Appendix A

In this appendix we will describe a particular design of a Conversational Base Machine (CBM). Much of the motivation for the features were developed in chapter 5. Other features reflect the author's personal preferences, and, as such, are not defended.

A1 The CBM Data Structures

For the purposes of this description, a set is an ordered collection of binary words on which we wish to place a "meaning!" The primary information about a set is part of the set itself. This is done for reasons of garbage collection and consistency of structure in general. Also, the different forms of sets, atomic, non-atomic, and programmed extensions are represented by a common format, mirroring the philosophy that the software is really an extension of the hardware. For primitive structures (those predefined in the CBM), the type may be dissectable into informational subfields such as the number of words per element and so on. Non-primitive data, hereafter called structures, contain type information and a pointer to an information block for the structure class (which contains size information, addresses of routines to be used to extend the basic machine operations, etc). Such structure blocks are themselves instances of structures which happen to need no other structure blocks to define them and which are primitive in the CBM.

The primary motivation behind such an approach is the desire that the hardware/software interface be consistent and flexible. That is, the subsuming of software functions into hardware for reasons of

(1) interpretive control is decentralized throughout the routines that perform actions corresponding to nodes in the parse tree;

(2) factored interpretation on trees requires less restrictions on the forms and times of changes which can be allowed to be made to programs and variables; and

(3) re-interpretation which is caused by such semantics changes is, in some sense, minimal in the case of a Tree Factored Interpreter.

Chapter 4, Differential Data Structures and an Extended TFI, gave a detailed discussion of implementations for variables and data in an IPS. Particular attention was given to the representation of symbol tables and to the means of implementing variables depending on what is known about specific variables. The latter part of the chapter was devoted to making the TFI scheme efficient with respect to memory utilization and to removing the remaining restrictions on the allowable times and forms of changes to variables in an IPS. Finally, we presented a method by which such a generalized TFI could act as a "normal" compiler with very little change.

Chapter 5, Bootstrapping an IPS, outlined a methodology for bootstrapping an IPS with an eye to being able to move the resultant system from machine to machine with some ease. The base on which the system is to be built was described in terms of the "primitive" operations needed by the TFI and other mechanisms in the foregoing parts of the work, with special attention given to subroutine and coroutine control and to the representation of data structures. A detailed description of a base which conforms to those specifications appears as Appendix A.

6E Summary

Chapter 1, Introduction, outlined a modus operandi for this work called the "Spectrum" philosophy: whenever there is more than one instance of a structure or feature of a system, those instances are to be considered as bands in a spectrum, and one should investigate and hopefully define its limits, other points of interest in the spectrum, or ways of moving automatically within it. A set of criteria for evaluating and designing interactive programming systems were derived from psychological, physical and intellectual needs of human programmers as well.

Chapter 2, Design Considerations for an IPS, developed a number of features for interactive programming, some of which were almost direct results of the criteria developed in Chapter 1. Among these were considerations of the form and function of the user/system interfaces, control structures for interaction, and the need for interpretation and compilation both within a single IPS.

Chapter 3, Interpretation and Compilation in an IPS, developed two central ideas. The first, called Factored Interpretation, provided the prime link between the notions of interpretation and compilation and showed how the latter could be obtained automatically from the former given suitable restrictions on the variability of the programs and variables being so used. The second idea was simply a means of using a form of parse tree program representation to drive interpretation. When coupled with the notion of a Factored Interpreter, such use of parse trees has the following properties:

Too much blue sky shines in such proposals for a simple Ph.D. dissertation! Nevertheless, some smaller and not completely absurd steps toward these objectives include the development of better communication facilities with machines, utilizing our already well developed natural modes of expression and reception: auditory, tactile, and optical.

This and work toward smooth system features involves joint research with behavioral psychologists; but in systems involving human interaction that is certainly long overdue.

Three recent developments of interest deserve mention as examples of a similar direction of research. The first is the development of the components of a many-user time-sharing system as a set of micro-coded processors [LS 70, BCC 70] for central processing units, a hardware scheduler (called the μscheduler), a swapping scheduler, and a sophisticated channel. The CPUs in that system have many high level language features such as array indexing, subroutine calling, and parameter passing encoded in the micro-code of the processors.

The second development is the design of an APL machine by Abrams [Ab 70] which is able to evaluate APL expressions in a highly optimal fashion. The last hardware development of note is the FLEX machine described in Kay's dissertation [Ka 69]. It pays a great deal of attention to the way in which the user communicates with his programs in both a physical and logical sense. It also contains an approach to the problem of attaining efficiency without loss of flexibility which is different than that developed in this work, but allows programs to exist in many different representations. Such fluidity of programs has been a hallmark of this research.

6D Long Range Research Goals

There is a strong temptation at this point to simply pull some extracts from science fiction writers such as Asimov or Clarke and reproduce them here. For, to some extent, the truly long range goals of research such as this are clear: sophisticated machines to work for and with us — tools to adapt our organic frames to more complex uses.

by a single common language suited to interaction is a result of this.

6C2 Hardware for Interactive Programming Systems

The discovery of a set of primitives for building an IPS, along with the bootstrapping approach suggests the development of hardware for an IPS. Iliffe's Basic Language Machine is, in fact, intended to be implemented as hardware; and the appearance of micro-programmable computers with read/write micro-store [SC 69, DSC 69] is very encouraging and we look forward to some interesting results in this direction in the next few years. Our Conversational Base Machine, for instance, could be implemented as "firmware" on such a machine.

Similar features have been incorporated in the command languages of a number of multilingual systems. This expansion of the command language facilities to include features normally associated with programming languages results from the importance of the user interface in an interactive environment. As users become sophisticated controllers of a multilingual system, more and more time is spent manipulating the features of the system and in switching between different processors such as text editors, compilers, etc. As a result, such features as recursively callable procedures, macro expansion (and thereby some syntax extension facilities) and arithmetic and logical abilities on numbers, address values, bit strings and character strings have become part of these command languages. An excellent example of such a command system is the CL-II command language of TSS/360 [IBM 69].

In short, the command language has evolved into a reasonably powerful interactive programming language. Unfortunately, in so doing, many of the lessons of systems such as JOSS, APL and LC^2 have been neglected; interactive control, especially, has not progressed to the degree available in monolingual systems.

As interactive programming language systems become more powerful and more widely used, we can expect this trand to more flexible command languages to continue, and we can expect the distinctions between multilingual and monolingual systems to fade. For, what the users are really doing is changing the human-system interface to make it easier to manipulate; replacing separate controls and languages

they can also be considered as tools for the designer. We propose an evolutionary approach for the use of the criteria: where they miss the mark, they should be amended or augmented in order to make them better approximations to the needs of the real world.

6C1 Monolingual and Multilingual Systems

It has been stated that "general" time-sharing systems such as TSS/360 [Le 68] and MULTICS [Sa 66] are more powerful and more useful than monolingual, dedicated systems such as JOSS, APL and LC^2. Since most dedicated systems have stressed ease and smoothness of use over efficiency, it should not be surprising that they are more limited, computationally, than the TSS and MULTICS type of system.

However, in terms of ease of use, acceptability by users and reliability, there is less agreement as to the dominance of general, multilingual, time-sharing systems over the monolingual, dedicated facilities. One main reason for this is that the "command language," the text editing facilities and the programming language in a monolingual system are one and the same; that is, there is one common syntax for these interrelated tasks, and each "sublanguage" may use the features of the other parts of the system. Thus, one can use the procedure feature of the programming language along with text editing statements to build text editing procedures, or one can include "command language" statement types in procedures in order to take advantage of the forms of control and parameter passing in the programming language.

suitable for that user, or class of users, and to encourage the evolution of interface facilities with use over time;

(3) the efficiency of the user's programs is not proportional to the speed of the IPS, as is the case with an interpreter; they can execute at the hardware speed of the computer hosting the system;

(4) the system encourages the development of programs which, by being independent and using a coroutine form of control, aids the building of large systems of programs; it is not meant simply as a replacement for a desk calculator or for running only small jobs; and

(5) reconstruction of the system for a new hardware computer is meant to be minimal; only the basic set of primitives need be reprogrammed in order to move the system from machine to machine.

6C Short Term Research Areas

One of the measuring rods for any research is the amount of other research which it suggests and prompts. The Algol-60 effort, for example, has prompted much work in parsing of programming languages, compiler construction, and language design and description.

This work has developed methods for achieving efficiency in programs and data structures within an IPS without sacrificing flexibility. It has also proposed an organizational structure for interactive programming systems and, perhaps just as importantly, put forth a set of criteria for evaluating an IPS. Those criteria, though well considered, must be viewed as only an initial stab at establishing such a set. To the extent that they are used in designing systems,

user, though to a lesser degree. In order to maintain the ability to change the representation of a program, it is necessary to maintain certain information — an audit trail — which facilitates such changes. In order to get rid of that overhead when change was no longer so necessary, the user must make a "contract" with the system, albeit a very loose one. That contract takes the following form: " compile program X; in return for that service I, the user, promise not to change the program or its variables unless it is inactive." Indeed, it is less a contract prohibiting change than a "gentlemen's agreement" on the circumstances under which changes are to be allowed.

Another view of the Interpretation/Compilation scheme which was originally given in section 2H is that it is an attempt to take advantage of de facto constancy while allowing the user to dispense with declaring such constancy. In order to build a system with this facility, declarations are required at some lower level: that level is low enough to make it a feasible basis on which to build an IPS of the type described in the thesis. An analysis of what that base looks like and how it can be used to bootstrap the IPS was the topic of chapter 5. The primitives which were given there in outline (and more specifically in Appendix A) were those developed in chapters 2, 3, and 4 of the thesis.

It is worth restating some of the properties of such an IPS:

(1) it is modifiable within itself, making available the system facilities as debugging aids for the system implementers and maintenance engineers;

(2) it can be changed easily to make its interface with the user

Among these were the concept of the user as a function, coroutine control and independent processes as a means to a well-structured and changeable system, and, most importantly, access to the system data structures as a prerequisite for an IPS changeable by the user. The natural result of this is a system which, being partly self-modifiable, is also bootstrappable to an extent.

6B The "Spectrum" Philosophy

Wherever more than one instance of a common phenomenon (such as interpretation and compilation for execution) has been found, we have attempted to understand it as a whole spectrum of instances of the phenomenon. Chapters 3 and 4 were heavily involved with understanding the execution of programs and the representation of data structures in an IPS. A means of having the system move automatically within the execution spectrum — between the extremities of compilation and interpretation — was developed, called the TFI (Tree Factored Interpretation), which changes the representation of a program depending on its usage.

In the case of data structures, we developed no automatic means of smoothly traversing the representation spectrum, but did find a number of discrete points of interest. Those points, corresponding to different representations for values, depended not only on the state of the data (whether or not the value was declared, for example), but also on the state of the user and his willingness to ascribe attributes to values as information which the system could use.

The TFI mechanism has this same failing of dependency on the

is completely untested by users in real situations until large amounts of coding have made the system difficult to change. The system which results after some years of usage normally bears only scant resemblance to the original structure, and has often degenerated into a "dirty" implementation with little of elegance or simplicity remaining.

Some morals which may be gleaned from this are:

(1) to fight change by building inflexible systems invites corruption;

(2) it would be better to expect evolutionary change in a system by building it that way than to resist it and ultimately have to accept revolutionary change (such as the so called "next generation" of systems); and

(3) system designers must concentrate much more on the human users of a system, bending and developing the system to their needs, and not vice versa: an excellent way of doing this is to make the designers wear users' shoes by demanding that they use and evaluate the system they are building.

There are more of the same given in Appendix A of [EER 68], but the above will suffice as motivation in retrospect for much of the thesis work.

Chapter 2, Design Considerations for an IPS, gave a step by step development of some particular features for an IPS. Those features were prompted by the material in the introduction on designing an IPS, plus the sketch of LC^2 under those considerations. Much of that development was aimed at allowing the user to be a close participant in the IPS, and also presented some mechanisms which facilitated that.

the way, describe themselves as a "Bootstrap Community") has been a source of encouragement and enlightment during this research. A statement of that community's goals and methodology, taken from [EER 68], describe this approach well:

> "... the aids developed and experimented with are those that promise to the Bootstrap Community the best payoff either in direct improvement of working abilities or in new understanding toward that end.
>
> "Implicit in the above, but deserving explicit comment, is the evolutionary nature of the system growth that results from this approach. Developments of various facets of this system, as well as our means to study, analyze, design and implement them, must all evolve together in a coordinated fashion."

Of course, there is no guarantee that such an approach will lead to better systems and better use of computers. Nevertheless, the probability certainly seems higher that such is the case than it does for the traditional scenario of system development. That scenario generally follows the pattern:

(1) think of features that a set of hypothetical users might need;

(2) make initial stabs at means of implementing the features derived in (1);

(3) if implementation is difficult for some features, reiterate through steps (1) and (2), modifying the features as necessary;

(4) construct the system as designed, with possible repetitions of steps (1) to (3) if the design does not work out;

(5) finally, release the system, largely untested, to the user community.

The glaring difference between this procedure and that advocated by Engelbart et al is that a system developed in the traditional manner

6 Conclusions, Future Research and Summary

The research reported here has been concerned with three aspects of interactive programming systems: design , architecture, and construction. The bulk of the thesis has dealt with the architecture of an IPS, especially as regards the flexibility and efficiency of programs and the data structures needed by both user and system.

The term aesthetics has been used as an umbrella for those aspects of the design of an IPS pertaining to the user's psychological, physical, and intellectual needs in terms of interactive computational tools. Such design considerations, while valuable in their own right, have also provided much motivation for the remainder of the research.

The inclusion in this work of a method for bootstrapping an IPS is the logical end-product of aesthetic considerations and flexible architecture.

6A Flexibility in Programming Systems

An over-riding belief buried in these pages is that if interactive programming is to be truly useful, then it must be used not just by the non-system programmer, but also by system programmers (which can be rephrased as: why should the users have all the fun?). Bootstrapping, as well as being an elegant way of building systems, also provides impetus for efficiency considerations and the use of the IPS by its implementers in its earliest stages of development.

In this regard, the work of Engelbart's group [EER 68] (who, by

activated by the occurrence of an interrupt with interrupt status information passed as a parameter to the routine. The FLEX machine [Ka 69] and the RC-4000 Multiprogramming System [Ha 69] are good examples of systems with such interrupt processes.

5E7 CBM Input/Output

The least well-specified part of the CBM is its set of input and output operations (this is a direct reflection of the state of I/O methods in both hardware and software). In general, data structures are to be allowed to reside on secondary storage; copying them into primary memory then obscures the actual I/O operation. Therefore, the COPY operation of the CBM will include most I/O operations, with processes handling I/O interrupts. In the RC-4000 Multiprogramming System, every I/O device corresponds to a process called an "external process." Such a notion is useful for exporting a system since many of the I/O processes can then be initially "faked" during the bootstrapping and exportation operations until they can later be written for the new machine.

5E8 Summary

The foregoing considerations give a broad brush outline of a CBM. A sample realization of them appears in Appendix A. That specific design is claimed to be true to these constraints. Other criteria have entered into the details of our example CBM, and little justification is given for them in terms of the needs of an IPS.

In order to implement processes, a special data structure called a port is needed. Ports act like channels over which only one message at a time may pass. Control also "flows" across ports between connected coroutines, usually associated with a request for input on the port or a request to output a message across the port.

In a subroutine call three distinct actions are combined together: creation of an incarnation of the subroutine (i.e., space for variables and state information), the passing of parameters from caller to callee, and finally the transfer of control to the subroutine with information being kept to later return control to the caller. These actions are separated in process (coroutine) control.

Creation occurs when some process (possibly the user) requests an incarnation of some process to be created. Since this is really the action of allocating space for a stack, static variables, and state information for the process, such actions require no special CBM instructions beyond the storage allocation abilities it has.

There are two times when parameters are "passed" between processes. The first is the passing of parameters from creator to created process when the new process begins execution. The second time occurs whenever control flows across a port as a "message" which passes over the port. The first type of parameter passage is really just the same as the subroutine parameter mechanism. The second is inextricably bound up with process control and also needs to be provided by the CBM.

Interrupt routines function much like processes which are

If a value is to be returned, that can be indicated in one of two ways:

(1) explicitly by having a return-with-value operation, or

(2) implicitly, by the fact that the top of the stack is not at the return control information but has an extra cell on top of it; that extra cell is the value to be returned.

In order to allow abnormal returns, such as the return-code mechanism discussed in section 5D2 (Control Within an IPS), the CBM needs to have such an operation. But it also requires a mechanism on the calling site which can act as a filter of abnormal returns, trapping those of interest while others are allowed to percolate back. There are a number of ways of doing this. One possible method is to attach a list of return-codes to each call site, which list is then inspected on an abnormal return. If the return-code which caused the inspection is one of those in the list, then a pre-specified routine for that call site is given control, and the return-code made accessible so the routine can deduce the condition which caused it to be invoked. Some means of filtering any abnormal return is also needed for routines which must reset variables or tables to a stable state before allowing the return-code to be passed back to the caller of the routine which "caught" the return-code on its way back.

The coroutine form of control advocated for processes may be thought of as "above" normal subroutine control. Thus, when control flows back and forth among a configuration of processes, it may do so even from within subroutines invoked from the process; in this sense coroutine control is above subroutine control.

be reclaimed. A set can only become inaccessible if no accessor points to it, and that can only occur when the last accessor referencing the set is destroyed. Thus, in the CBM, actions resulting in the copying or overwriting of accessors may have an effect on the information for the set accessed in order to maintain storage integrity. Such side effects are common in languages using lists such as LISP and it therefore seems appropriate to make them primitive in the CBM.

5E6 Control Structures

The TFI has relied heavily on a certain specific form of subroutine control; it is therefore a natural candidate for inclusion in the instruction set of the CBM. For calling a sequence of code, the following actions are done:

(1) check if the code is valid; if not, transfer control to the routine INTERP which will cause interpretation to take place;

(2) check whether the complete-bit for the code is on; if it is not, discard the code and transfer control to the routine INTERP;

(3) place information on the control stack (which may or may not be the same as the operand stack) recording where to return in the calling routine, and then transfer control to the called code.

The return sequence is very similar to a call:

(1) retrieve return information from the control stack;

(2) check if code to be returned to is valid; if not, activate REVERT (see section 4C3C);

(3) transfer control to the place indicated by the control information.

would be useful for passing parameters to subroutines since they would then not need to know exactly the depth of indirectness of a parameter, but could simply assume it to be direct. Another form of coercion which is related to Auto/Fetch is called Auto/Execute: if an operand for which a value is desired turns out to be a set of machine code, then the way to get its value is to evaluate it. These two examples and all other possible type-i to type-j coercions need to be stated in the design of a CBM.

5E4 Code Brackets

Another basic function in the CBM implements part of the code production scheme needed for a TFI; namely, the substitution of parameters into some prototype code string as operands of certain instructions in that code. This also emphasizes the point that one must be able to suppress coercion of operands when necessary. Otherwise, attempting to copy a code string would normally result in its execution due to Auto/Execute coercion.

5E5 Storage Allocation

In order to provide facilities for handling the allocation of storage for complex data structures, the CBM needs primitive operations for allocating and freeing memory blocks. These operations must not violate the protection mechanisms of the CBM or those features which guard against sets becoming inaccessible (such as normal arithmetic on accessors). For this reason each set in the CBM should have some mechanism, such as a count of all the accessors referencing that set, so that when the set is about to become inaccessible, its storage can

regular memory, or resides in the stack (which, of course, probably itself resides in primary memory).

As mentioned previously, operations should be able to act on entire sets as well as single elements, and combinations of scalars with sets. A good example of this might be the ADD operation. For the moment we will posit an ADD instruction such as

ADD A,B

for the CBM, meaning "add A to B, leaving the result of the addition in B." Depending on the structure of A and B, this could accomplish any of the following:

Structure of A	Structure of B	Effect of ADD
scalar	scalar	normal scalar addition
scalar	non-scalar (vector)	for I to size(B) do B[I] ← B[I]+A;
set	scalar	B ← 0; for I to size(A) do B ← B+A[I];
set	set	for I to size(B) do B[I] ← B[I]+A[I];

Since the operands of an instruction carry type information with them, the CBM should also handle automatically as much type changing as possible. For example, if one of the operands of an ADD were an integer and the other a floating-point number, the CBM should automatically convert the integer to be floating (just for the operation, not permanently) or vice versa depending on which is favored. This notion of coercion of types needs to be more general since there may be a number of different types in the CBM. One coercion called Auto/Fetch by Iliffe [I1 68] involves following a chain of accessors until a non-accessor operand is encountered. Such implicit indirect addressing

of operations into classes of similar instructions (arithmetic operations would be such a class), with the routine being invoked with the instruction to be executed as parameter. This reduces the number of routines needed, and the storage necessary to describe a structure class. It would also be desirable for the CBM to handle standard extensions for some classes of operators, if the accessor for the routine for that operator class were left unspecified for some structures. Many structures, when copied, for instance, are simply to be moved element by element to the copy site, without any side effects.

5E3 CBM Instructions

One thing which should be clear from the development of the TFI is that the CBM ought to be at least a stack machine: that is, there is a stack (and possibly more than one) to hold control information, local variables, and intermediate values generated during the evaluation of expressions, and operations to manipulate the stack. However, because of the ability to have variables whose lifetime is not suited to a stack discipline (such as ALLOCATED) the machine should not be just a stack machine, but also allow operations with operands anywhere in memory as well as on the stack, and mixtures of the two.

Thus, the instruction set of a CBM must be able to use operands no matter where they reside. And, if the number of operations is to be kept small, this should be reflected in the type of the operands to an instruction, rather than in the instruction code itself. This is just an extension of the notion that sets have type information associated with them: accessors for sets must tell whether the set is in

The set of operations for a CBM need to be minimal in order to keep the work of building or moving the base small. Having the CBM use type information assists this by allowing a single operation code to handle all interpretations of that action over all its possible operands. Also, doing this implies that operations can operate on entire sets instead of just single values. Combinations of operand types can then yield a number of APL-like operators as primitive in the CBM. Among these are distribution of a scalar over a vector, element by element distribution of an operator over two vectors of equal size, and the "distribution" of a vector over a scalar, by which is meant the APL reduction operation (e.g., $A \leftarrow +/B$).

In order to extend the CBM operations to extended data structures, the CBM needs a way of activating some routine when an instruction is "executed" with such data as an operand. There are a number of ways to do this; we will present three representative methods, ranging from one routine per operation to a single routine for all CBM operations.

The first method would simply require a vector of accessors of routines, one per CBM instruction, to be supplied for each class of extended data (not for each instance of that class). Of course, this means that the CBM must be able to associate a given extended type with a unique vector of routine accessors: this will be true of all the methods presented. At the other extreme, one routine per structure class would be sufficient if it were passed the instruction to be "simulated" when activated; it would then act as a machine-invoked software simulator for operations on that class of extended data.

We prefer an intermediate implementation which divides the set

What is important for the CBM is that the extended data structures be treated in the same manner as hardware primitives by the CBM programs using them. This is done for two reasons:

(1) extensibility in a bootstrapped IPS is a necessity and not a luxury; therefore, extensions should be able to be treated in as normal a manner as possible, just as if they were part of the CBM;

(2) since it is difficult to predict those data structures which will prove crucial in the efficiency of a particular IPS, we must be prepared to subsume non-primitive structures into the CBM, making them primitive and hopefully more efficient. Since this may be done during the useful lifetime of the system, and since many users may have suspended processes which presume to use those structures, whether or not a structure is primitive must not matter to programs operating on it. Indeed, our goal is that the programs be free of such knowledge.

Addresses or pointers are important data in most systems. Following Iliffe's lead, we place a restriction on address-values (which we will call accessors): they may not be operated on by the normal arithmetic and logical operations used for numbers, bit strings, etc. This, of course, means that the CBM must provide a comprehensive set of operations on accessors to make them useful. Such operations include copying accessors, sequencing through the set referenced by an accessor and creating and destroying accessors (which both imply the creation/destruction of a set, since that is what accessors point to). Accessors must also allow inter-process as well as intra-process accessing so that one process may, if necessary, use data or programs in another process.

it, a large amount of memory is wasted when each element of a set has the same form as every other (as is usual in vectors and arrays). Also, even though the BLM data structures are very general, one cannot extend the number of "types" of the machine to include new data structures. This, unfortunately exposes a difference between hardware and software which is unnecessary: machine instructions are not able to act on extensions in the same manner as primitive data and programs must be cognizant of this distinction when accessing or operating on such extensions. Since we wish to be able to extend the basic data types to include structures such as trees for an IPS, this point is one which will be strongly developed in the description of the CBM in Appendix A. The CBM has a structured memory, like Iliffe's BLM, and the structures in that memory are called sets.

5E2 The CBM's Data Structures

As in the BLM, the data in the CBM can be considered as n-tuples of "properties" such as type, length, protection, value, and so on. The properties other than the value-property assist the CBM in using the value of a datum better.

The basic data elements of the CBM are homogeneous sets of primitive values such as numbers, character and bit strings, etc. Another type of set which is needed to describe the structures of an IPS is a heterogeneous set: i.e., one in which the elements are not all of the same type. Some of these heterogeneous sets (called hetero-sets from now on) may be primitive because of the frequency of their use, or any other sufficient reason. Others represent extensions to the basic structures of the CBM.