.e. not an H representation), and make adequate provision for UNDEFined values. By 'typing' such functions as FN (rather than FNX) they will be distinguished by OUTV and evaluated and listed thus

{field name} {value of the function}

This is the only distinction between FN and FNX. After evaluation the Value will be freed. *For this and allied reasons it is necessary that 'fn' fields return values (V or H) that can safely be freed without affecting the REC of which they are part.* It is the nature of such functions to do so, but by making it a convention there is then a clear distinction between evaluated 'fn' fields and 'non-fn' field Values, i.e. between terms such as EVAL(S(Krec, Hstr)) and S(Krec, Hstr); the latter return Values which (if non-primary) are part of the REC.

It can also be useful to invoke routines and functions directly from the operator's VDU via the command language interface that would necessarily have to be implemented for any data management program. To facilitate this the function (FNX) is embedded in a REC together with fields specifying the number and type of the parameters. This REC is used by the command language interface to prompt for and validate the parameters prior to invoking the routine or function itself. An example is given in Section 7.

One further point about functions—and this applies to some of the system functions—concerns the type of their results. These can be either formal Values or H representations. The latter can be used to anticipate a VTOH conversion. For example predicate functions intended for use in host language 'IF' statements or conditional expressions should return Hbool. On the other hand such functions cannot be evaluated by OUTV.

5. TABLE OF FACILITIES

Tables 1(i) and 1(ii) present a summary of the functions and routines available to the system.

Table 1(i)

Facility	SCALARS	KEY	FN/FNX	LIST	REC	SET	TAB
size in words	1[1]	1	1	many	many	few	many
Interface with host system[2]							
M:Ktype, H→V	X	X[3]	X		← See next table →		
TYPEOF:V→Ktype	X	X	X		← See next table →		
VTOH:V→H	X	X	X				
Interface with I/O							
STRFORM:V→Vstr	X[4]	X					
PROMPT:Ktype, INPUT→V	X	X		X	[5]		
OUTV:V→OUTPUT	X[6]	X[6]	X[7]	X	X	X	X[7]
General functions							
FREE:V	X[8]	X		X	X	X	
COPY:V→V	X[8]	X		X	X	X	
EVAL:Vfn, . . .→V			X				
polyadic:V1, . . .Vk→V	X[9]						
variadic:V, V, . . .→V	X[10]						
Predicate functions							
UNDEF:V→Hbool[11]	X	X	X	X	X	X	X
EQUAL:V, V→Hbool[12]	X	X					
LTEQ:V, V→Hbool[13]	X	X					

[1] Except REAL = 2 and STR = 1.
[2] M(TYPEOF(V), VTOH(V)) = V; FREEing H does not affect V, and vice versa.
[3] M(Ktab, Hstr) yields the internal key corresponding to Hstr if present, else an UNDEF value of type Ktab.
[4] STRFORM(Vstr) = COPY(Vstr).
We can also define KEYSTR(K) = M(STR, VTOH(K)).
[5] Use PROMPTREC (described later *Input and Output*).
[6] = OUTS(VTOH(STRFORM(V))) where OUTS is the BCPL string output fn
OUTV(K) gives [TABname]keystring; OUTS(VTOH(K)) just gives keystring.
[7] Outputs TYPEOF(V) only.
[8] For primary values (i.e. size = 1W) FREEV is redundant and COPY can be replaced by assignment.
[9] The number of arguments is fixed, e.g. ADDDATE:Vdate, Vdate→Vdate.
[10] The number of arguments is variable, e.g. CONCAT:Vstr, Vstr, . . .→Vstr
CONCAT concatenates copies of the arguments into a single string, and *applies FREEV to the original arguments.*
[11] UNDEF simply discriminates on the def/undef bit.
[12] Returns FALSE if arguments are different type, ERROR if UNDEFined.
[13] Yields an ERROR if arguments are different type, or UNDEFined
Key comparison assumes alphanumeric ordering.

Table 1(ii)

Structure functions

function	LIST	REC	SET	TAB
M:	LIST, Ktype, L(. . .)→Vlist	Vrec, R(. . .)→Vrec	SET, Vlist→Vset	TAB, Hint[, Hstr, Hbool]→Vtab
S:	Vlist, Hint→V	Vrec, Hstr→V	N/A	K→V
A:	Vlist, Hint→AV	Vrec, Hstr→AV	N/A	K→AV
E:	AVlist, Hint, V	AVrec, Hstr, V	AVset, K	AVtab, Hstr, V→K
D:	AVlist, Hint	AVrec, Hstr	AVset, K	K
Q:	Vlist, V→Hint	Vrec, Hstr→K	Vset, K→Hbool	Ktab, Hstr→Hbool

In the above functions a key can be used in place of V or AV as first parameter.

BASE:	Vlist→K	N/A	Vset→K	N/A
LEN:	←		V→Hint	→
RESET	conventional		as for TAB	RESET:Krec
MOREIN	'FOR' loop		but with Vset	MOREIN:Krec→Hbool
NEXTIN	1 TO LEN(Vlist)		*as argument*	NEXTIN:*Krec*→K

Assignment

AS:AV, V For REC fields AS(Krec, Hstr, V) ≡ AS(A(Krec, Hstr), V)

Key functions

SUPOF:K→K = TYPEOF(STAB(K))

CASEOF:Ktype, K→Hbool if (Ktype = K*i* for some *i*) then TRUE else FALSE where K1 = K and K(*i* + 1) = SUPOF(K*i*)

The LIST functions satisfy the following relations

After an update E(AVlist, Hint, V) we have

S(Vlist', Hint') = S(Vlist, Hint' − 1) for Hint' > Hint
= S(Vlist, Hint') for Hint' < Hint
= V for Hint' = Hint

Vlist, Vlist' being the values at AVlist before and after updating.
The operation D(AVlist, Hint) restores the original value.

S(Vlist, Q(Vlist, V)) = V
S(Vlist, Hint) yields ERROR if Hint < 1 or HINT > LEN(Vlist)

Similar relations obtain for the other structure functions.
We can also define special LIST functions, for example

XL(Ktype, H1, H2, . . .) = M(LIST, Ktype, L(M(Ktype, H1), M(Ktype, H2), . . .))
SLAST(Vlist) = S(Vlist, LEN(Vlist))
DLAST(AVlist) = D(AVlist, LEN(!AVlist))
ELAST(AVlist, V) = E(AVlist, LEN(!AVlist) + 1, V)
DQLIST(AVlist, V) = D(AVlist, Q(!AVlist, V))
where ! is the BCPL indirection operator

6. FUNCTIONAL DESCRIPTION

The 'M' function

Examples of its use are:

M(INT, 25)	an INT Value
M(STR, "THIS IS A STRING")	a STR Value
M(STR, " ")	the empty STR Value
M(BOOL, TRUE)	the two BOOL
M(BOOL, FALSE)	Values
M(REAL, 3.125)	a REAL Value
M(DATE, 801001)	a DATE Value (1 October 1980)
M(TYPE, "PERSONS")	Key to the TAB PERSONS
M(PERSONS, "BLOGGS, J")	Key to an entry in the PERSONS TAB
M(FN, SALARY)	a FN Value

.... WHERE SALARY( ) = {some host language function for calculating salary}

Examples were given earlier of its use to create a LIST, a LIST template and a LIST matching the template; and a base SET and subSET value.

Incompatible parameters (where detectable) result in undefined values of the given type, for example

M(BOOL, 3.125)	
M(INT, 3.125)	
M(PERSONS, " ")	
M(PERSONS, "BLUGGS, J")	where "BLUGGS, J" is not present in the TAB

The last case is important because it is the means of determining whether a keystring is present in a TAB: the resulting 'key' is UNDEFined.

Input and output

PROMPT as its name suggests prompts the VDU operator for a particular type of value (the type is displayed). The operator can respond with a value of that type, or a different type provided that the latter is preceded by the type tag. For example

{prompt:}{reply}
[DATE]:801001⟨RT⟩
[INT]:37⟨RT⟩
[INT]:[REAL]37.5⟨RT⟩
[STR]:B.A⟨RT⟩
[STR]:⟨RT⟩
[STR]:[EMPTYSTR]⟨RT⟩

A reply takes the form '[Ktype]string⟨RT⟩' or 'string⟨RT⟩'. The string is used to create H in the Value M(Ktype, H), the Ktype being either the prompted type or the overruling version. An empty string results in an UNDEFined value of Ktype. Because of this, special provision has to be made for an empty STR value: see the last reply. *Note:* ⟨ESC⟩can be used instead of⟨RT⟩, but the distinction is only significant in PROMPTREC (see below) where the empty reply ⟨RT⟩ means 'leave the existing value of the field unchanged', and the empty reply ⟨ESC⟩ means 'replace it by an undefined value'.

PROMPTREC prompts the operator for the fields of a REC using the prefix hierarchy as a 'template'. For each field in turn the existing (default) type and value (unless UNDEFined) is displayed before PROMPTing for a replacement. As mentioned above ⟨RT⟩ signifies 'no change'. The operator can also insert new fields if he wishes. Section 7 illustrates its use.

PROMPTREC has two parameters. The first is the key of the REC concerned. The second is optional: it is a LIST of fieldnames to be PROMPTed. If omitted *all* fields are PROMPTed.

OUTV simply outputs its argument in as reasonably compact a form as possible. If the style is not acceptable the user must write his own customized version.

Select function: S

The result of a select function is the primary word of a component value: the extension if any is not copied. It is the user's responsibility to COPY it if necessary. See section on Storage Control.

In the case of a TAB the component is selected by key (which as explained earlier includes a reference to the TAB in question).

In the case of a LIST the component is selected by giving its index.

In the case of a REC the component is selected by giving the field name (as an Hstr). The scope of the field selector is all the RECs in the prefix hierarchy starting with the specified REC and moving upwards. The first field encountered with the given name is selected: it takes precedence over any fields of the same name higher up (i.e. defined earlier) in the hierarchy. Calls of the form S(Krec, . . .) (where Krec is coerced to yield the Vrec) make a copy of Krec in a global 'currency' variable 'THIS'. This enables a functional field in a *prefix* REC to access fields in the *prefixed* REC.

In all cases S will signal an ERROR if the selector is invalid, that is if a TAB key is undefined, a LIST index is out of range or the field of a REC is not present at any level in the hierarchy.

The extend function: E

This adds a new component to a structure. This may result in physical relocation of the structure if there is no immediate room for expansion. In this case the original node is freed. This is why the function takes the form of an update.

In the case of a TAB the Hstr, V parameters specify a keystring and associated value. If the keystring is already present it is updated, the old value being freed. (In the case of a disc TAB however an ERROR is signalled.) Otherwise the TAB is extended and the internal value of the new key returned as the result of the fn.

In the case of a LIST a new V is inserted at position I (= HINT), the existing Vs at I, I + 1, . . . being moved up one position. An ERROR is signalled if I is outside the range 1, LEN() + 1.

In the case of a REC the Hstr, V parameters specify a new field, value pair. If the field name is already present it is updated and the old value freed. The fn only applies to the specified REC, not to any REC in its prefix hierarchy.

In the case of a SET a new Key is added. An ERROR is signalled if it is not one of the base SET.

The delete function: D

This function also takes the form of an update although deleting a component does not necessarily entail relocation. Again ERROR is signalled if the selector is in some way invalid.

In the case of a TAB only a single parameter need be given: the Key to be deleted. The associated value is freed. In the case of a LIST the Ith component is deleted, those at I + 1, I + 2, . . . being moved down to close the gap. In the case of a REC the named field Hstr is deleted and the associated value freed (like the 'E' function it does not apply to the prefix hierarchy). Finally, in the case of a SET the specified K is deleted.

The invert function: Q

There are only three cases of interest: LIST (the result, Hint, is the lowest position number of V in Vlist—0 if not present); REC (the result, K, is the type of the lowest record in the hierarchy, of Vrec, which contains the field Hstr—if present in Vrec it is the type of Vrec; if not present at any level the result is NULL); and SET (the result, Hbool, is the truth value of 'the specified K is present in the SET'—again ERROR if K is not in the base SET). In the TAB case: Q(Ktab, Hstr) is equivalent to /UNDEF(M(Ktab, Hstr)).

The scanning functions: RESET, MOREIN, NEXTIN

As their names suggest these are intended for scanning the values in a TAB or SET. The EXAMPLES below illustrates their use. Note that it is MOREIN which

actually advances the scanner: NEXTIN merely returns the scanned value (and subsequent calls of NEXTIN return the same value). Scanning of LISTs is performed by conventional BCPL 'FOR' loops using the LEN function to set the upper limit.

Storage control: FREEV and COPY

Although responsibility for garbage collection is in the hands of the user, these operations are needed much less frequently than might be supposed. We have already explained that OUTV automatically frees any value resulting from the evaluation of a FN field. Similarly CONCAT frees its arguments before returning the result. In this case however care must be taken to supply arguments that can be freed. The function NAME.ETC (in Section 7) illustrates the point: the argument D(=S(K, "DEGREES")) has to be supplied as COPY(D).

In the case of assignments the value replaced must, if necessary, be freed by the user if the space is not to be 'lost'. Thus if a BCPL variable X holds a non-primary value, we must write FREEV(X) before reassigning to it with X := V. Alternatively we can call AS(@X, V) which does it automatically. In most cases however values consisting only of a primary word are involved, and the FREEV operation is unnecessary.

Similarly the update functions E and D free the original structure if it has to be physically relocated.

Derived update functions, such as ELAST, should take the same general form. See the function MERGEVALUE, and its use in SEL.POSTS, in Section 7.

7. EXAMPLES

These are based on a small extract from a possible database of university appointments. This covers appointments of persons to posts (including secondary posts such as Dean of School, Student Union Officers, professional consultants), appointment to committees, and the allocation of posts to depts, sections, etc. This particular fragment defines three subclasses of PERSON, namely EMPLOYEEs, STUDENTs, and ASSOCIATEs (persons who are not regular employees but who are associated with the University in some way). The fields of the PERSON REC are those attributes which are common to all persons, e.g. birth.date, sex, etc. In this case they include "CAP.POSTS", "SECOND.APPTS", "CMMTT.APPTS", because all types of persons can, theoretically, be appointed to committees and secondary posts. The field "CAP.POSTS" is a LIST of the keys to appointments (current and pending) held in another TAB (called POSTS). The next two fields are functions, which when evaluated yield LISTs of current SECOND and CMMTT appointments derived from the "CAP.POSTS" lists by selection using the function SEL.POSTS (which is also reproduced). An EMPLOYEE has a prime appointment which is also found from the same LIST by the field function PRIME.APPT.

EMPLOYEEs also have, in general, a TEL.NO and a ROOM.NO which are associated with the post they hold, and therefore given by S(PRIME.APPT(), "TEL.NO") (and similarly for ROOM.NO). The modified version used below provides for the possibility that these functions may be evaluated before the appointment REC is actually created—in which case PRIME.APPT() would return an UNDEF key, which as explained earlier would signal an ERROR (and abort whatever transaction was in progress). (An example of the difficulties which can arise in a database with incomplete information.) In the case of a STUDENT one is referred to the STUDENTS.UNION and in the case of an ASSOCIATE one gets the general enquiries number "0". In these two cases the ROOM.NO field is left undefined. Note that telephone and room numbers are held as strings in separate TABs, namely TEL.NOS and ROOM.NOS.

Only one field remains to be mentioned: that is NI.NO. This refers to an EMPLOYEE's national insurance number.

The functions PRIME.APPT and SEL.POSTS should be self-explanatory: they are typical examples of the use of LIST scanning operations. They also illustrate the CASEOF predicate.

In some cases the S function is called with the currency parameter 'THIS'. Particular care is needed with this feature: it is reset on every call of the form S(Krec, . . .), such calls being easily overlooked.

```
EM(TYPE, "PERSON", REC, R(
"OTHER.NAMES", USTR,
"STYLE",       USTR,
"ADDRESS",     USTR,
"DEGREES",     USTR,
"SEX",         USTR,
"BIRTH.DATE",  UDATE,
"NATIONALITY", USTR,
"CAP.POSTS",   ULIST,
"SECOND.APPTS", M(FN,
                  SECOND.APPTS),
"CMMTT.APPTS", M(FN,
                  CMMTT.POSTS)))

WHERE SECOND.APPTS() = SEL.POSTS(S(THIS,
"CAP.POSTS"), M(TYPE, "SECOND.POST"))
AND CMMTT.APPTS() = SEL.POSTS(S(THIS,
"CAP.POSTS"), M(TYPE, "CMMTT.POST"))

    EM(TYPE, "EMPLOYEE", "PERSON", R(
    "TEL.NO",      M(FN, TEL.NO),
    "ROOM.NO",     M(FN, ROOM.NO),
    "NI.NO",       USTR,
    "PRIME.APPT",  M(FN, PRIME.APPT)))
WHERE TEL.NO( ) =
  VALOF $( LET PA = PRIME.APPT( )
           RESULTIS
             UNDEF(PA)→M(TEL.NOS),    S(PA,
             "TEL.NO")
           $)
AND ROOM.NO( ) =
  VALOF $( LET PA =PRIME.APPT( )
           RESULTIS
             UNDEF(PA)→M(ROOM.NOS),   S(PA,
             "ROOM.NO")
           $)
```

```
AND PRIME.APPT( ) = VALOF
  $( LET L = S(THIS, "CAP.POSTS")
  FOR I = 1 TO LEN(L) DO
  $( LET KEY = S(L, I)
  IF CASEOF("PRIME.POST", KEY) AND
    S(KEY, "CURRENT") RESULTIS KEY
  $)
  RESULTIS M(POSTS)
  $)

  EM(TYPE, "STUDENT", "PERSON", R(
  "TEL.NO",          M(FN, TEL.NO),
  "ROOM.NO",         M(ROOM.NOS)))
WHERE TEL.NO( ) = S(M(TYPE, "STU-
  DENTS.UNION"), "TEL.NO")

  EM(TYPE, "ASSOCIATE", "PERSON", R(
  "TEL.NO",          M(TEL.NOS, "0"),
  "ROOM.NO",         M(ROOM.NOS)))
```

```
. . . AND SEL.POSTS(P, PREFIX) = VALOF
    $( LET NL = L( )
    FOR I = 1 TO LEN(P) DO
    $( LET POST = S(P, I)
    IF CASEOF(PREFIX, POST) THEN MER-
      GEVALUE(@NL, POST)
    $) RESULTIS NL
    $)

. . . AND MERGEVALUE(AL, V) = VALOF
    $( FOR I = 1 TO LEN(!AL) DO
    IF S(!AL, I) = V RETURN
    // the use of '=' rather than 'EQUAL' means
    // that V is restricted to primary values
    RESULTIS ELAST(AL, V)
    $)
```

Creating REC instances

RECs for individual persons can be created from the above 'templates'. These will be held in a TAB called PERSONS. RECs can be created within the program in the same way as the templates, thus for example:

```
EM(PERSONS, "BLOGGS, J", "EMPLOYEE", R(
"OTHER.NAMES",   M(STR, "JOE"),
"STYLE",         M(STR, "MR"),
"ADDRESS",       M(STR, "1 MAIN STREET,
                   NEWTOWN,
                   WIGSHIRE"),
"DEGREES",       M(STR, "B.A"),
"SEX",           M(STR, "MALE"),
"NATIONALITY",   M(STR, "BRITISH")))
```

This would leave "BIRTH.DATE" and "NI.NO" undefined. Also because the "CAP.POSTS" field has not been included with the above, the default value is ULIST, i.e. an undefined list. However an actual list, initially empty, must exist at the particular PERSON level for the purpose of updating with appointment keys. This could be done in the appointment procedure(s) but it would be making a special case of the first appointment and so arrangements are made to include the pair

"CAP.POSTS", XL(POSTS),

whenever a particular PERSON REC is created. In practice PERSON (and other) RECs are created via the facilities of a command language interface which enables the VDU operator to invoke various data management operations. One such facility is the COMMAND "NEW.PERSONS" which creates an empty REC for each person named in a PROMPT cycle. The field "CAP.POSTS" in each REC is initialized to XL(POSTS). Fields created in this way can be completed using the PROMPTREC facility, via another COMMAND "AMEND.PERSON". To create the above REC for "BLOGGS, J" at the VDU involves the following dialogue of prompts and replies

```
{prompt}:{reply}
→ # NEW.PERSONS⟨RT⟩
[PERSON]:EMPLOYEE⟨RT⟩
[STR]:BLOGGS, J⟨RT⟩
[STR]:⟨RT⟩
→ # AMEND.PERSON⟨RT⟩
OTHER.NAMES[STR]:JOE⟨RT⟩
STYLE[STR]:MR⟨RT⟩
ADDRESS[STR]:1 MAIN ST., NEWTOWN,
  WIGSHIRE⟨RT⟩
DEGREES[STR]:B.A⟨RT⟩
SEX[STR]:MALE⟨RT⟩
BIRTH.DATE[DATE]⟨RT⟩
NATIONALITY[STR]:BRITISH⟨RT⟩
NI.NO[STR]:⟨RT⟩

DO YOU WISH TO ADD ANY FURTHER
  FIELDS? Y/N:N
```

Note the empty replies for BIRTH.DATE and NI.NO which leave these particular fields as they were, namely UNDEF. If the reply to the last prompt is 'Y' the operator is invited to type in field-value pairs (terminated by an empty field), for example

```
PREV.UNIVS  [LIST](LONDON,
                   MANCHESTER)⟨RT⟩
LANGS       [LIST](FRENCH)⟨RT⟩
⟨RT⟩
```

These fields are only associated with this particular employee REC: they can be regarded as 'afterthoughts'. If it is desired to include such fields in all records then they be added (in the same fashion) to the relevant template REC. This can be done without affecting any particular employee RECs. Thereafter PROMPTREC will prompt for these fields automatically.

Redefinition of fields in REC subtypes

When creating a subtype of a REC type or when creating an individual REC (which is essentially a subtype with one member) a field can be redefined.

When PROMPTREC prompts for the value of a field this can refer either to an existing field in the REC which is the subject of the PROMPTREC, or to a 'default' field of the same name in the prefix hierarchy of the REC. In the latter case the field name and new value is included as a pair in the subject REC. In either case the type as well as the value itself can be changed (see example given earlier under PROMPT). One can also replace a function definition (FN or FNX), which may be applicable to most cases, by a specified value applicable to an

exceptional case. (One cannot however supply new function definitions via the PROMPT—or PROMPT REC—mechanism.)

Redefinition of a field at the lowest level is useful when dealing with anomalies in an otherwise orderly structure. For example all employees are paid on salary scales which are periodically renegotiated and 'restructured'. This process sometimes leaves individuals on personal protected salaries. There is no "SALARY" field in the PERSON REC because it is an attribute of a (prime) appointment, details of which are held in the RECs of the POSTs to which they refer. All (prime) POST RECs share a default definition of "SALARY". This is a function which takes "SALARY.SCALE", "GRADE" and "SALARY.POINT" (three other fields) and looks up the actual salary in a TABle. In any appointments where this routine does not apply, the "SALARY" field is simply redefined to take specified values, and the "SALARY.POINT" field is left undefined.

An illustration of the SET type

Specifying the valid grades of an appointment utilizes the economical representation of the SET type. Consider first a "CLERICAL.RELATED" post. The relevant fields are

```
"SALARY.SCALE",  CLERICAL.SCALES,
"VALID.GRADES",  MFULL(SET,
                       XL(CLERICAL.SCALES)),
"GRADE",         M(CLERICAL.SCALES),
```

The field "VALID.GRADES" is used by the appointment routine to check the value of the prompted "GRADE", i.e. its value is (in general) a subset of the keys (grades) of the salary scale TAB which applies to the post. A salary scale TAB contains a separate scale for each grade. In this case 'CLERICAL.SCALES' has 4 grades, namely "1", "2", "3", "4", and all are valid, so that the value of "VALID.GRADES" is the full set of keys of CLERICAL.SCALES.

In the case of computer operator staff ("COMP.OPR") the relevant salary scale (COMP.OPR.SCALES) also includes grades and salary scales for data processing staff, and so only a subset of these are applicable. We have

```
"SALARY.SCALE",  COMP.OPR.SCALES,
"VALID.GRADES",  M(SET,
                       XL(COMP.OPR.SCALES,
                         "TRAINEE",     "OPR",
                         "SEN.OPR",
                         "SHIFT.LEADER.A",
                         "SHIFT.LEADER.B",
                         "CHIEF.OPR"))
"GRADE",         M(COMP.OPR.SCALES),
```

the other grades being "DP.ASST", "SEN.DP.ASST", "CHIEF.DP.ASST".

An illustration of TAB scanning and output

An illustration of TAB scanning and output is the following fragment of BCPL which (based on the above) lists the names and birth dates of all employees whose 60th birthday is on or before 30th Sept 1980.

```
RESET(PERSONS)
WHILE MOREIN(PERSONS) DO
$ ( LET P = NEXTIN(PERSONS)
UNLESS CASEOF("EMPLOYEE", P) LOOP
IF S(P, "BIRTH.DATE") <M(DATE, 211001)
        $( NEWLINE( )
        OUTS(VTOH(P))
        SPACE( )
        OUTV(S(P, "BIRTH.DATE"))
        $)
$)
```

Sorting (in the working store)

Consider the following bubble sort routine, where AL is the address of a LIST of V's to be sorted, and P an order relation on (adjacent) V's.

```
LET SORT(AL, P) BE
    $( LET L = !AL AND N = LEN(!AL) AND
       B = NIL
       $( B := TRUE: N := N - 1
       FOR I = 1 TO N DO
       IF P(S(L, I), S(L, I + 1))
          THEN $( LET X = S(L, I)
                  !A(L, I) := S(L, I + 1); !A(L, I +
                  1) := X
                  B := FALSE
                  $)
       $) REPEATUNTIL B
       //I.E., NO MORE ITERATIONS
    $)
```

Where the V's are (sub)LISTS the interchanges are performed on their primary words so that the process amounts to 'sorting with detached keys'.

The same applies if the V's are RECs. For example by changing the IF statement in the previous example to

```
IF S(P, "BIRTH.DATE") < M(DATE, 211001)
    THEN ELAST(AL, COPY(S(P)))
```

(where AL is the address of some initially empty LIST), copies of the relevant RECs are accumulated in !AL. This can then be sorted on BIRTH.DATE by calling SORT(AL, P) where P is defined by

```
LET P(V1, V2) =
LEQ(S(V1,"BIRTH.DATE"), S(V2,"BIRTH.DATE"))
```

Sorting operations are confined to the working store because only LISTs can be sorted, and LISTs are working store constructs. *TABs cannot be sorted because their contents are already ordered (by keystring).* What is currently lacking in the system is a disc based LIST construct, which can be ordered, or reordered, on some attribute of the component values.

An example of 'string handling'

This arises when combining a person's name and

qualification letters in a manner suitable for the University calendar. For example given

NAME	DEGREES	FORMAT
"BLOGGS, J"	"B.A"	"J. BLOGGS, B.A"
"SOAP, J"	USTR	"J. SOAP, UNDEF"

The following function will produce this format

```
AND NAME.ETC(K) = VALOF
$( LET D = S(K, "DEGREES") AND N =
   INITS.NAME(K)
RESULTIS UNDEF(D) OR EQUAL(D, M(STR,
   " ")) → N, CONCAT(N, M(STR, ", "), COPY(D))
$)
```

where CONCAT is the variadic function of STRs listed earlier (note that it frees its arguments), and INITS.NAME reverses the order of name and initials. The possibility that D has been left UNDEFined (if the person has no degrees it should be set to the empty string) has to be examined because EQUAL could yield an ERROR. The details of INITS.NAME are given here simply to illustrate the flavour of the host language.

```
AND INITS.NAME(K) = VALOF
$( LET KS = KEYSTR(K) AND V, W = VEC 20,
   VEC 20 AND LEN = NIL AND I = NIL
UNPACKSTRING(KS, V); LEN := V!0
I := 2; WHILE V(I) /= ',' DO I + := 1
FOR J = 1 TO I − 1 DO W!(J + LEN − I + 1)
   := V!J
FOR J = I + 1 TO LEN DO W!(J − 1) := V!J
W!(LEN + 1 − I) := '.'; W!0 := LEN
RESULTIS(M(STR, PACKSTRING(W, NEW-
   VEC(L/5))))
$)
```

An example of a "COMMAND" REC

This enables a function to be invoked via the VDU command interface. Consider the "NEW.PERSONS" function referred to earlier. It enables the user to create "PERSON" RECs (subtype given by the parameter) in the PERSONS TAB, in response to PROMPTed keys, the process being terminated by an empty key. It also detects if the key is already in the PERSONS TAB. An essential feature of the operation (as explained earlier) is that their ULIST fields are initialized to empty lists.

```
EM(TYPE, "NEW.PERSONS", "COMMAND", R(
"BODY",    M(FNX, NEW.PERSONS),
"PARAMS", L("PERSON")))

WHERE NEW.PERSONS(PREFIX) BE
  $( LET NAME = PROMPT(STR, "NAME")
  IF UNDEF(NAME) RETURN
  TEST UNDEF(M(PERSONS, NAME))
  THEN
  EM(PERSONS, NAME, VTOH(PREFIX),
  R("CAP.POSTS",   XL(POSTS)))
  OR OUTS("NAME ALREADY PRESENT*C*L")
  $) REPEAT
```

A keyname in the "PARAMS" LIST is interpreted as follows.

keyname of	actual parameter is
TYPE	a value of this TYPE
TABle	a keyname to this TABle
REC type	a keyname to a subtype

Thus the actual parameter corresponding to "PERSON" must be one of "EMPLOYEE", "STUDENT", or "ASSOCIATE".

8. THE STRUCTURE OF A DATABASE PROGRAM

A BCPL program consists (in general) of a number of separately compiled text files. One such file holds all the manifest and external declarations; the remaining files contain the functions and routines which make up the program. (The breakdown into separate files is determined by considerations of recompiling to correct programming errors.) The general layout of a database program will be as follows

MANIFEST and EXTERNAL declarations	augmented by user
STATIC declarations START routine	augmented by user
Database functions	as specified in Table 1
AND CMD.RECS( ) BE $(make Transaction and Query functions $)	defined by user (e.g. "NEW.PERSON")
AND TYPE.RECS( ) BE $(make REC type definitions and associated functions $)	defined by user (e.g. "PERSON")
AND WS.DATA( ) BE $(initialize any working store data and the contents of any working store TABs $)	defined by user (can be overlayed after use)
AND DISC.DATA( ) BE $(initialize the contents of any disc based TABs $)	defined by user (can be overlayed after use)

(*Note:* 'functions' in the above means 'functions and routines').

The program is entered at the START( ) routine which also contains the main VDU command cycle. The structure of START is described in the next section.

The START routine (and MANIFEST and STATIC declarations)

```
MANIFEST
    $( NULL = ?; INT = ?; STR = ?; BOOL = ?
       REAL = ?; DATE = ?
```

```
    LIST = ?; REC = ?; TAB = ?; SET = ?
    FN = ?; FNX = ?
    TYPE = ?; FIELDS = ?

    // these identifiers are preset to the
    // (implementation dependant) values of the
    // internal keys corresponding to the first 14 entries
    // of the type TAB. There is a corresponding set of
    // values, UNULL, UINT, . . . , UFIELDS
    // representing the UNDEFined values of these
    // types.
    $)

STATIC
    $( VTYPE = NIL; MODE = NIL; THIS = NIL

    // In addition to these SYSTEM variables, a
    // further variable must be declared for each
    // application specific TAB introduced by the user.
    // This variable will hold the key of the TAB entry
    // within the TYPE TAB, for example
    // PERSONS = NIL anticipates a PERSONS :=
    // E(TYPE, "PERSONS", . . .) statement in the
    // START routine.
    $)

LET START( ) BE
$( INPUT := FINDTTY( ); OUTPUT := INPUT
                                // i/o medium is VDU
    VTYPE := M(TAB, 50)   // creates TAB
    VTYPE!? := VTYPE      // inserts self-reference
                // (implementation dependant)
    E(TYPE, "TYPE", VTYPE)
    E(TYPE, "NULL", UNULL)
    E(TYPE, "INT", UNULL)
    E(TYPE, "STR", UNULL)
    E(TYPE, "BOOL", UNULL)
    E(TYPE, "REAL", UNULL)
    E(TYPE, "DATE", UNULL)
    E(TYPE, "LIST", UNULL)
    E(TYPE, "REC", UNULL)
    E(TYPE, "TAB", UNULL)
    E(TYPE, "SET", UNULL)
    E(TYPE, "FN", UNULL)
    E(TYPE, "FNX", UNULL)
    E(TYPE, "FIELDS", M(TAB, 100))

    // The above are system entries.

    MODE := PROMPT(BOOL, "TYPE 'TRUE'
      (UPDATE MODE) OR 'FALSE' (INITIAL
      MODE)")

    // Next come the application specific entries, for
    // example PERSONS := E(TYPE, "PERSONS",
    // M(TAB, 1000, "PERSONS", MODE)) declares
    // an O/S file "PERSONS" to serve as a disc based
    // TAB of the same name.

TYPE.RECS( ); CMD.RECS( ); WS.DATA( )

    // All working store data is (re)generated by the
    // program at the start of each run. This is
    // particularly necessary for the TYPE and CMD
    // RECs in order that any procedural components,
    // whose location may vary from one run to another
    // if the program has been altered in any way (for
    // example, by correcting mistakes), may be
    // correctly referenced. WS.DATA refers to all
    // other working store data.

    UNLESS MODE THEN $( DISC.DATA( );
      MODE := M(BOOL, TRUE) $)

    // If MODE is FALSE (i.e. 'INITIAL') then
    // invoke DISC.DATA to initialize the disc-based
    // TABs.

    // The description of the next part, the
    // COMMAND cycle, is informal.

ESCAPE:
    $( LET STRING = PROMPT(STR)
    IF STRING = "EXIT" BREAK
    IF STRING = "ESCAPE" LOOP
    IF STRING is # key of a "COMMAND" REC
    invoke the associated function, first prompting for
    and checking the types of the actual parameters.
    The function may itself prompt for further
    information. If the function is a transaction it will
    update one or more files and issue an
    acknowledgement on the VDU; if the function is a
    query the result will be displayed on the VDU.
    IF STRING is :key to any other type (or instance)
    of REC the REC is displayed on the VDU (using
    OUTV), including any default values specified in
    the prefix hierarchy.
    $) REPEAT

EXIT: CLOSE.TABS( )

    // This routine scans the TAB entries in the TYPE
    // TAB. For each disc-based TAB the working
    // store component (the primary index) is restored
    // to its allocated disc blocks. Finally a tableau
    // giving the maximum size and actual size of all
    // TABs is displayed.
$)  // END OF START
```

The only significant omission from the above account is the possible need for a working store TAB of miscellaneous and frequently referred to data (e.g. the current date) which is user updateable, like the disc TABs.

9. CONCLUSION

A package of BCPL functions and routines has been specified which enables an applications programmer to build an 'experimental' database for use by a relatively small group of users. The package provides some of the benefits of LISP and SIMULA within the framework of a systems programming language. A similar system may be applicable to 'C'. It should lend itself to what has been called 'structure intensive' information systems, where the programmer prefers to see the structure manifest in the representation of the data, rather than work with a 'flat' relational model.[3]

As regards entities and relationships, TABs hold anything that can be accessed by a single key: an entity with attributes or a (1-many) relationship with attributes. This puts the emphasis on functional dependencies (on the key) rather than general relationships. To support a general relational algebra would entail a disc based structure capable of holding an unordered and arbitrarily large set of tuples of uniform type (fixed when the relation is created) with facilities for retrieving tuples via alternative and compound keys. Such an extension is planned. Nevertheless the TAB structure seems the

appropriate mechanism for representing entities, the internal key serving as a surrogate for use in relationships where it would be uneconomical to store the full external keystring. Moreover the REC type hierarchy (derived from SIMULA) captures precisely the notion which Smith & Smith[4] refer to as generalization (although they associate the notion with Hoare's discriminated union type, and PASCAL's variant records, which is a very different syntactic mechanism to the same end).

The reader may have noticed that no examples were given of records being used as *components* of other records (or lists), only as *prefix types*. Such structures can only be built by program means at present. The PROMPTREC mechanism would have to invoke itself recursively to deal with such structures, i.e. to prompt for the fields of the (sub)record, the type of which is specified in the (main)record template. This type may have subtypes, any (value) of which may be a valid component. If necessary the PROMPTREC mechanism should be able to remind the user of the permissible subtypes and their formats. The user must indicate which of these he is inputting. At the time of writing however this facility is not available. Another facility which is not available is the SET datatype. The examples given of its use were in fact simulated by LISTs in the application referred to below.

A pilot version of the package described here has been prepared for the DEC system 10 by the author. It is not portable because the internal representation of the data depends on the word structure of this machine. However the package amounts to less than 1000 BCPL statements so it should not be difficult to redesign it for another machine. Using this package the author was able to implement the structural part of the information system referred to in the text (it contains grading structures, salary tables, committee structures, etc., but no personnel records). Details of this system will be the subject of a separate report.

Acknowledgements

It is a pleasure to acknowledge helpful conversations with Jim Doran, Ken Moody, Peter Stocker and Bernard Sufrin who all read an earlier version of this paper, and also with Surendra Mayaramani.

REFERENCES

1. M. Richards and C. Whitby-Strevens, BCPL, The Language and its Compiler. Cambridge University Press (1979).
2. T. Winograd, Understanding Natural Language, p. 67. Edinburgh University Press (1972).
3. E. F. Codd, A relational model of data for large shared data banks. *Communications of the ACM* **13** (No. 6), 377–387 (June 1970).
4. J. M. Smith and D. C. P. Smith, Database abstractions: Aggregation and generalisation, *ACM Transactions on Database Systems* **2** (No. 2), 105–133 (1977).

Received November 1981