Calhoun: The NPS Institutional Archive

DSpace Repository

Theses and Dissertations | 1. Thesis and Dissertation Collection, all items

1985

The formal specification of an abstract database: design and implementation.

Zang, Klaus-Harald

Monterey, California. Naval Postgraduate School

https://hdl.handle.net/10945/21348

Downloaded from NPS Archive: Calhoun

http://www.nps.edu/library

Calhoun is the Naval Postgraduate School's public access digital repository for research materials and institutional publications created by the NPS community. Calhoun is named for Professor of Mathematics Guy K. Calhoun, NPS's first appointed -- and published -- scholarly author.

Dudley Knox Library / Naval Postgraduate School
411 Dyer Road / 1 University Circle
Monterey, California USA 93943

DUDLEY KNOX LIBRARY
NAVAL POSTGRADUATE SCHOOL
MONTEREY, CALIFORNIA 93943

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

THE FORMAL SPECIFICATION OF AN ABSTRACT DATABASE: DESIGN AND IMPLEMENTATION

by

Klaus-Harald Zang

December 1985

Thesis Advisor: Daniel L. Davis

Approved for public release; distribution is unlimited

T228065

REPORT DOCUMENTATION PAGE

REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS
SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.
DECLASSIFICATION / DOWNGRADING SCHEDULE	
PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
NAME OF PERFORMING ORGANIZATION aval Postgraduate School 6b. OFFICE SYMBOL (If applicable) Code 52	7a. NAME OF MONITORING ORGANIZATION Naval Postgraduate School
ADDRESS (City, State, and ZIP Code) onterey, California 93943-5100	7b. ADDRESS (City, State, and ZIP Code) Monterey, California 93943-5100
NAME OF FUNDING / SPONSORING ORGANIZATION 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO

TITLE (Include Security Classification)
THE FORMAL SPECIFICATION OF AN ABSTRACT DATABASE: DESIGN AND IMPLEMENTATION

PERSONAL AUTHOR(S)
Zang, Klaus-Harald

a TYPE OF REPORT	13b. TIME COVERED	14 DATE OF REPORT (Year, Month, Day)	15 PAGE COUNT
laster's Thesis	FROM _____ TO _____	1985, December	364

SUPPLEMENTARY NOTATION

COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Abstract Database; Formal Specification

ABSTRACT (Continue on reverse if necessary and identify by block number)

The technique of problem solving abstraction provides an appropriate tool for specifying an interface between the layers of computer hardware and software. Based on this methodology, the types of support and function calls that should be provided to application programs running on micro computers are described with respect to a database resource. The database is integrated with an abstract processor called AM, a

DISTRIBUTION / AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassifed
a. NAME OF RESPONSIBLE INDIVIDUAL rof. Daniel L. Davis	22b TELEPHONE (Include Area Code) (408) 646-3091 22c. OFFICE SYMBOL Code 52Vv

) FORM 1473, 84 MAR 83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE

#19 - ABSTRACT - (CONTINUED)

machine which focuses on eliminating the problems with portability and reusability of software, imposed by insufficient resource abstraction.

Approved for public release; distribution is unlimited.

The Formal Specification of an Abstract Database: Design and Implementation

by

Klaus-Harald Zang
Kapitänleutnant, German Navy

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
December 1985

ABSTRACT

The technique of problem solving abstraction provides an appropriate tool for specifying an interface between the layers of computer hardware and software. Based on this methodology, the types of support and function calls that should be provided to application programs running on micro computers are described with respect to a database resource. The database is integrated with an abstract processor called AM, a machine which focuses on eliminating the problems with portability and reusability of software, imposed by insufficient resource abstraction.

TABLE OF CONTENTS

I. INTRODUCTION ---------- 10

A. THE PORTABILITY PROBLEM ---------- 11

1. Abstraction ---------- 12

2. The Semantic Gap ---------- 14

B. THREE WAYS TO NARROW THE SEMANTIC GAP -------- 15

1. Formalism ---------- 15

2. Representation Independence ---------- 16

3. Intent Expressive Resource Abstraction --- 18

C. METHODOLOGY ---------- 18

II. THEORY ---------- 20

A. ABSTRACT DATA TYPES ---------- 22

B. STRUCTURES ---------- 24

C. ALGEBRAIC SPECIFICATIONS ---------- 27

1. Syntax Part ---------- 28

2. Axiom Part ---------- 30

3. Problems with Algebraic Specifications --- 32

4. Error Handling ---------- 38

III. DESIGN OF THE INTERFACE ---------- 45

A. BASIC DESIGN PRINCIPLES ---------- 45

1. Definition of the Problem ---------- 46

2. Informal Strategy ---------- 47

3. Formalization of the Strategy ---------- 47

B. BASIC DATABASE PRINCIPLES ---------- 50

C. THE ABSTRACT DATABASE RESOURCE ---------- 56

IV. SPECIFICATION METHODOLOGY ---------- 63

A. THE FUNDAMENTAL STRUCTURES ---------- 63

B. DEFINING OBJECTS AND THEIR PROPERTIES -------- 67

C. OBJECTCLASSES ---------- 74

1. Generalization ---------- 77

2. Aggregation ---------- 78

D. THE SPECIFICATION LANGUAGE ---------- 79

1. Grammar ---------- 79

2. The Macro Preprocessor ---------- 83

V. THE DESIGN ---------- 87

A. THE DATABASE CONCEPT ---------- 88

B. ABSTRACT DATABASE DATA TYPES ---------- 91

1. Property-Identification and Value -------- 92

2. Property and Propertyvalue ---------- 93

3. Object and Objectclass ---------- 97

4. Database ---------- 101

C. SPEC PARAMETERIZATION ---------- 108

D. THE LIST STRUCTURE APPLIED TO DATABASE DESIGN ---------- 119

E. LIST RETRIEVAL ---------- 123

1. Background on the Processor Resource ----- 123

2. The Queue ---------- 124

VI. IMPLEMENTATION ---------- 127

A. IMPLEMENTING DATA TYPES ---------- 128

B. MAPPING OPERATORS TO FUNCTIONS ---------- 134

C. ERROR HANDLING ---------- 136

D. EXECUTION ---------- 137

E. DATABASE IMPLEMENTATION ISSUES ---------- 140

VII. CONCLUSIONS ---------- 145

APPENDIX A: A GRAMMAR FOR ALGEBRAIC SPECIFICATIONS ---- 147

APPENDIX B: THE SPECIFICATION FOR AM (VERSION 3.0) ---- 150

APPENDIX C: A SIMPLE ASSEMBLER FOR AM ---------- 270

LIST OF REFERENCES ---------- 361

INITIAL DISTRIBUTION LIST ---------- 363

LIST OF FIGURES

II.1	Syntax Chart	29
II.2	A Simple Specification for the Queue	34
II.3	The Queue Principle	35
II.4	Syntax Part for an Index Queue	35
II.5	A Specification for the Queue Including the State	37
II.6	Illustration of Queue Operations	38
II.7	The Problem with Undefined Operations	42
III.1	Tree Representation of an Object	46
III.2	An Informal Strategy to Attack the Design Problem	48
III.3	Relational Diagram of the Formalized Strategy	49
III.4a	Traditional File Processing Approach	50
III.4b	Database Processing Approach	51
III.5	Architecture of a Database	52
III.6	Simplified Block Diagram of a Database System	55
III.7a	Block Diagram of Database on AM Resource	56
III.7b	Conventional Database System Operation	57
IV.1	Example of an Instance Value	65
IV.2	Creation of Properties	71
IV.3	Creation of Objects	73
IV.4	Objects Forming a Subclass	75
IV.5	Subclasses Forming an Objectclass	76
IV.6a	An Example of Generalization	77

IV.6b Creating a Class from Objects by Generalization - 78

IV.7 Creating an Object from Pairs ---------- 78

IV.8 General Structure of the Database ---------- 80

VI.1 Type Definitions for Natural ---------- 130

VI.2 Machine Values ---------- 132

VI.3 The Physical Resource ---------- 133

VI.4 Operator to Function Mapping for Type BOOL ------ 135

VI.5 Error Handling Routine for Property_id Type ----- 137

VI.6 Program Execution ---------- 138

VI.7 The Semantics for mov_m_m ---------- 139

I. INTRODUCTION

Traditionally, computer software evolved in connection with a particular hardware environment, and often assumed features closely related to characteristics of the underlying hardware. These so-called closed systems usually have a unique set of resources in both hardware and software. However, as systems became more general purpose, the requirement for portability and reusability of resources across systems increased and, consequently, the need for creating greater resource abstraction arose [Ref. 1].

The problem of formalizing the relationship between hardware and software resources was first addressed by Yurchak [Ref. 2] whose efforts resulted in the specification and implementation of an abstract machine, called AM.

New data types necessary to represent the abstraction of a bit-mapped display resource were added to AM by Hunter [Ref. 3] thereby creating AM (version 2.0) as a derivation of version 1.0.

This present research again is an extension of the work begun by Yurchak and Hunter with the goal to design and formally specify a portable, reusable abstract database (version 3.0). Its two major objectives are:

- investigate an appropriate methodology to specify an interface between the layers of computer hardware and software;

- find a way for applying such a methodology in order to describe the interface of a computing system with respect to a database.

The following is a modification of the introduction presented by Yurchak [Ref. 2] and Hunter [Ref. 3], and contains some of their ideas which contributes to a better understanding of the background and motivation for this research.

A. THE PORTABILITY PROBLEM

It is well known that porting large programs from one machine to another is an expensive ordeal. It is also well known that once the software has been moved to the new machine, it is anybody's guess whether or not it will be work as before. Even if our program seems to work, we may find it consumes more resources than we expected. Indeed, this may be just as bad as if it did not work at all.

There are a number of reasons why the portability problem is getting worse, not better:

- most architectures, even those which profess to be "language directed," reflect a bias toward making the machine look like what the programmer wants, or toward some engineering goal, such as maximizing the number of devices;
- both languages and machines are related to the data they manipulate in an implementation dependent way;
- language and hardware designers pursue their conflicting goals to the detriment of the poor compiler writer, who, with imprecise tools and methodologies is faced with the job of implementing ambiguous semantics on an informally designed resource.

Although these and other factors do adversely contribute to the imperfect task of moving software from one machine to

another, they add their weight to other difficult issues in language design, computer architecture, and software engineering. This study confines itself to treating the issues surrounding the interaction between programmer's view of the world as a problem, and the architect's view of the world as a resource.

The existing problem can widely be described as a matter of unsufficient resource abstraction. And there are examples that demonstrate the advantages brought along by consequently applied resource abstraction. For instance, many operating systems (OS) already provide a uniform and functional interface to the file system, and combined with a high level language and its associated runtime services, achieve a high degree of software portability [Ref. 1]. Current research work in the area of database machines indicates attempts to develop a system that would, like the OS, provide a uniform interface, the first step towards portability and reusability. And this trend should give some reason to be a little more optimistic.

1. Abstraction

"Abstraction" describes the separation of the defining properties of an object from other, unnecessary details about it. A programmer is primarily concerned with solving a problem. Appropriately, the tools at his disposal, programming languages, development aids, the programming environment, form a "problem solving abstraction." The hardware (and some of the software) on which this problem solving abstraction is implemented, however, is an abstraction of a different sort.

Addresses, registers, ports, most of the operating system service routines, all provide more or less efficient ways to manipulate the physical resources of the machine, they form a "physical resource abstraction."

The fuzzy area between these two abstractions, sometimes simplistically perceived as the boundary between hardware and software, exposes a number of shortcomings in language design and computer architecture collectively termed the "semantic gap."

As mentioned before, proper resource abstraction plays a major role in the attempt to resolve the portability problem. In areas other than the operating systems, abstraction however, seems to be rather difficult. Processors and visual displays are examples. The inability to establish a meaningful abstraction has impeded the formation of standard functional interfaces to these resources. Operating systems generally do not provide a functional interface to either the processor or the display.[1] Programs which access these resources directly, simply are not portable. High level languages (HLL) partially fill the gap left by OFs for the processor resource. Unfortunately, the interface level is high enough to force many applications to bypass the HLL for efficiency. Special graphics packages that extend the OS provide similar services for the display resource. But despite these efforts, the problem is

---

[1]Except in the most rudimentary way, OS function calls to the display are usually limited to character and string output.

still far from solved. The lack of formal means to specify the interface that the operating systems, high level languages and graphics packages attempt to provide is a serious shortcoming that impedes portability.

The same applies for database systems, too, which although more recently introduced in computer history not only offer a whole set of different and incompatible database models from which to choose, but also force the implementor to adopt the corresponding query language. The latest development is towards the so-called backend approach that reduces the workload of the host computer in a very impressive way by separating the database part from the mainframe and letting it run as autonomously as possible; but the above described interface problem remains unsolved. Thus, at present, the variety that originally was created to optimize a data base to meet the respective goals, strongly interferes with the idea of reusability.

2. The Semantic Gap

The semantic gap manifests itself anywhere a problem solving abstraction touches a physical resource abstraction. A detailed description may be found in Myers [Ref. 4]. He observes that the semantic gap contributes to the cost of software development, software unreliability, inefficiency, complexity, and the distortion of programming languages. Certainly no single development or methodology will eliminate this problem.

Narrowing the semantic gap requires significant changes in the fundamentals of computer architecture and language design. We chose to concentrate on three factors which significantly contribute to this problem:

- informally described semantics;
- representation dependent data types;
- arbitrarily designed instruction set architectures.

The implication, of course, is that through increased formalism, the introduction of representation independent data, and a more throughtful treatment of the instruction set, the semantic gap can be narrowed. The balance of this thesis is devoted to describing a methodology for doing just that.

B. THREE WAYS TO NARROW THE SEMANTIC GAP

1. Formalism

The benefits of formalism in the design process have been amply revealed in countless articles treating this issue from the standpoint of software engineering. Our concern will be limited to formalism as it applies to the specification of an abstraction. Various specification methodologies exist, many of which have been used with more or less success in projects of practical significance. But we caution the reader that by "formal" we mean a mathematical rigor rooted in proven theory. The idea of formalism as often applied to software engineering will not do here. A "formal specification" is a complete description of the meaning of an object. It forms the

basis for an abstraction and is ultimately a bridge over the semantic gap.

The benefits of formalism in which we are most interested are:

- it provides a firm basis for proving our assertions about a specification and its implementation;
- it encourages a discipline on the part of the designer to be rigorously precise;
- it compels us to find ways of describing things which are (implementation) independent.

2. Representation Independence

Conventional machines, in contrast to the AM, force us, as programmers, to develop our own abstractions of data. At a time when we are most concerned with developing clean algorithms the architecture obligates us to worry about status registers and word length. Certainly someone must ultimately deal with these physical properties of the hardware, but this should not fall as an "obligation" upon the programmer. The programmer should be free to ignore unnecessary detail.

Displays are equally difficult. Often the programmer is forced to deal with display data at a very low level. In order to create his display, it may be necessary for him to work at a level of poking bits out the processor port to the terminal. By defining data types that include objects which represent concepts appropriate to visual display processing, the programmer will be freed to work at a higher conceptual level.

Due to the common nature of this problem, with the database we find ourselves in a situation not much simpler than the one just described. Since a single database is usually designed to fulfill only a very specific task, for instance, running the passenger reservation system for an airline, first of all the logical structure of the database to be created must be developed. This can be done using a data structure diagram which contains all the required entities including the relationship among them. But while this step is achievable without consideration of the later implementation, we have to give up this kind of abstraction in the second phase when the diagram is transformed into a design that conforms to the limitations and peculiarities of a given database management system (DBMS). The programs themselves which may be created in parallel with the development of the logical database structure must apply to the standards of the database type chosen, thus putting the programmer in a similar situation as for all conventional machines.

We will attempt to minimize the dependence of data upon its representation through the use of "abstract data types." Our notion of data is very general. It ranges from integers, to image and database objects, and to program instructions. Data type representation will be hidden and abstract operations will be provided in the same way as with traditional abstract data types. If these data types can be kept representation independent, then portability is aided.

3. Intent Expressive Resource Abstraction

Conventional architectures do not permit us to unambiguously express our intent in a program. Artificial data types, combined with typical resource models, force ambiguity and the overloading of data structures. Stack frames are a good example of this. The semantics of the frame combine those of an array and those of a stack. Meanwhile, the whole thing is implemented in memory, with the data types overlaid on an array of fixed length cells.

We claim that applying methods similar to those used to describe abstract data types, we can describe an abstraction of the physical resource of a machine which benefits not only from the formalism used to specify it, but also permits the implementor to clearly interpret the intent of programs written for it.

C. METHODOLOGY

The goal of the research done by Yurchak and Hunter, and now of this thesis is to contribute something of practical significance to the study of software portability by treating an area which has been largely ignored, the design of a formal abstraction for the computing machine itself. We have innumerable high level programming languages, programming environments, graphics languages, database machines (backend processors), query languages, file systems, operating system command interpreters, a whole host of different abstractions tailored to the

task of providing us with just enough information to do everything we need to do, and nothing more. So why, then, have we failed to develop abstractions for the hardware resources, upon which we are so dependent, which are more than just a collection of registers, opcodes and some arbitrary rules about how they interact. A more difficult but certainly more important task than actually defining the abstraction is developing a methodology for producing other resource abstractions.

Our method has been to take a naive approach towards all areas of the design and implementation process not directly related to the specification itself. We do this for two reasons. First, we can take for granted the large body of research in programming languages and computer architecture, we are designing neither a language nor a processor, even though "ad hoc" examples were required to complete the implementation. Second, the research is intended to benefit programmers. Since it is unreasonable to expect those who may use this method to understand the theory behind the specification, the key to understanding the reasons for our design decisions lies in the way we coded it. Thus, cleverness has been eschewed in favor of clarity.

Our task in this thesis, then, is to examine a wide range of issues which impinge on the process of designing and implementing the specification of a database system, and then to describe how we went about actually doing it.

II. THEORY

The formal specification method used to define the Abstract Machine (AM) is based on algebraic semantics. This approach was chosen because algebras, due to their hierarchical structure, enable us to deal with complex problems or to control complex situations by decomposing them into simpler subproblems, with clearly determined interfaces.

Clean interfaces, on the other hand, provide a sound basis for modifying or combining existing programs, which is our intention in advancing AM from version 2.0 to version 3.0. But algebraic semantics also contribute to solve the portability problem for software systems, since they represent a high level of abstraction which is the only promising means to narrowing the semantic gap.

Abstraction allows us to deal with concepts apart from particular instances of those concepts and to concentrate on the essentials only. Abstract data types are the fundamental elements a formal specification is built upon. Guttag [Ref. 5] states that to describe an abstract data type precisely, its specification must comprise both the syntax and the semantics.

According to Guttag, a formal specification should meet the following criteria, which were taken from Hunter [Ref. 3], if it is to be useful:

- it must be restrictive enough to ensure that nothing unacceptable to the specifier will meet the requirements imposed by the specification;
- it must be sufficiently general to ensure that few, if any, acceptable entries are precluded;
- it must be understandable, so people can work with it.

From [Ref. 2], we note that to achieve true portability, we must be able to demonstrate the following properties in our implementation:

- the specified semantics actually implemented on the source machine are completely unambiguous;
- the implementation on the source machine is "correct."

Thus, our method of specifying must be formal enough to permit proofs of correctness. Exhaustive testing, however, especially when dealing with complex specifications, is usually not feasible and so the only true statement we can make is that our specification is correct at least with regard to those tests actually performed.

Algebraic specifications meet the above criteria for achieving true portability. Here we find a significant body of research already in place in the area of abstract data type specification. Goguen [Ref. 6] and Guttag [Ref. 5] treat this topic in great detail. We will not so here. Instead we give an overview of the important concepts of abstract data types, and direct the reader to the original works for more in-depth study of the underlying thoery. Davis [Ref. 7] provides the theoretical basis for the resource specification method.

Davis [Ref. 1] also provides additional background but with an emphasis on practical issues.

A. ABSTRACT DATA TYPES

The underlying principle that gives mathematics the powerful tool for generalization is its intention to specify the fundamental nature of a system by stating only a few basic properties. In order to describe an entire system by means of a small number of its characteristics it is necessary to find out what exactly these characteristics are and how to state them in terms that define all the systems of this particular category. This can be achieved effectively by applying the technique of data abstraction, which is a synonym for the term "abstract data type."

In general, abstract data types refer to the fact that permissible operations on the data ojects are emphasized, while details about the representation of the data objects are suppressed. Because data abstraction focuses mainly on functional properties but ignores unimportant things like representation details, only some of the many possible functional properties have usually to be specified. This allows us to define even complex systems by means of abstract data types, or in other words, via certain sets of data values together with the corresponding sets of primitive operations on those values.

The properties of abstract data types again are specified by algebraic axioms which define all mathematical systems and

provide the basis for deriving additional properties that are implied by the axioms. This will be discussed in more detail later in this chapter.

The stack, queue, or list serve as typical examples to demonstrate how data abstraction works; all of them can be characterized by simply defining the essential operations that may be performed on each individual system. For example, a list could be characterized by operations such as NIL that creates an empty list, FIRST that returns the first member of the list, PREFIX that adds a new member to the front of the list, and NULL that tests for an empty list. Using this approach the programmer need not care about how the list and its manipulating functions are actually implemented, which allows him to concentrate on his programming job. And this principle definition of the list can be used for all systems of type "list" because the description has been kept very general by means of abstraction.

Besides providing us with a very important mechanism for writing well-structured programs, abstract data types make program modifications easy. As long as the interfaces of the manipulating functions remain unchanged, internal details may be modified without affecting other program components in an unwanted way.

Thus, in applying the methodology of data abstraction we are forced to clearly specify our intentions, which in many situations is the stimulus to think intensively about what a

system really is or does, and then to describe the result in an unambiguous manner. Due to its clear style and high level of abstraction, the so-defined system not only is independent of its originator, but is also easier to understand.

B. STRUCTURES

So far we have discussed how to use the mechanism of abstraction in order to handle complex systems but we're primarily concerned with the consideration of the permissible operations on those systems, like the PREFIX operation for lists. In this paragraph we will now examine the more systematic definition of structures. Clocksin [Ref. 8] defines a structure as a single object that consists of a collection of other objects, called components. The components are grouped together to a single structure for convenience in handling them. Thus, a structure can be characterized by the kind of its components and the way they are arranged. Structures are helpful in organizing data since they allow a group of related information to be considered as a single object instead of different ones. The decomposition of data into individual components depends solely on the purpose which is to be achieved in solving the particular problem; so it is up to the programmer to create the structures that contribute most in reaching his intended goal.

To return to our former example, the list, which is a structure itself, what exactly are the kinds of components and their allowable arrangements? For the list as an ordered sequence of

elements the order of the elements in the sequence matters, or as stated in MacLennan [Ref. 9], the allowable arrangements are finite linear orderings. The components of a list may be any terms, including constants, variables, booleans, and lists themselves. Due to these properties lists can represent practically any kind of structure that might be convenient for symbolic computation.

So, for example, if one would like to describe a thing on the basis of certain attributes corresponding to it, the enumeration of these attributes could be considered as creating a list by which this thing can be defined:

thing: $attribute_1$, $attribute_2$,...,$attribute_n$

Bringing the attributes, or components of the list, then into a particular oder and specifying the values they may take, enables us to create classes of related lists. This can be very useful in grouping things of the same type together, as it is the case in all database systems:

class:

$thing_1$: $<attribute_{11}, attribute_{12},...,attribute_{1n}>$

$thing_2$: $<attribute_{21}, attribute_{22},...,attribute_{2n}>$

$\vdots \quad \vdots \quad \vdots \quad \vdots$

$thing_m$: $<attribute_{m1}, attribute_{m2},...,attribute_{mn}>$

This simple example will be discussed in much more detail in the following chapters.

In the previous paragraph we have already determined the primitive operations performed on lists. There are three classes of operations on structures in general:

- constructors which allow us to build a structure in accordance with its predefined characteristics (like PREFIX for lists);
- selectors which allow us to obtain a component from the structure (like FIRST);
- predicates which allow us to determine the arrangement of the components in a given structure and return a boolean value (like NIL).

Depending on the kind of structure one wishes to describe, there is a certain number of operations required; for lists it turns out that just six operations (two for each constructor, selector, and predicate) are sufficient, while, for example, the description of a stack would require only five operations (NEW and PUSH as constructors, POP and TOP as selectors, EMPTY as predicate):

STACK	CONSTRUCTORS	SELECTORS	PREDICATES
empty	NEW	-	EMPTY
nonempty	PUSH	POP	-
		TOP	

But apart from the number of actual operations which is variable, the three fundamental classes of operations remain unchanged for any structure. And there is one more point that should be mentioned: Selectors and constructors invert each other in some cases. For example, for the stack we have

POP(PUSH(stack,X)) = stack

which means that pushing an item X onto a stack and immediately popping it off again leaves the stack unchanged. The same

would be true for

PUSH(POP(stack),Y) = stack

if the same item Y just popped off the stack was immediately pushed back onto the stack again.

But as fine as the inversion of selectors and constructors works for stacks (unrestricted in the case of POP(PUSH)) or for lists, it would not work for queues because items are then added to the rear end while always removed from the front. Keeping these correlations in mind makes it easier to develop our algebraic specification later.

C. ALGEBRAIC SPECIFICATIONS

Specifications are particular and detailed descriptions of things; they contain essential information about the dimensions and peculiarities of the described objects. They furthermore represent a convenient way to describe the generally infinite objects of initial algebras in finite terms if we want to build our abstract data types on an algebraic basis.

Such a specification is also known as "operator signature" and consists of two major components:

- the syntax whose purpose is to define the constants and operations as well as the axioms as intermediate step in developing the semantics.
- the semantics of the data type as part of the realization which is mostly of concern for the implementation.

In mathematical notation a specification can be seen as a triple $<S,\Sigma,\varepsilon>$ where Σ is a S-sorted signature (this means, it

is based on the operands of sort S) and ε is a set of Σ-equations. Here $<S,\Sigma,\varepsilon>$ specifies an abstract data type by defining $T_{\Sigma,\varepsilon}$ which represents an isomorphism class of Σ-algebras.

An algebraic specification represents something between our intuitive sense of what we want and the actual computer code. This characteristic feature facilitates the translation of our ideas into working programs and narrows the scope of possible interpretation by the application programmer. It also allows us to augment an existing data type by introducing so-called derived operators without the need for rewriting the initial operations, or to extend existing specifications to make their data types available to others, thus avoiding unnecessary redundancy.

1. Syntax Part

The definition of an abstract data type itself should tell us all we need to know for using it effectively. But first of all, how can we be sure that we have selected the correct level of abstraction to begin with? MacLennan [Ref. 9] states that there is in fact no formula for calculating such a correct level, it rather depends on our individual view of the world and our needs at the moment. The chosen approach should, however, guarantee that it really models our abstraction of the part of the world we want to describe.

So, in order to start we list the constants and the primitive operations on the data types to be defined and, in doing so, we create the legal ways in which expressions involving these data types can be constructed.

For each primitive operation it is necessary to explicitly state the

- name of the operation;
- type(s) of its argument(s);
- type of value returned;

thereby setting the domain of the arguments and value for each operation:

operation: (argumenttype$_1$,argumenttype$_2$,...,argumenttype$_n$) → valuetype;

As an example, the syntax for integer addition could be written in the following way:

add(integer,integer) → integer;

The syntax chart for the general case which is a modification of the charts usually found in the ADA programming language is shown in Figure II.1.

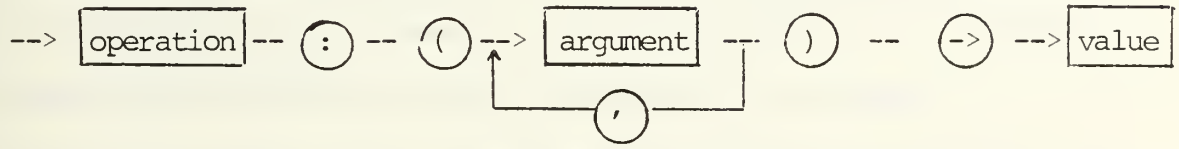


Figure II.1. Syntax Chart

In short, the purpose of specifying the syntax of an abstract data type is to define

1) the legal forms of expressions, and
2) the way in which the constants and operators can be combined into expressions.

Generally, it seems to be wise to build specifications in a bottom-up way, which means starting with the most primitive type of data and then proceeding in gradually reaching a higher level of abstraction on the basis of the more elementary types while hiding the details of the lower levels.

But in spite of this approach, each data type has to be considered independently of its later implementation and must be treated at any given level as if it were itself a primitive one which, on the other hand, could lead to some redundancy.

2. Axiom Part

Having defined the legal (or well-formed) expressions and the types of values they return in a rather symbolic way, we next consider the meaning of those primitive operations, this means, answering the question of what values they in fact compute when given legal inputs. This is done in the axiom or property part of the specification which therefore can be regarded as a refinement of the preceding syntax definition.

Semantics, in general, deal with the actual realization of our formal terms; here we are, in accordance with our intention to use an abstract terminology as far as possible, not yet interested in any programming language that could do the job, but prefer to give a set of mathematical equations, or axioms, which define the meaning of the operations in a way entirely independent of the final implementation. This approach of applying mathematical techniques eases the reasoning about well-formed expressions.

Trying to mathematically specify the axioms we face the problem of what it is that has to be specified and to what extent. As for determining the correct level of abstraction to start with, there is no definite solution to this problem. Basically, we want a set of equations which defines the properties of all well-formed expressions. It is mainly up to the creator of the specification to decide what kind of meaningful interactions between the operators should be included, as long as these equations are complete and define the result of a function for all legal inputs.

For example, the properties for the operation 'integer addition/subtraction'

```
+,- : (integer,integer) --> integer;
```

could be described as follows:

```
a+b = b+a;                          /*commutative*/
a+(b+c) = (a+b)+c;                  /*associative*/
a+0 = a;
a+(-a) = 0;
a-b = a+(-b);
```

The final step then would be to find the minimum complete set of equations, which means listing only those equations absolutely necessary to define the properties of the primitive operations, including implicit statements. But since it is sometimes more convenient to directly state equations which, thinking strictly mathematically, are already contained in other statements, it is legitimate to introduce them as so-called derived operators.

An example of such a derived operator is the boolean function IMPLIES:

IMPLIES(X,Y) = OR(NOT X,Y);

which is equivalent to the combination of two simpler functions containing NOT and OR. By this means we can add new operations and their defining equations to a data type, whenever it is useful within the specification.

3. Problems with Algebraic Specifications

From the preceding paragraphs it should become clear that there is no other way in creating a database for the Abstract Machine except of defining a formal specification first. This gives us the tool to concisely describe our intentions in an unambiguous and rigorous manner. But it also forces us to view the overall problem of what a database actually represents in a strictly mathematical way. This means, we first must determine the primitive names that form the syntactic realm and describe all legal operators. We then must specify the corresponding universe of discourse, which contains our primitive objects, and the functions that map each name from the syntactic domain to its counterpart in the semantic domain.

Although this approach often does not harmonize with our intuition, since the human mind tends to be more liberal than rigorous, it has the distinct advantage of providing us with a clear structure which is easier to understand and where each defined operation can be proved correct. We consider this basic type as the mathematical part of the specification.

This method furthermore has the advantage that, given the operation presently performed and the current state of the machine which corresponds, as Fairley [Ref. 10] states, with ". . . the information required to summarize the status of system entities at any particular point in time, . . . the next state can be determined."

During the research phase for this thesis, however, it became obvious that it often is very difficult to translate relatively simple models of the real world into terms of algebraic expressions. Besides avoiding unwanted inconsistencies the problem centered on the formal requirements for being precise and for specifying the fundamental parts of our database, wherever possible, by stating only the absolutely indispensible basic properties of the system from which all other operations can be derived.

Thus, in the beginning of our work the syntactic specification of the stack operations was studied. But due to its last-in first-out property which is not very helpful for database operations, the stack did not provide the paradigm with which to continue. As a more suitable example that manifested the difficulties in writing an algebraic specification the first-in first-out property of queues was then examined. The specification shown in Figure II.2 is built upon an example given by Fairley [Ref. 10]. In this example (Figure II.2), CREATE and WRITE are serving as constructors that build up or fill the queue. EMPTY and READFRONT describe its behavior.

Syntax:

Operation	Domain	Range
CREATE	( )	--> queue
WRITE	(queue,value)	--> queue
READFRONT	(queue)	--> value
DELETE	(queue)	--> queue
EMPTY	(queue)	--> boolean

Axioms:

EMPTY(CREATE) = true

EMPTY(WRITE(queue,value)) = false

READFRONT(CREATE) = error

DELETE(CREATE) = error

DELETE(WRITE(queue,value)) =
{ if EMPTY(queue) then CREAT
{ else
{ WRITE(DELETE(queue),value)

READFRONT(WRITE(queue,value)) =
{ if EMPTY(queue) then value
{ else
{ READFRONT(queue)

Figure II.2. A Simple Specification for the Queue

DELETE acts as a modifier to the queue. While READFRONT always returns a copy of the value sitting in the front position, the operation DELETE actually removes this value from the queue, thereby changing the state of the system.

The problem encountered here is that we are not able to clearly define all of the axioms but instead are forced to make some concessions in accordance with the prevailing state of the queue. This is caused by the fact that the value presently read from the queue is not necessarily the one entered last (compare Figure II.3).

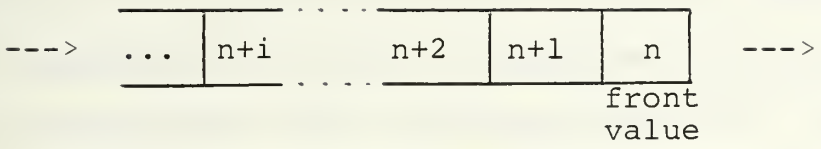


Figure II.3. The Queue Principle

To remain consistent it is therefore necessary to introduce the conditional "if-else" statement for a part of the axioms which, on the other hand, takes away some of the simplicity. Compared with the stack operations where it is sufficient to increase or decrease a counting device by a number of 1 with each item added to or subtracted from the stack, the queue operations require a more sophisticated mechanism to keep track of every single item in the queue, if one really wants to be precise. This precision could be achieved by indexing the items when inserted into the queue. The syntax for these operations is easily defined, as shown in in Figure II.4.

Syntax:

Operation	Domain		Range
CREATE	( )	-->	queue
WRITE	(queue,value$_i$)	-->	queue
READFRONT	(queue)	-->	value$_j$
DELETE	(queue)	-->	queue
EMPTY	(queue)	-->	boolean

Figure II.4. Syntax Part for an Index Queue

But since we have introduced the index values "$value_i$" and "$value_j$,"we now must describe their particular properties in the axiom part. This obviously is a complicated method of describing the relatively simple arrangement of a queue in terms of an algebraic specification, and also proves to be a rather difficult effort. But to our knowledge, there is no really elegant solution to this problem available at the present time.

The previously mentioned indexing methodology can only be simplified by reducing the queue to its basic operations in a similar way as done with the abstract specifications for the natural numbers or integers. Here, for example, the number 1 is expressed by the paraphrase "succnat(zeronat())," and each succeeding number can be described by adding just another "succnat" in front of this fundamental expression. Applying this technique to the queue problem allows us to keep track of every single value written to the queue and read or deleted from it.

So instead of introducing the extra indexing operation, we could integrate the state of the queue ('quaddr') and modify the specification as demonstrated in Figure II.5. On the basis of these few axioms it becomes possible to describe each state of the queue by an appropriate combination of the given operations. The following is an example of this:

```
READFRONT(WRITE(WRITE(INITIALIZE(),v1),v2)) = v1;    /*v:value*
READFRONT(DELETE(WRITE(WRITE(INITIALIZE(),v1),v2))) = v2;
READFRONT(DELETE(WRITE(INITIALIZE(),v))) = undef;
READFRONT(WRITE(DELETE(WRITE(INITIALIZE(),v1)),v2)) = v2;
```

Syntax:

Operation	Domain	Range
INITIALIZE	(qaddr,state)	--> state
READFRONT	(qaddr,state)	--> value
WRITE	(value,qaddr,state)	--> state
DELETE	(qaddr,state)	--> state

Axioms:

```
READFRONT(qaddr,INITIALIZE(qaddr,state)) = undef;
READFRONT(qaddr,WRITE(value,qaddr,INITIALIZE(qaddr,state)))
            = value;
DELETE(qaddr,INITIALIZE(qaddr,state)) = undef;
DELETE(qaddr,WRITE(value,qaddr,INITIALIZE())) = INITIALIZE();
DELETE(qaddr,WRITE(value,qaddr,state)) =
            WRITE(value,qaddr,DELETE(qaddr,state));
if not (INITIALIZE())
then READFRONT(qaddr,WRITE(value,qaddr,state)) =
             READFRONT(qaddr,state);
endif;
```

Figure II.5. A Specification for the Queue Including the State

From these examples, where we omitted the arguments "qaddr" and "state" in favor of clearness, the importance of placing the parentheses into the correct locations becomes obvious. The previously defined operations are illustrated in Figure II.6. A value different from the one residing in the front position can only be read from the queue after the front value has been deleted, so a

```
READFRONT(READFRONT(WRITE(INITIALIZE(),v)))
```

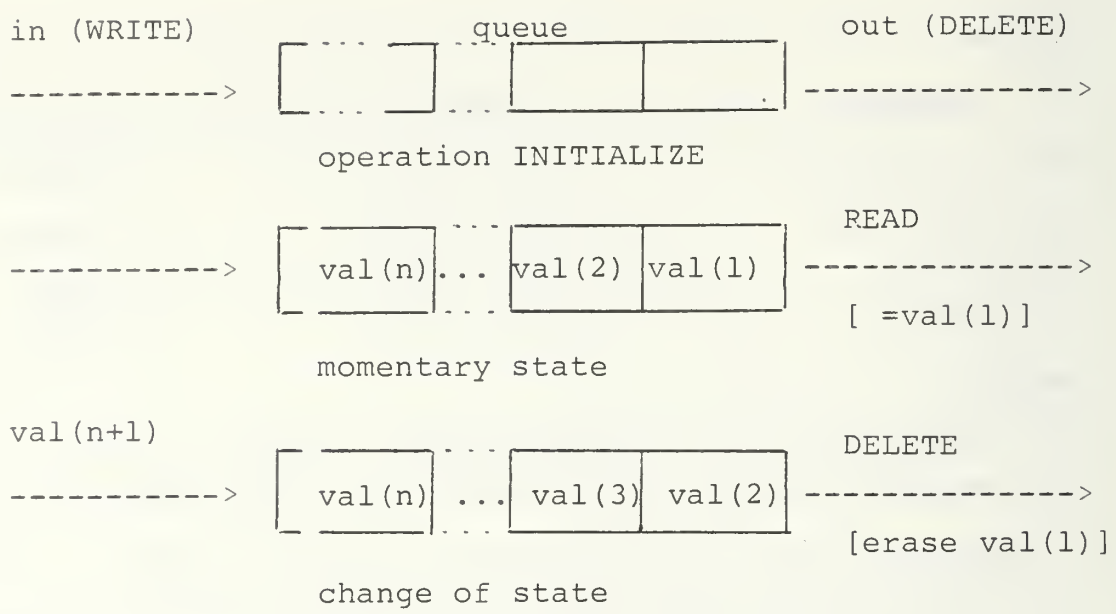


Figure II.6. Illustration of Queue Operations

would return two times the value "v", which is similar to the corresponding TOP operation for the stack. Although by this method some of the axioms and the "if-then-else" statements could be eliminated and furthermore, the requirements for an abstract specification can be satisfied, it does not seem to be an elegant solution to the queue problem either. The reason for integrating this deviation into our research work was to give the reader a better understanding of the problems that had to be managed in writing a formal specification for a database, which is not as simple as the queue.

4. Error Handling

A major aspect in creating a specification is how to deal with the situation should the user manipulate the defined

operations in a way that would result in an error. Because it is part of the human nature to fail once in a while, vital systems tend to be equipped with exception handling mechanisms which prevent the overall system from breaking down and becoming worthless, should a predefined type of error occur. This is also known as fault tolerance or lenience, and represents the opposite to the more mathematically sound term "strict," where a function becomes undefined whenever one or more of its arguments are undefined.

Error detection always causes a great deal of problems, and once having been successful it is a rather philosophical question of how to proceed, as long as a collapse of the entire system can be avoided and we get the information that a certain error has occurred. In general, every attempt to handle this problem should be based on the understanding that

- any operation which encounters an error is computationally meaningless;
- if an operation encounters an error, then any subsequent operation which utilizes the erroneous result must also return an error;
- errors must not be hidden, but must be known to the user.

These statements were directly taken from Hunter [Ref. 3], since he considerably modified the error handling routines for the Abstract Machine (version 2.0), which will be discussed later in this section.

One interesting but mathematically not indisputable approach was introduced by Guttag [Ref. 5] with the term

"undefined" for equations whose values were not determinable, for example, when attempting to read an item from the empty stack

READ(CREATE) = UNDEFINED.

But since the operation READ can only return a value from the stack, we either have to treat UNDEFINED itself as a value or READ as an only partially defined function.

In the meantime, this problem was solved by Davis [Ref. 1]; his method has been applied to the AM specification and will be described below.

Another approach that was used by Yurchak [Ref. 2] for the initial AM (version 1.0) is to modify the specification and include an error message. However, it soon became obvious, that in adding such an error message to the specification, care has to be taken of all the possible combinations the newly created error message could be involved in. This is in accordance with the above listed understanding that, if we get an error, any operation on it yields an error, too. But it also means that the number of additionally needed error axioms quickly leads to an extent which is no longer reasonable.

To get an idea what dimensions we easily reach in order to remain consistent, we only have to consider the part of the specification for natural numbers that deals with the special error axioms:

```
PREDNAT(ZERONAT() )            = NATERROR;
PREDNAT(NATERROR)              = NATERROR;
SUCCNAT(NATERROR)              = NATERROR:
SUMNAT(N,NATERROR)             = NATERROR;
SUMNAT(NATERROR,N)             = NATERROR;
SUMNAT(SUCCNAT(M),NATERROR)    = NATERROR;
SUBNAT(N,NATERROR)             = NATERROR;
SUBNAT(NATERROR,N)             = NATERROR;
MLTNAT(NATERROR,X)             = NATERROR;
```

where NATERROR would have to be specified as the extra operator class

```
ERROR
OP
     NATERROR: --> NAT.
```

Although these error axioms would reduce any term containing an error to the error message of the appropriate sort, thus eliminating unwanted elements of the carrier of sort NAT in the above example, it is obviously not practical to follow this approach. We therefore succeed with the concept of "undefined" as introduced by Davis [Ref. 1] and described by Hunter [Ref. 3], since this method allows us to keep the number of additional axioms manageably low.

The underlying principle is just a different way of viewing the mapping of elements from a given domain to their images using a certain function. For example, if we let f be a function from A to B and let A' be a subset of the domain A, then f(A') denotes a subset of B, the image of A' under f. We now consider A' as the domain of our constants and operations

defined in the syntax part of the specification. Furthermore, we are interested only in the corresponding values they are mapped to by the function f (see Figure II.7), while ignoring all the undefined operations in the set A-A', or in other words, the attempt of mapping an element from the undefined set results in an undefined value.

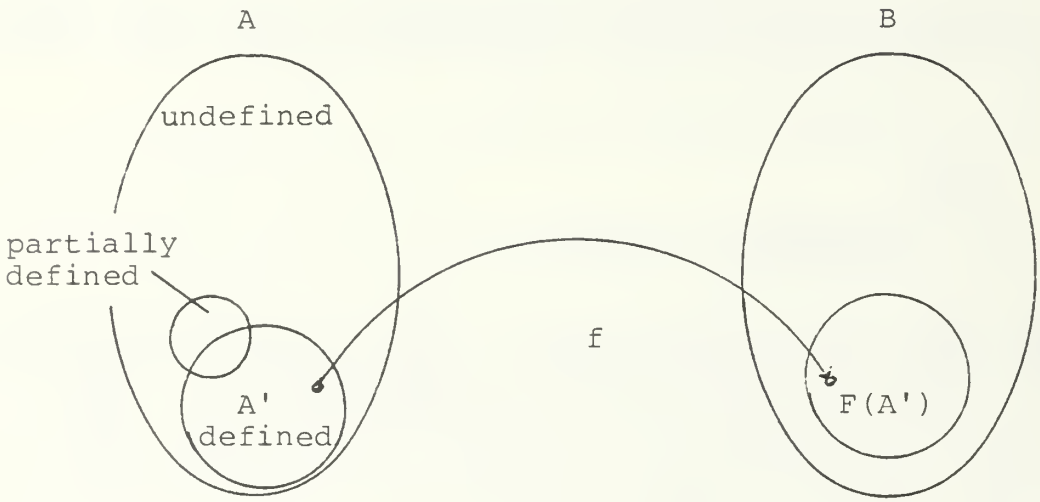


Figure II.7. The Problem with Undefined Operations

"Undefined" has the following properties:

- undef is used to describe the illegal operations;
- if t = undef then $A(x_1,x_2,...,t,...,x_n)$ = undef;

where "A" is any operator in the specification, and "x_i" is an expression;

- any equation containing undef is equivalent to undef;
- in a realization, if undef is encountered, the processing halts immediately and an appropriate error message is given.

So, instead of listing every single possible NATERROR, all we have to do is add the axiom

```
PREDNAT(ZERONAT() ) = undef;
```

to our specification of the natural numbers, which served only as an example for the general case, thereby solving the "predecessor of zero" problem without the introduction of a special error operator. The effect is to restrict the range of free variables that apply to an axiom. This becomes clearer if we look at the following construction, where we substitute ZERONAT() for the free variable n:

```
SUCCNAT(PREDNAT(n)) = n;          /*axiom to start with*/
now replace n by ZERONAT()
SUCCNAT(PREDNAT(ZERONAT() )) = ZERONAT();
SUCCNAT(undef) = ZERONAT();
undef = ZERONAT();
```

The evaluation of this axiom shows that substituting ZERONAT() for n leads to an undefined result, which is quite correct, but returns the appropriate value for all free variables otherwise.

Thus, PREDNAT(n) does not exactly belong to the set of constants and operations defined in A' because for some cases, or precisely when n is replaced by ZERONAT(), the value it maps to becomes undefined. We therefore say PREDNAT(n) is only partially defined and must be seen, similar to the POP operation for stacks as a member of the overlapping set of partially defined operations (Figure II.7).

Analogously, we receive a similar result for all those specified data types where a faulty user action may turn out as something undefined, like trying to read a value from the empty stack or queue. And since this method has already been applied to the Abstract Machine (AM) by Hunter, this is one more reason to continue with it.

III. DESIGN OF THE INTERFACE

Data that is stored more or less permanently to be manipulated by a computer resource represents in some way a simple database, where the software which uses or modifies these data is known as the database management system (DBMS). The difference that makes a database superior to file processing systems which can be considered as predecessors of the database and will be widely replaced by this newer technology, is their capability to provide via the DBMS more information from a given amount of data. The DBMS itself is a complex and usually large program that acts as a data librarian. If we follow the approach presented by Bjorner [Ref. 11], we can view the DBMS as the realization of a certain database model; this then allows us to divide the further treatment into the two parts abstraction and realization, which correspond to the database model and the database management system.

A. BASIC DESIGN PRINCIPLES

In this part of our work we are not yet concerned about the realization but rather concentrate on the problem of how to formulate a useful abstraction of the database resource. As seen, the desired state of abstraction can be reached by extracting the fundamental properties of the object of concern on a level which suits our intentions best. A way to do this

is to look briefly at the design methodology described by Booch [ref. 12] that works for the general case.

1. Definition of the Problem

To get started and to make the overall framework of a database more understandable, models are helpful tools because they enable us to express relatively complex things in a simpler and more evident way. A tree is often used to represent the data structure and the relationship between different members of the database. One of the typical requirements for a database system is to find a certain property among stored objects.

For example, an object with the properties A,B,C,D could be described by a simple tree (Figure III.1), where the object is represented by the root and the properties by the leaves of the tree.

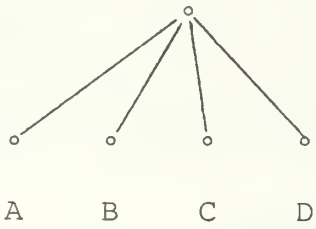


Figure III.1. Tree Representation of an Object

In combining different objects we can create very complicated trees (also called a hierarchy) which represent the relationship in a clear way that otherwise would be difficult to describe. Our goal now is to develop a system that checks the

leaves of the tree for a given condition which could be considered as a step towards the often applied function of searching for a particular object in a database.

2. Informal Strategy

While still ignoring the question whether such a tree will actually be represented as a sequential, linked, or inverted list in order to see if a certain condition is satisfied by any of the objects, we can apply the following informal strategy:

Find all stored members that satisfy condition B.

Figure III.2 gives an example how this informal strategy may be used. Thus, having shown how the informal method works, the next step is to find a way of formally expressing this strategy.

3. Formalization of the Strategy

First we need to identify the objects and their properties. To do this we look one more time at the informal part: Find all stored members that satisfy condition B. It is one of the advantages of the English language that an object is always represented by a noun, while adjectives describe the properties of an object. In our simple example this gives us right away 'member' as the object and 'condition B' as the required property. Next we have to identify the operation performed on the object which is not difficult either since action is described by verbs. In the discussed example, 'find' is the one looked for. Given the objects, properties, and the

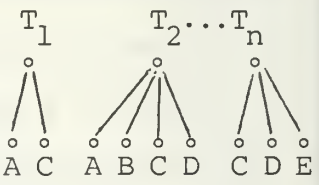
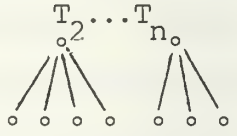
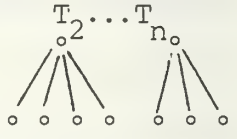
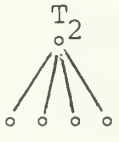
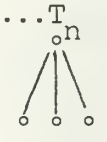
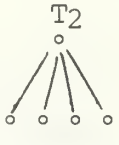
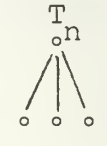
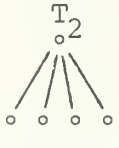
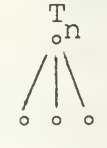
	comparand	comparator	indication	storage
initially			0	
set comparand to B and take first tree from storage	B		0	
check for condition B	B		0	
since no match get next tree from storage	B		0	
check for condition B	B		0	
since match set indication	B		1	
reset indication and get next tree from storage	B		0	
since no match get next tree	B		0	
since last tree halt				

Figure III.2. An Informal Strategy to Attack the Design Problem

operations we may perform on them, we now can describe the sequence of single steps necessary to 'Find all stored members that satisfy condition B' by the relational diagram shown in Figure III.3.

Given this design, we could continue and specify the interface of each box presented in Figure III.3. To complete the formalization part this would, indeed, be necessary. But we then would also find ourselves right in the middle of the implementation which is not what we want at this time. We therefore stop here short of coming up with a real formalization. The general idea about the possibilities to start abstracting the resource however should be a little more obvious by now.

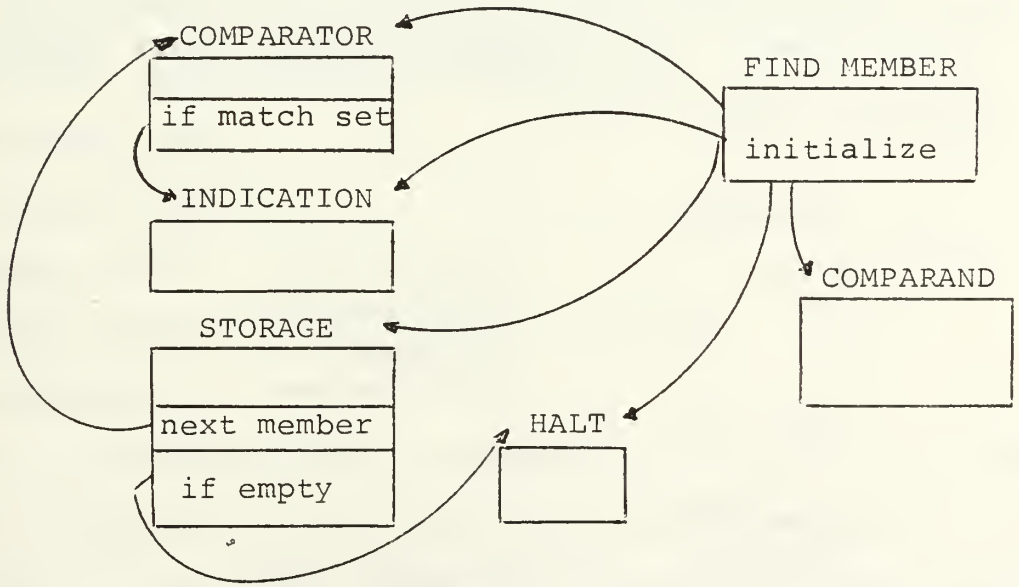


Figure III.3. Relational Diagram of the Formalized Strategy

B. BASIC DATABASE PRINCIPLES

Before we can start specifying the database resource we need some understanding of the fundamental principles every database, in general, is built upon. Although over the recent years there were various approaches in this area to make database work more efficiently, such as the introduction of the database machine (also known as 'backend'), or the continuing research on the multi-backend database headed by Professor Hsiao at the Naval Postgraduate School, we consider our database as an integrated part of the abstract machine built by extending the present AM. So, in order to formulate a useful abstraction we have to keep it simple and therefore, are interested only in the conventional, single user type of database.

A database is in fact nothing more than an elegant combination of several file processing systems under the control of the DBMS (Figure III.4).

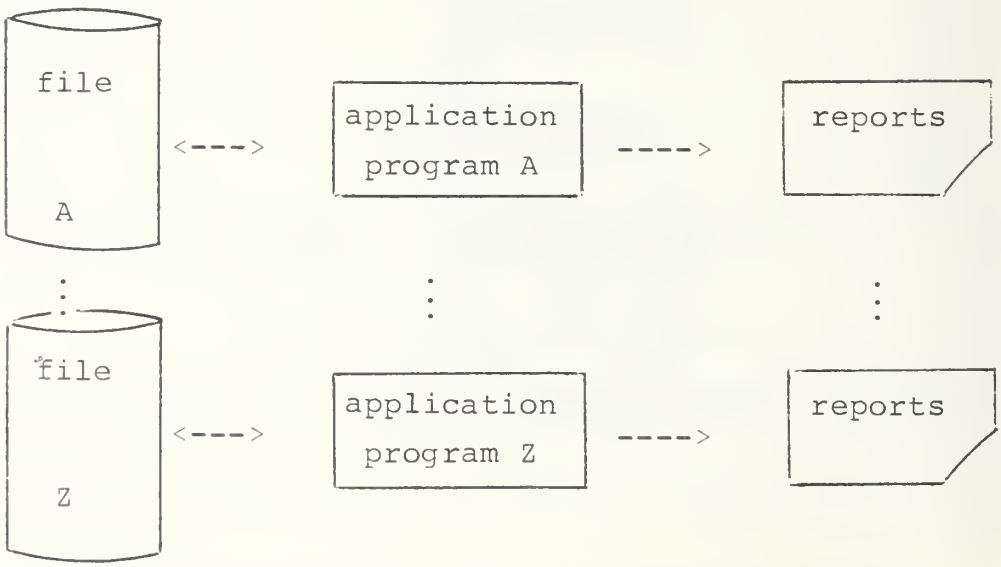


Figure III.4a. Traditional File Processing Approach

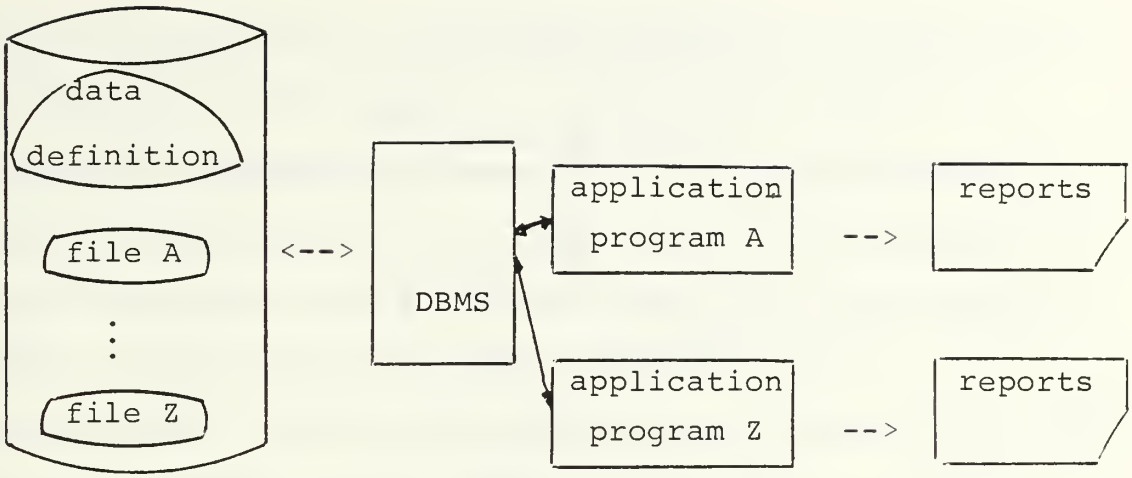


Figure III.4b. Database Processing Approach

For a general view of a database structure we refer to the architectural description presented by Deen [Ref. 13]. It is always a major question how to define and treat data that has to be processed by a machine. In the case of a database there are as many as five distinct levels data can be viewed from, namely, in a bottom-up fashion, the

- physical level
- storage level
- global level
- local level
- user level

The meaning of these levels (Figure III.5) is easier to understand if we start with the global level which represents the center part. This level refers to the overall logical description of the entire database without considering its storage representation. It shows the logical relationship among the

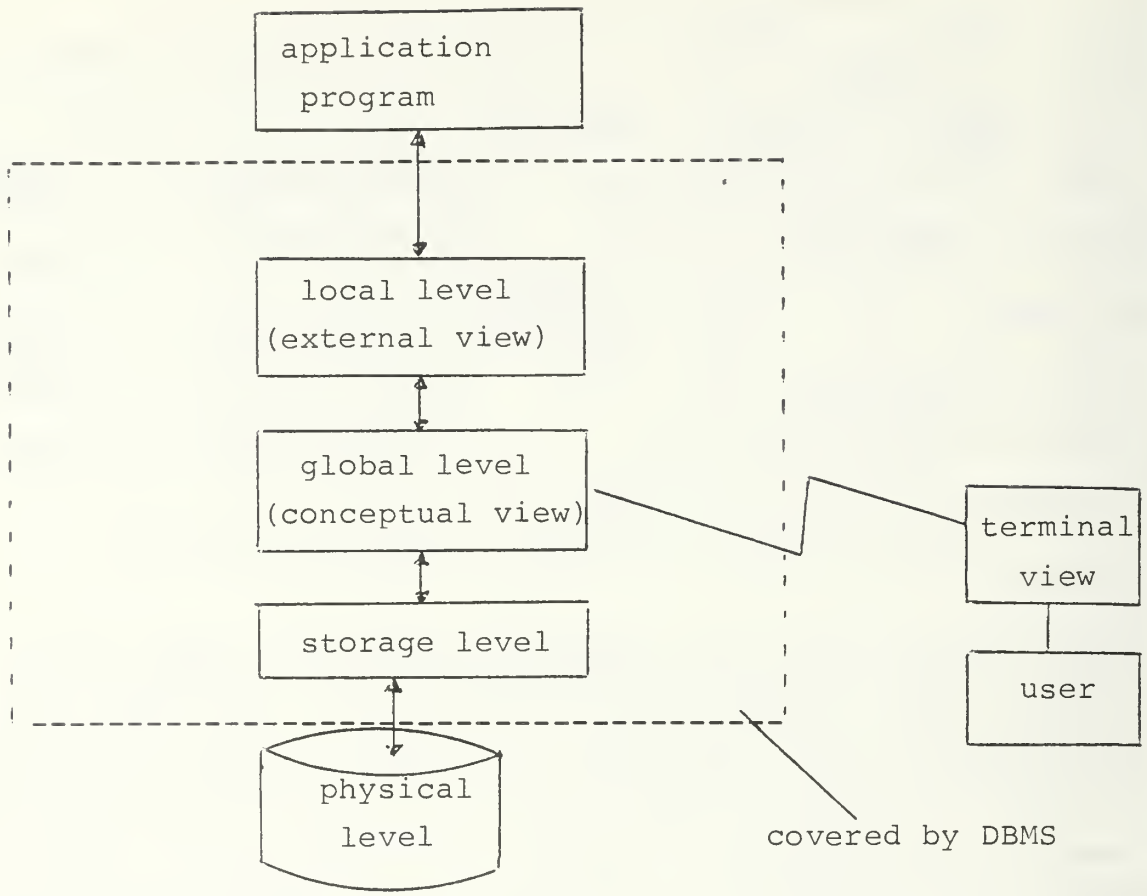


Figure III.5. Architecture of a Database: The Five Levels for Viewing Data

objects of the database and gives the conceptual view of the system.

The local level provides us with a subset of the database described at the global level. It was introduced with the intention to save the application programmer the inconvenience of invoking the whole global scheme while he usually is only interested in a few specific data items, since his need is local and his view is partial. So a subset is the application programmer's view of the database. It is a logical description

of the part in which he is interested, and therefore represents the external view of the system.

How the data should be organized for storage in the physical device is specified at the storage level which consists of entries for overflows, physical block sizes, and data placement techniques. Access paths can also be specified here.

Whereas global and local level are logical descriptions, the physical level is the physical database itself. The database is stored on physical devices in conformity with the specification of the particular method applied at the storage level, where the purpose of this method is to optimize the overall performance of the database whose logical description is given at the global level. The fifth level is the view of the database as seen by the end user from a remote terminal using a special, high level query language.

Since the view of data at the user level as well as the physical level are not of particular interest for our research, we shall exclude it from subsequent discussions, concentrating only on the application program, the storage, and the conceptual (global) and external (local) view. The application program is stored permanently and is always available to the user. It is usually written once, or maybe a few times, should the database description be changed, and can be invoked by special commands. To manipulate data in the database by the application program a sublanguage, the so-called query or data manipulation language (DML) is needed, one for each host language.

The DML acts as an interface language with the database which enables the application programmer to "navigate" through the database with a search strategy defined by the logical relationships of his data at the local level. An application program containing DML statements has to be compiled either by an extended host language compiler or by a special DML processor followed by a host language compiler.

In contrast to a typical compiler however, this device, the first of three major software pieces, also known as query processor, does not generate machine language but rather a sequence of commands that are passed to other parts of the database management system. The query processor needs to know about the structure of the database, so it can interpret special terms in the context of the particular system. This information about the database may already be built into the query processor itself.

The output of the query processor is fed into the database manager (Figure III.6), where it is translated into terms the third software component in our simplified database blockdiagram, the file manager, can understand, which means, into operations on files rather than on the more abstract data structures of the database description (global level). The file manager may be the general purpose file system provided by the underlying operating system (OS), or it may be a specialized file system that knows about the particular way in which the data is stored in the database.

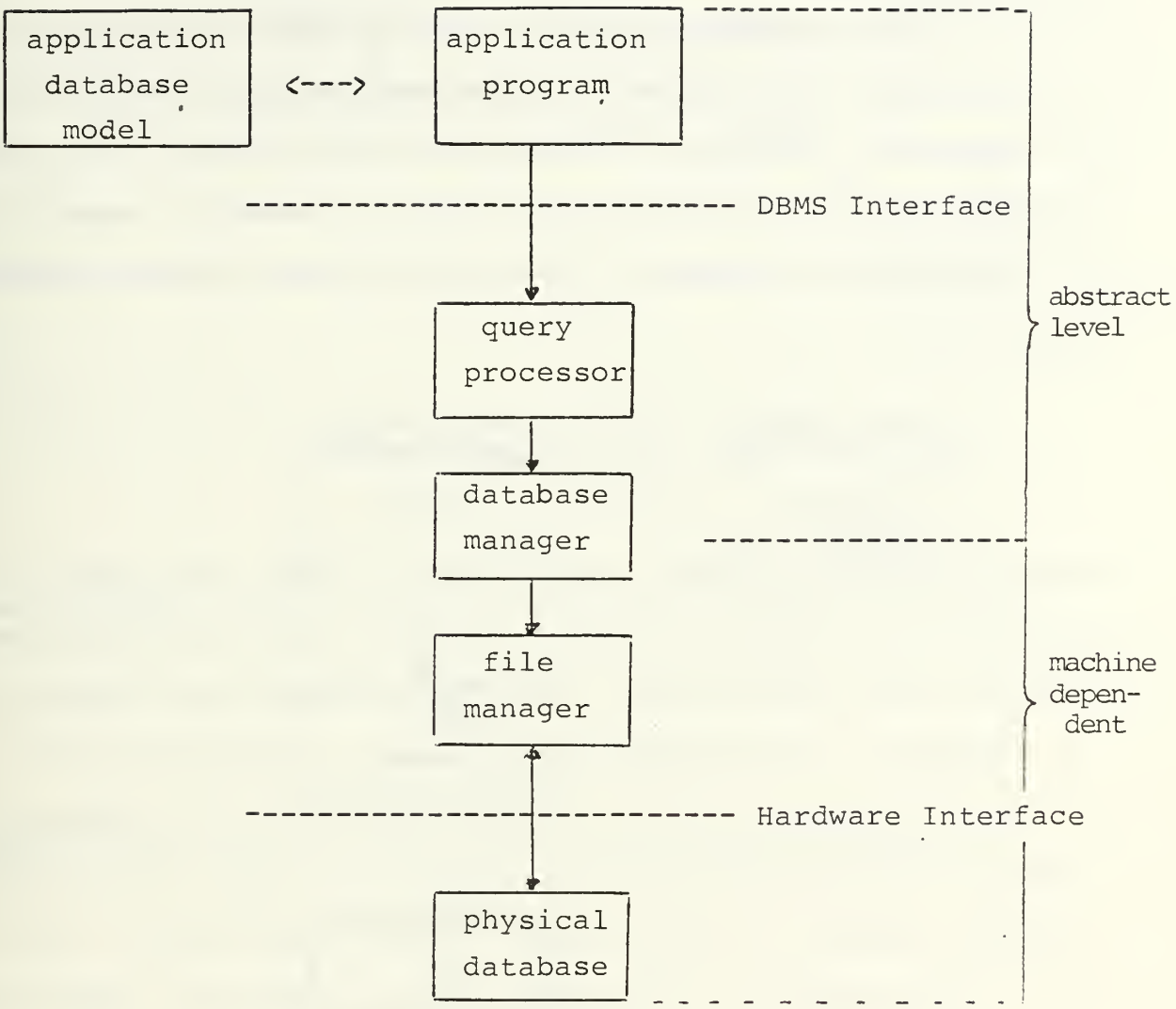


Figure III.6. Simplified Block Diagram of a Database System

The overall software that permits the use or modification of the data stored is a DBMS which basically covers all the software components we are mostly interested in. So this is the point in the system where the abstract database resource comes in.

C. THE ABSTRACT DATABASE RESOURCE

The AM database resource replaces two of the items shown in Figure III.6, namely the database manager and the file manager. Figure III.7a shows the block diagram for a database system using AM, while Figure III.7b presents a general view of the AM arrangement within the database system operation.

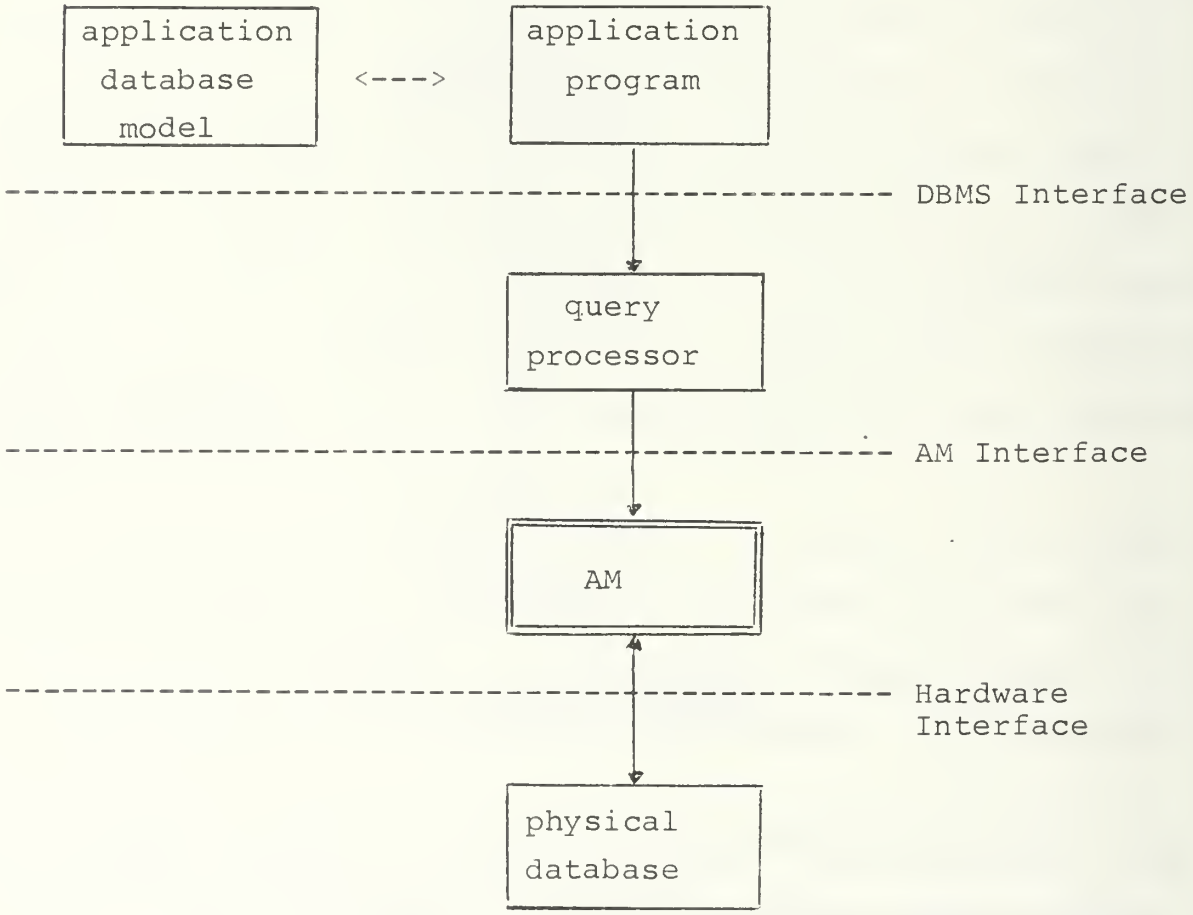


Figure III.7a. Block Diagram of Database on AM Resource

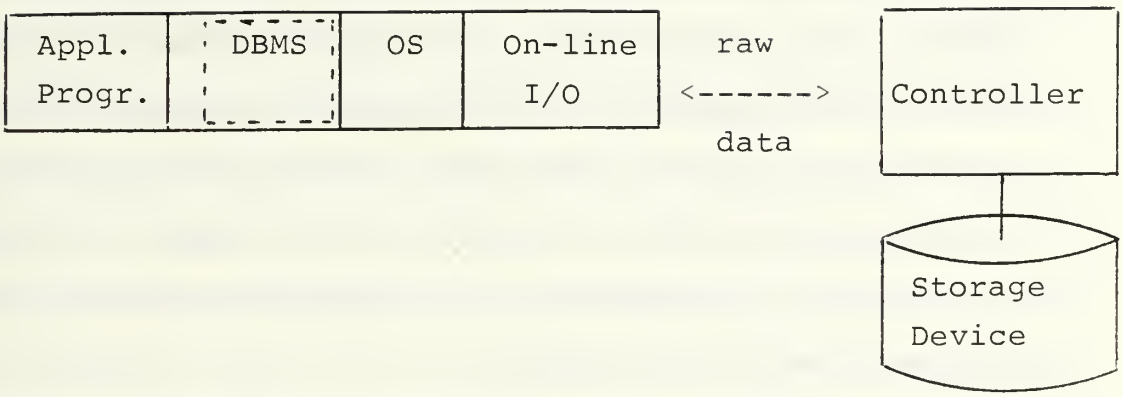


Figure III.7b. Conventional Database System Operation
(Compare Hsiao [Ref. 14])

So AM takes over right at the boundary where the system changes from working on a more abstract basis to an operation depending on the particular machine applied. Since AM does not include the query processor, it can not be considered as a complete replacement of the DBMS but instead covers only about two-thirds of the functions carried out by the DBMS before (Figure III.7b). We further recognize that we did not mention another specialized language, the data definition language (DDL), which in conventional systems serves as the tool to describe the entire database once the conceptual scheme is specified. However, to keep it as simple and clear as possible we do not want to introduce one more high level language, and therefore prefer to let this job be done by a special application program, using the query processor as mediator. Since installing the database is a one time matter this restriction seems feasible.

AM's database primitives are more low level and cover less aspects than, for example, the CODASYL[2] model. But they contain all the fundamental features for creating the database, for updating a given value, for inserting new elements into the database, and for retrieving special data. These operations are considered appropriate for this abstract level, and will be discussed in a later chapter.

In a similar way as for the conventional model, the database manager as the AM interface, which in fact is just a collection of routines, receives as its input the processed query primitives in a still machine independent form. This, however, is the only kind of input the database manager accepts because, since the former DDL entries are now handled by a special application program, there is no need for the manager to have a second input line. It further should be noticed that the security aspect, which means permitting access to certain information stored in the database to authorized persons only, as a task frequently carried out by the manager, has not been taken into account. This step can be justified with the definition of our database as a single user system. Thus, the general idea is to model the database resource 'on top' of the existing AM (version 2.0) by abandoning all of the usually required high level languages like DML and DDL, in order to level the resources with the AM operations, which is the major step for

---

[2] Conference on Data Systems Languages.

eliminating the semantic gap. But in contrast to the display resource defined by Hunter, AM is not entirely able to operate on the machine independent parts created by a conventional database system,as long as a separate DDL input is involved.

When dealing with the physical resource the question to be asked first is what purpose it is supposed to fulfill. From the programmer's point of view the database should enable him to:

- create objects, characterized by particular properties;
- connect the objects in a logical way to a file;
- store the objects without consideration of the physical storage method;
- operate on any of the stored objects in an uncomplicated manner.

In addition to the standard file system where the basic operations 'open', 'close', 'read', and 'write' will do, a database must also permit operations for retrieving a distinct object from the storage, modifying it or checking if a particular object is stored at all. The details will be described in the next chapter, but in general, the database represents the state of all the data stored as files and can be considered as 'just another' resource for AM.

As mentioned before, the programmer has to start his work with the creation of the logical concept tailored for the very special kind of database he intends to build. By this logical design, or model, he copies that certain portion of the real world which describes his view of selected activities best.

But as with every model, its capacity is limited, not all aspects can be covered. Thus, a careful selection of those portions which allow the required logical operations is necessary. The tool for compressing the parts of the real world in such a way that they can be stored within the database is the technique of Aggregation by forming a concept via abstracting a relationship between other concepts, called components, and Generalization by forming a concept via abstracting a class of other concepts, called categories. These techniques will be discussed in the next chapter.

Due to the compression however, some questions now become unanswerable. It is the task of the database designer to make sure that those questions which can no longer be answered are of the kind that never will be asked.

The standard primitives in the real world are the objects and certain properties, where objects as already stated, can be represented by nouns while properties can be considered as adjectives that characterize the objects. We will stick to this notation concerning the primitives throughout this paper because their meaning is more evident and they represent the most abstract level. This is one of the major differences to a conventional database where widely a different terminology is used. The reason for this is that in a database we actually can not work with the real world primitives since a model is not the real world itself. But instead we are working with representations of these primitives. So whenver a transaction

from the real world into the conceptual world takes place, the notation is changed to indicate this step. In a conventional design the objects are represented by the so-called entities, which are, in contrast to their physical implementation, still unrestricted by the constraints of the computer. Properties, in a similar way, are represented by so-called attributes which serve as a description for the entities and, while properties are characteristics of an object, attributes are representations of those characteristics. Thus, attributes are the characteristics of the data types (objects) themselves and, in fact, every entity has certain named attributes.

But as stated above, we do not follow this terminological excursion for the sake of staying as abstract and representation independent as possible. The intention finally, is to keep the structure simple, with emphasis on the permissible operations and to prevent the programmer from leaving the path of unambiguity.

In AM, database objects are abstract data types. Conceptually, database operation is accomplished in the following way. Objects are initially brought in from the disk and stored in main memory. To manipulate an object, it is first fetched from its memory location. It is then used as an operand in some database operation, and the resultant object is stored back into memory. At any instance, the memory may contain several objects, but the terminal is directed to view only some selected object(s) in accordance with the operation just

being performed. When these operations finally are completed, the objects temporarily residing in main memory are shifted back onto the disk under control of the operating system, which is not to be discussed here.

IV. SPECIFICATION METHODOLOGY

Because our database is considered as an extension of the existing AM we continue the work originated by Yurchak [Ref. 2] using essentially the same specification language which will be described later in this chapter. However, before we proceed with this, first some understanding is required about the approach we took in adding the database to AM. The purpose of the next section is therefore to make the reader familiar with the special methodology applied in order to design this resource, and the chronological steps that were done until finally the specification could be developed.

A. THE FUNDAMENTAL STRUCTURES

To define the operations that legally can be performed on the abstract data types for our database we need some tools to describe our intentions and also to preserve the necessary level of abstraction. Since the complexity of a database is not easily understood, a data model usually is formed as the simplified representation of a particular aspect of reality. In doing so, the questions that arise next are: what are the elements our model will be based on, what actually is it that we would like to represent in a database, or what are the specific aspects of the real world we are mostly interested in? Without claiming to be absolutely correct in the philosophical view of

things, the point to begin with is the fundamental structure, known as the primitives in the real world.

The first phenomenon here is the object, which may be a thing, a person, an event, an instruction or, in general, something solid that can be seen or touched. When objects can be put together under a common but more generalized notation, they may form an object class. But it must be mentioned that grouping distinct objects together is only achievable by ignoring their differences at the price of losing some specific information as a concession to the generalization.

The characteristic qualities owned by an object are its properties. For the above given examples, it might be the size of a thing, the name of a person, the date of an event, or the statement contained in an instruction. All the possible instances of a property again can be grouped together into a set defining the domain of all the values this property legally may take, which therefore will be called 'valueset' in our specification. This domain represents not just a collection of numbers or characters but instead has to be considered as the set of all values a given property can have. For example, for the object 'person' with the property 'name' the corresponding valueset would be the collection of all the names that might be found among people on earth.

An example of how objects and properties are related to each other is given in Figure IV.1, which is based on concept developed by Kroenke [Ref. 15:p. 207].

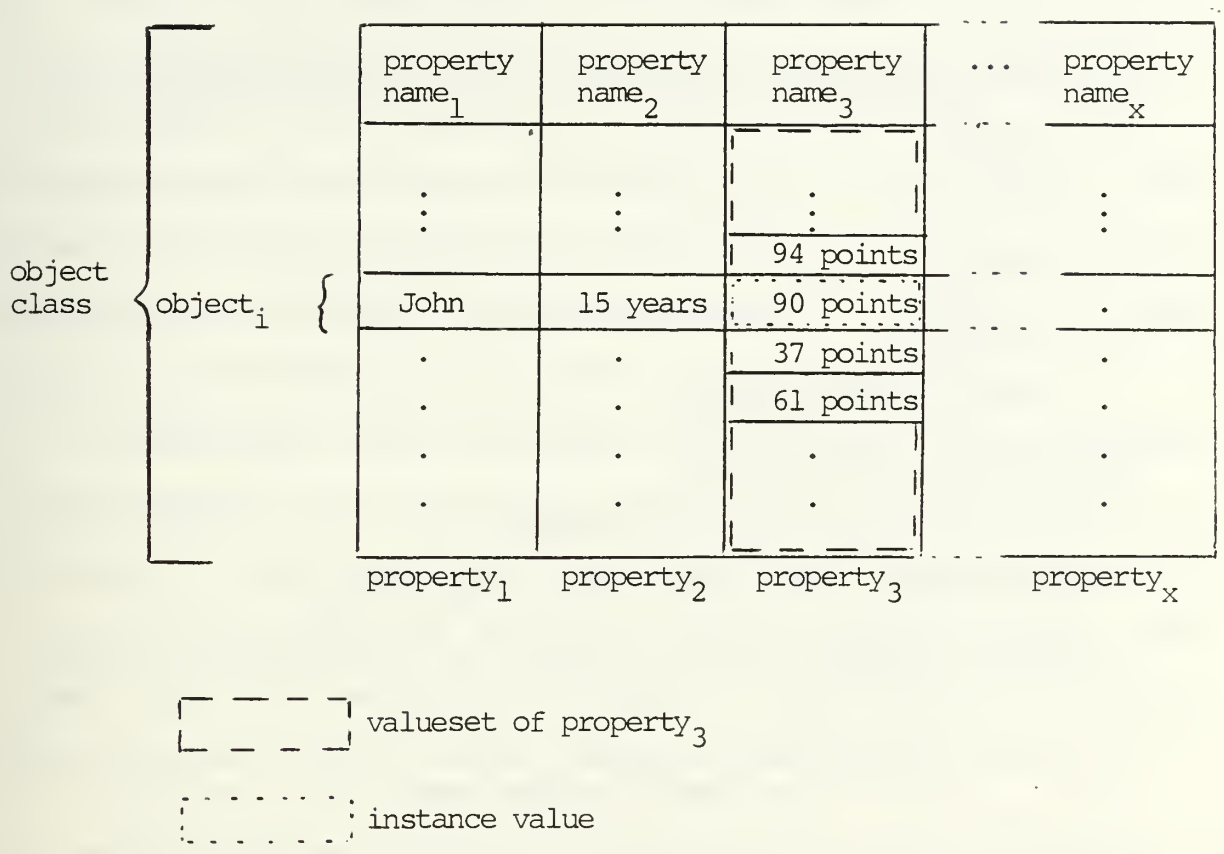


Figure IV.1. Example of an Instance Value as the Intersection Between an Object and a Valueset

Following this approach we can view the abstract model as consisting of objects and selected properties related to them, where each single property is composed of a certain name and a specific value from the predefined domain. Although one property may take different values, at any instance it can be considered as a pair containing a single name and a single value. This will be discussed in more detail in Section B of this chapter. So in short, we can write as follows:

$$object_i(pair_1, pair_2, \ldots, pair_n).$$

The basic operations performed on our simple database are:

- create
- insert
- modify
- retrieve
- test for membership

where briefly described:

create--installs a new database;

insert--adds a new object to the database;

modify--changes a certain object by altering one of its properties;

retrieve--retrieves an object identified by its particular properties from the database;

test for membership--returns a boolean value depending on the fact that a particular object does or does not belong to the database.

For futher information the reader is directed to the specification part of this thesis where the entire operations are defined in much more detail.

As already indicated in the previous chapter, we adopt the following view: whenever talking about objects and their properties we actually deal with primitives of the real world which are neither easy to handle nor can they be stored in a machine. To be more precise, in constructing a model we only work with a representation of the objects and properties but not with the primitives themselves. In doing so, the portion

of the real world our model tries to catch becomes manageable. For example, although we can not put the object STUDENT into a database, it is no problem to store the characterstring 'STUDENT' as a conceptual representation of this object. The same is true for properties. Since there is no way to store, for example, 15 YEARS, we instead extract the essential information ('15' in this case) which is more convenient.

Furthermore, there are some restrictions to be taken into consideration. For instance, the programmer should not be allowed to insert data that does not belong to the valueset of the specified property, nor modify a non present value. Such cases have to be covered by a special error handling routine.

B. DEFINING OBJECTS AND THEIR PROPERTIES

One approach to defining an object in terms of mathematical notation can be found in Hsiao [Ref. 14:p. 67]. Although Hsiao uses a different terminology, the proposed concept is as simple as it is clear:

> Let A be a set of 'attributes' and V be a set of 'values.' Then a 'record' R is a subset of the Cartesian product $A \times V$ in which each attribute has one and only one value. R can therefore be considered as a set of ordered pairs, or in short notation,
> $R = [(\text{attribute},\text{value})_1,\ldots,(\text{attribute},\text{value})_n]$.

The meaning of this equation is evident, however it does not necessarily ensure that a certain value will only be attached to an attribute for which it is explicitly defined in the corresponding domain. Thus, because our methodology is supposed to be strictly formal, the given equation can not simply be

translated by just changing terms. But we can adopt the basic idea.

The technique we apply must prevent us from mistakenly combining terms that are not defined for each other, and the way we described our properties supports this. Since a property is composed of a pair containing its name and the appropriate set of values which specifies all the legal values for this particular property, only combinations between members of this pair are possible. At each instance such a property identified by a certain name, may take any single value from the corresponding domain, thus representing one specific 'snapshot' lying within the range of feasible combinations. The following example illustrates our intentions.

property 1:

name: 'age'

set of values (domain): '10','11','12','13','14', '15'.

property 2:

name: 'city'

set of values (domain): 'Monterey', 'San Diego', 'Los Angeles', 'San Jose'.

Legal combinations representing different instances of the given properties would be:

'age','11'

'age','14'

'city','Monterey'

'city', 'San Diego'

however, combinations like:

'age','Los Angeles'

or 'city','Ll'

are not possible.

Thus, we have to attack the problem in two successive steps. We start with a particular property name N_i out of the set of all specified names N and with V_i as the corresponding value-set out of the set of given valuesets V. We then can define a property in a similar way as described above as the Cartesian product $N_i \times V_i$ where i>=1.

This ensures that a property won't have other values than those explicitly stated in its domain, where at any instance each property name has one and only one corresponding value. Having generally defined the property $P = N_i \times V_i$, i>=1, we now can easily describe an object as a sequence of one or more property instances P'

$$O = (P'_1,\ldots,P'_m).$$

Properties by themselves do not make much sense in a database, since it is the object we are interested in. But on the other hand, objects are made out of distinct property instances, and so both the object and its describing property instances represent the primitives in our database. Following a top-down or 'from the simple towards the more complex' strategy in developing a specification for the abstract database, the procedure of creating an object is illustrated in the next example.

property name		valueset		property	'snapshot'
N_1	X	V_1	domain of $property_1$		
		⋮		P_1	$P'_1 = (N_1, V_i)$
		V_m			
N_2	X	V_1	domain of $property_2$		
		⋮		P_2	$P'_2 = (N_2, V_j)$
		V_n			
⋮		⋮	⋮	⋮	⋮
N_z	X	V_1	domain of $property_z$		
		⋮		P_z	$P'_z = (N_z, V_k)$
		V_p			

An instance of an object 0_r can then be described by a sequence of properties P' which themselves contain a name and a particular value, or in other words, it represents several 'snapshots' of the properties P. The previously given description of an object should be clear by now:

$$0_r = (P'_1, P'_2, \ldots, P'_m).$$

A property name will be mapped together with any of the values assigned to the domain of the corresponding property, but will be restricted to only one value at a time when defining a certain object, although a single property name can take different values for different objects.

This probably becomes more obvious when looking at a graphical example, following in some sense the approach to a formal development for data structured in accordance with the different database models given by Hull [Ref. 16:pp. 518-528].

Notation: Let N be the distinguished set of property names and let V be a set of valuesets such that $N \cap V = 0$ (no common elements).

Definition: Then P is the set of all properties, where F maps to each subset P_i of P a property name N_i and a valueset V_i such that the following conditions apply (Figure IV.2):

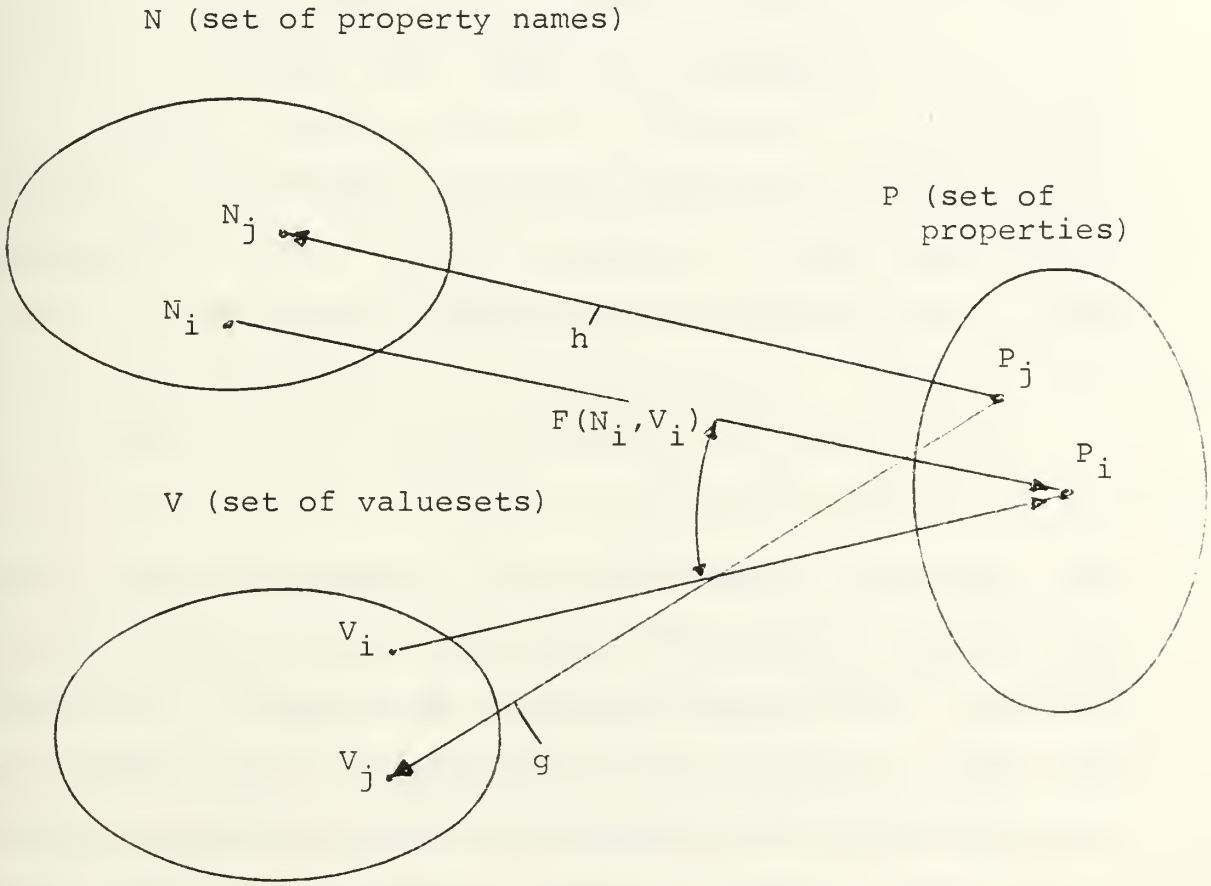


Figure IV.2. Creation of Properties

$F(N_i,V_i)\text{-->}P_i$;
g: $P_j\text{-->}V_j$;
h: $P_j\text{-->}N_j$;
$F(g(P_i),h(P_i)) = P_i$;

Restricting these mappings in such a way that values of a certain property can only be transferred from the valueset (domain) which is defined for this property type, properties consisting of different name/value pairs can be created.

Applying d = <P,F> then gives us the tool to describe the first part of the primitives, the properties. Next we need to define the remaining part, the objects. To achieve this, let 0 be a set of objects, where I maps certain pairs $(N_i,V_{ix}) \in P_i$, for 1<=i,x, to elements of 0 such that $I(N_i,) \neq I(N_j,)$ for every two N_i,N_j belonging to the same object. Then s = <<P,F>,I> = <d,I> can be used to describe any object 0_v by applying I one or more times to different $(N_i,V_{ix}) \in P_i$. The following conditions are true for this mapping (Figure IV.3):

$I(P_i,...,P_u) = I[(N_i,V_{ij}),...,(N_u,V_{uk})]\text{-->}0_v$;
l: $0_v\text{->}[P_i,...P_u = ((N_i,V_{ij}),...,(N_u,V_{uk}))]$;
$I(l(0_v)) = 0_v$;

The restrictions added are indeed necessary in order to prevent illegal operations on the sets, which otherwise would be possible. Both methods discussed above lead to the same result, and it was the goal of this section to give the reader an understanding of our strategy in defining objects and their properties. Although this strategy may appear somewhat

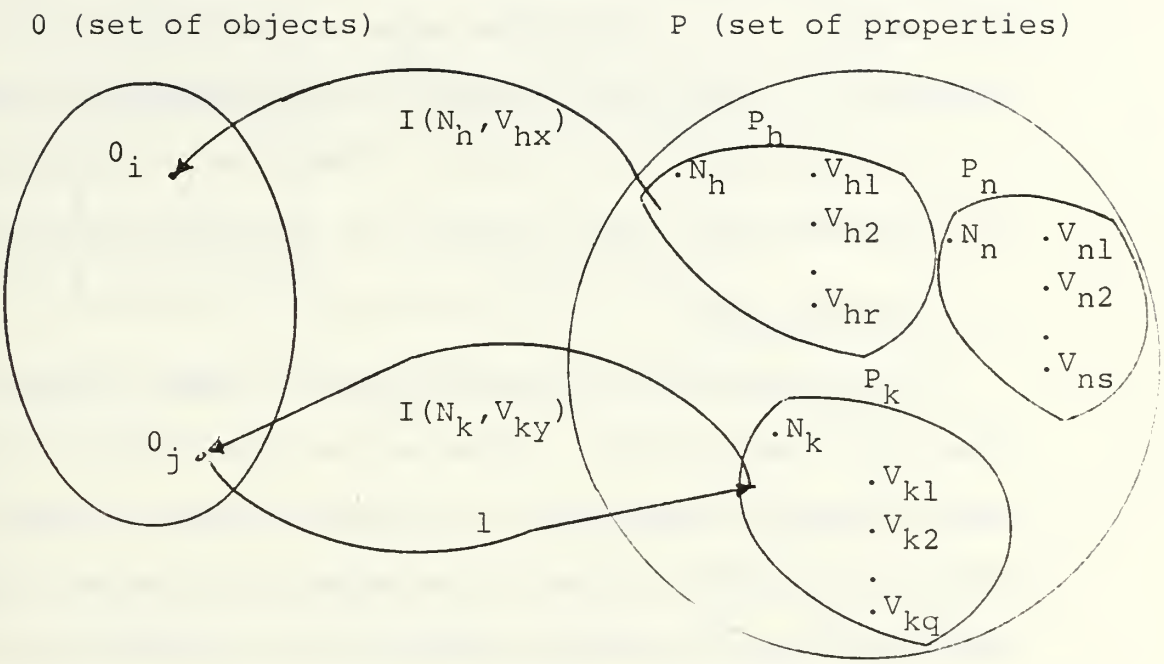


Figure IV.3. Creation of Objects

complex, it represents a serious attempt to handle the data-base primitives in a consistent way while staying as formal as possible. And before continuing with the next section which deals with the creation of object classes, a final note has to be attached: To distinguish between different objects it is necessary to include at least one name/value pair that uniquely identifies each object. This identifier is called the 'key.' If there is no such key available then there might be the case where it is impossible to distinguish one object from another.

Objects that are structured in a similar way, which means they are defined by the same property names appearing in the same order, can be grouped together to form a class. Such a class may then be identified by the kind of its property names

and the order they are arranged in. However, this is a tedious method. It is therefore more realistic to introduce a classname as identifier to distinguish between different classes.

In either case, the kind of identifier is considered to be an implementation issue and will be ignored at this point.

C. OBJECTCLASSES

A convenient way to handle a large number of objects is by grouping them together. This can be done with all objects that are related to each other in a logical sense. These sorted objects then form a class or subclass of objects, where each class has its own characteristics that distinguish it from another. Since objects are composed of several name/value pairs, the presence of those pairs and the order in which they appear is the criterion for associating any given object with a particular class. A graphical example of a subclass is presented in Figure IV.4. The subclass can be considered as a two-dimensional scheme with the objects arranged in horizontal order and the properties as different columns. From this representation it becomes obvious that, if a certain property is not contained in the subclass, none of the objects of that subclass can have this property at any instance. In mathematical notation this could be expressed as:

$$(object_x \epsilon subclass_y) \wedge (pair_z \epsilon object_x) \dashrightarrow (pair_z \epsilon subclass_y)$$

where $pair_z$ represents an instance of $property_z$ containing $name_z$ and one single value from the corresponding domain defined for $property_z$.

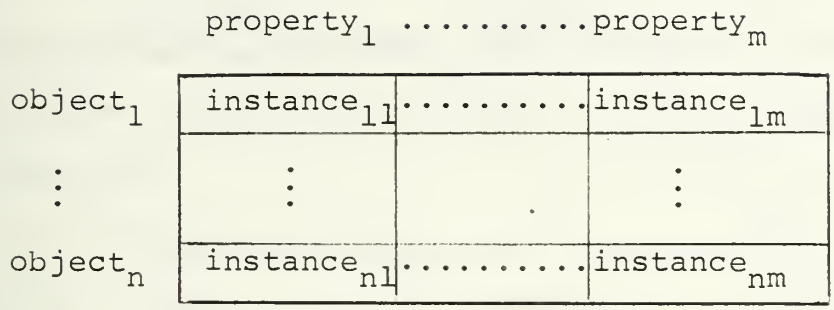


Figure IV.4. Objects Forming a Subclass

In the same way that objects can be grouped together to form a subclass, several subclasses again can be arranged to form an objectclass. Both are achieved by means of 'generalization,' a technique discussed in the next section.

An objectclass can be considered as a three-dimensional scheme consisting of 'layers' of subclasses as shown in Figure IV.4. To create this figure, turn the subclass in such a way that it fits into the horizontal plane, and then install each on top of the others. Using this method, the arrangement becomes more evident (Figure IV.5). So a subclass equals a certain level of this block whose shape depends on the number of properties, objects, and subclasses being applied. The entirety of all levels or subclasses forms the objectclass. Objects which do not belong to one subclass but have to be present because they are contained in another, are considered as just being left blank in all the subclasses in which they are not represented.

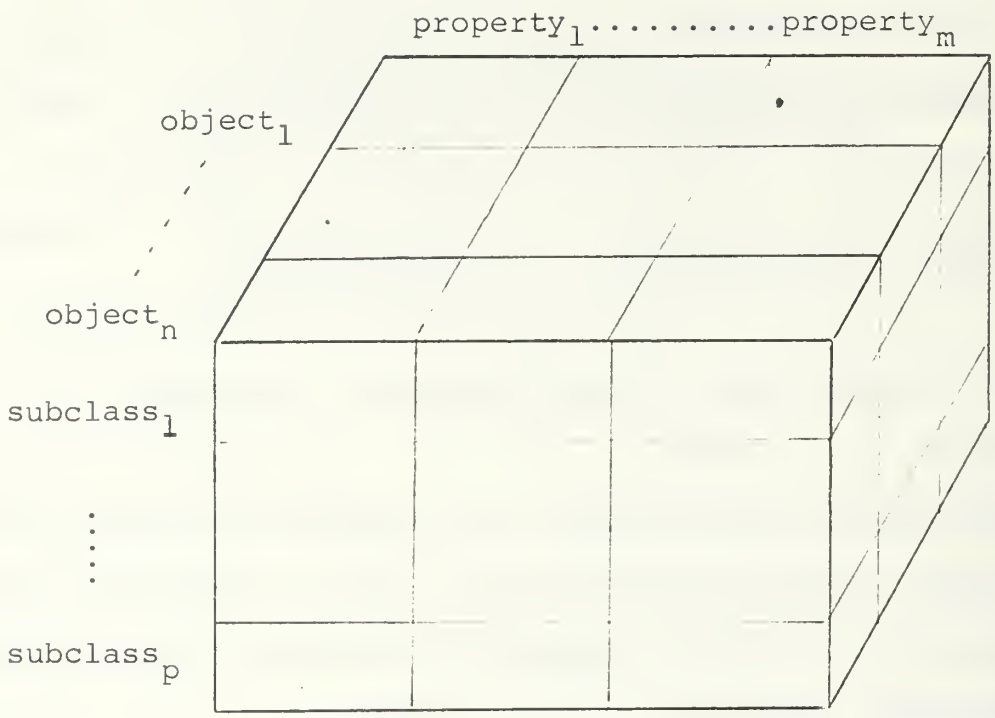


Figure IV.5. Subclasses Forming an Objectclass

If a certain property is not containted in the objectclass, none of the objects belonging to that class can have this property at any instance. In mathematical notation this can be expressed as:

$$(pair_z \in object_x) \wedge (object_x \in subclass_y) \wedge (subclass_y \in objectclass_n) \text{ ---> } (pair_z \in objectclass_n)$$

where, again $pair_z$ represents an instance of $property_z$ containing $name_z$ and a single value from the corresponding domain of $property_z$. So in searching for a particular name/value pair, first the objectclass can be checked for the matching property name, then if positive, the subclasses have to be

checked, and finally the object that responds to the requirements will be localized. By this means a search can be limited to those objects most likely to contain the requested name/value pair.

1. Generalization

Generalization is defined as ". . . an abstraction which enables a class of individual objects to be thought of generically as a single named object" [Ref. 17:p. 107]. Replacing the term 'named object' by the new term 'subclass' this technique provides a way for conveniently describing how subclasses and objectclasses can be constructed. This is true because, on the next higher level, subclasses themselves by abstraction build a generalized new 'object,' too, the objectclass (Figure IV.6a).

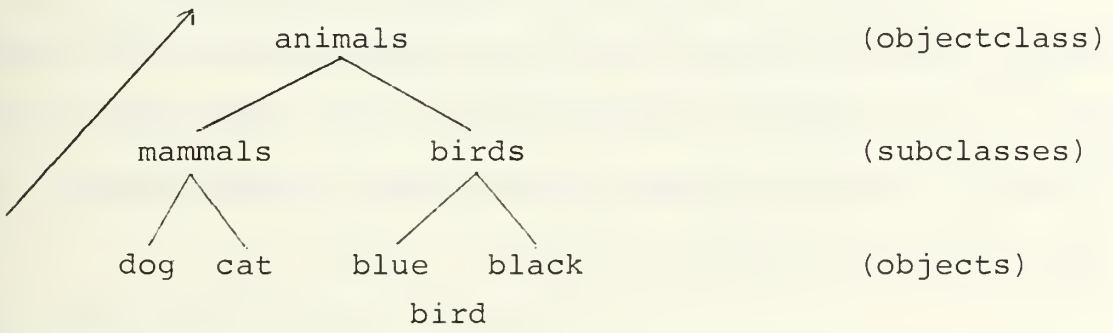


Figure IV.6a. An Example of Generalization

By generalization, which can be considered as a bottom-up technique, it is possible to create the abstraction necessary for the abstract database (Figure IV.6b).

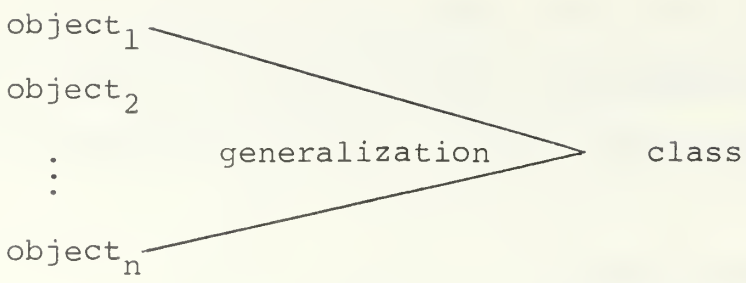


Figure IV.6b. Creating a Class from Objects by Generalizatio

This bottom-up approach, if all the objects belonging to one class are included, must then logically be reversible in such a way that a given class would lead to every single object being defined by that class. For example, the class 'mammals' naturally contains also the object 'horse,' and horses belong to the class of mammals.

2. Aggregation

In order to define the instance properties of an object, those properties must be determined for every object. This can be achieved by the technique of 'aggregation' by which different name/value pairs are grouped together so that they can be used to describe an object (Figure IV.7).

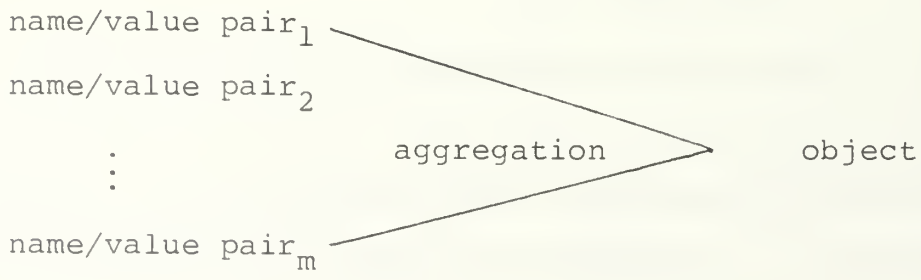


Figure IV.7. Creating an Object from 'pairs' by Aggregation

Although this method looks like a refinement of the object, aggregation in fact works the opposite way (bottom-up) and cannot be treated as an inversion of the generalization. Furthermore, aggregation is not automatically reversible, which means an object may have certain name/value pairs, however these pairs do not necessarily define this specific object in an unique way. For example, although a person can be described by the name/value pair ('age','10'), this particular pair need not necessarily refer to a person. So in contrast to the generalization, where an exhaustive installation of objects belonging to one class would guarantee reversibility, with aggregation this depends on the way an object is viewed.

Using the method of generalization and aggregation we are able to draw a picture of the general structure of the database (Figure IV.8). This drawing also clearly demonstrates that each instance property of any object belonging to a certain class, must itself belong to that class:

[((name/value pair)∈object)∈subclass]∈objectclass

D. THE SPECIFICATION LANGUAGE

1. Grammar

First developed by Yurchak (Ref. 2] and in a few parts modified by Hunter [Ref. 3], the grammar used for the specification language will be left unchanged to preserve the meaning of AM as the machine as originally designed. The grammar for the specification language as found in Appendix A is exactly

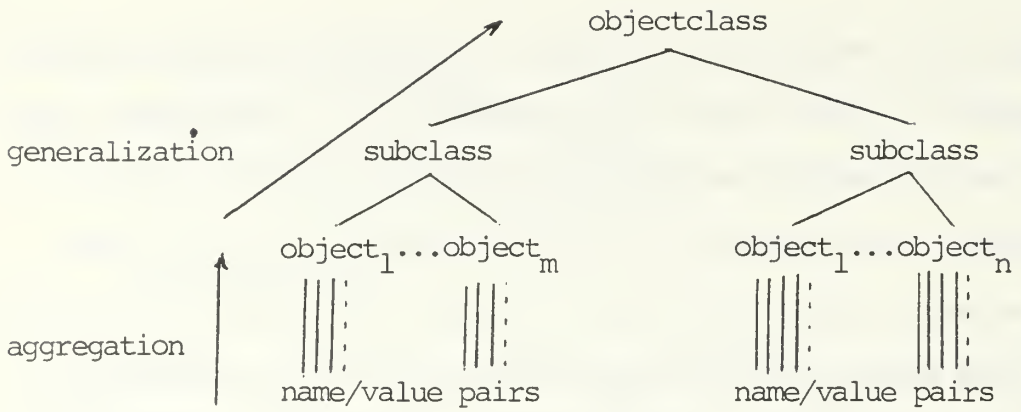


Figure IV.8. General Structure of the Database

the same as used by Hunter. The following description of the grammar and, in the next section, the preprocessor, represents an extract of Hunter's respectively Yurchak's work. It was inserted to give the reader some understanding of the underlying fundamentals.

The selected grammar is similar to examples found in the literature, but the specification language includes some features usually reserved to programming languages. A specification with modules called 'spec' is constructed first using this language while by means of the 'extension' operator it is possible to combine the specs in a hierarchical order. Each spec may introduce zero or more new 'sorts,' 'operators' and/or 'axioms.' A 'sort' can be considered as a data type and forms an object set from which the operands are selected for the operators. The elements in a sort are created from the listed 'operations.' Whenever feasible, one or more constants are declared in the beginning to provide a basis for other

elements. For example, the constant zeronat() would be such a basis for generating other elements in spec natural.

Sorts introduced in a spec may also be added to an existing spec through 'extension' of the spec(s) that will be taken as the basis, or they may form the primitives for a new 'branch' of the hierarchy. Extension provides the only means of relating the sorts and operators from different specs so that the newly declarated operators refer to both the new sort as well as to any sort from the extension.

Parameterized specifications are permitted but their use is minimized, as their properties are not well understood. Spec string is one representative of this type of specification.

The semantics and the overall structure of the specification must obey certain rules. All symbols must be unique. No symbol may be used unless it has first appeared as the name of a spec, in a sort definition, or to the left of a colon in an operator declaration. Following this rule guarantees that at no time the properties of the object inferred by the name are ambiguous. Thus, the structure of the specification is much like a Pascal program, but more restrictive. In short, there are no self referential specs, and no use of a spec is possible before it has been defined.

The specification language classifies all operators into one of three categories: 'primitive,' 'derived,' and 'hidden.'

- 'Primitive' operators are those which must be implemented to provide a full instantiation of the spec and form the basis of the resource description.

Although not every primitive operation needs to be directly implemented, the full functionality of each primitive operator must be present. It is up to the implementor whether he likes to exclude some of the primitives or some of the operators described by those primitives, as long as full functionality remains available from either set of operators.

- 'Derived' operators are those which can be derived from the primitives. The implementor may ignore these operations because their function always can be performed by the composition of primitives. Their inclusion is merely a matter of convenience. An example would be the derived operators 'or' and 'implies' in the spec boolean, whose functions are entirely covered by the primitive operators 'not' and 'and.'

- 'Hidden' operators are those to which the programmer has no access. They represent abstractions of the machine required to express a certain semantics. It might be convenient to have them in one case, while in another they may be essential to the semantic description. A typical example for a hidden operator is the READFRONT operation in the specification of a queue, as discussed in Chapter II. Here, this operator is required to build meaningful axioms.

The 'if-then' and 'if-then-else' constructs are used to build conditional axioms. Their function follows the same principles as it does in other languages, for instance, in Pascal. This means, when the evaluated 'boolean expression' is true, the 'then' part of the statement applies, otherwise the 'else' part. The 'boolean expression,' finally is defined as

expression meta_relop expression

where the term 'meta_relop' stands for the metalanguage symbols "=" (equality relation) or "!=" (inequality relation), and is used to decide about the truth of the given boolean expression.

So in some sense, the underlying grammar for our specification language is similar to the ones used for compiler compilers. In general, the application of a metalanguage provides an important tool to formulate various aspects of the developing design, since it can be used as a description for another language. Simple technical terms, such as 'if,' 'then,' 'else' or 'endif' were introduced to make our intentions more clear in both the grammar and specification. In order to distinguish between the metalanguage terminology and the regular language, metanames always are boldface. A typical representative for such a metalanguage is the BNF (Backus-Naur Form) which serves as a notation for describing the syntax of programming languages using ordinary technical English, supplemented by conventional mathematics.

2. The Macro Preprocessor

The main purpose of the macro preprocessor is to condense the amount of language wherever repetitions would swell the volume of specifications. This technique also improves readability because those parts of a specification sharing a common macro definition, can easily be identified. And since it is based on the same principle, understanding one macro definition is the starting point for understanding all of them.

As with the grammar, the idea of the preprocessor was originally introduced by Yurchak when he designed AM (version 1.0). This convenient technique has been continued by Hunter in developing AM (version 2.0), and it will also be used for

the abstract database. The fundamental theory remains unchanged. In the following section a description of the preprocessor is given as it was defined by Yurchak [Ref. 2] and adopted by Hunter [Ref. 3].

The basic form of a macro definition is

```
replace "text . . ." with "other text . . ."
```

Since the grammar of our specification language does not require quotes, they are used as delimiters for definition and equivalence strings. A macro with arguments appears like

```
replace (A,B,...,Z) "text . . ." with "other text . . ."
```

where the formal parameters must be capital letters. An uppercase letter always denotes a formal parameter to a macro, since there are no uppercase letters allowed within the spec itself. Thus, for the definition

```
replace (S)
          "typeof(S);"
with
          "typeS:-->type;
          atomofS: val-->S;
          valofS: S-->val;
```

then the string

```
          typeof(bool);
```

would be replaced by

```
          typebool:-->type;
          atomofbool: val-->bool;
          valofbool:bool-->val;
```

wherever it appeared. The utility of the macro becomes obvious when we look, for example, at the fetch and store operators, used to retrieve and store values of any type from/in primary memory. All AM data types map into a common sort called 'val,' which is returned from or passed to memory by these operators. In order to avoid the need for describing big numbers of virtually identical mapping functions, by means of macro definitions it is possible to describe the first data type and then just list all the others. This feature clearly simplifies the specification task.

Macro definitions are also excellent for expressing certain properties of operators such as commutativity, transitivity, etc., which are used throughout a specification. Instead of writing out the associated axioms repeatedly, which could prove to be tedious the definition of macros with appropriate parameters permits a more readable and explicit expression of these properties. The following example gives an illustration:

```
equint: int,int-->bool;
```

If the arguments are equivalent, then the operation should return true(), otherwise false(). In order to express eqint as the equivalence relation on objects of type int, three axioms are needed:

```
equint(i,i) = true();
equint(i,j) = equint(j,i);
implies(and(equint(i,j),equint(j,k)),equint(i,k))
            = true();
```

This by itself would be no reason for concern, but there can be a variety of relations like this within a specification, and for each single case these three axioms have to be repeated in some way. Macros provide the adequate solution, since a macro defined like

```
replace(X,S)
          "equivrel(X,S);"
with
          "for i in S
           X(i,i) = true();
           for i,j in S
           X(i,j) = X(j,i);
           for i,j,k in S
           implies(and(X(i,j),X(j,k)),X(i,k)) = true();"
```

enables us to use this definition as a template in which equint just has to be inserted

```
equivrel(equint,int);
```

thereby transforming equint into an equivalence relation on int in one step. Note that we are not required to explicitly specify the type of free variables, since this can normally be determined by context. We do so in the interest of clarity because there can be no doubt for which type 'equint' is an equivalence relation.

V. THE DESIGN

Having an operating AM processor available that already includes the control and primitive data type operations as well as the visual display device, the next step towards the goal of developing a fully operational machine was to add a database resource which could do a far better job than a conventional file handling system. With the design of the abstract database, now a model had to be created that was appropriate for manipulating data in a way to effectively support the programmer's requirements. Because a database is a complicated and complex subject, our intentions were to model a resource which includes only the fundamental operations as stated in the earlier chapters. This restriction had to be introduced in order to keep the time constraints given for this thesis. The complete specification for AM is presented in Appendix B.

However, one note of caution has to be added like the one originated by Hunter [Ref. 3] in his description of the bit-mapped display system: despite our best efforts to be thorough and rigorous, this AM specification may still contain some errors. This is not only so because extending a program written by others most likely supports this possibility, but first of all because it is a rather difficult matter to ensure that there is no ambiguity in the axioms. It also can not be guaranteed

that every portion of the spec is complete so that legal but undesirable implementations would not be permitted.

A. THE DATABASE CONCEPT

In contrast to the graphics part of AM, the database once installed by the application programmer is fairly limited to manipulating data in the predefined way. Creativity in the sense of trying and improving is only possible during the conceptual phase which always preceeds the actual installation. This means the programmer must have a clear concept about what to describe and how to arrange it in the most suitable way before the implementation finally can take place. A database represents a number of data being arranged in accordance with certain characteristics or particular relations of interest to the programmer. The main question to be answered is how to abstract a database to its fundamentals so that a programmer can work with it. Once the basic elements were identified as 'objects' and the 'properties' defining them, the next step was to develop the set of functions controlling the database that would support a natural way of thinking about the intended operations. In order to remain consistent, even the 'property' as the basic component of an object had further to be split into the subparts 'property_id' and 'valueset.'

This approach is certainly different from other methodologies because it required eleven separate specifications just to formally define the database and its abstract elements.

Each specification must be considered as a mandatory step on which succeeding specs are built. And each spec contributes in an important way to defining the abstract database resource and therefore can not be omitted. This number of specifications naturally caused some problems when translating every single function, in many cases required only for mathematical reasons of rigor and without practical usefulness for the programmer, into logical sets of operations. Although these operations had to be built in and are now available, it is anticipated that the application programmer will rather restrict himself to the more useful operations typical for database manipulations.

From the programmer's point of view, it is not of interest to retrieve all the values defined for a certain property once the domain has been fixed. He more likely wants to determine the name/value pairs associated with a particular object or find out whether one or more objects meet a given condition. Due to the underlying method objects are structured this can be achieved on the basis of list functions that will be discussed in some detail in the following sections. In general, we adopt the idea that a database can be considered as a big list. This approach was finally chosen because it facilitates our effort to describe the principles of the abstract database.

A standard database has to be created first in the mind of a programmer, then drawn on a piece of paper and, eventually, installed on the computer. This sequence represents a typical top-down approach, starting with the overall database,

partitioning it into different classes and subclasses and, finally, assigning objects to them. This approach keeps the programmer at a very low level of abstraction, where the computer can not provide much assistance. Adding, for example, a new object would require us to first specify the particular class to which it belongs. It would definitely ease the programmer's effort when he could work on a higher level of abstraction by using the power of the computing resource to insert such an additional object without caring about the specific class it refers to. But although this approach has some fascinating aspects it was not developed since object-classes are characterized by the kind of their property names, and thus have to be specified carefully. The probability of erroneous entries seemed too high to adopt this concept for our research.

As stated in Chapter III, we also abandon the introduction of a data definition language (DDL) and a query language since, due to their high level, a great deal of the intended abstraction would be taken away. This becomes quite obvious just by the fact that there are meanwhile several, non-compatible query languages established. Because our goal is to specify one particular part of a database, namely the interface between the conceptual level and the physical level, we confine our work to the fundamental principles and focus exclusively on the essential aspects of the database. For the interested reader it should however be mentioned that, once AM is completed,

there are plans for the near future to develop a high-level language for AM, too, which would naturally ease the workload of the programmer.

We now take a more detailed look at the issues and design of both the data abstractions and resource abstractions.

B. ABSTRACT DATABASE DATA TYPES

In this section we develop the abstraction of the database in detail along with all the data types needed to support it. In addition, we discuss issues concerning the design and examine how the specification captures the properties of the abstraction.

One of the problems encountered while writing the specs for the database was, that in contrast to the previous specifications which mostly operate on single data types called 'atoms,'[3] it became necessary to refer to the set theory which enables us to deal with single atoms as well as with strings of atoms. Since a database generally contains composed elements rather than simple atoms, the 'set' seemed to be the right means to tackle this problem. However, a set does not allow the same element to be represented more than once, which would restrict the operations in an unintended way. We therefore preferred to adopt the characteristics of a 'list,' where no similar limitations exist. This required the installation of an additional spec list that permits the more complicated

---

[3] An 'atom' represents a problem solving abstraction and is discussed in Chapter VI.

operations on strings. The list will be described in detail in Sections C and D of this chapter. But first of all, we consider the basic steps of how to abstract the database.

1. Property-Identification and Value

Each instance property of an object consists of a name/value pair which from now on will be referred to as 'property-value,' containing a particular property-identification ('pid') and a single value ('val'). Spec property_id expresses the properties of the 'pid' data type. A 'pid' can be a string of characters that qualifies as identifying notation for a database property. Different 'pids' are combined to a set forming the 'pidset' data type which can be considered as the description of the domain for all legal names properties may take. Spec property_idset models this domain. The operation naturally performed on this data type is creating such a set, starting with an initial 'pid' and then extending it by repeated application of the union operator with the option of using the empty set as well as the universe of all sets.

The values associated with a particular property are covered by spec value. This spec and the combination of its data type 'val' to a set, the new data type 'valset,' whose properties are described in spec valueset, are constructed in exactly the same way as the specs for 'pid' and 'pidset.' All values which may be meaningful in any context with the determined property names are permitted. For example, for the 'pid' age this could be the set of natural numbers from 1

to 100, while the 'pid' address might require a set of characterstrings. The joint operations 'unpidset' or 'unvalset' ensure that there are no redundancies in the same domain. But it is the programmer's responsibility to create the size and type of valuesets that fit his intentions best. In this stage it would be possible to build a domain containing different types of data which, if carelessly applied later, could lead to an erroneous result.

Both the 'pidset' and 'valset' data types provide a disjoint operation, and a test for membership and equivalence (relational operators). Regarding the intersection and union operations, provision has been made for associativity and commutativity.

2. Property and Propertyvalue

In order to specify a property in its entirety we need two parts: a 'pid' that describes the name of the property, and a corresponding 'valset' which determines the domain of all the values this particular property can legally have. A property is represented by sort 'prop' and is always constructed from the ordered pair 'pid' and 'valset'

prop = (pid,valset).

Spec property lists this data type and the possible operations.

To reduce a property to one of its two fundamental elements, the operators 'getid' and 'getvalset' have been introduced. While 'getid' returns the property name ('pid')

getid: prop --> pid,

'getvalset' returns the domain ('valset') of the property

getvalset: prop --> valset.

These two operators can be considered as reversion of the create operator 'crprop.' In combination with the empty value-set a property may be created just be defining a certain name, leaving the final determination of the corresponding valueset unspecified for the moment.

Spec propertyset provides the data type for different properties associated to a set and has a similar structure as spec property_idset or spec valueset. But since every property consists of the ordered pair 'pid' and 'valset,' by the 'getidset' operator all the 'pids' involved and specified as 'pidset' can be retrieved

getidset: propset --> pidset.

Defining the properties for the database in this way was the result of an analytic process which led to the understanding that a property indeed is composed of a single element identifying its type, and a set of values for this type. Since the sequence is important, we can treat a property as an ordered pair of single elements. But following this definition, now a certain property does not make much sense for describing an object, because it can not be used as a characteristic criterion. For example,

crprop: 'grade',('A','B','C','D','E','F') --> prop;

would result in the property ['grade',('A','B','C','D','E','F')] where 'grade' equals the 'pid' and ('A','B','C','D','E','F') the 'valset.' Would it be meaningful to describe any object by this property? It certainly would not, because this is a general statement about the property 'grade' containing all the defined values a grade can consist of. A specific object, however, should only contain a specific value that is characteristic for it. In Chapter IV we called such a combination a name/value pair or an instance property. Although we mentioned this subject before, it is our concern to illustrate the basic difference between a property and a particular propertyvalue.

Sort 'pval' in spec propertyvalue defines the data type to resolve the problem stated in the previous example. Operator 'crpval' enables us to create the required instance of a property that itself serves as the basis for describing any specific object. Referring to the above given example

crpval: 'grade','A' --> pval

would now result in the propertyvalue "'grade','A'" which then can be used for any object that would meet this condition. Besides the relational operators for equality and membership, as for all composed data types, there are operations available which return either the first element of the ordered pair, 'getpid' retrieves the 'pid'

getpid: pval --> pid,

or the second element which can be retrieved by the operator 'getval' returning the corresponding value of data type 'pval'

getval: pval --> val.

Since objects usually are described by more than just one propertyvalue, spec propertyvalueset was introduced. With its data type 'pvalset' it is possible to combine different propertyvalues into a set. This is the final step on our way towards defining an object which will be discussed next. In order to determine the different properties represented in such a propertyvalueset, the property names, or 'pids,' are of major interest. They can be retrieved by the operator 'getpidset' which accepts any 'pvalset' as input and returns the matching 'pidset'

getpidset: pvalset --> pidset.

Due to the fact that the data type 'pval' consists of the ordered pair 'pid' and 'val;' and that the combination of distinguished 'pvals' forms a 'pvalset,' this new data type is also composed of a set of ordered pairs itself, namely the set of the ordered pairs 'pid' and 'val.' It therefore was necessary to add two distinguished membership operations, one for testing whether a given propertyvalue is contained in a particular set of propertyvalues

mempvalset: pval,pvalset --> bool

and the other for checking if a given propertyvalueset belongs to a certain propertyset which includes information about the 'pids' involved as well as the domains for the corresponding values

mempset: pvalset,propset --> bool.

Again, as for all sets, the union operator 'unpvalset' ensures that there are no redundancies, while the disjoint operator 'intpvalset' would retrieve propertyvalues contained in both sets to be checked. Furthermore, the standard operations for associativity and commutativity have been included on principle in this more mathematical part of the specification.

3. Object and Objectclass

As mentioned before, the essential element of the database resource is the object. Each of the previously discussed specifications represents an indispensable portion that finally enables us to express the data type 'obj' by means of these more elementary data types. Its properties are specified in spec object. An object can be considered just as a particular set of propertyvalues, each containing a distinct 'pid' and a 'val'

$$obj = pvalset = [(pid,val)_1,\ldots,(pid,val)_n].$$

The function that initiates this operation is called 'crobj.' The kind or number of 'pvals' defining an object is of no interest for us at this point, although it will be later. So, theoretically, any combination of 'pvals' could be chosen to build up an object, even such containing the same 'pid' or 'pval' more than one time which actually would be meaningless. But since the installation of a database is always preceded by a rather precise concept, the grouping of different objects

into classes then should eliminate the possibility of an incidently induced redundancy on 'pids.'

The reverse operation to creating an object is 'getopvalset' which returns the entire 'pvalset' defining the object

getopvalset: obj --> pvalset

while the operator 'getopidset' retrieves only the corresponding set of 'pids'

getopidset: obj --> pidset.

In general, the type of properties is considered more important for structuring purposes, as this criterion forms a good basis for hierarchically combining related objects to classes (compare Chapter IV, Section C). We therefore did not, as the reader might have expected, introduce an analogous operation which would return all the values of a given object, but instead provided the operator 'getoval' that only retrieves one single value associated with one particular 'pid' of the object

getoval: obj,pid --> val.

Provisions are also made for an equality operator 'eqobj' and a membership operator 'haspval' which checks if a given 'pval' is contained in a certain object.

'Class' is the data type that represents a number of objects that are related in some kind to each other. This type has been discussed in some detail in Chapter IV. Its properties

are now specified in spec objectclass. If we want to insert an object into a particular class, this can be done by applying the operator 'insobj' which takes a class and an object and returns a class now including the new object. We must, however, ensure that only appropriate objects, which means, with matching 'pids,' will be inserted. This problem is solved by defining an operator for retrieving the 'pidset' of a class ('getcpidset') which takes a class as input and returns the corresponding 'pidset'

```
getcpidset: class --> pidset.
```

Together with the above specified operator 'getopidset' that accepts an object as input and returns its 'pidset,' the following axiom takes care of this

```
if eqpidset(getopidset(o),getcpidset(c)) = true()
then
   insobj(c,o) = c;
else insobj(c,o) = undef;
endif;
```

by determining whether the 'pidsets' of the object to be inserted and of the class both are equal. If they are, then the object can be added, if not, the operation becomes undefined and the object can not be added.

A similar approach was taken with the operator 'delobj' for the deletion of a selected object from a given class. It had to be ensured that any attempt to delete from a class something not contained in it was discovered. To solve this

problem the membership operator 'memclass' used in the axiom

```
if memclass(o,c) = true()
then
   delobj(o,c) = c;
else delobj(o,c) = undef;
endif;
```

to first check if the selected object is contained in the given class. This certainly is not a very efficient way of doing the deletion, but it avoids 'blowing-up' the machine by an operation that can not be handled.

The situation where a class is itself contained in another can be managed by the relational operator 'subclass.' This provision may be useful when the hierarchical structure is of importance. In connection with the intersection operator 'intclass' the boolean value of this relation can easily be determined:

```
if intclass(c1,c2) = c1
then
   subclass(c1,c2) = true();
endif;
```

As in most of the cases the 'if-then' part of this axiom could be reversed and the axiom would still be meaningful. Here it becomes obvious that the decision, what axioms to include and which to omit is a rather difficult matter and depends widely on the view of the designer. But since there is no sound recipe for how to proceed, this condition may be a source for

potential errors not discovered while the specification is written.

4. Database

The last data type, 'db,' defined in spec database represents the highest level and operates on all the data types previously discussed. So, we have now reached our goal of combining every single data type from 'pid' up to 'class.' As the reader might have expected, in order to constrcut a database one or more objectclasses must be available. This is a mandatory prerequisite since it would not make much sense to treat a couple of non-related objects like a database that always represents a particularly structured arrangement of data. Operator 'crdb' allows us to create a database; it takes a 'class' as its only argument and returns a database. A 'class' can be extended to any required size by the union operator defined in spec objectclass. This method not only avoids meaningless redundancies but also ensures that each 'pid' contained in one of the subordinated classes will be contained in the database, too.

Since 'db' is the data type of most interest for the application programmer, all the fundamental database operations have been provided in this specification. For example, 'getdbpidset' returns the set of 'pids' comprised in the database, which cannot be different from those of the corresponding classes

getdbpidset: db --> pidset.

This is expressed by the following axioms:

```
getdbpidset(insclass(crdb(c1)),c2) =
                unpidset(getcpidset(c1),getcpidset(c2));
```

which states that, if a new class c_2 is inserted into a database consisting of the class c_1, operator 'getdbpidset' would return the set of 'pids' equal to the joint 'pidsets' of c_1 and c_2 as determined in the right hand part of the equation.

Operator 'retclass' enables us to retrieve a given class from the database, object by object

```
retclass: db,class --> pvalset.
```

This function is more difficult to express in axiomatic terms

```
if and(
      and(
         (getopvalset(o) = pvs),
         (memclass(o,c) = true())
      ),
       (memdb(c,d) = true())
   ) = true()
then
   intpvalset(retclass(d,c),pvs) = pvs;
endif;
```

Here three conditions need to be fulfilled to activate the final statement. First, a given object o must have a particular 'pvalset' pvs, second, this object must be contained in a certain class c which itself has to be a member of the database d. Then the 'pvalset' pvs must also be contained in the database. So the intersect operation of all the 'pvalsets'

of this class c when retrieved from the database, and the particular 'pvalset' pvs must finally return precisely this pvs, since it is the only one contained in both the class and the object.

Provision for another operator has been made that retrieves an object whose 'pvalset' is matched by a given 'pval': 'retobj' accepts a database and a particular 'pval,' and searches the database for objects being defined by this 'pval.' Corresponding objects are then returned

```
retobj: db,pval --> obj.
```

A queue mechanism operating in accordance with the 'first-in, first-out' principle manages the case should more than a single object be retrieved. The axiom

```
if and(
      and(
         haspval(pv,o),
         memclass(o,c)
      ),
       memdb(c,d)
   ) = true()
then
   retobj(d,pv) = o;
endif;
```

states that, when an object o with a certain 'pval' pv is contained in a class c which itself is contained in the database d, then, when objects having this 'pval' are searched for by operator 'retobj,' these objects will be retrieved. If

there is only one object meeting condition pv, it will be presented as soon as it is discovered; otherwise a number of objects will be put on the queue and can then be retrieved object by object. Although this principle is simple, it ensures that each object with matching conditions can be determined and is available to the programmer at the end of one search.

The operator 'modobj' allows for changing a 'pval' of a given object, modifying it thereby. Three arguments are required: the database, the object itself and the new property-value. The database could have been omitted as an argument, but it guarantees that there actually is a certain structure available. The hard part is to defect any case for which the operation might not be defined, for example, if the given object does not belong to the indicated database, or if the replacing 'pval' is of a different type than the original or its value not defined in the domain of the associated property. To check whether all these premises are met, five conditions had to be added that must be satisfied in order to legally carry out the operation. The following axiom deals with these problems:

```
if and(
      and(
         and(
            and(
               memprop(pv,crprop(pd,vs)),
               mempidset(pd,getidset(prs))
            ),
```

```
                    (getopidset(o) = getidset(prs))
                ),
                 memclass(o,c)
            ),
             memdb(c,d)
        ) = true()
    then
       modobj(d,o,pv) = d;
    else modobj(d,o,pv) = unde;
    endif;
```

Going line by line through this axiom, it is stated that

1. the new 'pval' pv to replace the present one has to be contained in the property created from the 'pid' pd and the 'valset' vs, or in other words, since a 'pval' consists of a certain 'pid' and a single 'val,' the 'pids' will be identical while the 'val' of pv is contained in the corresponding 'valset';

2. 'pid' pd must be a member of 'propset' prs;

3. the 'pidset' of object o to be modified must be identical with the 'pidset' contained in 'propset' prs;

4. object o must be contained in class c;

5. class c must be a member of database d.

If all these conditions are met, 'modobj(d,o,pv)' is defined and the operation can be executed; otherwise it would be illegal and can not be carried out. In short, this axiom ensures before the operator can be applied, that the new propertyvalue may be inserted because the entered property_id is actually present in the object to be modified, and the new value is defined within the domain of the associated property. Thus, one instance of this property will be replaced by another instance also defined for the particular object.

The remaining operators for type 'db' are similar to the ones discussed for spec objectclass and are the relational operator for membership 'memdb' and the operator 'insclass' for the insertion of a class, respectively 'delclass' for its deletion. As analogously described before, a class can only be deleted if it is contained in the database; this is expressed in the axiom

```
if memdb(c,d) = true()
then
   delclass(d,c) = d;
else delclass(d,c) = undef;
endif;
```

If a given class is not contained in the database, it can not be deleted and the operation is undefined. With the insertion of a new class this is not quite as simple:

```
if and(
      or(
        memdb(c_1,d),
        memdb(c_2,d)
      ),
       eqpidset(getcpidset(c_1),getcpidset(c_2))
    ) = true()
 then
    insclass(d,c_1) = undef;
    insclass(d,c_2) = undef;
 else if and(
            and(
               memdb(c_1,d),
               not(memdb(c_2,d))
            ),
```

```
                    not(eqpidset(getcpidset(c1),getcpidset(c2)))
                ) = true()
        then and)
                (insclass(d,c2) = d),
                (getdbpidset(d) = unpidset(getcpidset(c1),
                getcpidset(c2))
            );
        endif;
```

Here the axiom defines that if either one of two objectclasses c_1,c_2 is already a part of the database d and both have the same set of property_ids 'eqpidset(getcpidset(c_1),getcpidset(c_2)),' then there is no way of inserting any of these classes since they are already represented. If one of the classes c_1 is part of the database while the other (c_2) is not and they do not have the same set of property_ids, which means they must be different then it would be a legal operation to insert the one not yet contained. The property_idset of the database must then be extended by the newly added 'pids'; this is done by the joint operator 'unpidset.'

These operations define spec database and thereby the fundamental part of all the individual specifications required for mathematically[4] describing the database. The remaining portion of Appendix B represents the transition towards corresponding operations that finally can be translated into machine instructions.

---

[4]Algebraic semantics describes what has to be done rather than how to do it.

C. SPEC PARAMETERIZATION

The characteristic properties of a list allow us to describe the essential database operations of retrieve, insert, modify, and delete in a convenient way. Lists not only provide a clearly structured method for treating strings of variable length but also support recursive operations, which proves to be very useful for searching the database. It is the purpose of this section to show how these basic operations can be managed by application of the list theory, while in the next section we describe its usefulness for our particular database design.

This, however, should not be seen as the attempt to narrow the spectrum of possibilities for the implementor or to guide his attention into one specific direction, since our methodology focuses on representation independence. The only reason for choosing this approach is that it provides an evident way to express our intentions.

Since the contents of our database can be considered as strings rather than as single atoms, it was necessary to make provision for this by introducing the additional spec list with data type 'elm.' This spec is a representative of the earlier mentioned parameterized specifications. It was used to define the special list operations including the recursions which offer a convenient way to carry out searches. Spec list allows the treatment of all previously described specifications as if they were of type 'elm,' simply by using a combination of spec

list with each of the former specs. For example, by application of the first discussed spec property_id using 'list (property_id),' now the new spec pidlist can be created which enables us to treat the initial data type 'pid' as 'elm' and the operator 'eqpid' as 'eqelm.' This procedure has the advantage that we can stay within or continue with the logical structure of AM specifications as they were developed by Yurchak [Ref. 2] and Hunter [Ref. 3]. It furthermore is a contribution to the clarity and simplicity of our work since we can adopt an available technique.

The typical list operators are 'nullst' which returns an empty list, 'firstelm' which returns the first element of a given list, 'firstlst' which retrieves the first list of a given list of lists, and 'restlst' which returns either the remaining elements or lists of a given list, except the first one. The meaning of these operators is expressed in several axioms

```
firstelm(makelst(k)) = k;
restlst(makelst(k)) = nullst;
```

where 'makelst'is itself an operator which takes a single element and returns a list containing this element as its only member

```
makelst: elm --> list.
```

Should a given list be empty, so that there is no first element or any rest, the application of these operators leads to an undefined result by the corresponding axioms.

In contrast to 'makelst,' the operator 'makenewlst' requires a list as an input and returns a list again. By this operator it then is possible to express the operation 'firstlst' in the following axiom:

$$\text{firstlst}(\text{makenewlst}(l)) = l;$$

thus operator 'makenewlst' has an important function in order to indisputably describe the meaning of operator 'firstlst.' In fact, it sometimes is necessary to define an additional operator just for the purpose of expressing an already developed operation in an unambiguous way.

Operator 'catlst' allows us to concatenate two lists into one and also serves as a significant tool for illustrating the meaning of the previous operations. The following axioms give an example:

$$\text{firstelm}(\text{catlst}(\text{makelst}(k),l)) = k;$$
$$\text{firstlst}(\text{catlst}(l_1,l_2)) = l_1;$$
$$\text{restlst}(\text{catlst}(l_1,l_2)) = l_2;$$

These are just a few representative axioms dealing with the fundamental list operations. Many more are required to actually express our intentions. They can be found in Appendix B. Again, we face the initially encountered problem of how to ensure that we did not miss any axiom of importance which possibly could result in an unwanted operation. So there is indeed no guarantee that our perception of the specified resource is precisely what the specification describes.

Some other, not necessarily typical list operations also had to be introduced for the particular purpose of handling all the recently defined database operations. Because these operators are more complicated they will be discussed in some detail in the following. Each of them requires searching the database first before it can be applied or successfully terminated. Consequently an iterating process was needed that could do the job. Due to its simplicity the recursive[5] approach was chosen, which in general requires a termination condition and a certain pattern that reduces the overall problem to smaller, solvable subcases. So, what we actually created was an archetype for each of the functions, containing its syntax and semantics, where the recursive definition can be considered as the prototype (compare [Ref. 9]).

Operator 'delst' is a function that accepts two lists as an input and returns a list. The first list is the one to be deleted from the list of lists entered as the second argument

e.g., delst[(l_b),(l_a,l_b,l_c)] --> (l_a,l_c) /*l_i: list */

Two conditions have to be met before this operator can legally be applied: The list from which we delete must not be empty, and the one being deleted must be contained in it, otherwise 'delst' becomes undefined. The axiom

---

[5]Other approaches, like repeatedly applied function calls, could also do the job.

```
if or(
     eqlst(l2,nullst()),
     not(memblst(l1,l2))
   ) = true()
then
   delst(l1,l2) = undef;
else if eqlst(l1,firstlst(l2)) = false()
then
   makenewlst(catlst(firstlst(l2),delst(l1,restlst(l2))));
else makenewlst(restlst(l2));
endif;
```

takes care of these cases. The next step then is to find the particular list l_1 which shall be erased. The only way to do this is by testing list after list of l_2 for identity with l_1. If the one being examined ('firstlst(l_2)') is not identical with the one to be deleted (l_1), it can not be thrown away but instead has to be saved, and the operation must be repeated with the rest of list l_2. This is achieved by the 'else-if' part of the axiom, until the identity is finally reached. By the 'else' part, only the rest of list l_2 is then concatenated to the previous portions of l_2, while the list searched for will be eliminated.

Example:

```
if or(
     eqlst[(la,lb,lc),()],
     not(memblst[(lb),(la,lb,lc)]
   ) = true()
then
   delst[(lb),(la,lb,lc)] = undef;
```

since this condition does not hold in this example, the 'else-if' part will be checked:

```
else if eqlst [(l_b),(l_a)] = false()
then
   makenewlst[catlst((l_a),delst((l_b),(l_b,l_c)))];
```

where 'delst' initializes the repeated application of the operation. During the second run the 'else-if' condition becomes true

```
eqlst[(l_b),(l_b)] = true()
```

and the 'else' part therefore is activated

```
makenewlst(l_c);
```

this leads to the concatenation

```
catlst(l_a,l_c);
```

which is the final result in this case.

This operation still retains the necessary level of abstraction since it would work for every data of type list. The structure of the database, as described in the eleven specifications at the begining of this chapter, supports the application of the list theory. This issue will be discussed in the next section.

Operator 'getlst' takes a list l_1 of lists and a particular list l_2 as input and returns the list corresponding to l_2

```
getlst[(l_a1,l_a2,l_b1,l_b2),(l_b1)] --> l_b2 /*l_i: list_i */
```

It is expressed in the axiom

```
if or(
     eqlst(l_1,nullst()),
     eqlst(l_2,nullst())
   ) = true()
then
   getlst(l_1,l_2) = undef;
else if eqlst(firstlst(l_1),l_2) = true()
```

```
then
   getlst(l1,l2) = firstlst(restlst(l1));
else if eqlst(firstlst(restlst(l1)),l2) = true()
then
   getlst(l1,l2) = firstlst(l1);
else getlst(restlst(restlst(l1)),l2);
endif;
```

The meaning may not be obvious at the first glance, so we will explain it. The underlying principle is that we consider a list of this type as a combination of several pairs of lists. Thus, entering a list of lists (l_{a1},l_{a2},l_{b1},l_{b2}) and one part (l_{b1}) of such a pair shall result in retrieving the matching second part of the corresponding pair from the list(l_{a1},l_{a2}, l_{b1},l_{b2}). Again, precaution has to be taken for cases where this operation can not be performed. For example, when either of the lists is empty, then 'getlst' becomes undefined. The axiom is constructed in such a way that always a pair of lists from the first list is checked against the second list which itself represents only one half of a pair. Should the first part of the list ('firstlst(l_{a1},l_{a2},l_{b1},l_{b2})') be identical with the second list (l_{b1}), then the matching part of the pair ('firstlst(restlst(l_{a1},l_{a2},l_{b1},l_{b2}))') will be returned as the result. Should the second part of such a pair ('firstlst (restlst(l_{a1},l_{a2},l_{b1},l_{b2}))') be identical with list (l_{b1}), then the first part of the corresponding pair will be retrieved. If no match occurs the 'else' condition applies, the recursion is activated and the remaining pairs from the first list will

be checked against the second list. So in case l_2 was not contained at all in l_1, list l_1 will eventually become empty and, since it is not possible to retrieve a part of a pair that does not exist in the given environment, the operation must become undefined at that time. This is exactly what will happen by means of the termination condition for the recursion.

Example:

```
if or(
    eqlst[(l_a1,l_a2,l_b1,l_b2),()],
    eqlst[(l_b1),()]
  ) = true()
then
   getlst[(l_a1,l_a2,l_b1,l_b2),(l_b1)] = undef;
```

since this condition is not true, the 'else-if' part will be checked:

```
else if eqlst[(l_a1),(l_b1)] = true()
```

which is not true either, so the next 'else-if' condition will be checked:

```
else if eqlst[(l_a2),(l_b1)] = true()
```

since this is false, the 'else' part is activated

```
else getlst[(l_b1,l_b2),(l_b1)];
```

which initializes the repeated application of this operator. In the second run, the first 'else-if' condition becomes true

```
else if eqlst[(l_b1),(l_b1)] = true()
```

so the following statement will be executed:

```
then
   getlst[(l_b1,l_b2),(l_b1)] = l_b2;
```

returning l_{b2} which is the list that corresponds with the second argument l_{b1} in the described example.

The next operator, 'sofirstlst' (set of first lists), takes a list as input and returns a list or, more precisely, it requires a list of lists and gives a set of lists back

sofirstlst[l_{a1},l_{a2},l_{b1},l_{b2}] --> (l_{a1},l_{b1}).

This special function was introduced to manage the operations where a set of first lists shall be retrieved:

```
if ltnat(lenlst(l),succnat(succnat(zeronat()))) = true()
then
   sofirstlst(l) = undef;
else if eqnat(lenlst(l),succnat(succnat(zeronat())))
   = true()
then
   sofirstlst(l) = firstlst(l);
else catlst(firstlst(l),sofirstlst(restlst(restlst(l))));
endif;
```

where the criterion for the operation to be defined is, that the list must have at least two elements that themselves are lists or lists of lists, which is stated in the 'if-then' part of the axiom. This is so because it would not make sense to apply 'sofirstlst' to anything else besides a list of lists. The termination condition will be reached when the list has been reduced to just two sublists ('else-if' part). In all the other cases the first element of the first list ('firstlst (firstlst(l))') will be concatenated to the iterated operation 'sofirstlst,' now applied to the remaining portion containing all other lists except the first one. Thus, finally, this operator returns every first list from the sublists of list l.

Example:

```
if [lenlst(l_a1,l_a2,l_b1,l_b2) < 2] = true()
then
   sofirstlst[l_a1,l_a2,l_b1,l_b2] = undef;
```

since the length of list l = 4, this condition does not hold and the 'else-if' part is tested

else if [lenlst(l_{a1},l_{a2},l_{b1},l_{b2}) = 2] = true()

which is not true either, so the 'else' part is applied:

else catlst[l_{a1},sofirstlst($l_{b1}$$l_{b2}$)];

thereby initializing the repeated application of the operation. In the second run, the 'else if' condition becomes true, since (l_{b1},l_{b2}) now has length = 2, so the 'then' statement is applied:

sofirstlst(l_{b1},l_{b2}) = l_{b1};

which leads to

catlst[l_{a1},l_{b1}] = (l_{a1},l_{b1})

thereby returning the set of first lists from the given list l.

The last operator to be discussed is called 'retobjlst'; it takes two lists as an input and returns a list. It was introduced with the intention to retrieve all the lists which meet a particular condition

retobjlst[((l_{a1},l_{a2},l_{a3}),(l_{b1},l_{b2}),(l_{c1},l_{c2},l_{c3})),(l_{b2})] --> (l_{b1},l_{b2}).

So in some sense, this is the most useful operation because it allows us to search a big list for certain sublists without having to remember all the details about the sublists. The axiom is short:

if eqlst(firstlst(l_1),nullst()) = true()
then
  retobjlst(l_1,l_2) = nullst();
else if intlst(l_2,firstlst(l_1)) = l_2
then
  catlst(firstlst(l_1),retobjlst(restlst(l_1),l_2));

else retobjlst(restlst(l_1),l_2);
endif;

and states that, if the overall list to be searched (l_1) for a sublist is empty, this list can not be contained in it. Thus, the result is the empty list itself ('if-then' part), which also serves as the termination condition for the recursion. Should the intersection between the entered list l_2 and the first list of l_1 be equivalent to l_2, the entire first list of l_1 will be concatenated to the repeated operation now applied to the remaining lists of l_1 ('else-if-then' part). This ensures that the list will be completely scanned since more than one of the contained sublists could meet the given condition comprised in l_2. If l_2 does not occur in the particular list being searched at the moment, the intersection of both can not be equal to l_2, and the 'else' part is activated. In this case the operation continues without saving the non-matching portion of l_1. When finally terminated, a concatenated list is available that comprises every single list of l_1 meeting the predefined requirement.

Example:

if eqlst[(l_{a1},l_{a2},l_{a3}),()] = true()

since this first list of ll is not empty, the condition does not hold and the 'else-if' part is tested

else if intlst[(l_{b2}),(l_{a1},l_{a2},l_{a3})] = l_{b2}

does not hold either, so the 'else' part is applied:

retobjlst[((l_{b1},l_{b2}),(l_{c1},l_{c2},l_{c3})),(l_{b2})]

which initializes the repeated application of 'retobjlst.' In the second run, the 'if' condition is still not true, so

the 'else-if' part is tested

else if intlst[(l_{b2}),(l_{b1},l_{b2})] = l_{b2}

since the condition holds, the 'then' statement is executed

then

catlst[(l_{b1},l_{b2}),retobjlst((l_{c1},l_{c2},l_{c3}),(l_{b2}))];

which, again, initializes 'retobjlst.'

In the third run, this leads to the application of

retobjlst[(),(l_{b2})]

but now the 'if' condition is true and the entire operation results in the concatenation of [(l_{b1}),(l_{b2})] with the empty list, which gives us (l_{b1},l_{b2}) as the final result.

As discussed earlier, the queue mechanism has to be used as an intermediate storage process to ensure that none of the retrieved lists will be lost. By means of parameterization all the operators described in the mathematical part of the specification (spec property_id through spec database) can be applied using the adequate list operations. This is achieved by short hybrid specifications (spec pidlist through spec dblist) which combine the parameters defined in 'spec list' with the corresponding operators of the original specifications, giving access to all the operations of data type 'lst.'

D. THE LIST STRUCTURE APPLIED TO DATABASE DESIGN

In this section we describe how the application of the list theory supports the fundamental database operations. Since the structure of the abstract database can be compared with a large list, this concept will be discussed in more detail. Starting with the basic elements 'property_id' and 'value,' each of the related data types may be considered as forming a

list containing just the single element <pid> or <val>, while their sets are represented by lists consisting of as many sublists as required, for example,

<<pid1>,<pid2>> or <<val1>,<val2>,<val3>>.

Consequently, a 'property' which is defined by a 'pid' and a 'valset' can be expressed by means of a list containing these two major sublists as elements, where the second list is itself composed of a number of single lists:

< <pid>,< <val1>,<val2>,...,<valn> > >.

Adding another '<' at the beginning and one '>' at the end combines several properties into a 'propertyset.' Applying this technique to 'propertyvalue' which represents an instance of a property, the resultant looks like the following:

< <pid>,<val> >

and the corresponding set can be created by combining the necessary number of adequate ordered pairs

< < <pid1>,<val1> >,< <pid2>,<val2> >,...,< <pidi>,<vali>

To construct an 'object' is now straightforward since the object is nothing more than a particular 'propertyvalueset' itself. The 'objectclass' then can be considered as a number of different objects put into the same list. But caution has to be taken that we do not violate our definition of a class. Since objects can only be grouped together if they are structured in a similar way, it is mandatory that they contain exactly the same number of corresponding 'pids.'

A 'class' can be expressed by a list of the form:

```
< < < <pid1>,<vall1> >,...,< <pidi>,<vali1> > >,
  < < <pid1>,<vall2> >,...,< <pidi>,<vali2> > >,
        :        :       :::      :        :
  < < <pid1>,<vallj> >,...,< <pidi>,<valij> > > >;
```

where the kind and number of 'pids' is the only criterion for associating a given object to a certain class, while any 'value' necessary for describing an object can legally be attached to a 'pid' as long as it is defined in the appropriate domain.

Finally, the database can be treated like a big list containing several lists of the class type just described, where the same criteria must be met on a higher level. To see how an operation on this list structure works, let's consider the disjoint operator for 'pidsets':

```
intpidset(pidset1,pidset2) --> pidset3;
```

where, for example,

```
pidset1 =  < <pid1>,<pid2> >;
pidset2 =  < <pid1>,<pid3> >.
```

Invoking the parameterized spec pidsetlist gives us access to both spec list and spec property_idset. 'Intpidset' is then replaced by 'intlst' which by substitution leads to the operation

```
intlst(< <pid1>,<pid2> >,< <pid1>,<pid3> >).
```

Since operator 'intlst' is handled by the recursive axiom

```
if or(
    (eqlst(l1,nullst()) = true()),
    (eqlst(l2,nullst()) = true())
  ) = true()
then
   intlstIfirstlst(l1),l2) = nullst();
else if memblst(firstlst(l1),l2) = true()
then
   catlst(firstlst(l1),intlst(restlst(l1),l2));
endif;
```

first the termination condition is checked, which means, if either of the two lists is empty then the intersection must be equal to the null-list ('if-then' part). In our particular case they are not empty, so the 'else-if' part will be tested. Since the first list ('<pid1>') of l1 ('<<pid1>,<pid2>>') is contained in l2 ('<<pid1>,<pid3>>'), this condition holds and the concatenated list ('catlst[<pid1>,intlst((<pid2>),(<pid1>, <pid3>))]') is created where 'intlst' invokes a recursive operation on the rest of l1 ('<pid2>') and l2 ('<pid1>,<pid3>'). This time, since neither the 'if' nor the 'else-if' part is true, the 'else' condition is applied which leads to a repeated operation on the rest of l1 and on l2:

intlst[<>,(<pid1>,<pid3>)].

Now the 'if' part of the axiom becomes true which returns the empty list. This results in the concatenation

catlst(<pid1>,<>)

and gives '<pid1>' as the intersection of l1 with l2.

E. LIST RETRIEVAL

In the previous sections we developed the abstractions of the database resource and discussed the set of operators that apply to database programming. In AM (version 2.0) the 'state' of the machine consists of the aggregation of the memory, register, stack cell contents, display register and monitor. We will now extend the 'state' to include the new entity 'queue.'

1. Background on the Processor Resource

For a better understanding of the applied extension, in the following paragraph a brief description of AM, taken from Hunter [Ref. 3], will be presented.

In AM (version 1.0) the five primitive data types, boolean, natural, integer, character, and string, form the atomic data types and are referred to as 'atoms.' Yurchak [Ref. 2] as the implementor of AM discussed the impact of the relationship between the data and a conventional machine on portability issues in detail, and identified the following properties of AM which were used to reduce the "semantic gap" and give AM its uniqueness:

- in the organization of primary storage, the next logical data item is in the next logical address;
- except as formally specified, no data type may be accessed in any way, as another data type;
- given any arbitrary logical address, the value stored there and its type can always be determined.

The processor portion of AM is an abstraction of a conventional "Von Neumann" machine with some unconventional properties. The only machine element is called a 'value.' All data

primitives (atoms) map into values. Spec typing, as introduced by Hunter, describes the relationship of 'values' and 'atoms.' As an illustration of this relationship consider the intersect operation on two 'pidsets.' We fetch the value representation of the first 'pid' of each set from two registers, and convert each value to its 'pidset' atom with the 'atomofpidset' operator. The 'pids' are intersected in accordance with the 'pidset' data type, and the resulting 'pid' is converted back into a value with 'valofpidset' for storage into a register. The operation will be continued recursively until both 'pidsets' are completely intersected.

Primary storage is an array of one or more memory segments, each of which may contain an arbitrary number of cells. Each cell is capable of "containing" any legal data value. Both programs and data may reside together in a single segment. For high speed storage, there are one or more register segments, each of which contains an arbitrary number of registers. AM also has one or more stacks, a heap, a crude file system, and now a queue. Again, every register, stack and queue cell is capable of containing any type of data.

The basic atomic data types are augmented by several others needed for the execution of programs. These are instructions and memory, register, stack, file addresses, and the queue.

2. The Queue

The value representation of the new data types 'pid,' 'pidset,' 'val,' 'valset,' 'prop,' 'propset,' 'pval,' 'pvalset,'

'obj,' 'class,'and 'db,' may be placed in any memory, register, or stack cells with one exception: whenever a set of 'pids,' 'vals,' 'pvals' or 'objs' will be retrieved, they can not be displayed until they are first placed in the queue. The concept of the queue is similar to a stack. Since we do not want the programmer to have access to the "inside" of the queue nor want to provide facilities for altering the queue in any way, we make its use only available for the very special purpose of acting as a buffer for the data retrieved from the database so it can be returned to the programmer when a search is complete. The reason for introducing the queue is that the order always matters in a database. Thus, a stack which reverses the sequence between inputs and outputs would not work for this case. The queue preserves the order in which data is entered, and although queue operations are more difficult to specify than the stack operations, it finally was adopted.

To make the queue mechanism operational in a similar way as the stack, instructions were installed for opening/closing and reading/writing ('spec instructions'), while the operations for defining the state of the queue and their meaning were added to 'spec amstate.' The program portion is described in 'spec am' which makes the queue an integrated part of AM.

The database resource can be invoked by the operator 'opendb' which requires a characterstring as identifier, a database, a state, and returns a state. Provision for closing

the database is made by the operator 'closedb' which requires a database and a state as input and returns a state, thereby terminating the access.

VI. IMPLEMENTATION

At present, only the original AM, version 1.0, is implemented and operates as a finite state machine interpreter. It comprises approximately 12,000 lines of C code, including the assembler. Developed by Yurchak, the overall concept for the assembler is as simple as it is effective. A text file representing an assembly language program is translated by the assembler into a relocatable object module. A loader, part of the AM interpreter, then loads this object module into the appropriate cells, and AM executes it. The reader is referred to Appendix C for more details about the assembler.

Since Yurchak [Ref. 2] as the originator provides a complete description of the AM implementation, we will repeat major portions of his work but also consider points of interest found in Hunter's [Ref.3] description of the version 2.0 extension while finally adding some examples and discussion of our latest modification towards AM, version 3.0.

For time reasons, neither the bit-mapped display nor the database resource have actually been implemented. Rehosting the original AM, version 1.0-Unix from a VAX 780 to a Zenith Z100 microcomputer by Hunter manifested once again all the typical difficulties known as "the portability problem." This rehosting required approximately 350 functions to be renamed throughout the 12,000 lines of code, since the compiler and

linker now used operate on shorter character names only. Another problem Hunter faced during the re-implementation of version 1.0 was that, although the Unix C compiler allowed passing of structures by value, the Lattice C compiler for the Zenith version does not, so the entire program had to be converted to passing structures by pointer. Thus major parts of the initial AM had to be modified or rewritten by Hunter.

But despite these problems usually encountered when porting software, the number of test programs developed for the Unix version run on the Zenith Z100 with the same results. So far the assembler has been revised to handle the full extension for both versions 2.0 and 3.0, including all new data types, the resource extensions, plus some additional operators for the original data types, as mentioned before. The machine itself however, has not yet been extended to handle the new data types introduced by the gradual modification. This remains for some future work.

After this description of the present state of the AM development, we will now continue with the overview of the implementation. There are four main areas: the representation of data types, the mapping of operators in the specification to functions in the interpreter, the handling of errors, and the execution of a program.

A. IMPLEMENTING DATA TYPES

Since it provided an easy translation from the specification, C was adopted by Yurchak as the corresponding programming

language for AM. But as he states in his work, another language, like LISP might have done the job as well.

AM is a tagged architecture. Each data element or value must be self-descriptive. As Hunter points out, it is important to realize the distinction between an atom which corresponds with a data type, and a value. In contrast to an atom that is referred to as sort in our specifications and represents a problem solving abstraction like 'pid' or 'obj' for the database, a value embodies a machine element. Furthermore, an atom is representation independent and keeps its level of abstraction, while a value is the specific representation of such an atom. Representation independence is achieved by certain conversion functions (Appendix B) that map all atoms into appropriate values and vice versa. This translation technique enables us to determine the type of a value solely from the value itself, which is one of the distinct properties of AM and gives the machine its tagged architecture, introduced to ease the "portability problem." The most likely construct to provide this feature is a structure (record).

Each atom is represented in C-language as a structure consisting of a 16-bit tag field, and a value field. The size of the value field varies with the type. Each sort in the specification, as the equivalent to an atom, is assigned a 16-bit code. Whenever an atom is created, or copied, it is tagged with the appropriate code. Figure VI.1 lists some fragments from the header files used by our interpreter and represents

the 'natural' data type which has a simple value field. Hunter [Ref. 3] compressed the initial term 'NAT_TYPE' to the handier 'T_NAT' for the reasons given above.

```
#define T_NAT    0x0002      /*natural type tag*/

typedef unsigned intnat;

typedef struct {
      short type;
      nat   val;
} NAT;
```

Figure VI.1. Type Definitions for Natural

By using a fixed size tag field as the first field in each record, we build in some additional robustness since, even in the event of a mistyped structure being copied into the formal parameter of a function, we can rely upon the first word to be a valid code (the type).

The next step is to describe the structure for machine values that must be capable of containing any atom. To manage this problem, Yurchak introduced the union operation which involves every single sort defined in the specifications so that any atom can be represented by the value structure. Due to the "Z100" characteristics, the value structure is divided into portions of two bytes for the tag field and four bytes for a pointer; the data type's value will be represented either directly in such a value field or, if it can not be expressed within the space of the four bytes available,

a pointer to its real location is used. String and list structures are examples that use pointers since their size is variable and usually large.

Figure VI.2 shows the concept of the union structure for machine values VAL. Because the number of data types increased very much in the latest modification of AM, only a sample of the values actually present is given. Hunter [Ref. 3] also notices that INSTR itself represents a VAL pointing to another VAL which contains the instruction's opcode. By this technique it becomes possible to fetch and store instructions, thereby allowing us to put a program into memory and to execute it.

The primary physical resources are also defined as structures. A sample of these resources is presented in Figure VI.3. Registers, display registers, memory and stacks are represented as arrays of arrays of pointers to values. The reader should note that a simple change to the constants in the header files can completely alter the configuration of the machine. We can specify an arbitrary number of arbitrary long memory, register and display register segments, as well as different sizes for an arbitrary number of stacks and the queue. Database and file are represented as an array of structures, with the files containing an input/output buffer in addition to the status information contained in both. The number of separate databases or the number and type of files can be changed by recompiling the corresponding module of the interpreter.

```
typedef short opcode;

typedef struct {
      short type;
      union value *val;
} INSTR

typedef union value {
      short   type;
      opcode  opcdval;      /*this is the compressed version
                             of the initially used term
                             'opcodeval'*/
      BOOL    boolval;      /*starting here the data types
      INT     intval;        are listed*/

       :       :

      FONT    fontval;
      LIST    listval;

       :       :

      MAD     madval;       /*memory address*/
      QADDR   qaddrval;     /*queue address*/

       :       :

      INSTR   instrval;
      MOP     mopval;       /*monadic operator*/
      DOP     dopval;       /*dyadic operator*/
} VAL;
```

Figure VI.2. Machine Values

```
typedef struct {                        /*memory segment*/
      int  size;
      VAL  **val;
} memseg;

typedef struct {                        /*stack segment*/
      int  size;
      int sp;
      VAL  **val;
} stkseg;

typedef struct {                        /*queue segment*/
      int  size;
      VAL  **val;
} qseg;

typedef struct {                        /*file segment*/
      int  stat;
      int  mode;
      int  type;
      int  val;
} fileseg;

typedef struct {                        /*database segment*/
      int  stat;
      int  val;
} dbseg;

#define_NUMMEMSEG   1024
#define_NUMSTKSEG      1
#define_NUMQSEG        1
#define_NUMFILES      16
#define_NUMDB          1               /*defined for 1 database*/

memseg_mem[_NUMMEMSEG] = {
       1024,0
       1024,0 } ;
stkseg_stk[_NUMSTKSEG] = {
       512,512,0 } ;
qseg_q[_NUMQSEG] = {
       512,512,0 } ;
```

Figure VI.3. The Physical Resource

With respect to the characteristic requirements for retrieving selected contents from the database, the queue was provided to act as a buffer. Since this is the primary reason for the queue, its accessibility has been limited to serve just this purpose. There are in fact only three ways of accessing the queue, either directly or via main memory and register operations that directly lead to the physical address of the data. The database must be opened similar to a file in order to perform the desired operations, and must be closed again when the operations are terminated.

B. MAPPING OPERATORS TO FUNCTIONS

It seems natural, although incorrect, to look at the operators in a spec as functions. However, in the implementation, this makes perfect sense. Figure VI.4 lists the code for the AM module which implements the boolean type. The header files which provide the constant definitions are omitted here. Notice that, where possible, we rely upon the operations provided by the C language, rather than slow down an already slow interpreter with axiomatic implementations of the operators.

As the implementation proceeds to more complex specifications, the program relies less upon C and more upon the operators which we have defined. In fact, the more complex operators are implemented as calls to previously defined functions which almost directly mimic the axioms from which they are derived.

```
BOOLtrue = { T_BOOL,1 };          /*the initially used BOOL_
                                  TYPE was replaced by the
                                  more complex term T_BOOL*/
BOOLfalse  { T_BOOL,0 };
BOOL *not(a)
BOOL a;
{
  BOOL *tmp;                      /*'tmp' was installed by
                                  Hunter*/
  tmp = (BOOL*)tmalloc(sizeof(BOOL));
  tmp->type = T_BOOL;
  tmp->val!= a->val;
  return(tmp);
}
BOOL *and(a,b)
BOOL *a,*b;
{
  BOOL *tmp;

  tmp = (BOOL*)tmalloc(sizeof(BOOL));
  tmp->type = T_BOOL;
  tmp->val = (a->val&&b->val);
  return(tmp);
}
BOOL *eqbool(a,b)
BOOL *a,*b;
{
  BOOL *tmp;

  tmp = (BOOL*)tmalloc(sizeof(BOOL));
  tmp->type = T_BOOL;
  tmp->val = (a->val ==  b->val);
  return(tmp);
}
BOOL  *nebool(a,b)
BOOL  *a,*b;
}
  BOOL  *tmp;

  tmp = (BOOL*)tmalloc(sizeof(BOOL));
  tmp->type = T_BOOL;
  tmp->val = (a->val !=  b->val);
  return(tmp);
}
BOOL  *or(a,b)
BOOL  *a,*b;
{
  BOOL *tmp;
```

Figure VI.4 Operator to Function Mapping for Type BOOL

```
        tmp = (BOOL*)tmalloc(sizeof(BOOL));
        tmp->type = T_BOOL;
        tmp->val = (a->val | b->val);
        return(tmp);
    }
```

Figure VI.4 (CONTINUED)

C. ERROR HANDLING

The method of treating errors was entirely revised by Hunter. All errors in the specifications are now described with the 'undef' operator. By definition, that makes all errors fatal, but they need not be. Those errors which are not, must then be defined explicitly in the specification. As remarked earlier, a more detailed treatment of errors would be an area for further study.

AM flags most errors in the operators which perform data conversions. This is a natural place for this to occur, since it is difficult to see how the type of a data element may be changed at any other time. Figure VI.5 shows a fragment which implements the 'property_id' conversion routines. The routine 'error' does not return, but terminates execution after writing the error message to 'stderr.' Notice that, even if a much larger structure was passed to 'atomofpid' or 'valofpid,' the error would be detected and handled gracefully.

This type of error checking is also performed in the functions which implement data operations.

```
PID  *atpid(v)                    /*short form for atomofpid*/
VAL  *v;
{
  PID  *b;

  if(v->type != V_PID)            /*reduced term for PID_VAL*/
       error("value not of type PID -%x", v->type);
  b = (PID*)tmalloc(sizeof(PID));
  b->type = T_PID;                /*reduced term for PID_TYPE*/
  b->val = v->pidval.val;
  return(b);
}
VAL  *vlpid(b)                    /*short form for valofpid*/
PID  *b;
{
  VAL  *v;

  if(b->type != T_PID)
       error("atom not of type PID -%x", b->type);
  v = (VAL*)tmalloc(sizeof(VAL));
  v->pidval.type = V_PID;
  v->pidval.val = b->val;
  return(v);
}
```

Figure VI.5. Error Handling Routine for Property_id Type

D. EXECUTION

The final point of interest involves actually executing a program. The method is also illustrative of the way in which the program mimics the axioms of the specification. Here, too, we resort to subterfuge to implement in a finite way a specification which could require the expenditure of an infinite resource (an implied stack in this case). The problem is the corecursive relationship between the functions 'xeq' and 'prog.' We eliminate this problem by never actually returning from 'xeq.' We rely on a dangerous but effective C idiom, 'setjmp' and 'longjmp.' Figure VI.6 illustrates this.

```
main(argc,argv)
char *argv[];
{
   int ap;                          /*check for toggles*/
   for (ap=1;ap<argc;ap++){
       if (*argv[ap] == '-'){
          if (*(argv[ap]+1) == 'x'){
             traceflag = 1;         /*added by Hunter*/
             xtraceflag = 1;
          }
          if (*(argv[ap]+1) ==  't')
             traceflag = 1;
       }
}
 initiam();                         /*main body*/
 amload();
 setjmp(_context);
 Q = prog(&_pc,Q);
 exit(0);
}


STATE   prog(m,q)                   /*program for prog*/
MAD *m;
STATE   q;

   q = xeq(atinstr(fetchm(m,q)),m,q);
                                    /*short term for atomofinstr*/
}

STATE    xeq(i,m,q)                 /*program for xeq*/
INSTR *i;
MAD *m;
STATE   q;
{
   opnd *p;
   if (i->type!=T_INSTR)            /*short term for INSTR_TYPE*/
      error("attempt to execute non-instruction -%x", i->type)
      p = i->val;
      switch(getopcode(p[0].opcodeval)){
      /*a case and semantics for each valid opcode goes
       here*/
      default:
      error("attempt to execute an illegal instruction -%x",
            p[0].opcodeval);
      }
      longjmp(_context,1);
}
```

Figure VI.6. Program Execution

In 'main,' 'initam' configures AM and invokes all of the initialization operators. 'Amload' loads a program from secondary storage into the appropriate cells as directed by the linker directives in the object module. 'Setjmp' then saves the state of the "real" machine. The variable '_pc' is the program counter which is set inside 'amload.' Now everything is set. The program is loaded and ready to run.

'Prog' is now called. Notice that 'prog' simply invokes 'xeq.' Recall now the axiom which defines the semantics of execution:

```
prog(m,q) = xeq(atomofinstr(fetchm(m,q)),m,q);
```

The value of a language which permits usefully long names is obvious in this case. Within 'xeq' a large case statement decodes the instruction and executes it according to the semantics provided for that case. This semantics is very closely modeled on the axioms in the specification. Figure VI.7 shows one such case and its accompnaying semantic action.

```
case IM_M_M;
  q = storem(
        fetchm(
          &p[1].madval,          /*val of memaddr pointed to
                                   by p[1]*/
          q                      /*q:state*/
        ),
         &p[2].madval,
         q
  );
   _pc.val = nxtmad(m)->val;
   break
```

Figure VI.7. The Semantics for 'mov_m_m'

Now compare Figure VI.7 to the axiom for mov_m_m.

```
xeq(mov_m_m(m1,m2),m,q) =
      prog(
          nxtmad(m),
          storem(
                fetchm(m1,q),
                m2,
                q
          )
      );
```

The similarities are not accidental. This should make the point that it is beneficial for the implementation language to permit such a close modeling of the specification. Obviously, this made the implementation easier to write, debug and understand.

E. DATABASE IMPLEMENTATION ISSUES

Similar to a file system, the database consists of two major parts: the information contained in it and the program that allows the user to manipulate this implementation. Once implemented, both program and content of the databases are rather fixed, although the information part can gradually be changed by iterated application of the appropriate commands. Thus in general the user is limited to retrieving or modifying the stored information, but this is the main purpose of a database. Should a conceptual change of the contents become necessary after a while, it is more convenient to revise the kind and arrangement of the data, and let the entire database then be re-implemented by an application programmer.

Theoretically, there exist no boundaries for the size of our abstract database, which means objects can be defined by any number of propertyvalues, and classes may contain any number of objects. But in reality we cannot ignore the capability of the available physical resources. Since databases tend to increase rapidly, the capacity of the attached storage device will set the natural limit.

Practically speaking, a user would need a DDL tool to effectively create a database, just as a compiler would be needed to effectively write programs for AM. The purpose of this thesis however, is to give a precise specification of the low level resources needed for a database.

The particular commands accessible by the user will be fully integrated into the AM instruction set. They mainly consist of the operations described in spec objectclass and spec database and are considered sufficient to perform all the necessary data manipulations. The operators provided permit the insertion and deletion of an object into/from a given class, the update of an object by modification of its contents, and the selection of one or more objects in accordance with a predefined condition. As soon as the database part of AM is invoked (instruction 'opendb') these commands can be applied to the contents representing this database after they are brought into main memory first. For any change, data are fetched from their memory location, loaded into a register, and stored back when the operation is completed. After

termination of the desired activities the database must be closed (instruction 'closedb') and the data will be transferred back to secondary storage. This procedure ensures that the data residing in secondary storage at the end of the operation always represent the actual state of the database. A presorting of data outside main memory is not feasible, since we do not presume the existence of an additional processor which is usually known as back-end computer.

To select an object on the basis of a certain entity by which it is defined, called 'propertyvalue' ('pv') in our terminology, the characterizing 'pv' will be loaded into a register as a comparand. Identified by the corresponding 'property_id' of this 'pv,' the class possibly containing the required object is then localized and the entire 'pv' set of its first object will be loaded into a separate set of registers. If a match occurs between the comparand and the register containing the adequate 'pv' of the object, the total 'pv' set is copied into the queue and the next object will be loaded. If no match takes place the procedure will be continued without storing the object in the queue. When the entire class has been searched, the 'select' operation terminates and the resultant object(s) can be read from the queue. The state of the database will not be changed by this operation since only a partial copy is taken. But it can not be excluded that none of the objects would meet the criterion searched for. In this case no copy will reside in the queue and the returned 'pv' set is empty.

In contrast to the 'select' operation, which can be considered as a read function, insertion, deletion, and modification actually do change the given state of the database, so these operations are a little more complicated. Inserting a new object requires, besides the 'pv' set that defines it, the class to which it is inserted. This technique ensures that no object accidentally will be inserted which is not attached to a certain class. Without this restriction, the structure of the database could be changed in an unacceptable way. When the class has been identified by comparison of the corresponding 'property_ids' the new object will be added at the end of that class. An error handling routine is invoked should the required class not exist.

At this point the question must be answered where the special identifier or 'key,' mentioned in a previous chapter, would best fit. Such a device is necessary to distinguish between 'pv' sets which incidentally are identical, although they may represent different objects, or to detect an unwanted redundancy. Only an identifier that is unique to every single object can meet this requirement. However, the simple arrangement of the objects in a numerical order would not work, since by mistake the same object could be listed under different numbers without the means of recognizing the error. The only way to solve this problem is by introducing a key value that can only be applied in connection with the particular object it defines, like a social security number or a similar characteristic

attribute. We adopt this technique but leave the selection of the proper criterion up to the application programmer. However, it is considered advantageous always to define the first 'property_id' of a class as the key element. This would allow the arrangement of the objects in a numerical or alphabetical order with regard to their identifiers.

Deletion and modification have one thing in common. The particular object must be retrieved first before the operation can be applied. This will be achieved in a similar way as for the 'selection' operator. The 'pv' set of the object in question is loaded into a set of registers and the appropriate class is identified. Then the 'pv' set representing the first object of the class is loaded into a separate set of registers and checked for identity. As soon as a match occurs the search stops. In the case of a modification the up-dated 'pv' set will be stored in memory and the pointers are adapted to the new location. For a deletion operation the pointers are advanced and the "erased" object will be by-passed. If the indicated object cannot be found in the appropriate class, an error handling routine is activated and the state of the database will not be changed.

VII. CONCLUSIONS

Interface standards that are precise, understandable and enforceable can provide a way to improve efforts toward portable software. With the abstraction of a database, we not only extended AM by adding another basic resource to the processor and the visual display device, but also showed a way to reduce the database to its fundamental properties. Rather than being concerned with a specific data definition or query language, our abstraction of a database is intended to provide a uniform, abstract, and functional interface to the computing system.

By this concept the application programmer retains all the freedom he needs to actually implement the database resource in a way that fits his purpose best. And although it may turn out that the AM machine becomes even slower as the result of the additional data types we introduced, the specified axioms fully describe the operations in a precise, unambiguous and easily understandable manner, thus leaving no room for any different interpretation by the programmer.

Based on the principle of resource abstraction, the AM specification intensively supports a strong typing such that objects of a given type can not take other values than the ones appropriate to the type, and no operations can legally be applied to an object which are not defined for its type. All these decisions naturally reduce efficiency, but this loss will be compensated by gains in clearness and accuracy.

It is difficult to foresee how much AM can be modified for efficiency without compromising the level of abstraction presently achieved. To test for resource equivalence or to prove the correctness of implementations of resource specifications is a nontrivial matter, and this problem certainly will increase with every change attempted. So, for the near future, it seems that we have to pay the price for implementing in a strictly formal way, since no promising theory is yet known to reduce the large number of necessary function calls within the specification.

Further basic resources that could be taken into consideration for a possible AM extension are a so-called mouse device with properties similar to the joy stick cursor, and a keyboard.

APPENDIX A

A GRAMMAR FOR ALGEBRAIC SPECIFICATIONS

```
abstraction:
        (abstraction spec)?

spec:
        (spechead|parmhead) specbody specend

spechead:
        nameblk 'is'

parmhead:
        nameblk 'parm' specbody 'is'

specend:
        'end' specname ':'

nameblk:
        'spec' specname

specbody:
        extension? specblk

extension:
        extendblk specblk 'end' 'extend' ';'

extendblk:
        'extend' specnames 'with'

specnames:
        specname
       |specnames ',' specname

specblk:
        useblk
       |sortblk? opblk axiomblk?

useblk:
        'use' specname '(' specname ')' mapping? specblk
        'enduse'

mapping:
        'where' eqivlist

equivlist:
        equivalence ';'
       |equivlist equivalence ';'
```

```
equivalence:
        sortname 'is' sortname
       |opname 'is' opname

sortblk:
        'sort' sortnames

sortnames:
        sortname ';'
       |sortnames sortname ';'

opblk:
        primblk? dervblk? hiddenblk?

primblk:
        'primitive' 'op' ops

ops:
        op ';'
       |ops op ';'

op:
        opname ';' arglist? '->' sortname

arglist:
        sortname
       |arglist ',' sortname

dervblk:
        dervops dervdef

dervops:
        'derived' 'op' ops

dervdef:
        'derived' 'def' axioms

hiddenblk:
        'hidden' 'op' ops

axiomblk:
        'axiom' axioms

axioms:
        axiom ';'
       |axioms axiom ';'

axiom:
        conditional
       |('for' varlist 'in' sortname)? termexpr '=' termexpr
```

```
termexpr:
        factor
       |multiplier? opname '(' factors ')'

factors:
        factor
       |factors ',' factor

factor:
        multiplier? opname '(' ')'
       |freevar

varlist:
        freevar
       |varlist ',' freevar

multiplier:
        '[' positive_number ']'

conditional:
        'if' termexpr meta_relop termexpr then else? 'endif'

meta_relop:
        '='
       |'!='

then:
        'then' axioms

else:
        'else' axioms
```

APPENDIX B

THE SPECIFICATION FOR AM (VERSION 3.0)

```
replace()
   "NUMINTENS"
with
   "199"


replace()
   "DISPLAYSIZE"
with
   "9999"


replace(X,S)
   "equivrel(X,S);"
with
   "X(i,i) = true();
   X(i,j) = X(j,i);
   implies(and(X(i,j),X(j,k)),X(i,k)) = true();"


replace(X,S)
   "reflexive(X,S);"
with
   "X(i,i) = true();"


replace(X,S)
   "commutative(X,S);"
with
   "X(i,j) = X(j,i);"


replace(X,S)
   "transitive(X,S);"
with
   "implies(and(X(i,j),X(j,k)),X(i,k)) = true();"


replace(X,S)
   "associative(X,S);"
with
   "X(i,X(j,k)) = X(X(i,j),k);"
```

```
replace(X,S)
   "irreflexive(X,S);"
with
   "X(i,i) = false();"


replace(X,S)
   "symmetric(X,S);"
with
   "implies(X(i,j),X(j,i)) = true();"


replace(X,S)
   "antisymmetric(X,S);"
with
   "implies(and(X(i,j),X(j,i)),(i == j)) = true();"


replace(S,T)
   "idopers(S,T);"
with
   "startT: → S;
   nextT:S → S;
   prevT:S → S;
   eqS:S,S → bool;"


replace(S,T)
   "idaxioms(S,T);"
with
   "prevS(startT()) = undef;
   prevS(nextS(i)) = i;
   if i != startT() then
      nextS(prevS(i)) = i;
   endif;
   equivrel(eqS,S);"


replace(S)
   "typingopers(S);"
with
   "typeS: → type;
   atomofS: val → S;
   valofS: S → val;"


replace(S)
   "typingaxioms(S);"
with
   "whattype(valofS(t)) = typeS();
   atomofS(valofS(t)) = t;
```

```
   if whattype(v) = typeS()
      then valofS(atomofS(v)) = v;
      else atomofS(v) = undef;
   endif;"


replace(S,T)
   "relop(S,T);"
with
   "applyrop(ST(),vl,v2) = valofbool(TS(atomofS(vl),
    atomofS(v2)) );"


replace(S)
   "isops(S);"
with
   "if whattype(v) = typeS()
      then applybop(isS(),V) = valofbool(true());
      else applybop(isS(),v) = valofbool(false());
   endif;"


replace(S,T)
   "stateaxioms(S,T);"
with
   "fetchS(a,initam()) = undef;
   storeS(fetchS(a,q),a,q) = q;
   implies(
      eqT(al,a2),
      fetchS(al,storeS(v,a2,q)) = v
   ) = true();
   implies(
      not(eqT(al,a2)),
      fetchS(al,storeS(v,a2,q)) = fetch(al,q)
   ) = true();"

replace                              /* database part */
   "crpidset"
with
   "pidsetlist.makelst"


replace
   "unpidset"
with
   "pidsetlist.unlst"


replace
   "intpidset"
with
   "pidsetlist.intlst"
```

```
replace
   "mempidset"
with
   "pidsetlist.memblst"


replace
   "crvalset"
with
   "valsetlist.makelst"


replace
   "unvalset"
with
   "valsetlist.unlst"


replace
   "intvalset"
with
   "valsetlist.intlst"


replace
   "memvalset"
with
   "valsetlist.memblst"


replace
   "getid"
with
   "proplist.firstlst"


replace
   "getvalset"
with
   "proplist.restlst"


replace
   "crprop"
with
   "proplist.catlst"


replace
   "crpropset"
with
   "propsetlist.makenewlst"
```

```
replace
   "unpropset"
with
   "propsetlist.unlst"


replace
   "intpropset"
with
   "propsetlist.intlst"


replace
   "getidset"
with
   "propsetlist.sofirstlst"


replace
   "mempropset"
with
   "propsetlist.memblst"


replace
   "crpropval"
with
   "pvallist.catlst"


replace
   "getpid"
with
   "pvallist.firstlst"


replace
   "getval"
with
   "pvallist.restlst"


replace
   "memprop"
with
   "pvallist.memblst"


replace
   "crpvalset"
with
   "pvalsetlist.makenewlst"
```

```
replace
   "unpvalset"
with
   "pvalsetlist.unlst"


replace
   "intpvalset"
with
   "pvalsetlist.intlst"


replace
   "mempvalset"
with
   "pvalsetlist.memblst"


replace
   "mempset"
with
   "pvalsetlist.memblst"


replace
   "getpidset"
with
   "pvalsetlist.sofirstlst"


replace
   "crobj"
with
   "objlist.makenewlst"


replace
   "readobj"
with
   "objlist.makenewlst"


replace
   "haspval"
with
   "objlist.memblst"


replace
   "getopidset"
with
   "objlist.sofirstlst"
```

```
replace
   "getoval"
with
   "objlist.getlst"


replace
   "crclass"
with
   "classlist.makenewlst"


replace
   "unclass"
with
   "classlist.unlst"


replace
   "intclass"
with
   "classlist.intlst"


replace
   "memclass"
with
   "classlist.memblst"


replace
   "subclass"
with
   "classlist.memblst"


replace
   "getcpidset"
with
   "classlist.sofirstlst"


replace
   "insobj"
with
   "classlist.catlst"


replace
   "delobj"
with
   "classlist.delst"
```

```
replace
   "crdb"
with
   "dblist.makenewlst"


replace
   "memdb"
with
   "dblist.memblst"


replace
   "insclass"
with
   "dblist.catlst"


replace
   "delclass"
with
   "dblist.delst"


replace
   "retclass"
with
   "dblist.intlst"


replace
   "retobj"
with
   "dblist.retobjlst"


replace
   "getbdbidset"
with
   "dblist.sofirstlst"


replace
   "modobj"
with
   "dblist.modlst"                /* database part */
```

```
spec boolean
is
   sort
      bool;
   primitive
   op
      true: → bool;
      false: → bool;
      not: bool → bool;
      and: bool,bool → bool;
   derived
   op
      or: bool,bool → bool;
      implies: bool,bool → bool;
   derived
   def
      or(b1,b2) = not(and(not(b1),not(b2)) );
      implies(b1,b2) = not(and(b1,not(b2)) );
   axiom
      false() = not(true());
      not(not(b)) = b;
      and(true(),b) = b;
      and(false(),b) = false();
      commutative(and,bool);
end boolean;
```

```
spec natural
is
   extend
      boolean
   with
      sort
         nat;
      primitive
      op
         zeronat: → nat;                  /* zero * /
         prednat: nat → nat;              /* predecessor */
         succnat: nat → nat;              /* successor */
         sumnat: nat,nat → nat;           /* addition */
         subnat: nat,nat → nat;           /* subtraction */
         mltnat: nat,nat → nat;           /* multiplication */
         divnat: nat,nat → nat;           /* division */
         eqnat: nat,nat → bool;           /* equal */
         gtnat: nat,nat → bool;           /* greater than */
      derived"
      op
         ltnat: nat,nat → bool;           /* less than */
         genat: nat,nat → bool;           /* greater or equal */
         lenat: nat,nat → bool;           /* less or equal */
         nenat: nat,nat → bool;           /* not equal */
```

```
   derived"
   def
      ltnat(n,m) = not(or(gtnat(n,m),eqnat(n,m)) );
      genat(n,m) = not(ltnat(n,m));
      lenat(n,m) = not(gtnat(n,m));
      nenat(n,m) = not(eqnat(n,m));
   axiom
      prednat(zeronat()) = undef;
      prednat(succnat(n)) = n;
      succnat(prednat(n)) = n;
      sumnat(n,zeronat()) = n;
      sumnat(n,succnat(m)) = succnat(sumnat(n,m));
      subnat(n,zeronat()) = n;
      if gtnat(n,m) = true()
      then
         subnat(n,succnat(m)) = prednat(subnat(n,m));
      else
         subnat(n,succnat(m)) = undef;
      endif;
      mltnat(x,zeronat()) = zeronat();
      mltnat(x,succnat(zeronat()) ) = x;
      mltnat(x,y) = sumnat(x,mltnat(x,prednat(y)) );
      if y = zeronat()
      then
         divnat(x,y) = undef;
      else if ltnat(x,y) = true()
      then
         divnat(x,y) = zeronat();
      else
         divnat(x,y) = sumnat(
            succnat(zeronat()),
            divnat(subnat(x,y),y)
         );
      endif;
      endif;
      eqnat(n,m) = eqnat(succnat(n),succnat(m));
      gtnat(succnat(n),n) = true();
      equivrel(eqnat,nat);
      irreflexive(gtnat,nat);
      irreflexive(ltnat,nat);
      transitive(gtnat,nat);
      transitive(ltnat,nat);
      transitive(genat,nat);
      transitive(lenat,nat);
      antisymmetric(genat,nat);
      antisymmetric(lenat,nat);
      symmetric(nenat,nat);
      commutative(sumnat,nat);
      commutative(mltnat,nat);
      associative(sumnat,nat);
      associative(mltnat,nat);
   end extend;
end natural;
```

```
spec integer
is
   extend
      boolean,
      nat
   with
      sort
         int;
      primitive
      op
         zeroint: → int;
         ntoi: nat → int;                    /* nat to int */
         iton: int → nat;                    /* int to nat */
         absint: int → int;                  /* absolute value */
         predint: int → int;
         succint: int → int;
         sumint: int,int → int;
         subint: int,int → int;
         mltint: int,int → int;
         divint: int,int → int;
         modint: int,int → int;              /* modulo */
         eqint: int,int → bool;
         gtint: int,int → bool;
      derived
      op
         ltint: int,int → bool;
         geint: int,int → bool;
         leint: int,int → bool;
         neint: int,int → bool;
      derived
      def
         ltint(n,m) = not(or(gtint(n,m),eqint(n,m)) );
         geint(n,m) = not(ltint(n,m));
         leint(n,m) = not(gtint(n,m));
         neint(n,m) = not(eqint(n,m));
      axiom
         predint(succint(n)) = n;
         succint(predint(x)) = x;
         ntoi(zeronat()) = zeroint();
         ntoi(succnat(n)) = sumint(succint(zeroint()),
                                   ntoi(n));
         iton(zeroint()) = zeronat();
         if ltint(x,zeroint()) = true()
         then
            iton(x) = undef;
         else
            iton(succint(x)) = sumnat(succnat(zeronat),
                                      iton(x));
         endif;
```

```
if ltint(x,zeroint()) = true()
then
   absint(x) = subint(zeroint(),x);
else
   absint(x) = x;
endif;
sumint(n,zeroint()) = n;
sumint(n,succint(m)) = succint(sumint(n,m));
subint(x,zeronat()) = x;
subint(x,succnat(y)) = predint(subint(x,y));
mltint(x,zeroint()) = zeroint();
mltint(x,succint(zeroint()) ) = x;
mltint(x,y) = sumint(x,mltint(x,predint(y)) );
if y = zeroint()
then
   divint(x,y) = undef;
else if ltint(absint(x),absint(y)) = true()
then
   divint(x,y) = zeroint();
else if or(
   and(
      gtint(x,zeroint()),
      gtint(y,zeroint())
   ),
   and(
      ltint(x,zeroint()),
      ltint(y,zeroint())
   )
) = true()
then
   divint(x,y) = sumint(
      succint(zeroint()),
      divint(subint(x,y),y)
   );
else
   divint(x,y) = sumint(
      predint(zeroint()),
      divint(sumint(x,y),y)
   );
endif;
endif;
endif;
if gtint(m,zeroint()) = true()
then
   if ltint(n,zeroint()) = true()
   then
      modint(n,m) = modint(sumint(n,m),m);
   else
      modint(n,m) = subnat(n,mltnat(m,divint(n,m)) );
   endif;
else
   modint(n,m) = undef;
endif;
```

```
            eqint(x,y) = eqint(succint(x),succint(y)) );
            gtint(succint(n),n) = true();
            equivrel(eqint,int);
            irreflexive(gtint,int);
            irreflexive(lting,int);
            transitive(gtint,int);
            transitive(ltint,int);
            transitive(geint,int);
            transitive(leint,int);
            antisymmetric(geint,int);
            antisymmetric(leint,int);
            symmetric(neint,int);
            commutative(sumint,int);
            commutative(mltint,int);
            associative(sumint,int);
            associative(mltint,int);
   end extend;
end integer;


spec character
is
   extend
      boolean
   with
      sort
         char;
      primitive
      op
         'A','B','C',...,'Z': → char;
         'a','b','c',...,'z': → char;
         '!','@','#','$','%','^','&','*','(',')': → char;
         '-','_','+','=','~','`','{','}','[',']': → char;
         '''','"',':',';',',',',','.'<','>','?','/': → char;
         '|','': → char;
         '1','2','3','4','5','6','7','8','9','0': → char;
         NUL: → char;
         SOH: → char;
         STX: → char;
         ETX: → char;
         EOT: → char;
         ENQ: → char;
         ACK: → char;
         BEL: → char;
         BS: → char;
         HT: → char;
         LF: → char;
         VT: → char;
         FF: → char;
         CR: → char;
         SO: → char;
         SI: → char;
         DLE: → char;
```

```
    DC1: → char;
    DC2: → char;
    DC3: → char;
    DC4: → char;
    NAK: → char;
    SYN: → char;
    ETB: → char;
    CAN: → char;
    EM: → char;
    SUB: → char;
    ESC: → char;
    FS: → char;
    GS: → char;
    RS: → char;
    US: → char;
    SP: → char;
    DEL: → char;
    eqchar: char,char → bool;
    gtchar: char,char → bool;
derived
op
    ltchar: char,char → bool;
    gechar: char,char → bool;
    lechar: char,char → bool;
    nechar: char,char → bool;
derived
def
    ltchar(n,m) = not(or(gtchar(n,m),eqchar(n,m)) );
    gechar(n,m) = not(ltchar(n,m));
    lechar(n,m) = not(gtchar(n,m));
    nechar(n,m) = not(eqchar(n,m));
axiom
    gtchar('DEL','~') = true();
    gtchar('~','}') = true();
    gtchar('}','|') = true();
    gtchar('|','{') = true();
    gtchar('{','z') = true();
    gtchar('z',...,'a') = true();
    gtchar('a',' ') = true();
    gtchar(' ','_') = true();
    gtchar('_','^') = true();
    gtchar('^',']') = true();
    gtchar(']','') = true();
    gtchar('','[') = true();
    gtchar('[','Z') = true();
    gtchar('Z',...,'A') = true();
    gtchar('A','@') = true();
    gtchar('@','?') = true();
    gtchar('?','>') = true();
    gtchar('>','=') = true();
    gtchar('=','<') = true();
    gtchar('<',';') = true();
```

```
gtchar(';',':') = true();
gtchar(':','9') = true();
gtchar('9',...,'0') = true();
gtchar('0','/') = true();
gtchar('/','.') = true();
gtchar('.','-') = true();
gtchar('-',',') = true();
gtchar(',','+') = true();
gtchar('+','*') = true();
gtchar('*',')') = true();
gtchar(')','(') = true();
gtchar('(',''') = true();
gtchar(''','&') = true();
gtchar('&','%') = true();
gtchar('%','$') = true();
gtchar('$','#') = true();
gtchar('#','"') = true();
gtchar('"','!') = true();
gtchar('!',SP) = true();
gtchar(SP,US) = true();
gtchar(US,RS) = true();
gtchar(RS,GS) = true();
gtchar(GS,FS) = true();
gtchar(FS,ESC) = true();
gtchar(ESC,SUB) = true();
gtchar(SUB,EM) = true();
gtchar(EM,CAN) = true();
gtchar(CAN,ETB) = true();
gtchar(ETB,SYN) = true();
gtchar(SYN,NAK) = true();
gtchar(NAK,DC4) = true();
gtchar(DC4,DC3) = true();
gtchar(DC3,DC2) = true();
gtchar(DC2,DC1) = true();
gtchar(DC1,DLE) = true();
gtchar(DLE,SI) = true();
gtchar(SI,SO) = true();
gtchar(SO,CR) = true();
gtchar(CR,FF) = true();
gtchar(FF,VT) = true();
gtchar(VT,LF) = true();
gtchar(LF,HT) = true();
gtchar(HT,BS) = true();
gtchar(BS,BEL) = true();
gtchar(BEL,ACK) = true();
gtchar(ACK,ENQ) = true();
gtchar(ENQ,EOT) = true();
gtchar(EOT,ETX) = true();
gtchar(ETX,STX) = true();
gtchar(STX,SOH) = true();
gtchar(SOH,NUL) = true();
equivrel(eqchar,char);
```

```
         irreflexive(gtchar,char);
         irreflexive(ltchar,char);
         transitive(gtchar,char);
         transitive(ltchar,char);
         transitive(gechar,char);
         transitive(lechar,char);
         antisymmetric(gechar,char);
         antisymmetric(lechar,char);
         symmetric(nechar,char);
   end extend;
end character;


spec string
parm
   extend
      boolean
   with
      sort
         lm;
      primitive
      op
         eqlm: lm,lm → bool;
         gtlm: lm,lm → bool;
      derived
      op
         ltlm: lm,lm → bool;
         gelm: lm,lm → bool;
         lelm: lm,lm → bool;
         nelm: lm,lm → bool;
      derived
      def
         ltlm(n,m) = not(or(gtlm(n,m),eqlm(n,m)) );
         gelm(n,m) = not(ltlm(n,m));
         lelm(n,m) = not(gtlm(n,m));
         nelm(n,m) = not(eqlm(n,m));
      axiom
         equivrel(eqlm,lm);
         irreflexive(gtlm,lm);
         irreflexive(ltlm,lm);
         transitive(gtlm,lm);
         transitive(ltlm,lm);
         transitive(gelm,lm);
         transitive(lelm,lm);
         antisymmetric(gelm,lm);
         antisymmetric(lelm,lm);
         symmetric(nelm,lm);
   end extend;
is
   extend
      natural
      boolean
```

```
with
   sort
      str;
   primitive
   op
      nullstr: → str;                  /* null string */
      makestr: lm → str;               /* make */
      lenstr: str → nat;               /* string length */
      headstr: str → lm;               /* string head */
      tailstr: str → str;              /* string tail */
      catstr: str,str → str;           /* concatenation */
      eqstr: str,str → bool;
      gtstr: str,str → bool;
   derived
   op
      ltstr: str,str → bool;
      gestr: str,str → bool;
      lestr: str,str → bool;
      nestr: str,str → bool;
   derived
   def
      ltstr(n,m) = not(or(gtstr(n,m),eqstr(n,m)) );
      gestr(n,m) = not(ltstr(n,m));
      lestr(n,m) = not(gtstr(n,m));
      nestr(n,m) = not(eqstr(n,m));
   axiom
      lenstr(nullstr) = zeronat();
      lenstr(makestr(l)) = succnat(zeronat());
      lenstr(catstr(sl,s2)) = sumnat(lenstr(sl),
                                lenstr(s2));
      headstr(makestr(l)) = l;
      tailstr(makestr(l)) = nullstr;
      headstr(catstr(makestr(l),s)) = l;
      tailstr(catstr(makestr(l),s2)) = s2;
      headstr(nullstr) = undef;
      tailstr(nullstr) = nullstr;
      catstr(catstr(sl,s2),s3) = catstr(sl,
                                   catstr(s2,s3));
      catstr(nullstr,s) = catstr(s,nullstr) = s;
      implies(eqlm(ll,l2),eqstr(makestr(ll),
                           makestr(l2)) ) = true();
      implies(gtlm(ll,l2),gtstr(makestr(ll),
                           makestr(l2)) ) = true();
      gtnat(lenstr(makestr(l)),lenstr(nullstr))
                           = true();
      implies(gtnat(lenstr(sl),lenstr(s2)),
                           gtstr(sl,s2)) = true();
      if lenstr(sl)! = zeronat()
      then
         gtnat(lenstr(catstr(sl,s2),lenstr(s2))
                           = true();
```

```
            else
               eqnat(lenstr(catstr(s1,s2),lenstr(s2))
                                 = true();
            endif;
            equivrel(eqstr,str);
            irreflexive(gtstr,str);
            irreflexive(ltstr,str);
            transitive(gtstr,str);
            transitive(ltstr,str);
            transitive(gestr,str);
            transitive(lestr,str);
            antisymmetric(gestr,str);
            antisymmetric(lestr,str);
            symmetric(nestr,str);
   end extend;
end string;


spec str.chartype
is
   extend
      character
   with
      use
         string(character)
      where
         char is lm;
         eqchar is eqlm;
         gtchar is gtlm;
         ltchar is ltlm;
         gechar is gelm;
         lechar is lelm;
         nechar is nelm;
   end extend;
end str.chartype;


spec intensity                              /* display part */
is
   extend
      boolean
   with
      sort
         intens;
      primitive
      op
         minintens: → intens;          /* minimum intensity */
         maxintens: → intens;          /* maximum intensity */
         nullintens: → intens;         /* null intensity */
         predintens: intens → intens;
```

```
   succintens: intens → intens;
   sumintens: intens,intens → intens;
   subintens: intens,intens → intens;
   eqintens: intens,intens → bool;
   gtintens: intens,intens → bool;
derived
op
   ltintens: intens,intens → bool;
   geintens: intens,intens → bool;
   leintens: intens,intens → bool;
   neintens: intens,intens → bool;
derived
def
   ltintens(n,m) = not(or(gtintens(n,m),eqintens(n,m)) );
   geintens(n,m) = not(ltintens(n,m));
   leintens(n,m) = not(gtintens(n,m));
   neintens(n,m) = not(eqintens(n,m));
axiom
   predintens(minintens()) = undef;
   predintens(nullintens()) = undef;
   succintens(maxintens()) = undef;
   succintens(nullintens()) = undef;
   sumintens(i.nullintens()) = undef;
   subintens(i,nullintens()) = undef;
   maxintens() = [NUMINTENS]succintens(minintens())
   sumintens(i,minintens()) = i;
   subintens(i,minintens()) = i;
   sumintens(i,succintens(j)) = succintens
                                (sumintens(i,j));
   if gtintens(i,j) = true()
   then
      subintens(i,succintens(j)) =
                        predintens(subintens(i,j));
   else
      subintens(i,succintens(j)) = undef;
   endif;
   eqintens(i,j) = eqintens(succintens(i),
                        succintens(j));
   eqintens(i,j) = eqintens(predintens(i),
                        predintens(j));
   eqintens(i,succintens(predintens(i)) ) = true();
   eqintens(i,predintens(succintens(i)) ) = true();
   if or(
      eqintens(i,nullintens()),
      eqintens(j,nullintens())
   ) = true()
   then
      gtintens(i,j) = undef;
   endif
   gtintens(succintens(i),i) = true();
   equivrel(eqintens,intens);
```

```
         irreflexive(gtintens,intens);
         irreflexive(ltintens,intens);
         transitive(gtintens,intens);
         transitive(ltintens,intens);
         transitive(geintens,intens);
         transitive(leintens,intens);
         antisymmetric(geintens,intens);
         antisymmetric(leintens,intens);
         symmetric(neintens,intens);
         commutative(sumintens,intens);
         associative(sumintens,intens);
   end extend;
end intensity;


spec pointcolor
is
   extend
      boolean
      intensity
   with
      sort
         color;
      primitive
      op
         nullcolor: → color;              /* null color */
         redcompnt: color → intens;       /* red component */
         grncompnt: color → intens;       /* green component */
         blucompnt: color → intens;       /* blue component */
         eqcolor: color,color → bool;     /* equal color */
         defcolor: intens,intens,intens   color;
                                          /* define color */
      axiom
         redcompnt(nullcolor()) = nullintens();
         grncompnt(nullcolor()) = nullintens();
         blucompnt(nullcolor()) = nullintens();
         if and(
            or(
               or(
                  eqintens(i1,nullintens()),
                  eqintens(i2,nullintens())
               ),
               eqintens(i3,nullintens())
            ),
            or(
               or(
                  not(eqintens(i1,nullintens()) ),
                  not(eqintens(i2,nullintens()) )
               ),
               not(eqintens(i3,nullintens()) )
            )
         ) = true()
```

```
         then
            defcolor(il,i2,i3)) = undef;
         else
            redcompnt(defcolor(il,i2,i3)) = il;
            grncompnt(defcolor(il,i2,i3)) = i2;
            blucompnt(defcolor(il,i2,i3)) = i3;
         endif;
         eqcolor(c1,c2) = and(
            and(
               eqintens(redcompnt(c1),redcompnt(c2)),
               eqintens(grncompnt(c1),grncompnt(c2))
            ),
            eqintens(blucompnt(c1),blucompnt(c2))
         );
         equivrel(eqcolor,color);
   end extend;
end pointcolor;


spec point
is
   extend
      boolean,
      natural,
      integer
   with
      sort
         pnt;
      primitive
      op
         xcord: pnt → int;              /* x coordinate */
         ycord: pnt → int;              /* y coordinate */
         locpnt: int,int → pnt;         /* point location */
         eqpnt: pnt,pnt → bool;         /* equal point */
         gtpnt: pnt,pnt → bool;         /* right & above */
         ltpnt: pnt,pnt → bool;         /* left & below */
         gepnt: pnt,pnt → bool;         /* right & above,
                                           or right inline or above
                                           inline */
         lepnt: pnt,pnt → bool;         /* left & below,
                                           or left inline or below
                                           inline */
         offsetpnt: int,int,pnt → pnt;  /* point offset */
      axiom
         xcord(locpnt(il,i2)) = il;
         ycord(locpnt(il,i2)) = i2;
         eqpnt(p1,p2) = and(
            eqint(xcord(p1),xcord(p2)),
            eqint(ycord(p1),ycord(p2))
         );
```

```
gtpnt(p1,p2) = and(
   gtint(xcord(p1),xcord(p2)),
   gtint(ycord(p1),ycord(p2))
);
ltpnt(p1,p2) = and(
   ltint(xcord(p1),xcord(p2)),
   ltint(ycord(p1),ycord(p2))
);
gepnt(p1,p2) = and(
   or(
      gtint(xcord(p1),xcord(p2)),
      eqint(xcord(p1),xcord(p2))
   ),
   or(
      gtint(ycord(p1),ycord(p2)),
      eqint(ycord(p1),ycord(p2))
   )
);
lepnt(p1,p2) = and(
   or(
      ltint(xcord(p1),xcord(p2)),
      eqint(xcord(p1),xcord(p2))
   ),
   or(
      ltint(ycord(p1),ycord(p2)),
      eqint(ycord(p1),ycord(p2))
   )
);
if x = zeroint()
then
   xcord(offsetpnt(x,y,p)) = xcord(p);
else if gtint(x,zeroint()) = true()
then
   xcord(offsetpnt(x,y,p)) = succint(xcord
            (offsetpnt(predint(x),y,p)) );
else
   xcord(offsetpnt(x,y,p)) = predint(xcord
            (offsetpnt(succint(x),y,p)) );
endif;
endif;
if y = zeroint()
then
   ycord(offsetpnt(x,y,p)) = ycord(p2);
else if gtint(y,zerint()) = true()
then
   ycord(offsetpnt(x,y,p)) = succint(ycord
            (offsetpnt(x,predint(y),p)) );
else
   ycord(offsetpnt(x,y,p)) = predint(ycord
            (offsetpnt(x,succint(y),p)) );
endif;
endif;
```

```
         equivrel(eqpnt,pnt);
         reflexive(gepnt,pnt);
         reflexive(lepnt,pnt);
         irreflexive(gtpnt,pnt);
         irreflexive(ltpnt,pnt);
         transitive(gtpnt,pnt);
         transitive(ltpnt,pnt);
         transitive(gepnt,pnt);
         transitive(lepnt,pnt);
   end extend;
end point;


spec rectangle
is
   extend
      boolean,
      integer,
      point
   with
      sort
         rct;
      primitve
      op
         origin: rct → pnt;              /* lower left
                                 corner */
         corner: rct → pnt;              /* upper right
                                 corner */
         xdimrct: rct → int;             /* x dimension */
         ydimrct: rct → int;             /* y dimension */
         area: pnt,pnt → rct;            /* define rct */
         inrct: pnt,rct → bool;          /* pnt inside rct
                                 test */
         disjrct: rct,rct → bool;        /* disjoint rcts */
         intsctrct: rct,rct → rct;       /* rct intersection */
         putrct: pnt,rct → rct;          /* put rct at
                                 location */
         shiftrct: int,int,rct → rct;    /* shift rct */
      axiom
         if ltint(xcord(p2),xcord(pl)) = true()
         then
            xcord(origin(area(pl,p2)) ) = xcord(p2);
         else
         xcord(origin(area(pl,p2)) ) = xcord(pl);
         endif;
         if ltint(ycord(p2),ycord(pl)) = true()
         then
            ycord(origin(area(pl,p2)) ) = ycord(p2);
         else
            ycord(origin(area(pl,p2)) ) = ycord(pl);
         endif;
```

```
if gtint(xcord(p1),xcord(p2)) = true()
then
   xcord(corner(area(p1,p2)) ) = xcord(p1);
else
   xcord(corner(area(p1,p2)) ) = xcord(p2);
endif;
if gtint(ycord(p1),ycord(p2)) = true()
then
   ycord(corner(area(p1,p2)) ) = ycord(p1);
else
   ycord(corner(area(p1,p2)) ) = ycord(p2);
endif;
inrct(p,r) = and(
   gepnt(p,origin(r)),
   lepnt(p,corner(r))
);
disjrct(r1,r2) =
   not(or(
      or(
         inrct(origin(r2),r1),
         inrct(corner(r2),r1)
      ),
      or(
         inrct(
            locpnt(xcord(origin(r2)),
                   ycord(corner(r2)) ),
            r1
         ),
         inrct(
            locpnt(xcord(corner(r2)),
                   ycord(origin(r2)) ),
            r1
         )
      )
   ));
if disjrct(r1,r2) = true()
then
   intsctrct(r1,r2) = undef;
else
   inrct(p,intsctrct(r1,r2) = and(
      inrct(p,r1),
      inrct(p,r2)
   );
endif;
shiftrct(x,y,r) = area(
   offsetpnt(x,y,origin(r)),
   offsetpnt(x,y,corner(r))
);
putrct(p,r) = area(
   p,
   offsetpnt(xdimrct(r),ydimrct(r),p)
);
```

```
         xdimrct(r) = subint(
            xcord(origin(r)),
            xcord(corner(r))
         );
         ydimrct(r) = subint(
            ycord(origin(r)),
            ycord(corner(r))
         );
   end extend;
end rectangle;


spec imageform
is
   extend
      boolean,
      pointcolor,
      point,
      rectangle
   with
      sort
         form;
      primitive
      op
         initform: rct → form;          /* initialize form */
         farea: form → rct;             /* rct area of
                                 form */
         getcolor: pnt,form → color;    /* get pnt color */
         fillform: color,form → color;  /* fill form */
         setcolor: pnt,color,form → form;
                                 /* set pnt color */
            /****************************************
            * invform - inverse form
            * given color A, color B, form F
            * map F foreground colors to A
            * map F background to B
            * /
         invform: color,color,form → form;
      axiom
         farea(initform(r)) = r;
         getcolor(p,initform(r)) = nullcolor();
         if inrct(p,farea(f)) = true()
            then getcolor(p,setcolor(p,c,f)) = c;
            else getcolor(p,f) = nullcolor();
         endif;
         if inrct(p,farea(f)) = true() then
            getcolor(p,fillform(c,f)) = c;
         endif;
         if inrct(p,farea(f)) = false() then
            setcolor(p,c,f) = undef;
         endif;
```

```
         if inrct(p,farea(f)) = true() then
            if getcolor(p,f) = nullcolor()
            then
               getcolor(p,invform(c1,c2,f)) = c2;
            else
               getcolor(p,invform(c1,c2,f)) = c1;
            endif;
         endif;
   end extend;
end imageform;


spec iconfont
is
   extend
      boolean,
      natural,
      pointcolor,
      rectangle,
      imageform,
      pntblktrans
   with
      sort
         font;
      primitive
      op
         initfont: rct → font;          /* initialize font */
         rctfont: font → rct;           /* rct of font
                                icons */
         lenfont: font → nat;           /* number of icons
                                in font */
         spmap: rct,pnt → pnt;          /* map spot (font
                                loc) to pnt */
         psmap: rct,pnt → pnt;          /* map pnt to spot
                                (font loc) */
         infont: nat,font → bool;       /* for given index,
                                does font have icon */
         delfont: nat,font → font;      /* delete icon
                                from font */
         getfont: form,nat,font → font; /* put icon into
                                font */
         offsetfont: int,int,font,pnt → pnt;
                                        /* offset in
                                multiples of font rcts */
      axiom
         rctfont(initfont(r)) = r;
         lenfont(initfont(r)) = zeronat();
         spmap(r,p) = locpnt(
            mltint(xcord(p),xdimrct(r)),
            mltint(ycord(p),ydimrct(r))
         );
```

```
         psmap(r,p) = locpnt(
            divint(xcord(p),xdimrct(r)),
            divint(ycord(p),ydimrct(r))
         );
         infont(id,initfont(r)) = false();
         infont(id,delfont(id,ft)) = false();
         infont(id,setfont(f,id,ft)) = true();
         if and(
            eqint(xdimrct(rctfont(ft)),xdimrct(farea(f)) ),
            eqint(ydimrct(rctfont(ft)),ydimrct(farea(f)) )
         ) = false()
         then
            setfont(f,id,ft) = undef;
         endif;
         if infont(id,ft) = true()
         then
            lenfont(setfont(f,id,ft)) = lenfont(ft);
         else
            lenfont(setfont(f,id,ft)) = succnat(lenfont(ft));
         endif;
         if infont(id,ft) = true()
         then
            lenfont(delfont(id,ft)) = prednat(lenfont(ft));
         else
            lenfont(delfont(id,ft)) = lenfont(ft);
         endif;
         if infont(id,ft) = false() then
            getfont(id,ft) = undef;
         endif;
         rctfont(ft) = farea(getfont(id,ft));
         getfont(id,setfont(f,id,ft)) = f;
         getfont(id,setfont(a,id,setfont(b,id,ft)) ) = a;
         offsetfont(x,y,ft,p) = locpnt(
            sumint(xcord(p),mltint(x,xdimrct(rctfont(ft)) )),
            sumint(ycord(p),mltint(y,ydimrct(rctfont(ft)) ))
         );
   end extend;
end iconfont;


spec pntblktrans
is
   extend
      natural,
      integer,
      point,
      rectangle,
      form
   with
      sort
         ptblt;
```

```
primitive
op
   initptblt: → ptblt;                  /* initialize ptblt */
   getsrct: ptblt → rct;                /* get source rct */
   getdrct: ptblt → rct;                /* get destination
                        rct */
   getcrct: ptblt → rct;                /* get clipping rct */
   getrule: ptblt → nat;                /* get copy rule */
   setsrct: rct,ptblt → ptblt;          /* set source rct */
   setdrct: rct,ptblt → ptblt;          /* set destination
                        rct */
   setcrct: rct,ptblt → ptblt;          /* set clipping rct */
   setrule: nat,ptblt → ptblt;          /* set copy rule */
      /*************************************
      * copyblt - form copy operation:
      * given source, mask destination forms;
      * call cpyrecur with origin of wksrct
      * ptblt controls operation;
      */
   copyblt: ptblt,form,form,form → form;
      /*************************************
      * drawline - draws line between two pnts:
      * given start pnt, stop pnt, brush, destination
        mask;
      * calls recursive h/v drawloop depending on
        slope of line
      * drawloop constructs line using repeated
      * calls to copyblt using source form as a
        brush
      */
   drawline: pnt,pnt,ptblt,form,form,form → form;
      /*************************************
      * copyfont - copy icon from font to a given
        point in the dest form
      * the source and dest rct in ptblt are
        automatically set
      */
   copyfont: pnt,ptblt,nat,font,form,form → form;
      /*************************************
      * invcopyfont - same as copyfont bit with
        inverse coloring on the
      * the font form source.
      */
   invcopyfont: color,color,pnt,ptblt,nat,font,form,
      form → form;
hidden
op
      /*************************************
      * wksrct - working source rct
      * intersection of source form farea
      * and the ptblt source rct
      */
```

```
wksrct: form,ptblt → rct;
   /*************************************
   * wkdrct - working destination rct
   * intersection of destination form farea
   * and the ptblt destination rct
   */
wkdrct: form,ptblt → rct;
   /*************************************
   * modpnt - pnt modulo (2D modulo):
   * given pnt P, form F
   * if P in F
   *     then P
   *     else(wrap P around into F)
   *        reduce coord of P
   *        by dim of F
   *        until P in F
   */
modpnt: pnt,form → pnt;
   /*************************************
   * getmcolor - applies the masking rules
   * given pnt P, source S, dest D, mask M,
   *     ptblt B
   * returns color MS (masked source color)
   * based on:
   *     masking policy
   *     S color @ P
   *     M color @ modpnt of
   *        matchpnt of P,S,D,B
   */
getmcolor: pnt,ptblt,form,form,form → color;
   /*************************************
   * nextpnt - given pnt P, returns next pnt
   *     in wksrct
   * based on sequential ordering imposed on rct:
   *     start at origin
   *     if right neighbor of P in rct
   *        then return right neighbor of P
   *        else
   *           move left to rct boundary
   *           return pnt above
   */
nextpnt: pnt,ptblt,form → pnt;
   /*************************************
   * matchpnt - find corresponding pnt in dest
   * given pnt P,source S, dest D, ptblt B
   * returns pnt that is offset XY from the
   *     origin of the wkdrct
   * where XY is the offset from the
   *     origin of the wksrct
   *     that equals P
   */
matchpnt: pnt,ptblt,form,form → pnt;
```

```
      /****************************************
      * copypnt - set color at pnt in dest
      * given pnt P, source S, dest D, mask M,
           ptblt B
      * set color @ matchpnt of P,S,D,B
      * based on:
      *     B copy rule
      *     MS color from getmcolor of P,S,M
      *     D color @ matchpnt of P,S,D,B
      */
   copypnt: pnt,ptblt,form,form,form → form;
      /****************************************
      * cpyrecur - recursive function of copybit
      * given pnt P
      * if P in wksrct
      *     then
      *        call copypnt with P
      *        call cpyrecur with nextpnt of P
      *     else
      *        stop recursion
      */
   cpyrecur: pnt,ptblt,form,form,form → form;
      /****************************************
      * hdrawloop - recursive function of drawline
      * used when absolute value of slope is < 45
           degrees
      * walks line one horizontal point at a time
      *     moving vertically as required,
      * at each step:
      *     sets ptblt destination rct
      *     calls copyblt
      */
   hdrawloop: nat,int,int,int,int,int,form,
                form,ptblt → form;
      /****************************************
      * vdrawloop - recursive function of drawline
      * used when absolute value of slope is >= 45
           degrees
      * walks line one vertical point at a time
      *     moving horizontally as required
      * at each step:
      *     sets ptblt destination rct
      *     calls copyblt
      */
   vdrawloop: nat,int,int,int,int,int,form,
                form,ptblt → form;
axiom
   getsrct(initptblt()) = area(
      locpnt(zeroint(),zeroint()),
      locpnt(zeroint(),zeroint())
   );
```

```
getdrct(initptblt()) = area(
   locpnt(zeroint(),zeroint()),
   locpnt(zeroint(),zeroint())
);
getcrct(initptblt()) = area(
   locpnt(zeroint(),zeroint()),
   locpnt(zeroint(),zeroint())
);
getrule(initptblt()) = zeronat();
getsrct(setsrct(r,pb)) = r;
getdrct(setdrct(r,pb)) = r;
getcrct(setcrct(r,pb)) = r;
getrule(setrule(n,pb)) = n;
wksrct(f,pb) = intsctrct(farea(f),getsrct(pb));
wkdrct(f,pb) = intsctrct(farea(f),getdrct(pb));
modpnt(p,f) = offsetpnt(
   modint(xcord(p),xdimrct(farea(f)) ),
   modint(ycord(p),ydimrct(farea(f)) ),
   origin(farea(f))
);
   /*****************************************
   * matchpnt
   * p: pnt in source
   * pb: ptblt
   * s: source
   * d: dest
   *****************************************/
matchpnt(p,pb,s,d) = offsetpnt(
   subint(
      xcord(p),
      xcord(origin(wksrct(s,pb)) )
   ),
   subint(
      ycord(p),
      ycord(origin(wksrct(s,pb)) )
   ),
   origin(wkdrct(d,pb))
);
   /*****************************************
   *getmcolor
   * p: pnt in source
   * pb: ptblt
   * s: source
   * m: mask
   * d: destination
   *****************************************/
if or(
   eqcolor(getcolor(p,s),nullcolor()),
   eqcolor(
      getcolor(modpnt(matchpnt(p,pb,s,d),m),m),
```

```
      nullcolor()
   )
) = true()
then
   getmcolor(p,pb,s,m,d) = getcolor(p,s);
else
   getmcolor(p,pb,s,m,d) = getcolor(modpnt
            (matchpnt(p,pb,s,d),m),m);
endif;
   /*****************************************
   * nextpnt
   * p: pnt in source
   * pb: ptblt
   * s: source
   *****************************************/
if ltint(
   xcord(p),
   xcord(corner(wksrct(s,pb)) )
) = true()
then
   nextpnt(p,pb,s) = locpnt(
      succint(xcord(p)),
      ycord(p)
   );
else
   nextpnt(p,pb,s) = locpnt(
      xcord(origin(wksrct(s,pb)) ),
      succint(ycord(p))
   );
endif;
   /*****************************************
   * copypnt
   * p: pnt in source
   * pt: ptblt
   * s: source
   * m: mask
   * d: destination
   *****************************************/
if inrct(
   matchpnt(p,pb,s,d)
   intsctrct(
      wkdrct(d,pb),
      getcrct(pb)
   )
) = true()
then
   if getrule(pb) = zeronat()
   then
      copypnt(p,pb,s,m,d) = d;
   else if getrule(pb) = [l]succnat(zeronat())
```

```
then
   if and(
      not(eqcolor(getmcolor(p,pb,s,m,d),
            nullcolor()) ),
      not(eqcolor(
         getcolor(
            matchpnt(p,pb,s,d),
            d
         ),
         nullcolor()
      ))
   ) = true()
   then
      copypnt(p,pb,s,m,d) = setcolor(
         matchpnt(p,pb,s,d),
         getmcolor(p,pb,s,m,d),
         d
      );
   else
      copypnt(p,pb,s,m,d) = d;
   endif;
else if getrule(pb) = [2]succnat(zeronat())
then
   if and(
      not(eqcolor(getmcolor(p,pb,s,m,d),
            nullcolor()) ),
      eqcolor(
         getcolor(
            matchpnt(p,pb,s,d),
            d
         ),
         nullcolor()
      )
   ) = true()
   then
      copypnt(p,pb,s,m,d) = setcolor(
         matchpnt(p,pb,s,d),
         getmcolor(p,pb,s,m,d),
         d
      );
   else
      copypnt(p,pb,s,m,d) = d;
   endif;
else if getrule(pb) = [3]succnat(zeronat())
then
   if getmcolor(p,pb,s,m,d) != nullcolor()
   then
      setcolor(
         matchpnt(p,pb,s,d),
         getmcolor(p,pb,s,m,d),
         d
      );
```

```
   else
      copypnt(p,pb,s,m,d) = d;
   endif;
else if getrule(pb) = [4]succnat(zeronat())
then
   if and(
      eqcolor(getmcolor(p,pb,s,m,d),
                  nullcolor()),
      not(eqcolor(
         getcolor(
            matchpnt(p,pb,s,d),
            d
         ),
         nullcolor()
      ))
   ) = true()
   then
      copypnt(p,pb,s,m,d) = setcolor(
         matchpnt(p,pb,s,d),
         nullcolor(),
         d
      );
   endif;
else if getrule(pb) = [5]succnat(zeronat())
then
   if getcolor(
      matchpnt(p,pb,s,d),
      d
   )) != nullcolor()
   then
      copypnt(p,pb,s,m,d) = setcolor(
         matchpnt(p,pb,s,d),
         getmcolor(p,pb,s,m,d),
         d
      );
   else
      copypnt(p,pb,s,m,d) = d;
   endif;
else if getrule(pb) = [6]succnat(zeronat())
then
   if and(
      or(
         eqcolor(getmcolor(p,pb,s,m,d),
                  nullcolor()),
         not(eqcolor(
            getcolor(
                  matchpnt(p,pb,s,d),
                  d
            ),
            nullcolor()
         ))
      ),
```

```
            or(
               not(eqcolor(getmcolor(p,pb,s,m,d),
                        nullcolor())) ),
               eqcolor(
                  getcolor(
                        matchpnt(p,pb,s,d),
                        d
                     ),
                     nullcolor()
               )
            )
         ) = true()
         then
            copypnt(p,pb,s,m,d) = setcolor(
               matchpnt(p,pb,s,d),
               getmcolor(p,pb,s,m,d),
               d
            );
         else
            copypnt(p,pb,s,m,d) = d;
         endif;
      else if getrule(pb) = [7]succnat(zeronat())
      then
         if or(
            not(eqcolor(getmcolor(p,pb,s,m,d),
                        nullcolor()) ),
            not(eqcolor(
               getcolor(
                  matchpnt(p,pb,s,d),
                  d
               ),
               nullcolor()
            ))
         ) = true()
         then
            copypnt(p,pb,sm,d) = setcolor(
               matchpnt(p,pb,s,d),
               getmcolor(p,pb,s,m,d),
               d
            );
         else
            copypnt(p,pb,s,m,d) = d;
      else
            copypnt(p,pb,s,m,d) = d;
         endif;
      endif;
      endif;
      endif;
      endif;
      endif;
      endif;
```

```
   endif;
   endif;
endif;
   /*************************************
   * cpyrecur
   * p: pnt in source
   * pb: ptblt
   * s: source
   * m: mask
   * d: destination
   *************************************/
if inrct(p,wksrct(s,pb)) = true()
then
   /* copy pnt and continue */
   cpyrecur(p,pb,s,m,d) = cpyrecur(
      nextpnt(p,pb,s),
      pb
      s,
      m,
      copypnt(p,pb,s,m,d),
   );
else
   /* all source pnts visited */
   cpyrecur(p,pb,s,m,d) = d;
endif;
   /*************************************
   * copyblt
   * pb: ptblt
   * s: source
   * m: mask
   * d: destination
   *************************************
if or(
   disjrct(farea(s),getsrct(pb)),
   disjrct(farea(d),getdrct(pb))
) = true()
then
   copyblt(pb,s,m,d) = d;
else
   copyblt(pb,s,m,d) =
      cpyrecur(
         origin(wksrct(s,pb)),
         pb
         s,
         m,
         d,
      );
endif;
   /*************************************
   * hdrawloop
   * n: dist to go (major axis)
```

```
   * p: minor axis move counter (vertical)
   * dx: xDelta sign
   * dy: yDelta sign
   * px: yDelta abs
   * py: xDelta abs
   * s: source form
   * d: dest form
   * m: mask form
   * pb: ptblt
   ****************************************/
/* is it the last step? */
if n = succnat(zeronat())
then
   /* time to move in minor direction? */
   if ltint(subint(p,px),zeroint()) = true()
   then
      /* move minor */
      hdrawloop(n,p,dx,dy,px,py,s,d,m,pb =
                  copyblt(
         setdrct(shiftrct(dx,dy,getdrct(pb)),pb)
         s,
         m,
         d,
      );
   else
      /* move major */
      hdrawloop(n,p,dx,dy,px,py,s,d,m,pb) =
                  copyblt(
         setdrct(shiftrct(dx,zeroint(),
                  getdrct(pb)),pb)
         s,
         m,
         d,
      );
   endif;
else if ltint(subint(p,px),zeroint()) = true()
then
   /* move minor and continue */
   hdrawloop(n,p,dx,dy,px,py,s,d,m,pb) =
                  hdrawloop(
      /* reduce distance to go */
      subnat(n,succnat(zeronat()),
      /* reset counter for next minor move */
      sumint(subint(p,px),py),
      dx,
      dy,
      px,
      py,
      s,
      /* move minor and major then copy brush */
```

```
      copyblt(
         setdrct(shiftrct(dx,dy,getdrct(pb)),pb)
         s,
         m,
         d,
      ),
      m,
      setdrct(shiftrct(dx,dy,getdrct(pb)),pb)
   );
else
   /* move major and continue */
   hdrawloop(n,p,dx,dy,px,py,s,d,m,pb) =
               hdrawloop(
      /* reduce dist to go */
      subnat(n,succnat(zeronat()),
      /* reduce count till next minor move */
      subint(p,px),
      dx,
      dy,
      px,
      py,
      s,
      /* move major then copy brush */
      copyblt(
         setdrct(shiftrct(dx,zeroint(),
                     getdrct(pb)),pb)
         s,
         m,
         d,
      ),
      m,
      setdrct(shiftrct(dx,zeroint(),getdrct(pb)),
                     pb)
   );
endif;
endif;
   /*****************************************
   * vdrawloop
   * n: dist to go (major axis)
   * p: minor axis move counter (horizontal)
   * dx: xDelta sign
   * dy: yDelta sign
   * px: yDelta abs
   * py: xDelta abs
   * s: source form
   * d: dest form
   * m: mask form
   * pb: ptblt
   *****************************************/
/* is it the last step? */
if n = succnat(zeronat())
```

```
then
   /* last step */
   if ltint(subint(p,py),zeroint()) = true()
   then
      /* move minor */
      vdrawloop(n,p,dx,dy,px,py,s,d,m,pb) =
                     copyblt(
         setdrct(shiftrct(dx,dy,getdrct(pb)),pb)
         s,
         m,
         d,
      );
   else
      /* move major */
      vdrawloop(n,p,dx,dy,pz,py,s,d,m,pb) =
                     copyblt(
         setdrct(shiftrct(zeroint(),dy,
                     getdrct(pb)),pb)
         s,
         m,
         d,
      );
   endif;
else if ltint(subint(p,py),aeroint()) = true()
then
   /* move minor and continue walk */
   vdrawloop(n,p,dx,dy,px,py,s,d,m,pb) =
                     vdrawloop(
      /* reduce dist to go */
      subnat(n,succnat(zeronat()) ),
      /* set counter for next minor move */
      sumint(subint(p,py),px),
      dx,
      dy,
      px,
      py,
      s,
      /* move minor and major then copy brush */
      copyblt(
         setdrct(shiftrct(dx,dy,getdrct(pb)),pb)
         s,
         m,
         d,
      ),
      m,
      setdrct(shiftrct(dx,dy,getdrct(pb)),pb)
   );
else
   /* move major and continue walk */
   vdrawloop(n,p,dx,dy,px,py,s,d,m,pb) =
                     vdrawloop(
```

```
      /* reduce dist to go */
      subnat(n,succnat(zeronat()) ),
      /* reduce count till minor move */
      subint(p,py),
      dx,
      dy,
      px,
      py,
      s,
      /* move major and copy brush */
      copyblt(
         setdrct(shiftrct(zeroint(),dy,
                  getdrct(pb)),pb)
         s,
         m,
         d,
      ),
      m,
      setdrct(shiftrct(zeroint(),dy,getdrct(pb)),
                  pb)
   );
endif;
endif;
   /****************************************
   * drawline
   * p1: start pnt
   * p2: end pnt
   * pb: ptblk
   * s: brush form
   * m: mask form
   * d: dest form
   ****************************************
if and(
   subint(ycord(p2),ycord(p1)),
   subint(xcord(p2),xcord(p1))
) = true()
then
/* line is a single pnt */
   drawline(p1,p2,pb,s,m,d) = copyblt(pb,s,m,d);
else if ltint(
   absint(subint(ycord(p2,ycord(p1)) ),
   absint(subint(xcord(p2),xcord(p1)) )
) = true()
then
   /* line is horizontal */
   drawline(p1,p2,pb,s,m,d) =
      hdrawloop(
         /* distance to go */
         iton(absint(subint(xcord(p2),xcord(p1)) )),
         /* dist till minor move counter */
```

```
            divint(
               absint(subint(xcord(p2),xcord(pl)) )),
               [2]succint(zeroint())
            ),
            /* dx */
            divint(
               subint(xcord(p2),xcord(pl)),
               absint(subint(xcord(p2),xcord(pl)) )
            ),
            /* dy */
            divint(
               subint(ycord(p2),ycord(pl)),
               absint(subint(ycord(p2),ycord(pl)) )
            ),
            /* px */
            absint(subint(ycord(p2),ycord(pl)) ),
            /* py */
            absint(subint(xcord(p2),xcord(pl)) ),
            s,
            copyblt(pb,s,m,d),
            m,
            pb
         );
   else
      /* line is vertical */
      drawline(pl,p2,pb,s,m,d) =
         vdrawloop(
            /* dist to go */
            iton(absint(subint(ycord(p2),ycord(pl)) )),
            /* dist till minor move counter */
            divint(
               absint(subint(ycord(p2),ycord(pl)) ),
               [2]succint(zeroint())
            ),
            /* dx */
            divint(
               subint(xcord(p2),xcord(pl)),
               absint(subint(xcord(p2),xcord(pl)) )
            ),
            /* dy */
            divint(
               subint(ycord(p2),ycord(pl)),
               absint(subint(ycord(p2),ycord(pl)) )
            ),
            /* px */
            absint(subint(ycord(p2),ycord(pl)) ),
            /* py */
            absint(subint(xcord(p2),xcord(pl)) ),
            s,
            copyblt(pb,s,m,d),
```

```
                m,
                pb
            );
        endif;
        endif;
        /*****************************************
        * copyfont
        * p: position in destination for lower left
                        corner of source form
        * pb: ptblt
        * id: index number
        * ft: font with source form
        * m: mask
        * d: destination form
        *****************************************/
    copyfont(p,pb,id,ft,m,d) = copyblt(
        setdrct(
            putrct(p,rctfont(ft)),
            setsrct(rctfont(ft),pb)
        )
        getfont(id,ft),
        m,
        d,
    );
        /*****************************************
        * invcopyfont
        * c1: foreground color
        * c2: background color
        * p: position in destination for lower left
                    corner of source form
        * pb: ptblt
        * id: index number
        * ft: font with source form
        * m: mask
        * d: destination form
        *****************************************
    invcopyfont(c1,c2,p,pb,id,ft,m,d) = copyblt(
        setdrct(
            putdrct(p,rctfont(ft)),
            setsrct(rctfont(ft),pb)
        )
        invform(c1,c2,getfont(id,ft)),
        m,
        d,
    );
  end extend;
end pntblktrans;
```

```
spec identifiers
is
   extend
      boolean
   with
      sort
         memid;
         regid;
         stkid;
         dregid;
         fid;
         qid;                                /* queue ID */
         dbid;                               /* database ID */
      primitive
      op
         idopers(memid,memseg);              /* memory seg id */
         idopers(regid,regseg);              /* register seg id */
         idopers(stkid,stkseg);              /* stack seg id */
         idopers(dregid,dregseg);            /* display register
                                                seg id */
         idopers(qid,qseg);                  /* queue segment ID */
      axiom
         idaxioms(memid,memseg);
         idaxioms(regid,regseg);
         idaxioms(stkid,stkseg);
         idaxioms(dregid,dregseg);
         idaxioms(qid,qseg);
   end extend;
end identifiers;


spec memaddress
is
   extend
      identifiers,
      boolean
   with
      sort
         memaddr;
      primitive
      op
         startmemaddr: memid → memaddr;
         nextmemaddr: memaddr → memaddr;
         prevmemaddr: memaddr → memaddr;
         getmemid: memaddr → memid;
         offset: int,memaddr → memaddr; /* offset from
                                           memaddr */
         eqmemaddr: memaddr,memaddr → bool;
      axiom
         prevmemaddr(startmemaddr(i)) = undef;
         prevmemaddr(nextmemaddr(m)) = m;
```

```
         nextmemaddr(prevmemaddr(m)) = m;
         offset(succint(n),m) = nextmemaddr(offset(n,m));
         if offset(n,m) = startmemaddr()
         then
            offset(predint(n),m) = undef;
         else
            offset(predint(n),m) = prevmemaddr(offset(n,m));
         endif;
         eqmemid(i,getmemid(offset(n,startmemaddr(i)) )) =
                              true();
         eqmemaddr(startmemaddr(il),startmemaddr(i2)) =
                              eqmemid(il,i2);
         eqmemaddr(startmemaddr(i),nextmemaddr(a)) =
                              false();
         eqmemaddr(nextmemaddr(al),nextmemaddr(a2)) =
                              eqmemaddr(al,a2);
         offset(zeroint(),m) = m;
         equivrel(eqmemaddr,memaddr);
   end extend;
end memaddress;


spec regaddress
is
   extend
      identifiers,
      boolean
   with
      sort
         regaddr;
      primitive
      op
         startregaddr: regid → regaddr;
         nextregaddr: regaddr → regaddr;
         prevregaddr: regaddr → regaddr;
         getregid: regaddr → regid;
         eqregaddr: regaddr,regaddr → bool;
      axiom
         prevregaddr(startregaddr(i)) = undef;
         prevregaddr(nextregaddr(m)) = m;
         nextregaddr(prevregaddr(m)) = m;
         eqregaddr(startregaddr(il),startregaddr(i2)) =
                              eqregid(il,i2);
         eqregaddr(startregaddr(i),nextregaddr(a)) =
                              false();
         eqregaddr(nextregaddr(al),nextregaddr(a2)) =
                              eqregaddr(al,a2);
         equivrel(eqregaddr,regaddr);
   end extend;
end regaddress;
```

```
spec stkaddress
is
   extend
      identifiers,
      boolean
   with
      sort
         stkaddr;
      primitive
      op
         getstkid: stkaddr → stkid;
         eqstkaddr: stkaddr,stkaddr → bool;
      axiom
         eqstkaddr(nextstkaddr(a1),nextstkaddr(a2)) =
                                 eqstkaddr(a1,a2);
         equivrel(eqstkaddr,stkaddr);
   end extend;
end stkaddress;


spec qaddress                              /* database part */
is
   extend
      identifiers,
      boolean
   with
      sort
         qaddr;
      primitive
      op
         getqid: qaddr → qid;
         eqqaddr: qaddr,qaddr → bool;
      axiom
         eqqaddr(nextqaddr(a1),nextqaddr(a2)) =
                                 eqqaddr(a1,a2);
         equivrel(eqqaddr,qaddr);
   end extend;
end qaddress;                              /* database part */


spec dregaddress                           /* display part */
is
   extend
      identifiers,
      boolean
   with
      sort
         dregaddr;
      primitive
      op
         startdregaddr: dregid → dregaddr;
```

```
      nextdregaddr: dregaddr → dregaddr;
      prevdregaddr: dregaddr → dregaddr;
      getdregid: dregaddr → dregid;
      eqdregaddr: dregaddr,dregaddr → bool;
   axiom
      prevdregaddr(startdregaddr(i)) = undef;
      prevdregaddr(nextdregaddr(m)) = m;
      nextdregaddr(prevdregaddr(m)) = m;
      eqdregaddr(startdregaddr(il),startdregaddr(i2)) =
                  eqdregid(il,i2);
      eqdregaddr(startdregaddr(i),nextdregaddr(a)) =
                  false();
      eqdregaddr(nextdregaddr(al),nextdregaddr(a2)) =
                  eqdregaddr(al,a2);
      equivrel(eqdregaddr,dregaddr);
end extend;
end dregaddress;


spec monitorattribute
is
   extend
      boolean
   with
      sort
         mattribute;
      primitive
      op
         xpixels: → mattribute;
         ypixels: → mattribute;
         hscrnsize: → mattribute;
         vscrnsize: → mattribute;
         intenscapbl: → mattribute;
         colorcapbl: → mattribute;
         backgnd: → mattribute;
         dselect: → mattribute;
         eqmattribute: mattribute,mattribute → bool;
      axiom
         equivrel(eqmattribute,mattribute);
   end extend;
end monitorattribute;                    /* display part */


spec files
   is
      extend
         identifiers,
         boolean
      with
         sort
            file;
```

```
         primitive
         op
            getfile: fid → file;
            eqfile: file,file → bool;
         axiom
            eqfile(getfile(i1),getfile(i2)) = eqfid(i1,i2);
            equivrel(eqfile,file);
   end extend;
end files;

spec operatorclasses
is
   sort
      mop;
      dop;
      top;
      qop;
      sop;
      oop;
      rop;
      bop;
end operatorclasses;


spec instructiontype
is
   sort
      instr;
end instructiontype;


spec property_id                          /* database part */
is
   extend
      boolean
   with
      sort
         pid;
      primitive
      op
         pid1: → pid;                     /* 1st property_id */
         pid2: → pid;                     /* 2nd property_id */
          .       .
          .       .
          .       .
         pidn: → pid;                     /* nth property_id */
         eqpid: pid,pid → bool;           /* equal property_id */
      axiom
         equivrel(eqpid,pid);
   end extend;
end property_id;
```

```
spec property_idset
is
   extend
      boolean,
      property_id
   with
      sort
         pidset;
      primitive
      op
         0: → pidset;                          /* empty set */
         u: → pidset;                          /* universe */
         crpidset: pid → pidset;               /* create */
         unpidset: pidset,pidset → pidset;
                                               /* union */
         intpidset: pidset,pidset → pidset;
                                               /* intersection */
         mempidset: pid,pidset → bool;         /* member */
         eqpidset: pidset,pidset → bool;
                                               /* equal */
      axiom
         eqpidset(0,0) = true();               /* empty pidset */
         if eqpidset(ps1,ps2) = true()
         then
            (eqpidset(unpidset(ps1,crpidset(pd1)),
                      unpidset(ps2,crpidset(pd2))) =
             eqpid(pd1,pd2));
         endif;                                /* ps: 'pidset' */
                                               /* pd: 'pid' */
         mempidset(pd1,crpidset(pd1)) = true();
         if mempidset(pd2,crpidset(pd1)) = true()
         then
            eqpid(pd1,pd2) = true();
         endif;
         if and(
               eqpidset(ps1,ps2),
               eqpidset(ps2,ps3)
            ) = true()
         then
            eqpidset(unpidset(ps1,ps2),ps3) = true();
            eqpidset(intpidset(ps1,ps2),ps3) = true();
         endif;
         if and(
               eqpidset(ps1,ps2),
               eqpidset(ps2,ps3)
            ) = true()
         then
            unpidset(ps1,ps2) = unpidset(ps2,ps3);
            intpidset(ps1,ps2) = intpidset(ps2,ps3);
         else unpidset(
                      unpidset(ps1,ps2),
```

```
                        unpidset(ps2,ps3)
                ) = unpidset(
                        unpidset(psl,ps2),
                        ps3
                    );
                    intpidset(
                        intpidset(psl,ps2),
                        intpidset(ps2,ps3
                    ) = intpidset(
                            intpidset(psl,ps2),
                            ps3
                        );
            endif;
            mempidset(pdl,unpidset(psl,ps2))
                    = or(
                        mempidset(pdl,psl),
                        mempidset(pdl,ps2)
                     );
            mempidset(pdl,intpidset(psl,ps2))
                    = and(
                        mempidset(pdl,psl),
                        mempidset(pdl,ps2)
                     );
            if eqpidset(unpidset(psl,ps2),intpidset(psl,ps2))
                            = true()
            then
               eqpidset(psl,ps2) = true();
            endif;
            if and(
                  and(
                     not(eqpid(pdl,pd3)),
                     not(eqpid(pd2,pd3))
                  ),
                   eqpidset(unpidset(crpidset(pdl),
                            crpidset(pd2)),ps3)
               ) = true()
            then
               mempidset(pd3,ps3) = false();
            endif;
            associative(unpidset,pidset);
            associative(intpidset,pidset);
            commutative(unpidset,pidset);
            commutative(intpidset,pidset);
            equivrel(eqpidset,pidset);
      end extend;
   end property_idset;


   spec value
   is
      extend
         boolean
```

```
   with
      sort
         val;
      primitive
      op
         val1: → val;                    /* 1st value */
         val2: → val;                    /* 2nd value */
           .      .
           .      .
           .      .
         valn: → val;                    /* nth value */
         eqval: val,val → bool;          /* equal value */
      axiom
         equivrel(eqval,val);
   end extend;
end value;


spec valueset
is
   extend
      boolean,
      value
   with
      sort
         valset;
      primitive
      op
         0: → valset;                    /* empty set */
         u: → valset;                    /* universe */
         crvalset: val → valset;         /* create */
         unvalset: valset,valset → valset;
                                         /* union */
         intvalset: valset,valset → valset;
                                         /* intersection */
         memvalset: val,valset → bool;   /* member */
         eqvalset: valset,valset → bool;
                                         /* equal */
      axiom
         eqvalset(0,0) = true ();        /* empty valset */
         if eqvalset(vs1,vs2) = true()   /* vs: 'valset' */
         then
            (eqvalset(unvalset(vs1,crvalset(v1)),
                      unvalset(vs2,crvalset(v2))) =
             eqval(v1,v2));              /* v: 'val' */
         endif;
         memvalset(v,crvalset(v)) = true();
         if memvalset(v2,crvalset(v1)) = true()
         then
            eqval(v1,v2) = true();
         endif;
```

```
if and(
      eqvalset(vsl,vs3),
      eqvalset(vs2,vs3)
   ) = true()
then
   eqvalset(unvalset(vsl,vs2),vs3) = true();
   eqvalset(intvalset(vsl,vs2),vs3) = true();
endif;
if and(
      eqvalset(vsl,vs2),
      eqvalset(vs2,vs3)
   ) = true()
then
   unvalset(vsl,vs2) = unvalset(vs2,vs3);
   intvalset(vsl,vs2) = intvalset(vs2,vs3);
else unvalset(
            unvalset(vsl,vs2),
            unvalset(vs2,vs3)
      ) = unvalset(
               unvalset(vsl,vs2),
               vs3
         );
         intvalset(
               intvalset(vsl,vs2),
               intvalset(vs2,vs3)
         ) = intvalset(
                     intvalset(vsl,vs2),
                     vs3
            );
endif;
memvalset(vl,unvalset(vsl,vs2))
         = or(
            memvalset(vl,vsl),
            memvalset(vl,vs2)
           );
memvalset(vl,intvalset(vsl,vs2))
         = and(
            memvalset(vl,vsl),
            memvalset(vl,vs2)
           );
if eqvalset(unvalset(vsl,vs2),intvalset(vsl,vs2))
                     = true()
then
   eqvalset(vsl,vs2) = true();
endif;
if and(
      and(
         not(eqval(vl,v3)),
         not(eqval(v2,v3))
      ),
```

```
                eqvalset(unvalset(crvalset(v1,
                                  crvalset(v2)),vs3)
            ) = true()
         then
            memvalset(v3,vs3) = false();
         endif;
         associative(unvalset,valset);
         associative(intvalset,valset);
         commutative(unvalset,valset);
         commutative(intvalset,valset);
         equivrel(eqvalset,valset);
   end extend;
end valuset;


spec property
is
   extend
      boolean,
      property_id,
      property_idset,
      value,
      valueset
   with
      sort
         prop;
      primitive
      op
         crprop: pid,valset → prop;      /* create */
         eqprop: prop,prop → bool;       /* equal */
         getid: prop → pid;              /* get property_id */
         getvalset: prop → valset;       /* get valueset */
      axiom
         if and(
               eqpid(getid(pr1),getid(pr2)),
               eqvalset(getvalset(pr1),getvalset(pr2))
            ) = true()
         then                            /* pr: property */
            eqprop(pr1,pr2) = true();    /* pd: property_id */
         endif;
         getid(crprop(pd1,vs1)) = pd1;
         getvalset(crprop(pd1,vs1)) = vs1;
         equivrel(eqprop,prop);
   end extend;
end property;


spec propertyset
is
   extend
      boolean,
      property_id,
```

```
   property_idset,
   value,
   valueset,
   property
with
   sort
      propset;
   primitive
   op
      0: → propset;                    /* empty set */
      u: → propset;                    /* universe */
      crpropset: prop → propset;       /* create */
      unpropset: propset,propset → propset;
                                       /* union */
      intpropset: propset,propset → propset;
                                       /* intersection */
      mempropset: prop,propset → bool;
                                       /* member */
      getidset: propset → pidset;      /* get pidset */
      eqpropset: propset,propset → bool;
                                       /* equal propset */
   axiom
      eqpropset(0,0) = true();         /* empty propset */
      if eqpropset(prs1,prs2) = true()
                                       /* prs: 'propset' */
      then
         (eqpropset(unpropset(prs1,crpropset(pr1)),
                    unpropset(prs2,crpropset(pr2))) =
          eqprop(pr1,pr2));
      endif;                           /* pr: 'prop' */
      mempropset(pr,crpropset(pr)) = true();
      if mempropset(pr2,crpropset(pr1)) = true()
      then
         eqprop(pr1,pr2) = true();
      endif;
      if and(
            eqpropset(prs1,prs3),
            eqpropset(prs2,prs3)
         ) = true()
      then
         eqpropset(unpropset(prs1,prs2),prs3) = true();
         eqpropset(intpropset(prs1,prs2),prs3) = true();
      endif;
      if and(
            eqpropset(prs1,prs2),
            eqpropset(prs2,prs3)
         ) = true()
      then
         unpropset(prs1,prs2) = unpropset(prs2,prs3);
         intpropset(prs1,prs2) = intpropset(prs2,prs3);
```

```
        else unpropset(
                 unpropset(prs1,prs2),
                 unpropset(prs2,prs3)
           ) = unpropset(
                    unpropset(prs1,prs2),
                    prs3
               );
           intpropset(
                 intpropset(prs1,prs2),
                 intpropset(prs2,prs3)
           ) = intpropset(
                    intpropset(prs1,prs2),
                    prs3
               );
        endif;
        mempropset(pr,unpropset(prs1,prs2))
              = or(
                 mempropset(pr,prs1),
                 mempropset(pr,prs2)
               );
        mempropset(pr,intpropset(prs1,prs2))
              = and(
                 mempropset(pr,prs1),
                 mempropset(pr,prs2)
               );
        if eqpropset(unpropset(prs1,prs2),
                    intpropset(prs1,prs2)) = true()
        then
           eqpropset(prs1,prs2) = true();
        endif;
        if and(
           and(
              not(eqprop(pr1,pr3)),
              not(eqprop(pr2,pr3))
           ),
            eqpropset(unpropset(crpropset(pr1),
                      crpropset(pr2)),prs3)) = true()
           ) = true()
        then
           mempropset(pr3,prs3) = false();
        endif;
        if mempropset(crprop(pd,vs),prs) = true()
                                        /* pd: 'pid' */
        then                            /* vs: 'valset' */
           mempidset(pd,getidset(prs)) = true();
        endif;
        associative(unpropset,propset);
        associative (intpropset,propset);
        commutative(unpropset,propset);
        commutative(intpropset,propset);
        equivrel(eqpropset,propset);
   end extend;
end propertyset;
```

```
spec propertyvalue                          /*pr:property*/
is
   extend
      boolean,
      property_id,
      property_idset,
      value,
      valueset,
      property
   with
      sort
         pval;
      primitive
      op
         crpval: pid,val → pval;          /* create */
         getpid: pval → pid;              /* get property_id */
         getval: pval → val;              /* get
                                             propertyvalue */
         eqpval: pval,pval → bool;
         memprop: pval,prop → bool;
      axiom
         getpid(crpval(pd,va)) = pd;      /* pd: property_id */
         getval(crpval(pd,va)) = va;      /* va: value */
         if and(
               eqpid(getpid(pv),getid(pr)),
               memvalset(getval(pv),getvalset(pr))
            ) = true()                    /* pv:
                                             propertyvalue */
         then
            memprop(pv,pr) = true();
         endif;
         if and(
               memprop(pv1,pr),
               memprop(pv2,pr)
            ) = true()
         then
            eqpid(getpid(pv1),getpid(pv2)) = true();
         endif;
         if eqpval(crpval(pd1,val),crpval(pd2,va2))
                                  = true()
         then and(
               eqpid(pd1,pd2),
               eqval(val,va2)
            ) = true();
         endif;
         equivrel(eqpval,pval);
   end extend;
end propertyvalue;
```

```
spec propertyvalueset
is
   extend
      boolean,
      property_id,
      property_idset,
      value,
      valueset,
      property,
      propertyvalue
   with
      sort
         pvalset;
      primitive
      op
         0: → pvalset;                    /* empty set */
         u: → pvalset;                    /* universe */
         crpvalset: pval → pvalset;       /* create */
         unpvalset: pvalset,pvalset → pvalset;
                                          /* union */
         intpvalset: pvalset,pvalset → pvalset;
                                          /* intersection */
         mempvalset: pval,pvalset → bool;
                                          /* member */
         mempset: pvalset,propset → bool;
                                          /* member propset */
         getpidset: pvalset → pidset;     /* get pidset */
         eqpvalset: pvalset,pvalset → bool;
                                          /* equal */
      axiom
         eqpvalset(0,0) = true();         /* empty pvalset */
         if eqpvalset(pvsl,pvs2) = true()
                                          /* pvs: 'pvalset' */
         then
            (eqpvalset(unpvalset(pvsl,crpvalset
                                 (crpval(pdl,vl))),
                       unpvalset(pvs2,crpvalset
                                 (crpval(pd2,v2)))) =
             eqpval(crpval(pdl,vl),crpval(pd2,v2)));
         endif;                           /* pd: 'pid' */
                                          /* v: 'val' */
         mempvalset(pv,crpvalset(pv)) = true();
                                          /* pv: 'pval' */
         if mempvalset(pv2,crpvalset(pvl)) = true()
         then
            eqpval(pvl,pv2) = true();
         endif;
         if and(
               eqpvalset(pvsl,pvs3),
               eqpvalset(pvs2,pvs3)
            ) = true()
```

```
then
   eqpvalset(unpvalset(pvsl,pvs2),pvs3) = true();
   eqpvalset(intpvalset(pvsl,pvs2),pvs3) = true();
endif;
if and(
      eqpvalset(pvsl,pvs2),
      eqpvalset(pvs2,pvs3)
   ) = true()
then
   unpvalset(pvsl,pvs2) = unpvalset(pvs2,pvs3);
   intpvalset(pvsl,pvs2) = intpvalset(pvs2,pvs3);
else unpvalset(
         unpvalset(pvsl,pvs2),
         unpvalset(pvs2,pvs3)
   ) = unpvalset(
            unpvalset(pvsl,pvs2),
            pvs3
      );
   intpvalset(
         intpvalset(pvsl,pvs2),
         intpvalset(pvs2,pvs3)
   ) = intpvalset(
            intpvalset(pvsl,pvs2),
            pvs3
      );
endif;
mempvalset(pv,unpvalset(pvsl,pvs2))
      = or(
         mempvalset(pv,pvsl),
         mempvalset(pv,pvs2)
        );
mempvalset(pv,intpvalset(pvsl,pvs2))
      = and(
         mempvalset(pv,pvsl)
         mempvalset(pv,pvs2)
        );
if eqpvalset(unpvalset(pvsl,pvs2),
                intpvalset(pvsl,pvs2)) = true()
   ) = true()
then
   eqpvalset(pvsl,pvs2) = true()
endif;
if and(
    and(
      not(eqpval(pvl,pv3)),
      not(eqpval(pv2,pv3))
   ),
    eqpvalset(unpvalset(crpvalset(pvl),
            crpvalset(pv2)),pvs3)) = true()
then
   mempvalset(pv3,pvs3) = false();
endif;
if mempvalset(crpval(pd,v),pvs) = true()
then
   mempidset(pd,getpidset(pvs)) = true();
endif;
```

```
         if  eqpidset(getpidset(pvs),getidset(prs)) = true()
         then                                  /* prs: 'propset' */
            mempset(pvs,prs) = true();
         endif;
         associative(unpvalset,pvalset);
         associative(intpvalset,pvalset);
         cummutative(unpvalset,pvalset);
         commutative(intpvalset,pvalset);
         equivrel(eqpvalset,pvalset);
   end extend;
end propertyvalueset;


spec object
is
   extend
      boolean
      property_id
      property_idset,
      value,
      propertyvalue,
      propertyvalueset
   with
      sort
         obj;
      primitive
      op
         crobj: pvalset → obj;              /* create */
         getopvalset: obj → pvalset;        /* get
                                               propertyvalueset */
         getopidset: obj → pidset;          /*get
                                               property_idset */
         getoval: obj,pid → val;            /* get value */
         haspval: pval,obj → bool;          /* has
                                               propertyvalue */
         eqobj: obj,obj → bool;             /* equal */
      axiom
         getopvalset(crobj(pvs)) = pvs; /* pvs: 'pvalset' */
         if mempvalset(pv,pvs) = true()
                                            /* /pv: 'pval' */
         then
            haspval(pv,crobj(pvs)) = true();
         endif;
         getopidset(crobj(pvs)) = getpidset(pvs);
         if and(
               (crpval(pd,v) = pv),         /* pd: 'pid' */
               mempvalset(pv,pvs)           /* v: 'val' */
            ) = true()
         then
            getoval(crobj(pvs),pd) = v;
         endif;
```

```
         if eqpvalset(pvsl,pvs2) = true()
         then
            eqobj(crobj(pvsl),crobj(pvs2)) = true();
         endif;
         eqobj(o1,o2) =                    /* o: 'obj' */
            eqpvalset(getopvalset(o1),getopvalset(o2));
         equivrel(eqobj,obj);
   end extend;
end object;


spec objectclass
is
   extend
      boolean,
      property_id,
      property_idset,
      object
   with
      sort
         class;
      primitive
      op
         0: → class;                       /* empty class */
         u: → class;                       /* universe */
         crclass: obj → class;             /* create */
         unclass: class,class → class;     /* union */
         intclass: class,class → class;    /* intersection */
         subclass: class,class → bool;     /* subclass */
         memclass: obj,class → bool;       /* member */
         getcpidset: class → pidset;       /* get pidset
                                              of class */
         insobj: class,obj → class;        /* insert */
         delobj: obj,class → class;        /* delete */
         eqclass: class,class → bool;      /* equal */
      axiom
         eqclass(0,0) = true();            /* empty class */
         if eqclass(c1,c2) = true()        /* c: 'class' */
         then
            eqclass(unclass(c1,crclass(crobj(pvsl))),
                    unclass(c2,crclass(crobj(pvs2)))) =
            eqpvalset(pvsl,pvs2);          /* pvs: 'pvalset' */
         endif;
         if and(
               eqobj(o1,o2),               /* o: 'obj' */
               eqclass(c1,c2)
            ) = true()
         then
            eqclass(insobj(o1,c1),insobj(o2,c2)) = true();
         endif;
```

```
memclass(o,crclass(o)) = true();
if memclass(o2,crclass(o1)) = true()
then
   eqobj(o1,o2) = true();
endif;
if and(
      eqclass(c1,c3),
      eqclass(c2,c3)
   ) = true()
then
   eqclass(unclass(c1,c2),c3) = true();
   eqclass(intclass(c1,c2),c3) = true();
endif;
if and(
      eqclass(c1,c2),
      eqclass(c2,c3)
   ) = true()
then
   unclass(c1,c2) = unclass(c2,c3);
   intclass(c1,c2) = intclass(c2,c3);
else unclass(
         unclass(c1,c2),
         unclass(c2,c3)
     ) = unclass(
            unclass(c1,c2),
            c3
         );
     intclass(
            intclass(c1,c2),
            intclass(c2,c3)
     ) = intclass(
            intclass(c1,c2)
            c3
         );
endif;
memclass(o,unclass(c1,c2))
      = or(
         memclass(o,c1),
         memclass(o,c2)
       );
memclass(o,intclass(c1,c2))
      = and(
         memclass(o,c1),
         memclass(o,c2)
       );
if and(
      and(
         not(eqobj(o1,o2)),
         not(eqobj(o2,o3))
      ),
       eqclass(unclass(crclass(o1),crclass(o2)),c3)
   ) = true()
```

```
            then
               memclass(o3,c3) = false();
            endif;
            if and(
                  memclass(o1,c),
                  memclass(o2,c)
               ) = true()
            then
               eqpidset(getopidset(o1),getopidset(o2)) = true();
            endif;
            if eqclass(unclass(c1,c2),c1) = true()
            then
               subclass(c2,unclass(c1,c2)) = true();
            else if eqclass(unclass(c1,c2),c2) = true()
            then
               subclass(c1,unclass(c1,c2)) = true();
            else and(
                  subclass(c1,unclass(c1,c2)),
                  subclass(c2,unclass(c1,c2))
               ) = true();
            endif;
            delobj(o,insobj(c,o)) = c;
            getcpidset(crclass(o)) = getopidset(o);
            if not(memclass(o1,c)) = true()
            then
               delobj(o1,c) = undef;
            else delobj(o1,unclass(crclass(o1),crclass(o2)))
                              = crclass(o2);
            endif;
            if not (eqpidset(getopidset(o1),getcpidset(c)))
                              = true()
            then
               insobj(c,o1) = undef;
            else insobj(crclass(o2),o1) = unclass(crclass(o2),
                              crclass(o1));
            endif;
            if eqclass(intclass(c1,c2),c1) = true()
            then
               subclass(c1,c2) = true();
            endif;
            associative(unclass,class);
            associative(intclass,class);
            commutative(unclass,class);
            commutative(intclass,class);
            equivrel(eqclass,class);
      end extend;
end objectclass;
```

```
spec database
is
   extend
      property_id,
      property_idset,
      value,
      valueset,
      property,
      propertyset,
      propertyvalue,
      propertyvalueset,
      object,
      objectclass,
      identifiers
   with
      sort
         db;
      primitive
      op
         crdb: dbid,class → db;          /* create */
         insclass: db,class → db;        /* insert new class */
         delclass: class,db → db;        /* delete class */
         retclass: db,class → pvalset;   /* retrieve pvalset
                                            of class */
         retobj: db,pval → obj;          /* retrieve obj by
                                            pval */
         getdbpidset: db → pidset;       /* get pidset */
         modobj: db,obj,pval → db;       /* modify obj */
         getdb: dbid → db;               /* get db by ID */
         getdbid: db → dbid;             /* get ID of db */
         memdb: class,db → bool;         /* member */
         eqdb: db,db → bool;             /* equal db */
         eqdbid: dbid,dbid → bool;       /* equal ID */
      axiom
         if eqdb(d1,d2) = true()         /* d: 'db' */
         then
            eqdb(insclass(d1,c1),insclass(d2,c2)) =
            eqclass(c1,c2);
         endif;
         memdb(c,crdb(i,c)) = true();    /* c: 'class' */
         memdb(c,insclass(d,c)) = true();
                                         /* i: ID */
         delclass(c,insclass(d,c)) = d;
         if and(
               and(
                  eqpvalset(getopvalset(o),pvs),
                                         /* o: 'obj' */
                  memclass(o,c)          /* pvs: 'pvalset' */
               ),
                memdb(c,d)
            ) = true()
```

```
then
   intpvalset(retclass(d,c),pvs) = pvs;
endif;
if and(
      and(
         haspval(pv,o),        /* pv: 'pval' */
         memclass(o,c)
      ),
       memdb(c,d)
   ) = true()
then
   retobj(d,pv) = o;
endif;
if not(memdb(cl,d)) = true()
then
   delclass(cl,d) = undef;
else delclass(cl,crdb(i,unclass(cl,c2))) =
                     crdb(i,c2);
endif;
if memdb(cl,d) = true()
then
   insclass(d,cl) = undef;
else if and(
           and(
               not(memdb(cl,d)),
               memdb(c2,d)
           ),
            not(eqpidset(getcpidset(cl),
                         getcpidset(c2)))
      ) = true()
then and(
      eqclass(insclass(crdb(i,c2),cl),
                       crdb(i,unclass(cl,c2))),
      eqpidset(getdbpidset(d),unpidset
               (getcpidset(cl),getcpidset(c2)))
   ) = true();
endif;
if and(
      and(
         memclass(o,c),
         haspval(pv,o)
      ),
       memdb(c,d)
   ) = true()
then
   mempvalset(pv,retclass(d,c)) = true();
endif;
```

```
         if and(
               and(
                  and(
                     and(                    /* pv: 'pval' */
                        memprop(pv,crprop(pd,vs)),
                                             /* pd: 'pid' */
                        mempidset(pd,getidset(prs))
                                             /* vs: 'valset' */
                     ),                      /* prs: 'propset' */
                      eqpidset(getopidset(o),getidset(prs))
                  ),                         /* o: 'obj' */
                   memclass(o,c)             /* d: 'db' */
               ),
                memdb(c,d)
            ) = true()
         then and(
               haspval(pv,o),
               memdb(crclass(o),modobj(d,o,pv))
            ) = true();
         else modobj(d,o,pv) = undef;
         endif;
         getdbid(crdb(i,c)) = i;
         if eqdb(getdb(i1),getdb(i2)) = true()
         then
            eqdbid(i1,i2) = true();
         endif;
         equivrel(eqdb,db);
         equivrel(eqdbid,dbid);
   end extend;
end database;


spec list
parm
   extend
      boolean,
      string
   with
      sort
         elm;
      primitive
      op
         eqelm: elm,elm → bool;          /* equal */
      axiom
         equivrel(eqelm,elm);
   end extend;
is
   extend
      natural,
      boolean
```

```
with
   sort
      lst;
   primitive
   op
      nullst: → lst;                    /* empty list */
      makelst: elm → lst;               /* make list from
                                           elm */
      makenewlst: lst → lst;            /* make list from
                                           list */
      firstelm: lst → elm;              /* first elm of
                                           list */
      firstlst: lst → lst;              /* first lst of
                                           list */
      restlst: lst → lst;               /* rest of list */
      catlst: lst,lst → lst;            /* concatenate two
                                           lst */
      catelm: elm,lst → lst;            /* concatenate elm
                                           to lst */
      memelm: elm,lst → bool;           /* elm member of
                                           lst */
      memblst: lst,lst → bool;          /* lst member of
                                           lst */
      lenlst: lst → nat;                /* length of lst */
      unlst: lst,lst → lst;             /* union */
      intlst: lst,lst → lst;            /* intersection */
      inslst: lst,lst → lst;            /* insert */
      delst: lst,lst → lst;             /* delete */
      getlst: lst,lst → lst;            /* get list */
      sofirstlst: lst → lst;            /* set of first
                                           lists */
      retobjlst: lst,lst → lst;         /* retrieve objlst */
      modlst: lst,lst → lst;            /* modify */
      eqlst: lst,lst → bool;            /* equal */
   axiom
      firstelm(makelst(k)) = k;
      firstelm(catlst(makelst(k),l)) = k;
      firstelm(nullst()) = undef;
      firstelm(makenewlst(makelst(k))) = k;
      restlst(catlst(makelst(k),l)) = l;
      restlst(nullst()) = undef;
      restlst(makelst(k)) = nullst();
      restlst(catlst(l1,l2)) = l2;
      firstlst(catlst(l1,l2)) = l1;
      if makelst(k) = l
      then
         firstlst(makelst(k)) = l;
      endif;
      lenlst(nullst()) = zeronat();
      lenlst(makelst(k)) = succnat(zeronat());
```

```
lenlst(restlst(l)) = subnat(lenlst(l),
                     succnat(zeronat()));
lenlst(catlst(l1,l2)) = sumnat(lenlst(l1),lenlst(l2));
if and(
      (l != nullst()),
      (restlst(l) = nullst())
   ) = true()
then
   lenlst(l) = succnat(zeronat());
endif;
catlst(catlst(l1,l2),l3) = catlst(l1,catlst(l2,l3));
catlst(nullst(),l) = catlst(l,nullst()) = l;
implies(eqelm(k1,k2),eqlst(makelst(k1),
                     makelst(k2))) = true();
gtnat(lenlst(makelst(k)),lenlst(nullst()) = true();
if (lenlst(l1) != zeronat())
then
   gtnat(lenlst(catlst(l1,l2),lenlst(l2))) = true();
else eqnat(lenlst(catlst(l1,l2),lenlst(l2)))
                                 = true();
endif;
if and(
      (l1 = nullst()),
      (l2 = nullst())
   ) = true()
then
   eqlst(l1,l2) = true();
else if (firstelm(l1) != firstelm(l2))
then
   eqlst(l1,l2) = false();
else eqlst(restlst(l1),restlst(l2));
endif;
if (l = nullst())
then
   catelm(k,l) = makelst(k);
else if (makelst(k) = nullst())
then
   catelm(k,l) = l;
else firstelm(l) = k;
endif;
if (l1 = nullst())
then
   unlst(l1,l2) = l2;
else if memelm(firstelm(l1),l2) = true()
then
   unlst(restlst(l1),l2);
else catelm(firstelm(l1),unlst(restlst(l1),l2));
endif;
memelm(firstelm(l),l) = true();
memelm(k,makelst(k)) = true();
eqlst(l,makenewlst(restlst(l))) = false();
equivrel(eqlst,lst);
```

```
if eqlst(firstlst(l1),nullst()) = true()
                                  /* recursion for
                                     memblst */
then
   memblst(l1,l2) = true();
else if and(
         not(eqlst(firstlst(l1),nullst()) = true()),
         (eqlst(firstlst(l2),nullst()) = true())
      ) = true()
then
   memblst(l1,l2) = false();
else if eqlst(firstlst(l1),firstlst(l2)) = true()
then
   memblst(restlst(l1),restlst(l2));
else memblst(firstlst(l1),restlst(l2));
endif;
if or(                            /* recursion for
                                     intlst */
   (eqlst(l1,nullst()) = true()),
   (eqlst(l2,nullst()) = true())
  ) = true()
then
  intlst(firstlst(l1),l2) = nullst();
else if memblst(firstlst(l1),l2) = true()
then
  catlst(firstlst(l1),intlst(restlst(l1),l2));
else intlst(restlst(l1),l2);
endif;
if or(                            /* recursion for
                                     getlst */
   (eqlst(l1,nullst()) = true()),
   (eqlst(l2,nullst()) = true())
  ) = true()
then
  getlst(l1,l2) = undef;
else if eqlst(firstlst(l1),l2) = true()
then
  getlst(l1,l2) = firstlst((restlst(l1));
else if eqlst(firstlst(restlst(l1)),l2) = true()
then
  getlst(l1,l2) = firstlst(l1);
else getlst(restlst(restlst(l1)),l2);
endif;
if or(                            /* recursion for
                                     delst */
   (eqlst(l2,nullst()) = true()),
   (not(memblst(l1,l2)) = true())
  ) = true()
then
   delst(l1,l2) = undef;
else if not(eqlst(l1,firstlst(l2))) = true()
```

```
        then
           makenewlst(catlst(firstlst(l2),delst(l1,
                              restlst(l2))));
        else makenewlst(restlst(l2));
        endif;
        if                                  /* recursion for
                                               sofirstlst */
           ltnat(lenlst(l),succnat(succnat(zeronat())))
                                            = true()
        then
           sofirstlst(l) = undef;
        else if eqnat(lenlst(l),succnat(succnat(zeronat())))
                                            = true()
        then
           sofirstlst(l) = firstlst(l);
        else catlst(firstlst(l),sofirstlst(restlst
                                (restlst(l)));
        endif;
        if                                  /* recursion for
                                               retobjlst */
           eqlst(firstlst(l1),nullst()) = true()
        then
           retobjlst(l1,l2) = nullst();
        else if intlst(l2,firstlst(l1)) = l2
        then
           catlst(firstlst(l1),retobjlst(restlst(l1),l2));
        else retobjlst(restlst(l1),l2);
        endif;
        if and(
              and(
                 and(
                    and(
                       memblst(l1,makenewlst(unlst(l2,l3))),
                       memblst(l2,sofirstlst(l4))
                    ),
                     (sofirstlst(l5) = sofirstlst(l4))
                 ),
                  memblst(l5,l6)
              ),
               memblst(l6,l7)
           ) = true()
        then
           memblst(makenewlst(l5),modlst(l7,l5,l1))
                                 = true();
        else modlst(l7,l5,l1) = undef;
        endif;
   end extend;
end list;
```

```
spec pidlist
is
   use
      list(property_id)
   where
      pid is elm;
      eqpid is eqelm;
end pidlist;


spec pidsetlist
is
   use
      list(property_idset)
   where
      pidset is elm;
      eqpidset is eqelm;
end pidsetlist;


spec vallist
is
   use
      list(value)
   where
      val is elm;
      eqval is eqelm;
end vallist;


spec valsetlist
is
   use
      list(valueset)
   where
      valset is elm;
      eqvalset is eqelm;
end valsetlist;


spec proplist
is
   use
      list(property)
   where
      prop is elm;
      eqpropis eqelm;
end proplist;
```

```
spec propsetlist
is
   use
      list(propertyset)
   where
      propset is elm;
      eqpropset is eqelm;
end propsetlist;


spec pvallist
is
   use
      list(propertyvalue)
   where
      pval is elm;
      eqpval is eqelm;
end pvallist;


spec pvalsetlist
is
   use
      list(propertyvalueset)
   where
      pvalset is elm;
      eqpvalset is eqelm;
end pvalsetlist;


spec objlist
is
   use
      list(object)
   where
      obj is elm;
      eqobj is eqelm;
end objlist;


spec classlist
is
   use
      list(objectclass)
   where
      class is elm;
      eqclass is eqelm;
end classlist;
```

```
spec dblist
is
   use
      list(database)
   where
      db is elm;
      eqdb is eqelm;
end dblist;                                  /* database part */


spec typing
is
   extend
      boolean,
      natural,
      integer,
      character,
      str.chartype,
      intensity,
      pointcolor,
      point,
      rectangle,
      imageform,
      pntblktrans,
      iconfont,
      identifiers,
      memaddress,
      regaddress,
      stkaddress,
      dregaddress,
      monitorattribute,
      files,
      operatorclasses,
      instructiontype,
      pidlist,                               /* database part */
      pidsetlist,
      vallist,
      valsetlist,
      proplist,
      propsetlist,
      pvallist,
      pvalsetlist,
      objlist,
      classlist
      dblist                                 /* database part */
   with
      sort
         type;
         val;
```

```
primitive
op
   typingopers(bool);
   typingopers(nat);
   typingopers(int);
   typingopers(char);
   typingopers(str.char);
   typingopers(intens);
   typingopers(color);
   typingopers(pnt);
   typingopers(rct);
   typingopers(form);
   typingopers(ptblt);
   typingopers(font);
   typingopers(memid);
   typingopers(regid);
   typingopers(stkid);
   typingopers(dregid);
   typingopers(fid);
   typingopers(memaddr);
   typingopers(regaddr);
   typingopers(stkaddr);
   typingopers(dregaddr);
   typingopers(mattribute);
   typingopers(file);
   typingopers(mop);
   typingopers(dop);
   typingopers(top);
   typingopers(qop);
   typingopers(sop);
   typingopers(oop);
   typingopers(rop);
   typingopers(bop);
   typingopers(instr);
   typingopers(pidlist.lst);        /* database part */
   typingopers(pidsetlist.lst);
   typingopers(vallist.lst);
   typingopers(valsetlist.lst);
   typingopers(proplist.lst);
   typingopers(propsetlist.lst);
   typingopers(pvallist.lst);
   typingopers(pvalsetlist.lst);
   typingopers(objlist.lst);
   typingopers(classlist.lst);
   typingopers(dblist.lst);         /* database part */
hidden
op
   whattype: val → type;
   eqtype: type,type → bool;
```

```
      axiom
         typingaxioms(bool);
         typingaxioms(nat);
         typingaxioms(int);
         typingaxioms(char);
         typingaxioms(str.char);
         typingaxioms(intens);
         typingaxioms(color);
         typingaxioms(pnt);
         typingaxioms(rct);
         typingaxioms(form);
         typingaxioms(ptblt);
         typingaxioms(font);
         typingaxioms(memid);
         typingaxioms(regid);
         typingaxioms(stkid);
         typingaxioms(dregid);
         typingaxioms(fid);
         typingaxioms(memaddr);
         typingaxioms(regaddr);
         typingaxioms(stkaddr);
         typingaxioms(dregaddr);
         typingaxioms(mattribute);
         typingaxioms(file);
         typingaxioms(mop);
         typingaxioms(dop);
         typingaxioms(top);
         typingaxioms(qop);
         typingaxioms(sop);
         typingaxioms(oop);
         typingaxioms(rop);
         typingaxioms(bop);
         typingaxioms(instr);
         typingaxioms(pidlist.lst);      /* database part */
         typingaxioms(pidsetlist.lst);
         typingaxioms(vallist.lst);
         typingaxioms(valsetlist.lst);
         typingaxioms(proplist.lst);
         typingaxioms(propsetlist.lst);
         typingaxioms(pvallist.lst);
         typingaxioms(pvalsetlist.lst);
         typingaxioms(objlist.lst);
         typingaxioms(classlist.lst);
         typingaxioms(dblist.lst);       /* database part */
         equivrel(eqtype,type);
   end extend;
end typing;
```

```
spec operators
is
   extend
      operatorclasses,
      typing
   with
      primitive
      op
         boolnot: → mop;
         booland: → dop;
         boolor: → dop;
         natpred: → mop;
         natsucc: → mop;
         natsum: → dop;
         natsub: → dop;
         nateq: → rop;
         natgt: → rop;
         natlt: → rop;
         intpred: → mop;
         intsucc: → mop;
         intabs: → mop;
         intntoi: → mop;
         intiton: → mop;
         intsum: → dop;
         intsub: → dop;
         intmlt: → dop;
         intdiv: → dop;
         intmod: → dop;
         inteq: → rop;
         intgt: → rop;
         intlt: → rop;
         chareq: → rop;
         chargt: → rop;
         charstrlen: → mop;
         charmakestr: → mop;
         charheadstr: → mop;
         chartailstr: → mop;
         charcatstr: → dop;
         str.chareq: → rop;
         str.chargt: → rop;
         intenspred: → mop;
         intenssucc: → mop;
         intenssum: → dop;
         intenssub: → dop;
         intenseq: → rop;
         intensgt: → rop;
         colorredcompnt: → mop;
         colorgrncompnt: → mop;
         colorblucompnt: → mop;
         colordef: → top;
         coloreq: → rop;
```

```
pntxcord: → mop
pntycord: → mop;
pntloc: → dop;
pntoffset: → top;
pnteq: → rop;
pntgt: → rop;
pntlt: → rop;
pntge: → rop;
pntle: → rop;
rctorigin: → mop;
rctcorner: → mop;
rctxdim: → mop;
rctydim: → mop;
rctarea: → dop;
rctin: → dop;
rctdisj: → dop;
rctint: → dop;
rctput: → dop;
rctshift: → top;
forminit: → mop;
formfarea: → mop;
formgetcolor: → dop;
formfill: → dop;
formsetcolor: → top;
forminv: → top;
fontinit: → mop;
fontrct: → mop;
fontlen: → mop;
fontspmap: → dop;
fontpsmap: → dop;
fontin: → dop;
fontdel: → dop;
fontget: → dop;
fontset: → top;
fontoffset: → qop;
ptbltgetsrct: → mop;
ptbltgetdrct: → mop;
ptbltgetcrct: → mop;
ptbltgetrule: → mop;
ptbltsetsrct: → dop;
ptbltsetdrct: → dop;
ptbltsetcrct: → dop;
ptbltsetrule: → dop;
ptbltcopy: → qop;
ptbltdrawline: → sop;
ptbltfont: → sop;
ptbltfontinv: → oop;
pidlist.eqlst: → rop;          /* database part */
pidsetlist.makenewlst: → mop;
pidsetlist.unlst: → dop;
pidsetlist.intlst: → dop;
pidsetlist.memblst: → rop;
```

```
pidsetlist.eqlst: → rop;
vallist.eqlst: → rop;
valsetlist.makenewlst: → mop;
valsetlist.unlst: → dop;
valsetlist.intlst: → dop;
valsetlist.memblst: → rop;
valsetlist.eqlst: → rop;
proplist.firstlst: → mop;
proplist.restlst: → mop;
proplist.catlst: → dop;
proplist.eqlst: → rop;
propsetlist.makenewlst: → mop;
propsetlist.sofirstlst: → mop;
propsetlist.unlst: → dop;
propsetlist.intlst: → dop;
propsetlist.memblst: → rop;
propsetlist.eqlst: → rop;
pvallist.firstlst: → mop;
pvallist.restlst: → mop;
pvallist.catlst: → dop;
pvallist.memblst: → rop;
pvallist.eqlst: → rop;
pvalsetlist.makenewlst: → mop;
pvalsetlist.sofirstlst: → mop;
pvalsetlist.unlst: → dop;
pvalsetlist.intlst: → dop;
pvalsetlist.memblst: → rop;
pvalsetlist.eqlst: → rop;
objlist.makenewlst: → mop;
objlist.sofirstlst: → mop;
objlist.getlst: → dop;
objlist.eqlst: → rop;
objlist.memblst: → rop;
classlist.makenewlst: → mop;
classlist.sofirstlst: → mop;
classlist.unlst: → dop;
classlist.intlst: → dop;
classlist.catlst: → dop;
classlist.delst: → dop;
classlist.memblst: → rop;
classlist.eqlst: → rop;
dblist.makenewlst: → mop;
dblist.sofirstlst: → mop;
dblist.catlst: → dop;
dblist.delst: → dop;
dblist.intlst: → dop;
dblist.modlst: → top;
dblist.memblst: → rop;          /* database part */
isbool: → bop;
isnat: → bop;
```

```
    isint: → bop;
    ischar: → bop;
    isstr.char: → bop;
    isintens: → bop;
    iscolor: → bop;
    ispnt: → bop;
    isrct: → bop;
    isform: → bop;
    isptblt: → bop;
    isfont: → bop;
    ismemid: → bop;
    isregid: → bop;
    isstkid: → bop;
    isdregid: → bop;
    isfid: → bop;
    ismemaddr: → bop;
    isregaddr: → bop;
    isstkaddr: → bop;
    isdregaddr: → bop;
    isfile: → bop;
    ismop: → bop;
    isdop: → bop;
    istop: → bop;
    isqop: → bop;
    issop: → bop;
    isoop: → bop;
    isrop: → bop;
    isbop: → bop;
    isinstr: → bop;
    ispidlist.lst: → bop;          /* database part */
    ispidsetlist.lst: → bop;
    isvallist.lst: → bop;
    isvalsetlist.lst: → bop;
    isproplist.lst: → bop;
    ispropsetlist.lst: → bop;
    ispvallist.lst: → bop;
    ispvalsetlist.lst: → bop;
    isobjlist.lst: → bop;
    isclasslist.lst: → bop;
    isdblist.lst: → bop;           /* database part */
hidden
op
    applymop: mop,val → val;
    applydop: dop,val,val → val;
    applytop: top,val,val,val → val;
    applyqop: qop,val,val,val,val → val;
    applysop: sop,val,val,val,val,val,val → val;
    applyoop: oop,val,val,val,val,val,val,val,val → val;
    applyrop: rop,val,val → val;
    applybop: bop,val → val;
```

```
axiom
   applymop(boolnot(),v) = valofbool(not(
                atomofbool(v)) );
   applydop(booland(),vl,v2) = valofbool(and(
                atomofbool(vl),atomofbool(v2)) );
   applydop(boolor(),vl,v2) = valofbool(or(
                atomofbool(vl),atomofbool(v2)) );
   applymop(natpred(),v) = valofnat(prednat(
                atomofnat(v)) );
   applymop(natsucc(),v) = valofnat(succnat(
                atomofnat(v)) );
   applydop(natsum(),vl,v2) = valofnat(sumnat(
                atomofnat(vl),atomofnat(v2)) );
   applydop(natsub(),vl,v2) = valofnat(subnat(
                atomofnat(vl),atomofnat(v2)) );
   applymop(intpred(),v) = valofint(predint(
                atomofint(v);
   applymop(intsucc(),v) = valofint(succint(
                atomofint(v);
   applymop(intabs(),v) = valofint(absint(
                atomofint(v);
   applymop(intntoi(),v) = valofint(ntoi(
                atomofnat(v)) );
   applymop(intiton(),v) = valofnat(iton(
                atomofint(v)) );
   applydop(intsum(),vl,v2) = valofint(sumint(
                atomofint(vl),atomofint(v2)) );
   applydop(intsub(),vl,v2) = valofint(subint(
                atomofint(vl),atomofint(v2)) );
   applydop(intmlt(),vl,v2) = valofint(mltint(
                atomofint(vl),atomofint(v2)) );
   applydop(intdiv(),vl,v2) = valofint(divint(
                atomofint(vl),atomofint(v2)) );
   applydop(intmod(),vl,v2) = valofint(modint(
                atomofint(vl),atomofint(v2)) );
   applymop(charstrlen(),v) = valofnat(lenstr.char(
                atomofstr.char(v)) );
   applymop(charmakestr(),v) = valofstr.char(
                makestr.char(atomofchar(v)) );
   applymop(charheadstr(),v) = valofchar(headstr.char(
                atomofstr.char(v)) );
   applymop(chartailstr(),v) = valofstr.char(
                tailstr.char(atomofstr.char(v)) );
   applydop(charcatstr(),vl,v2) = valofstr.char(
                catstr.char(atomofstr.char(vl),
                atomofstr.char(v2)
   ));
   applymop(intenspred(),v) = valofintens(
                predintens(atomofintens(v);
   applymop(intenssucc(),v) = valofintens(
                succintens(atomofintens(v);
```

```
applydop(intenssum(),vl,v2) = valofintens(
        sumintens(atomofintens(vl),
        atomofintens(v2)
));
applydop(intenssub(),vl,v2) = valofintens(
        subintens(atomofintens(vl),
        atomofintens(v2)
));
applymop(colorredcompnt(),v) = valofintens(
        redcompnt(atomofcolor(v)) );
applymop(colorgrncompnt(),v) = valofintens(
        grncompnt(atomofcolor(v)) );
applymop(colorblucompnt(),v) = valofintens(
        blucompnt(atomofcolor(v)) );
applytop(colordef(),vl,v2,v3) = valofcolor(
        defcolor(atomofintens(vl),
                 atomofintens(v2),
                 atomofintens(v3)
));
applymop(pntxcord(),v) = valofint(xcord(
                 atomofpnt(v)) );
applymop(pntycord(),v) = valofint(ycord(
                 atomofpnt(v)) );
applydop(pntloc(),vl,v2) = valofpnt(locpnt(
        atomofint(vl),atomofint(v2)) );
applytop(pntoffset(),vl,v2,v3) = valofpnt(
        offsetpnt(atomofint(vl),
                  atomofpnt(v2),
                  atomofpnt(v3)
));
applymop(rctorigin(),v) = valofpnt(origin(
        atomofrct(v)) );
applymop(rctcorner(),v) = valofpnt(corner(
        atomofrct(v)) );
applymop(rctxdim(),v) = valofint(xdimrct(
        atomofrct(v)) );
applymop(rctydim(),v) = valofint(ydimrct(
        atomofrct(v)) );
applydop(rctarea(),vl,v2) = valofrct(area(
        atomofpnt(vl),atomofpnt(v2)) );
applydop(rctin(),vl,v2) = valofbool(inrct(
        atomofpnt(vl),atomofrct(v2)) );
applydop(rctdisj(),vl,v2) = valofbool(disjrct(
        atomofrct(vl),atomofrct(v2)) );
applydop(rctint(),vl,v2) = valofrct(intsctrct(
        atomofrct(vl),atomofrct(v2)) );
applydop(rctput(),vl,v2) = valofrct(putrct(
        atomofpnt(vl),atomofrct(v2)) );
applytop(rctshift(),vl,v2,v3) = valofrct(shiftrct(
        atomofint(vl),
        atomofint(v2),
        atomofint(v3),
));
```

```
applymop(forminit(),v) = valofform(initform(
        atomofrct(v)) );
applymop(formfarea(),v) = valofrct(farea(
        atomofform(v)) );
applydop(formgetcolor(),vl,v2) = valofcolor(
        getcolor(atomofpnt(vl),atomofform(v2)) );
applydop(formfill(),vl,v2) = valofform(fillform(
        atomofcolor(vl),atomofform(v2)) );
applytop(formsetcolor(),vl,v2,v3) = valofform(
        setcolor(atomofpnt(vl),
                 atomofpnt(v2),
                 atomofpnt(v3)
));
applytop(forminv(),vl,v2,v3) = valofform(
        invform(atomofcolor(vl),
                atomofcolor(v2),
                atomofcolor(v3)
));
applymop(fontinit(),v) = valoffont(initfont(
        atomofrct(v)) );
applymop(fontrct(),v) = valofrct(rctfont(
        atomoffont(v)) );
applymop(fontlen(),v) = valofnat(lenfont(
        atomoffont(v)) );
applydop(fontspmap(),vl,v2) = valofpnt(spmap(
        atomofrct(vl),atomofpnt(v2)) );
applydop(fontpsmap(),vl,v2) = valofpnt(psmap(
        atomofrct(vl),atomofpnt(v2)) );
applydop(fontin(),vl,v2) = valofbool(infont(
        atomofnat(vl),atomoffont(v2)) );
applydop(fontdel(),vl,v2) = valoffont(delfont(
        atomofnat(vl),atomoffont(v2)) );
applydop(fontgetfont(),vl,v2) = valofform(getfont(
        atomofnat(vl),atomoffont(v2)) );
applytop(fontset(),vl,v2,v3) = valoffont(setfont(
        atomofform(vl),
        atomofnat(v2),
        atomoffont(v3)
));
applyqop(fontoffset(),vl,v2,v3,v4) = valofpnt(
        offsetfont(atomofint(vl),
                   atomofint(v2),
                   atomoffont(v3),
                   atomofpnt(v4)
));
applymop(ptbltgetsrct(),v) = valofrct(getsrct(
        atomofptblt(v)) );
applymop(ptbltgetdrct(),v) = valofrct(getdrct(
        atomofptblt(v)) );
```

```
applymop(ptbltgetcrct(),v) = valofrct(getcrct(
        atomofptblt(v)) );
applymop(ptbltgetrule(),v) = valofnat(getrule(
        atomofptblt(v)) );
applydop(ptbltsetsrct(),vl,v2) = valofptblt(
        setsrct(atomofrct(vl),atomofptblt(v2)) );
applydop(ptbltsetdrct(),vl,v2) = valofptblt(
        setdrct(atomofrct(vl),atomofptblt(v2)) );
applydop(ptbltsetcrct(),vl,v2) = valofptblt(
        setcrct(atomofrct(vl),atomofptblt(v2)) );
applydop(ptbltsetrule(),vl,v2) = valofptblt(
        setrule(atomofnat(vl),atomofptblt(v2)) );
applyqop(ptbltcopy(),vl,v2,v3,v4) = valofform(
        copyblt(atomofptblt(vl),
                atomofform(v2),
                atomofform(v3),
                atomofform(v4)
));
applysop(ptbltdrawline(),vl,v2,v3,v4,v5,v6) =
        valofform(drawline(atomofpnt(vl),
                           atomofpnt(v2),
                           atomofblt(v3),
                           atomofform(v4),
                           atomofform(v5),
                           atomofform(v6)
));
applysop(ptbltfont(),vl,v2,v3,v4,v5,v6) =
        valofform(copyfont(atomofpnt(vl),
                           atomofptblt(v2),
                           atomofnat(v3),
                           atomoffont(v4),
                           atomofform(v5),
                           atomofform(v6),
));
applyoop(ptbltfontinv(),vl,v2,v3,v4,v5,v6,v7,v8) =
        valofform(invcopyfont(atomofcolor(vl),
                              atomofcolor(v2),
                              atomofpnt(v3),
                              atomofptblt(v4),
                              atomofnat(v5),
                              atomoffont(v6),
                              atomofform(v7),
                              atomofform(v8)
));
applymop(pidsetlist.makenewlst(),v) =
        valofpidsetlist(pidsetlist.makenewlst(
        atomofpidlist(v)
));                              /* database part */
applydop(pidsetlist.unlst(),vl,v2) =
        valofpidsetlist(pidsetlist.unlst(
        atomofpidsetlist(vl),
        atomofpidsetlist(v2)
));
```

```
applydop(pidsetlist.intlst(),vl,v2) =
        valofpidsetlist(pidsetlist.intlst(
        atomofpidsetlist(vl),
        atomofpidsetlist(v2)
));
applymop(valsetlist.makenewlst(),v) =
        valofvalsetlist(valsetlist.makenewlst(
        atomofvallist(v)) );
applydop(valsetlist.unlst(),vl,v2) =
        valofvalsetlist(valsetlist.unlst(
        atomofvalsetlist(vl),
        atomofvalsetlist(v2)
));
applydop(valsetlist.intlst(),vl,v2) =
        atomofvalsetlist(valsetlist.intlst(
        atomofvalsetlist(vl),
        atomofvalsetlist(v2)
));
applymop(proplist.firstlst(),v) = valofpidlist(
        proplist.firstlst(atomofproplist(v)) );
applymop(proplist.restlst(),v) = valofvalsetlist(
        proplist.restlst(atomofproplist(v)) );
applydop(proplist.catlst(),vl,v2) = valofproplist(
        proplist.catlst(atomofpidlist(vl),
                        atomofvalsetlist(v2)
));
applymop(propsetlist.makenewlst(),v) =
        valofpropsetlist(propsetlist.makenewlst(
        atomofproplist(v)) );
applymop(propsetlist.sofirstlst(),v) =
        valofpidsetlist(propsetlist.sofirstlst(
        atomofpropsetlist(v)) );
applydop(propsetlist.unlst(),vl,v2) =
        valofpropsetlist(propsetlist.unlst(
        atomofpropsetlist(vl),
        atomofpropsetlist(v2)
));
applydop(propsetlist.intlst(),vl,v2) =
        valofpropsetlist(propsetlist.intlst(
        atomofpropsetlist(vl),
        atomofpropsetlist(v2)
));
applymop(pvallist.firstlst(),v) = valofpidlist(
        pvallist.firstlst(atomofpvallist(v)) );
applymop(pvallist.restlst(),v) = valofvallist(
        pvallist.restlst(atomofpvallist(v)) );
applydop(pvallist.catlst(),vl,v2) = valofpvallist(
        pvallist.catlst(atomofpidlist(vl),
                        atomofvallist(v2)
));
```

```
applymop(pvalsetlist.makenewlst(),v) =
        pvalsetlist.makenewlst(atomofpvallist(v)) );
applymop(pvalsetlist.sofirstlst(),v) =
        valofpidsetlist(pvalsetlist.sofirstlst(
        atomofpvalsetlist(v)) );
applydop(pvalsetlist.unlst(),vl,v2) =
        valofpvalsetlist(pvalsetlist,unlst(
        atomofpvalsetlist(vl),
        atomofpvalsetlist(v2)
));
applydop(pvalsetlist.intlst(),vl,v2) =
        valofpvalsetlist(pvalsetlist.intlst(
        atomofpvalsetlist(vl),
        atomofpvalsetlist(v2)
));
applymop(objlist.makenewlst(),v) = valofobjlist(
        objlist.makenewlst(atomofpvalsetlist(v)) );
applymop(objlist.makenewlst(),v) = valofpvalsetlist(
        objlist.makenewlst(atomofobjlist(v)) );
applymop(objlist,sofirstlst(),v) = valofpidsetlist(
        objlist.sofirstlst(atomofobjlist(v)) );
applydop(objlist.getlst(),vl,v2) = valofvallist(
        objlist.getlst(atomofobjlist(vl),
                       atomofpidlist(v2),
));
applymop(classlist.makenewlst(),v) =
        valofclasslist(classlist.makenewlst(
        atomofpidsetlist(v)) );
applymop(classlist.sofirst.lst(),v) =
        valofpidsetlist(classlist.sofirstlst(
        atomofclasslist(v)) );
applydop(classlist.unlst(),vl,v2) = valofclasslist(
        classlist.unlst(atomofclasslist(vl),
                        atomofclasslist(v2)
));
applydop(classlist.intlst(),vl,v2) =
        valofclasslist(classlist.intlst(
        atomofclasslist(vl),
        atomofclasslist(v2)
));
applydop(classlist.catlst(),vl,v2) =
        valofclasslist(classlist.catlst(
        atomofclasslist(vl),
        atomofobjlist(v2)
));
applydop(classlist.delst(),vl,v2) =
        valofclasslist(classlist.delst(
        atomofobjlist(vl),
        atomofclasslist(v2)
));
```

```
applymop(dblist.makenewlst(),v) =
         valofdblist(dblist.makenewlst(
         atomofclasslist(v)) );
applymop(dblist.sofirstlst(),v) =
         valofpidsetlist(dblist.sofirstlst(
         atomofdblist(v)) );
applydop(dblist.catlst(),vl,v2) = valofdblist(
         dblist.catlst(atomofdblist(vl),
                       atomofclasslist(v2)
));
applydop(dblist.delst(),vl,v2) = valofdblist(
         dblist.delst(atomofclasslist(vl),
                      atomofdblist(v2)
));
applydop(dblist.intlst(),vl,v2) =
         valofpvalsetlist(dblist.intlst(
         atomofdblist(vl),
         atomofclasslist(v2)
));
applydop(dblist.retobjlst(),vl,v2) =
         valofobjlist(dblist.retobjlst(
         atomofdblist(vl),
         atomofpvallist(v2)
));
applytop(dblist.modlst(),vl,v2,v3) =
         valofdblist(dblist,modlst(
         atomofdblist(vl),
         atomofobjlist(v2),
         atomofpvallist(v3)
));                                  /* database part */
relop(nat,eq);
relop(nat,gt);
relop(nat,lt);
relop(int,eq);
relop(int,gt);
relop(int,lt);
relop(char,eq);
relop(char,gt);
relop(str.char,eq);
relop(str.char,gt);
relop(intens,eq);
relop(intens,gt);
relop(intens,lt);
relop(colorkeq);
relop(pnt,eq);
relop(pnt,gt);
relop(pnt,lt);
relop(pnt,ge);
relop(pnt,le);
```

```
relop(pidlist,eqlst);          /* database part */
relop(pidsetlist,eqlst);
relop(vallist,eqlst);
relop(valsetlist,eqlst);
relop(proplist,eqlst);
relop(propsetlist,eqlst);
relop(pvallist,eqlst);
relop(pvalsetlist,eqlst);
relop(objlist,eqlst);
relop(classlist,eqlst);
relop(dblist,eqlst);
relop(pidsetlist,memblst);
relop(valsetlist,memblst);
relop(propsetlist,memblst);
relop(pvallist,memblst);
relop(pvalsetlist,memblst);
relop(classlist,memblst);
relop(dblist,memblst);         /* database part */
isops(bool);
isops(nat);
isops(int);
isops(char);
isops(str.char);
isops(intens);
isops(color);
isops(pnt);
isops(rct);
isops(form);
isops(ptblt);
isops(font);
isops(memid);
isops(regid);
isops(stkid);
isops(dregid);
isops(fid);
isops(memaddr);
isops(regaddr);
isops(stkaddr);
isops(dregaddr);
isops(file);
isops(mop);
isops(dop);
isops(top);
isops(qop);
isops(sop);
isops(oop);
isops(rop);
isops(bop);
isops(instr);
```

```
         isops(pidlist.lst);              /* database part */
         isops(pidsetlist.lst);
         isops(vallist.lst);
         isops(valsetlist.lst);
         isops(proplist.lst);
         isops(propsetlist.lst);
         isops(pvallist.lst);
         isops(pvalsetlist.lst);
         isops(objlist.lst);
         isops(classlist.lst);
         isops(dblist.lst);               /* database part */
   end extend;
end operators;


spec instructions
is
   extend
      natural,
      integer,
      memaddress,
      regaddress,
      stkaddress,
      dregaddress,
      operatorclasses,
      instructiontype,
      typing,
      qaddress
   with
      primitive
      op
         org: → instr;
         extern: → instr;
         globl: → instr;
         mbegin: → instr;
         mend: → instr;
         offst: int,regaddr → instr;
         link: regaddr,nat → instr;
         unlink: regaddr,nat → instr;
         getdwin: dregaddr,regaddr → instr;
         setdwin: regaddr,dregaddr → instr;
         getmtr: mattribute,regaddr → instr;
         setmtr: mattribute,regaddr → instr;
         modads: mop,regaddr → instr;
         monad: mop,regaddr,regaddr → instr;
         monadi: mop,val,regaddr → instr;
         dyads: dop,regaddr,regaddr → instr;
         dyadsi: dop,val,regaddr → instr;
         dyad: dop,regaddr,regaddr,regaddr → instr;
         dyadi: dop,val,regaddr,regaddr → instr;
```

```
triads: top,regaddr,regaddr,regaddr → instr;
triadsi: top,val,regaddr,regaddr → instr;
triad: top,regaddr,regaddr,regaddr,regaddr → instr;
triadi: top,val,regaddr,regaddr,regaddr → instr;
quads: qop,regaddr,regaddr,regaddr,regaddr → instr;
quad: qop,regaddr,regaddr,regaddr,regaddr,
            regaddr → instr;
sexads: sop,regaddr,regaddr,regaddr,regaddr,
            regaddr,regaddr → instr;
sexad: sop,regaddr,regaddr,regaddr,regaddr,
            regaddr,regaddr,regaddr → instr;
octads: sop,regaddr,regaddr,regaddr,regaddr,
            regaddr,regaddr,regaddr,regaddr → instr;
octad: sop,regaddr,regaddr,regaddr,regaddr,regaddr,
            regaddr,regaddr,regaddr,regaddr → instr;
movi_m: val,memaddr → instr;
movi_pcr: val,int → instr;
movi_r: val,regaddr → instr;
movi_ri: val,regaddr → instr;
movi_rid: val,regaddr,int → instr;
movi_ridn: val,regaddr,nat,int → instr;
mov_m_m: memaddr,memaddr → instr;
mov_m_r: memaddr,regaddr → instr;
mov_m_ri: memaddr,regaddr → instr;
mov_m_rid: memaddr,regaddr,int → instr;
mov_m_ridn: memaddr,regaddr,nat,int → instr;
mov_m_d: memaddr,dregaddr → instr;
mov_pcr_pcr: int,int → instr;
mov_pcr_r: int,regaddr → instr;
mov_pcr_ri: int,regaddr → instr;
mov_pcr_rid: int,regaddr,int → instr;
mov_pcr_ridn: int,regaddr,nat,int → instr;
mov_pcr_d: int,dregaddr → instr;
mov_r_m: regaddr,memaddr → instr;
mov_r_pcr: regaddr,int → instr;
mov_r_r: regaddr,regaddr → instr;
mov_r_ri: regaddr,regaddr → instr;
mov_r_rid: regaddr,regaddr,int → instr;
mov_r_ridn: regaddr,regaddr,nat,int → instr;
mov_r_d: regaddr,dregaddr → instr;
mov_ri_m: regaddr,memaddr → instr;
mov_ri_pcr: regaddr,int → instr;
mov_ri_r: regaddr,regaddr → instr;
mov_ri_ri: regaddr,regaddr → instr;
mov_ri_rid: regaddr,regaddr,int → instr;
mov_ri_ridn: regaddr,regaddr,nat,int → instr;
mov_ri_d: regaddr,dregaddr → instr;
mov_rid_m: regaddr,int,memaddr → instr;
mov_rid_pcr: regaddr,int,int → instr;
mov_rid_r: regaddr,int,regaddr → instr;
mov_rid_ri: regaddr,int,regaddr → instr;
```

```
mov_rid_rid: regaddr,int,regaddr,int → instr;
mov_rid_ridn: regaddr,int,regaddr,nat,int → instr;
mov_rid_d: regaddr,int,dregaddr → instr;
mov_ridn_m: regaddr,nat,int,memaddr → instr;
mov_ridn_pcr: regaddr,nat,int,int → instr;
mov_ridn_r: regaddr,nat,int,regaddr → instr;
mov_ridn_ri: regaddr,nat,int,regaddr → instr;
mov_ridn_rid: regaddr,nat,int,regaddr,int → instr;
mov_ridn_ridn: regaddr,nat,int,regaddr,int,int →
                    instr;
mov_ridn_d: regaddr,nat,int,dregaddr → instr;
mov_d_m: dregaddr,memaddr → instr;
mov_d_pcr: dregaddr,int → instr;
mov_d_r: dregaddr,regaddr → instr;
mov_d_ri: dregaddr,regaddr → instr;
mov_d_rid: dregaddr,regaddr,int → instr;
mov_d_ridn: dregaddr,regaddr,nat,int → instr;
mov_d_d: dregaddr,dregaddr → instr;
push_i: val,stkaddr → instr;
push_m: memaddr,stkaddr → instr;
push_pcr: int,stkaddr → instr;
push_r: regaddr,stkaddr → instr;
push_ri: regaddr,stkaddr → instr;
push_rid: regaddr,int,stkaddr → instr;
push_ridn: regaddr,nat,int,stkaddr → instr;
push_d: dregaddr,stkaddr → instr;
pop_x: stkaddr → instr;
pop_m: stkaddr,memaddr → instr;
pop_pcr: stkaddr,int → instr;
pop_r: stkaddr,regaddr → instr;
pop_ri: stkaddr,regaddr → instr;
pop_rid: stkaddr,regaddr,int → instr;
pop_ridn: stkaddr,regaddr,nat,int → instr;
pop_d: stkaddr,dregaddr → instr;
nop: → instr;
stop: → instr;
jmp: memaddr → instr;
jmp_i: memaddr → instr;
jmp_r: regaddr → instr;
bra: int → instr;
bra_r: regaddr → instr;
if: relop,regaddr,regaddr,memaddr → instr;
ifi: relop,regaddr,val,memaddr → instr;
ifte: relop,regaddr,regaddr,memaddr,memaddr → instr;
iftei: relop,regaddr,val,memaddr,memaddr → instr;
if_pcr: relop,regaddr,regaddr,int → instr;
ifi_pcr: relop,regaddr,val,int → instr;
ifte_pcr: relop,regaddr,regaddr,int,int → instr;
iftei_pcr: relop,regaddr,val,int,int → instr;
test: bop,regaddr,memaddr → instr;
testm: bop,memaddr,memaddr → instr;
```

```
         teste: bop,regaddr,memaddr,memaddr → instr;
         testme: bop,memaddr,memaddr,memaddr → instr;
         test_pcr: bop,regaddr,int → instr;
         testm_pcr: bop,memaddr,int → instr;
         teste_pcr: bop,regaddr,int,int → instr;
         testme_pcr: bop,memaddr,int,int → instr;
         jsr: memaddr,stkaddr → instr;
         jsr_i: memaddr,stkaddr → instr;
         jsr_r: regaddr,stkaddr → instr;
         bsr: int,stkaddr → instr;
         bsr_r: regaddr,stkaddr → instr;
         rts: stkaddr → instr;
         open: stkaddr → instr;
         close: stkaddr → instr;
         read: stkaddr → instr;
         write: stkaddr → instr;
                                          /* database part */
         write_i: val,qaddr → instr;      /* write to queue */
         write_m: memaddr,qaddr → instr;
         write_r: regaddr,qaddr → instr;
         delete_x: qaddr → instr;         /* delete value
                                             from queue */
         delete_m: qaddr → instr;
         delete_r: qaddr → instr;
         read_m: qaddr,memaddr → instr; /* read value from
                                             queue */
         read_r: qaddr,regaddr → instr;
         open: val → instr;               /* open database */
         close: val → instr;              /* close database */
                                          /* database part */
   end extend;
end instructions;


spec amstate
is
   extend
      boolean,
      natural,
      integer,
      str.chartype,
      memaddress,
      regaddress,
      stkaddress,
      dregaddress,
      files,
      identifiers,
      typing,
      qaddr,
      db
```

```
with
   sort
      state;
   primitive
      fetchm: memaddr,state → val;    /* memory */
      fetchr: regaddr,state → val;    /* register */
      fetchd: dregaddr,state → val;   /* display register */
      fetchdwin: dregaddr,state → val;
                                      /* display window */
      fetchmtr: mattribute,state → val;
                                      /* monitor attribute */
      storem: val,memaddr,state → state;
      storer: val,regaddr,state → state;
      stored: val,dregaddr,state → state;
      storedwin: val,dregaddr,state → state;
      storedmtr: val,mattribute,state → state;
      initam: → state;                /* initialize
                                         machine */
      initstk: stkaddr,state → state;
                                      /* initialize stack */
      topstk: stkaddr,state → val;    /* get top val of
                                         stack */
      pushstk: val,stkaddr,state → state;
                                      /* push stack */
      popstk: stkaddr,state → state;  /* pop stack */
      lalloc: nat,state → memid;      /* get memory block
                                         from heap */
      lfree: memid,state → state;     /* free memroy block */
      indir: nat,memaddr → memaddr;   /* memaddr for n
                                         levels of
                                         indirection */
      infile: file,state → val;       /* read from file */
      outfile: val,file,state → state;
                                      /* write to file */
      openfile: str.char,file,int,int,state → state;
                                      /* open file */
      closefile: file,state → state;  /* close file */
      rmode: → int;                   /* read mode */
      wmode: → int;                   /* write mode */
      rwmode: → int;                  /* read/write mode */
      openerr: → int;                 /* open error */
      openok: →int;                   /* open ok */
      valdata: → int;                 /* file ops w/ AM
                                         sort val data */
      chardata: → int;                /* file ops w/
                                         character data */
                                      /* database part */
      initq: qaddr,state → state;     /* initialize queue */
      read: qaddr,state → val;        /* read front value
                                         from queue */
```

```
   write: val,qaddr,state → state;
                                        /* write to queue */
   delete: qaddr,state → state;         /* delete front value
                                           from queue */
   open: db,str.char,state → state;
                                        /* open database */
   close: db,str.char,state → state;
                                        /* close database */
                                        /* data base part */
hidden
op
      /****************************************
      * active - lalloc flag
      * true when memory block is allocated w/ lalloc
      * false initially and after memory block
        released with lfree
      * used to prevent offsetting into non-
        allocated memory
      */
   active: memid,state   bool;
axiom
   if whattype(v) != formtype() then
      stored(v,a,q) = undef;
   endif;
   if whattype(v) != pnttype() then
      storedwin(v,a,q) = undef;
   endif;
   if whattype(v) != nattype()
   then
      storemtr(v,xpixels(),q) = undef;
      storemtr(v,ypixels(),q) = undef;
      storemtr(v,hscrnsize(),q) = undef;
      storemtr(v,vscrnsize(),q) = undef;
      storemtr(v,intenscapbl(),q) = undef;
      storemtr(v,colorcapbl(),q) = undef;
   endif;
   if whattype(v) != colortype() then
      storemtr(v,backgnd(),q) = undef;
   endif;
   if whattype(v) != dregaddr() then
      storemtr(v,dselect(),q) = undef;
   endif;
   topstk(s,initstk(s)) = undef;
   popstk(s,initstk(s)) = undef;
   popstk(s,initam()) = undef;
   stateaxioms(m,memaddr);
   stateaxioms(r,regaddr);
   stateaxioms(d,dregaddr);
   stateaxioms(dwin,dregaddr);
   stateaxioms(mtr,mattribute);
```

```
topstk(s,pushstk(v,s,q)) = v;
popstk(s,pushstk(v,s,q)) = q;
active(m,initam()) = false;
active(lalloc(n,q),q) = true;
active(m,lfree(m,q)) = false;
active(m,storer(v,a,q)) = active(m,q);
active(m,storem(v,a,q)) = active(m,q);
active(m,stored(v,a,q)) = active(m,q);
active(m,storedwin(v,a,q)) = active(m,q);
active(m,storexscrnsize(v,a,q)) = active(m,q);
active(m,storeyscrnsize(v,a,q)) = active(m,q);
active(m,storeintenscapbl(v,a,q)) = active(m,q);
active(m,storecolorcapbl(v,a,q)) = active(m,q);
active(m,storebackgnd(v,a,q)) = active(m,q);
active(m,storedregaddr(v,a,q)) = active(m,q);
active(m,initstk(a,q)) = active(m,q);
active(m,pushstk(v,a,q)) = active(m,q);
active(m,popstk(a,q)) = active(m,q);
active(m,outfile(v,f,q)) = active(m,q);
active(m,openfile(s,f,x,y,q)) = active(m,q);
active(m,closefile(f,q)) = active(m,q);
if active(m,q) = false() then
   fetchm(offset(n,m),q) = undef;
endif;
if active(m,q) = false() then
   storem(offset(n,m),q) = undef;
endif;
if ltint(n,ntoi(n2)) = true()
then
   offset(n,offset(nl,startmemaddr(
      lalloc(n2,q)) )) =
         offset(
            sumint(n,nl),
            startmemaddr(lalloc(n2,q))
         );
else
   offset(n,offset(nl,startmemaddr(lalloc(n2,q)) )) =
      undef;
indir(zeronat(),m) = m;
if whattype(fetchm(indir(n,m),q)) = typememaddr()
then
   indir(succnat(n),m) = atomofmemaddr(
      fetchm(indir(n,m),q));
else
   indir(succnat(n),m) = undef;
endif;
openfile(s,f,n,openfile(s,f,m,x,q)) = undef;
closefile(f,openfile(s,f,n,x,q)) = q;
infile(f,initam()) = undef;
infile(f,close(d,q)) = undef;
infile(f,openfile(s,f,wmode(),x,q)) = undef;
```

```
            outfile(v,f,initam()) = undef;
            outfile(v,f,close(f,q)) = undef;
            outfile(v,f,openfile(s,f,rmode(),x,q)) = undef;
            outfile(f,openfile(s,f,m,chardata(),q)) = undef;
                                          /* database part */
            read(qu,initq(qu,q)) = undef;  /* qu: queue */
            delete(qu,initam()) = undef;
                                          /* q: state */
            delete(qu,write(v,qu,initq(qu,q))) = v;
                                          /* v: value */
            read(qu,write(v,qu,initq(qu,q))) = v;
            delete(write(v,qu,initq())) = initq();
            delete(write(v,qu,q)) = write(v,qu,delete(qu,q));
            if not (initq()) = true()
            then
               read(qu,write(v,qu,q)) = read(qu,q);
            endif;
            active(m,initq(a,q)) = active(m,q);
            active(m,write(v,a,q)) = active(m,q);
            active(m,delete(a,q)) = active(m,q);
            active(m,open(s,d,q)) = active(m,q);
                                          /* s: string char */
            active(m,close(s,d,q)) = active(m,q);
                                          /* d: database */
                                          /* database part */
      end extend;
   end amstate;


   spec displaywindow
   is
      extend
         rectangle
         dregaddress
      with
         primitive
         op
            dwin: dregaddr → rct;
         axiom
            xdimrct(dwin(a)) = [DISPLAYSIZE]succint(zeroint());
            ydimrct(dwin(a)) = [DISPLAYSIZE]succint(zeroint());
            origin(dwin(a)) = atomofpnt(fetchdwin(a,q));
      end extend;
   end displaywindow;
```

```
spec am
is
   extend
      memaddress,
      instructiontype,
      typing,
      amstate
   with
      primitive
      op
            /****************************************
            * prog - AM execution
            * corecursive - calls xeq
            */
         prog: memaddr,state → state;
      hidden
      op
            /****************************************
            * cond - implements conditionals
            * returns one of two input memaddrs
            *     based on bool value
            */
         cond: val,memaddr,memaddr → memaddr;
            /****************************************
            * xeq - corecursive function
            * calls prog
            * used for AM execution
            */
         xeq: instr,memaddr,state → state;
      axiom
         prog(a,q) = xeq(atomofinstr(fetchm(a,q),a,q));
         cond(valofbool(true()),a1,a2) = a1;
         cond(valofbool(false()),a1,a2) = a2;
         xeq(offst(i,r),m,q) =
            prog(
               nextmemaddr(m),
               storer(
                  valofmemaddr(offset(i,atomofmemaddr(
                        fetchr(r,q)) )),
                  r,
                  q
               )
            );
         xeq(link(r,n),m,q) =
            prog(
               nextmemaddr(m),
               storer(
                  valofmemaddr(startmemaddr(lalloc(n,q)) ),
                  r,
                  storem(
                     fetchr(r,q),
```

```
            startmemaddr(lalloc(n,q),q)
         )
      )
   );
xeq(unlink(r),m,q) =
   prog(
      nextmemaddr(m),
      lfree(
         getmemid(atomofmemaddr(fetchr(r,q)) ),
         storer(
            fetchm(atomofmemaddr(fetchr(r,q)),q),
            r,
            q
         )
      )
   );
xeq(getdwin(d,r),m,q) =
   prog(
      nextmemaddr(m),
      storer(
         fetchdwin(d,q),
         r,
         q
      )
   );
xeq(setdwin(r,d),m,q) =
   prog(
      nextmemaddr(m),
      storedwin(
         fetchr(r,q),
         d,
         q
      )
   );
xeq(getmtr(t,r),m,q) =
   prog(
      nextmemaddr(m),
      storer(
         fetchmtr(t,q),
         r,
         q
      )
   );
xeq(setmtr(r,t),m,q) =
   prog(
      nextmemaddr(m),
      storemtr(
         fetchr(r,q),
         t,
         q
      )
   );
```

```
xeq(monads(o,rl),m,q) =
   prog(
      nextmemaddr(m),
      storer(
         applymop(
            o,
            fetchr(rl,q)
         ),
         rl,
         q
      )
   );
xeq(monad(o,rl,r2),m,q) =
   prog(
      nextmemaddr(m),
      storer(
         applymop(
            o,
            fetchr(rl,q)
         ),
         r2,
         q
      )
   );
xeq(monadi(o,v,rl),m,q) =
   prog(
      nextmemaddr(m),
      storer(
         applymop(o,v),
         rl,
         q
      )
   );
xeq(dyads(o,rl,r2),m,q) =
   prog(
      (nextmemaddr(m),
      storer(
         applydop(
            o,
            fetchr(rl,q),
            fetchr(r2,q)
         ),
         r2,
         q
      )
   );
xeq(dyadsi(o,v,rl),m,q) =
   prog(
      nextmemaddr(m),
      storer(
```

```
            applydop(
               o,
               v,
               fetchr(r1,q)
            ),
            r1,
            q
         )
      );
   xeq(dyad(o,r1,r2,r3),m,q) =
      prog(
         nextmemaddr(m),
         storer(
            applydop(
               o,
               fetchr(r1,q),
               fetchr(r2,q)
            ),
            r3,
            q
         )
      );
   xeq(dyadi(o,v,r1,r2),m,q) =
      prog(
         nextmemaddr(m),
         storer(
            applydop(
               o,
               v,
               fetchr(r1,q)
            ),
            r2,
            q
         )
      );
   xeq(triads(o,r1,r2,re),m,q) =
      prog(
         nextmemaddr(m),
         storer(
            applytop(
               o,
               fetchr(r1,q),
               fetchr(r2,q),
               fetchr(r3,q)
            ),
            r3,
            q
         )
      );
```

```
xeq(triadsi(0,v,rl,r2),m,q)=
   prog(
      nextmemaddr(m),
      storer(
          applytop(
             o,
             v,
             fetchr(rl,q),
             fetchr(r2,q)
          ),
          r2,
          q
      )
  );
xeq(triad(0,rl,r2,r3,r4),m,q)=
   prog(
      nextmemaddr(m),
      storer(
          applytop(
             o,
             fetchr(rl,q),
             fetchr(r2,q),
             fetchr(r3,q)
          ),
          r4,
          q
      )
   );
xeq(triadi(o,v,rl,r2,r3),m,q)=
   prog(
      nextmemaddr(m),
      storer(
          applytop(
             o,
             v,
             fetchr(rl,q),
             fetchr(r2,q)
          ),
          r3,
          q
      )
   );
xeq(quads(o,rl,r2,r3,r4),m,q)=
   prog(
      nextmemaddr(m),
      storer(
          applyqop(
             o,
             fetchr(rl,q),
             fetchr(r2,q),
             fetchr(r3,q),
```

```
                fetchr(r4,q)
            ),
            r4,
            q
        )
    );
xeq(quad(o,r1,r2,r3,r4,r5),m,q)=
    prog(
        nextmemaddr(m),
        storer(
            applyqop(
                o,
                fetchr(r1,q),
                fetchr(r2,q),
                fetchr(r3,q),
                fetchr(r4,q)
            ),
            r5,
            q
        )
    );
xeq(sexads)o,r1,r2,r3,r4,r5,r6),m,q)=
    prog(
        nextmemaddr(m),
        storer(
            applysop(
                o,
                fetchr(r1,q),
                fetchr(r2,q),
                fetchr(r3,q),
                fetchr(r4,q),
                fetchr(r5,q),
                fetchr(r6,q)
            ),
            r6,
            q
        )
    );
xeq(sexad(o,r1,r2,r3,r4,r5,r6,r7),m,q)=
    prog(
        nextmemaddr(m),
        storer(
            applysop(
                o,
                fetchr(r1,q),
                fetchr(r2,q),
                fetchr(r3,q),
                fetchr(r4,q),
                fetchr(r5,q),
                fetchr(r6,q)
```

```
            ),
            r7,
            q
        )
    );
xeq(octads(o,r1,r2,r3,r4,r5,r6,r7,r8),m,q)=
    prog(
        nextmemaddr(m),
        storer(
            applyoop(
                o,
                fetchr(r1,q),
                fetchr(r2,q),
                fetchr(r3,q),
                fetchr(r4,q),
                fetchr(r5,q),
                fetchr(r6,q),
                fetchr(r7,q),
                fetchr(r8,q)
            ),
            r8,
            q
        )
    );
xeq(octad(o,r1,r2,r3,r4,r5,r6,r7,r8,r9),m,q)=
    prog(
        nextmemaddr(m),
        storer(
            applyoop(
                o,
                fetchr(r1,q),
                fetchr(r2,q),
                fetchr(r3,q),
                fetchr(r4,q),
                fetchr(r5,q),
                fetchr(r6,q),
                fetchr(r7,q),
                fetchr(r8,q)
            ),
            r9,
            q
        )
    );
xeq(movi_m(v,ml),m,q)=
    prog(
        nextmemaddr(m),
        storem(v,ml,q)
    (;
xeq(movi_pcr(v,i),m,q)=
    prog(
        nextmemaddr(m),
        storem(
```

```
            v,
            offset(i,m),
            q
        )
    );
xeq(movi_r(v,r),m,q)=
    prog(
        nextmemaddr(m),
        storem(
            v,
            offset(i,m),
            q
        )
    );
xeq(movi_r(v,r),m,q)=
    prog(nextmemaddr(m),storer(v,r,q));
xeq(movi_ri(v,r),m,q)=
    prog(
        nextmemaddr(m),
        storem(
            v,
            atomofmemaddr(fetchr(r,q)),
            q
        )
    );
xeq(movi_rid(v,r,n),m,q)=
    prog(
        nextmemaddr(m),
        storem(
            v,
            offset(
                n,
                atomofmemaddr(fetchr(r,q))
            ),
            q
        )
    );
xeq(movi_ridn(v,r,il,i2),m,q)=
    prog(
        nextmemaddr(m),
        storem(
            v,
            offset(
                i2,
                indir(
                    il,
                    atomofmemaddr(fetchr(r,q))
                )
            ),
            q
        )
    );
```

```
xeq(movi_d(v,r),m,q)=
   prog(nextmemaddr(m),stored(v,r,q));
xeq(mov_m_m(m1,m2),m,q)=
   prog(
      nextmemaddr(m),
      storem(
          fetchm(m1,q),
          m2,
          q
      )
   );
xeq(mov_m_r(m1,r),m,q)=
   prog(nextmemaddr(m),storer(fetchm(m1,q),r,q));
xeq(mov_m_ri(m1,r),m,q)=
   prog(
      nextmemaddr(m),
      storem(
          fetchm(m1,q),
          atomofmemaddr(fetchr(r,q))
          ,q
      )
   );
xeq(mov_m_rid(m,l,r,n),m,q)=
   prog(
      nextmemaddr(m),
      storem(
          fetchm(m1,q),
          offset(
              n,
              atomofmemaddr(fetchr(r,q))
          ),
          q
      )
   );
xeq(mov_m_ridn(m1,r,i1,i2),m,q)=
   prog(
      nextmemaddr(m),
      storem(
          fetchm(m1,q),
          offset(
              ie,
              indir(
                  il,
                  atomofmemaddr(fetchr(r,q))
              )
          ),
          q
      )
   );
xeq(mov_pcr_pcr(i1,i2),m,q)=
prog(
```

```
        nextmemaddr(m),
        storem(
            fetchm(offset(il,m),q),
            offset(i2,m),
            q
        )
    );
xeq(mov_pcr_r(i,r),m,q)=
    prog(
        nextmemaddr(m),
        storer(
            fetchm(offset(i,m),q),
            r,
            q
        )
    );
xeq(mov_pcr_ri(i,r),m,q)=
    prog)
        nextmemaddr(m),
        storem(
            fetchm(offset(i,m),q),
            atomofmemaddr(fetchr(r,q)),
            q
        )
    );
xeq(mov_pcr_rid(il,r,i2),m,q)=
    prog(
        nextmemaddr(m),
        storem(
            fetchm(offset(il,m),q),
            offset(
                i2,
                atomofmemaddr(fetchr(r,q))
            ),
            q
        )
  );
xeq(mov_pcr_ridn(il,r,n,i2),m,q)=
    prog(
        nextmemaddr(m),
        storem(
            fetchm(offset(il,m),q),
            offset(
                i2,
                indir(
                    n,
                    atomofmemaddr(fetchr(r,q))
                )
            ),
            q
        )
  );
```

```
xeq(mov_m_d(ml,r),m,q)=
   prog(nextmemaddr(m),stored(fetchm(ml,q),r,q));
xeq(mov_r_m(r,ml),m,q)=
   prog(nextmemaddr(m),storem(fetchr(r,q),ml,q));
xeq(mov_r_pcr(r,i),m,q)=
   prog(
      nextmemaddr(m),
      storem(
          fetchr(r,q),
          offset(i,m),
          q
      )
   );
xeq(mov_r_r(rl,r]),m,q)=
prog(nextmemaddr(m),storer(fetchr(rl,q),r2,q));
xeq(mov_r_ri(rl,r2),m,q)=
   prog(
      nextmemaddr(m),
      storem(
          fetchr(rl,q),
          atomofmemaddr(fetchr(r2,q)),
          q
      )
   );
xeq(mov_r_rid(rl,r2,n),m,q)=
   prog(
      nextmemaddr(m),
      storem(
          fetchr(rl,q),
          offset(
              n,
              atomofmemaddr(fetchr(r2,q))
          ),
          q
      )
   );
xeq(mov_r_ridn(rl,r2,il,i2),m,q)=
   prog(
      nextmemaddr(m),
      storem(
          fetchr(rl,q),
          offset(
              i2,
              indri(
                  il,
                  atomofmemaddr(fetchr(r2,q))
              )
          ),
          q
      )
   );
```

```
xeq(mov_r_d(r1,r2),m,q)=
   prog(nextmemaddr(m),stored(fetchr(r1,q),r2,q));
xeq(mov_ri_m)r,m1),m,q)=
   prog(
      nextmemaddr(m),
      storem(
          fetchm(atomofmemaddr(fetchr(r,q)),q),
          m1,
          q
      )
   );
xeq(mov_ri_pcr(r,i),m,q)=
   prog(
      nextmemaddr(m),
      storem(
          fetchm(atomofmemaddr(fetchr(r,q)),q),
          offset(i,m),
          q
      )
   );
xeq(mov_ri,r(r1,rq),m,q)=
   prog(
      nextmemaddr(m),
      storer(
          fetchm(atomofmemaddr(fetchr(r1,q)) ),
          r2,
          q
      )
   );
xeq(mov_ri_ri(r1,r2),m,q)=
   prog(
      nextmemaddr(m),
      storem(
          fetchm(atomofmemaddr(fetchr(r1,q)),q),
          atomofmemaddr(
             fetchr(r2,q)
          ),
          q
      )
   );
xeq(mov_ri_rid)r1,r2,n),m,q)=
   prog(
      nextmemaddr(m),
      storem(
          fetchm(atomofmemaddr(fetchr(r1,q)),q),
          offset(
               n,
               atomofmemaddr(fetchr(r2,q))
          ),
          q
      )
   );
```

```
xeq(mov_ri_ridn(r1,r2,i1,i2)m,q)=
   prog(
      nextmemaddr(m),
      storem(
          fetchm(atomofmemaddr(fetchr(r1,q)),q),
          offset(
              i2,
              indir(
                  i1,
                  atomofmemaddr(fetchr(r2,q))
              )
          ),
          q
      )
   );
xeq(mov_ri_d(r1,r2),m,q)=
   prog(
      nextmemaddr(m),
      stored(
          fetchm(atomofmemaddr(fetchr(r1,q)) ),
          r2,
          q
      )
   );
xeq(mov_rid_m(r,i,m1),m,q)=
   prog(
      nextmemaddr(m),
      storem(
          fetchm(
           .  offset(
                  i,
                  atomofmemaddr(fetchr(r,q))
              ),
              q
          ),
          m1,
          q
      )
   );
xeq(mov_rid_pcr)r,i1,i2),m,q)=
   prog(
      nextmemaddr(m),
      storem(
          fetchm(
              offset(
                  i1,
                  atomofmemaddr(fetchr(r,q))
              ),
              q
          ),
          offset(i2,m),
          q
```

```
        )
    );
xeq(mov_rid_r(rl,n,r2),m,q)=
    prog(
        nextmemaddr(m),
        storer(
            fetchm(
                offset(
                    n,
                    atomofmemaddr(fetchr(rl,q))
                ),
                q
            ),
            r2,
            q
        )
    );
xeq(mov_rid_ri(rl,i,r2),m,q)=
    prog(
        nextmemaddr(m),
        storem(
            fetchm(
                offset(
                    i,
                    atomofmemaddr(fetchr(rl,]))
                ),
                q
            ),
            atomofmemaddr(fetchr(r2,q)),
            q
        )
    );
xeq(mov_rid_rid(rl,il,r2,i2),m,q)=
    prog(
        nextmemaddr(m),
        storem(
            fetchm(
                offset(
                    il,
                    atomofmemaddr(fetchr(rl,q))
                ),
                q
            ),
            offset(
                i2,
                atomofmemaddr(fetchr(r2,q))
            ),
            q
        )
    );
xeq(mov_rid_ridn(rl,il,r2,i,2,i3),m,q)=
    prog(
```

```
        nextmemaddr(m),
        storem(
            fetchm(
                offset(
                    il,
                    atomofmemaddr(fetchr(rl,q))
                ),
                q
            ),
            offset(
                i3,
                indir(
                   i2,
                   atomofmemaddr(fetchr(r2,q))
                )
            ),
            q
        )
    );
xeq(mov_rid_d(rl,n,r2),m,q)=
    prog(
       nextmemaddr(m),
      stored(
          fetchm(
              offset(
                  n,
                  atomofmemaddr(fetchr(rl,q))
              ),
              q
          ),
          r2,
          q
       )
    );
xeq(mov_ridn_m(r,n,i,ml),m,q)=
    prog(
       nextmemaddr(m),
       storem(
           fetchm(
               offset(
                   i,
                   indir(
                       n,atomofmemaddr(fetchr(r,q))
                   )
               ),
               q
           ),
           q
       )
    );
xeq(mov_ridn_pcr(r,n,il,i2),m,q)=
    prog(
```

```
        nextmemaddr(m),
        storem(
            fetchm(
                offset(
                    il,
                    indir(
                        n,
                        atomofmemaddr(fetchr(r,q))
                    )
                ),
                q
            ),
            offset(i2m,),
            q
        )
    );
xeq(mov_ridn_r(rl,il,i2,r2),m,q)=
    prog(
        nextmemaddr(m),
        storer(
            fetchm(
                offset(
                    i2,
                    indir(
                        il,
                        atomofmemaddr(fetchr(rl,q))
                    )
                ),
                q
            ),
            r2,
        )   q
    );
xeq(mov_ridn_r(rl,il,i2,r2),m,q)=
    prog(
        nextmemaddr(m),
        storem(
            fetchm(
                offset(
                    i2,
                    indir(
                        il,
                        atomofmemaddr(fetchr(rl,q))
                    )
                ),
                q
            ),
            atomofmemaddr(
               fetchr(r2,q)
            ),
            q
      •)
    );
```

```
xeq(mov_ridn_rid(r1,i1,i2,r2,i3),m,q)
   prog(
      nextmemaddr(m),
      storem(
          fetchm(
              offset(
                  i2,
                  indir(
                     i1,
                     atomofmemaddr(fetchr(r1,q))
                  )
              ),
              q
          ),
          offset(
              i3,
              atomofmemaddr(fetchr(r2,q))
          ),
          q
      )
   );
xeq(mov_ridn_ridn(r1,i1,i2,r,i3,i4),m,q)=
   prog(
      nextmemaddr(m),
      storem(
          fetchm(
              offset(
                  i2,
                  indir(
                      i1,
                      atomofmemaddr(fetchr(r1,q))
                  )
              ),
              q
          ),
          offset(
              i4,
              indir(
                  i3,
                  atomofmemaddr(fetchr(r2,q))
              )
          ),
          q
      )
   );
xeq(mov_ridn_d(r1,i1,ir,r2),m,q)=
   prog(
      nextmemaddr(m),
      stored(
          fetchm(
```

```
                offset(
                    i2,
                    indir(
                        il,
                        atomofmemaddr(fetchr(rl,q))
                    )
                ),
                q
            ),
            r2,
            q
        )
    );
xeq(mov_d_m(r,ml),m,q)=
    prog(nextmemaddr(m),storem(fetchd(r,q),ml,q));
xeq(mov_d_pcr(r,i),m,q)=
    prog(
        nextmemaddr(m),
        storem(
            fetchd(r,q),
            offset(i,m),
            q
        )
    );
xeq(mov_d_r(rl,r2),m,q)=
    prog(nextmemaddr(m),storer(fetchd(rl,q),r2,q));
xeq(mov_d_ri(rl,r]),m,q)=
    prog(
        nextmemaddr(m),
        storem(
            fetchd(rl,q),
            atomofmemaddr(fetchr(r2,q)),
            q
        )
    );
xeq(mov_d_rid(rl,r2,n),m,q)=
    prog(
        nextmemaddr(m),
        stroem(
            fetchd(rl,q),
            offset(
                n,
                atomofmemaddr(fetchr(r2,q))
            ),
            q
        )
    );
xeq(mov_d_ridn(rl,r2,il,i2),m,q)=
    prog(
        nextmemaddr(m),
        storem(
            fetchd(rl,q),
```

```
            offset(
                i2,
                indir(
                    il,
                    atomofmemaddr(fetchr(r2,q))
                )
            ),
            q
        )
    );
xeq(mov_d_d(r1,r2),m,q)=
    prog(nextmemaddr(m),stored(fetchd(r1,q),r2,q));
xeq(push_i(v,s),m,q)=
    prog(nextmemaddr(m),pushstk(v,s,q));
xeq(push_pcr(i,s),m,q)=
    prog(
        nextmemaddr(m),
        pushstk(
            fetchm(offset(i,m),q),
            s,
            q
        )
    );
xeq(push_r(r,s),m,q)=
    prog(nextmemaddr(m),pushstk(fetchr(r,q)s,q));
xeq(push_ri(r,s),m,q)=
    prog(
        nextmemaddr(m),
        pushstk(
            fetchm(atomofmemaddr(fetchr(r,q)),q),
            s,
            q
        )
    );
xeq(push_rid(r,n,s),m,q)=
    prog(
        nextmemaddr(m),
        pushstk(
            fetchm(
                offset(
                    n,
                    atomofmemaddr(fetchr(r,q))
                ),
                q
            ),
            s,
            q
        )
    );
xeq(push_ridn(r,il,i2,s),m,q)=
    prog(
        nextmemaddr(m),
```

```
        pushstk(
            fetchm(
                offset(
                    i2,
                    indir(
                        il,
                        atomofmemaddr(fetchr(r,q))
                    )
                ),
                q
            ),
            s,
            q
        )
    );
xeq(push_d(r,s),m,q)=
    prog(nextmemaddr(m),pushstk(fetchd(r,q),s,q));
xeq(pop_x(s),m,q)=
    prog(nextmemaddr(m),popstk(s,q));
xeq(pop_m(s,ml),m,q)=
    prog(
        nextmemaddr(m),
        popstk(
            s,
            storem(
                topstk(s,q),
                ml,
                q
            )
        )
    );
xeq(pop_pcr(s,i),m,q)=
    prog(
        nextmemaddr(m),
        popstk(
            s,
            storem(
                topstk(p,q),
                offset(i,m),
                q
            )
        )
    );
xeq(pop_r(s,r),m,q)=
    prog(
        nextmemaddr(m),
        popstk(
            s,
            storer(
                topstk(s,q),
                r,
                q
```

```
            )
        )
    );
xeq(pop_ri(s,r),m,q)=
    prog(
        nextmemaddr(m),
        popstk(
            s,
            storem(
                topstk(s,q),
                atomofmemaddr(fetchr(r,q)),
                q
            )
        )
    );
xeq(pop_rid(s,r,n),m,q)=
    prog(
        nextmemaddr(m),
        popstk(
            s,
            storem(
                topstk(s,q),
                offset(
                    n,
                    atomofmemaddr(fetchr(r,q))
                ),
                q
            )
        )
    );
xeq(pop_ridn(s,r,il,i2),m,q)=
    prog(
        nextmemaddr(m),
        popstk(
            s,
            storem(
                topstk(s,q),
                offset(
                    i2,
                    indir(
                        il,
                        atomofmemaddr(fetchr(r,q))
                    )
                ),
                q
            )
        )
    );
xeq(pop_d(s,r),m,q)=
prog(
    nextmemaddr(m),
    popstk(
```

```
            s,
            stored(
               topstk(s,q),
               r,
               q
            )
         )
      );
   xeq(nop,m,q) = prog(nextmemaddr(m),q);
   xeq(stop,m,q) = prog(m,q) = q;
   xeq(jmp(ml),m,q) = prog(ml,q);
   xeq(jmp_i(ml),m,q) = prog(atomofmemaddr(fetchm(ml,q)),q);
   xeq(jmp_r(r),m,q) = prog(atomofmemaddr(fetchr(r,q)),q);
   xeq(bra(n),m,q) = prog(offset(n,nextmemaddr(m)),q);
   xeq(bra_r,r,m,q) = prog(offset(atomofint(fetchr(r,q)),
                           nextmemaddr(m)),q);
   xeq(if(o,rl,r2,ml),m,q)=
      prog(
         cond(
            applyrop(
               o,
               fetchr(rl,q),
               fetchr(r2,q),
            ),
            ml,
            nextmemaddr(m)
         ),
         q
      );
   xeq(ifi(o,r,v,ml),m,q)=
      prog(
         cond(
            applyrop(
               o,
               fetchr(r,q),
               v
            ),
            ml,
            nextmemaddr(m)
         ),
         q
      );
   xeq((ifte(o,rl,r2,ml,m2),m,q)=
      prog(
         cond(
            applyrop(
               o,
               fetchr(rl,q),
               fetchr(r2,q)
            ),
            ml,
            m2
```

```
      ),
      q
   );
xeq(iftei(o,r,v,m1,m2),m,q)=
   prog(
      cond(
         applyrop(
            o,
            fetchr(r,q),
            v
         ),
         m1,
         m2
      ),
      q
   );
xeq(if_pcr)o,r1,r2,n),m,q)=
   prog(
      cond(
         applyrop(
            o,
            fetchr(r1,q)
            fetchr(r2,q)
         ),
         offset(n,nextmemaddr(m)),
         nextmemaddr(m)
      ),
      q
   );
xeq(ifi_pcr(o,r,v,n),m,q)=
   prog(
      cond(
         applyrop(
            o,
            fetchr(r,q),
            v
         ),
         offset(n,nextmemaddr(m)),
         nrxtmemaddr(m)
      ),
      q
   );
xeq(ifte_pcr(o,r1,r2,i1,i2),m,q)=
   prog(
      cond(
         applyrop(
            o,
            fetchr(r1,q),
            fetchr(r2,q)
         ),
         offset(i1,nextmemaddr(m)),
```

```
            offset(i2,nextmemaddr(m))
         ),
         q
      );
   xeq(iftei_pcr(o,r,v,il,i2),m,q)=
      prog(
         cond(
            applyrop(
               o,
               fetchr(r,q),
               v
            ),
            offset(il,nextmemaddr(m)),
            offset(i2,nextmemaddr(m))
         ),
         q
      );
   xeq(test(o,rl,ml),m,q)=
      prog(
         cond(
            applybop(o,fetchr(rl,q)),
            ml,
            nextmemaddr(m)
         ),
         q
      );
   xeq(testm(o,m2,ml),m,q)=
      prog(
         cond(
            applybop(o,fetchm(m2,q)),
            ml,
            nextmemaddr(m)
         ),
         q
      );
   xeq(teste(o,rl,ml,m2),m,q)=
      prog(cond(applybop(o,fetch(rl,q)),ml,m2),q);
   xeq(testme(o,m3,ml,m2),m,q)=
      prog(cond(applybop(o,fetchm(m3,q)),ml,m2),q);
   xeq(test_pcr(o,rl,n),m,q)=
      prog(
         cond(
            applybop(o,fetchr(rl,q)),
            offset(n,nextmemaddr(m)),
            nextmemaddr(m);
         ),
         q
      );
   xeq(tesm_pcr(o,m2,n),m,q)=
      prog(
         cond(
```

```
            applybop(o,fetchm(m2,q)),
            offset(n,nextmemaddr(m)),
            nextmemaddr(m)
         ),
         q
      );
   xeq(teste_pcr(o,rl,il,i2)m,q)=
      prog(
         cond(
            applybop(o,fetchr(rl,q)),
            offset(il,nextmemaddr(m)),
            offset(i2,nextmemaddr(m))
         ),
         q
      );
   xeq(testme_pcr(o,m2,il,i2),m,q)=
      prog(
         cond(
            applybop(o,fetchm(m3,q)),
            offset(il,nextmemaddr(m)),
            offset(i2,nextmemaddr(m))
         ),
         q
      );
   xeq(jsr(ml,s),m,q)=
      prog(ml,pushstk(valofmemaddr(nextmemaddr(m)),s,q));
   xeq(jsr_i(ml,s),m,q)=
      prog(
         atomofmemaddr(fetchm(ml,q)),
         pushstk(valofmemaddr(nextmemaddr(m)),s,q)
      );
   xeq(jsr_r(r,s),m,q)=
      prog(
         atomofmemaddr(fetchr(r,q)),
         pushstk(valofmemaddr(nextmemaddr(m)),s,q)
      );
   xeq(bsr(n,s),m,q)=
      prog(
         offset(n,nextmemaddr(m)),
         pushstk(valofmemaddr(nextmemaddr(m)),s,q)
      );
   xeq(bsr_r(r,s),m,q)=
      prog(
         offset(
             atomofint(fetchr(r,q)),
             nextmemaddr(m)
         ),
         pushstk(valofmemaddr(nextmemaddr(m)),s,q)
      );
   xeq(rts s,m,q)=
      prog(atmoofmemaddr(topstk(s,q)),popstk(s,q));
   xeq(open(s),m,q)=
      prog(
```

```
        nextmemaddr(m),
        openfile(
            atomofstr,char(topstk(s,popstk(s,popstk
               (s,popstk(s,q)) )) ),
            atomoffile(topstk(s,popstk(s,popstk(s,q)) )),
            atomofint(topstk(s,popstk(s,q)) ),
            atomofint(topstk(s,q)),
            popstk(s,q)
        )
    );
xeq(close(s),m,q)=
    prog(
        nextmemaddr(m),
        closefile(
             atomoffile(topstk(s,q)),
             popstk(s,q)
        )
    );
xeq(read(s),m,q)=
    prog(
        nextmemaddr(m),
        storem(
            infile(
                 atomoffile(topstk(s,popstk(s,q)) ),
                 popstk(s,q)
            ),
            atomofmemaddr(topstk(s,q)),
            popstk(s,q)
        )
    );
xeq(write(s),m,q)=
    prog(
        nextmemaddr(m),
        outfile(
           fetchm(
               atomofmemaddr(topstk(s,popstk(s,q)) ),
               popstk(s,q)
           ),
           atomoffile(topstk(s,q)),
           popstk(s,q)
        )
    );
xeq(write_i(v,qu),m,q)=        /* database part */
    prog(nextmemaddr(m),write(v,qu,q));
xeq(write_m(ml,qu),m,q)=
    prog(nextmemaddr(m),write(fetchm(ml,q),qu,q));
xeq(write_r(r,qu),m,q)=
    prog(nextmemaddr(m),write(fetchr(r,q),qu,q));
xeq(delete_x(qu),m,q)=
    prog(nextmemaddr(m),delete(qu,q));
```

```
        xeq(delete_m(qu,ml),m,q)=
           prog(
               nextmemaddr(m),
               delete(
                     qu,
                     storem(
                           read(qu,q),
                           ml,
                           q
                     )
           );  )
        xeq(delete_r(qu,r),m,q)=
           prog(
               nextmemaddr(m),
               delete(
                     qu,
                     storer(
                           read(qu,q),
                           r,
                           q
                     )
               )
           );
        xeq(open(s),m,q)=
           prog(
               nextmemaddr(m),
               open(
                   atomofstr.char(vl),
                   atomofdb(v2)
               )
           );
        xeq(close(s),m,q)=
           prog(
               nextmemaddr(m),
               close(
                    atomofstr.char(vl),
                    atomofdb(v2)
               )
           );                              /* database part */
   end extend;
end am;
```

APPENDIX C

A SIMPLE ASSEMBLER FOR AM

1. Introduction

This document is adapted from Yurchak [Ref. 2], Appendix C, and constitutes the reference manual for both version 2.0-Z100, developed by Hunter [Ref. 3], and the latest modification towards version 3.0. It provides a comprehensive description of the syntax and semantics of the assembler as well as a description of the salient features of the AM machine and a definition of the opcodes executed by AM.

AMASM is an assembler which generates a relocatable load module for the abstract machine interpreter AM. It is to the extent possible written in portable C. The parser and scanner were produced using the Unix YACC and LEX utilities. The output from these utilities require several patches to allow compilation on the Z100 using Lattice 'C.' Readers desiring to port the code to other machines may have to make slight changes to "defines." In this implementation, longs are assumed to occupy 32 bits, both int and short - 16 bits, and char - 8 unsigned bits. NOte: if the int size changes, then the infile and outfile functions in amstate.c must be changed.

The input syntax of AMASM is similar to that of other assemblers. It supports symbolic addresses and constants and a typical set of directives, but has no macro capabilities. The assembler accepts an ASCII source file created on a conventional text editor and produces an output file containing relocation information and AM opcodes. Invoking AM causes the output file "a.am" to be loaded and executed.

2. Differences from Version 1.0

Since it was our intention to primarily specify and describe the abstraction of a database resource, the assembler part for AM was considered to be of less importance for this thesis. Due to the limited time the adaption of AMASM to the database requirements is still incomplete, only some examples are given which indicate a way of how to integrate this latest resource. Thus, for instance, although the read/write commands for the queue were developed, the method of actually retrieving objects from the queue has not yet been defined. The same is true for the database itself, where the only commands described are those for opening and closing the

database, while the other operations were left undefined for the above reasons. So, in fact, AM (version 3.0) represents only a partial extension of AM (version 2.0-Z100).

3. Usage

AMASM is invoked with the following command line syntax:

```
amasm [-t][-x][-s][-l] file ...
```

AMASM produces a single load module "a.am," which forms the input to the AM loader. The optional "-t" switch sends a debugging trace to "stdout," the "-x" switch provides an extended version of the trace, and the "-s" switch provides trace of the recognized scanner tokens. The optional "-l" switch generates the listing and cross-reference file "a.x." Appended to this file is a hex dump of "a.am."

4. Lexical Conventions

Assembler tokens include identifiers (alternatively, "symbols" or "names"), literal constants, operators and delimiters.

4.1. Identifiers

Legal identifiers are described by the following regular expression:

```
[A-Za-z][A-Za-z0-9]*
```

Identifiers consist of a letter or underline "_" followed by a string of zero or more letters, decimal digits and underlines. Upper and lower case are distinct. Identifiers may represent symbolic constants, instruction mnemonics, labels, addresses and type names.

4.2. Operators

The following are considered to be operators:

```
== != < <= > >=
+ - * / % & |
```

The meaning of the above symbols varies with context.

4.3. Literal Constants

Decimal and hexadecimal constants are described by the following regular expressions respectively:

```
[-+][0-9]+| [0-9]+
$[0-9A-Fa-f]+
```

Decimal constants consist of an optional sign followed immediately by one or more decimal digits. Hexadecimal constants consist of the character "$" followed immediately by a string of one or more decimal digits and upper or lower case letters "A"through "F." Numeric constants may represent addresses, integer and natural numbers, boolean and character values.

Character constants consist of a single quote "'", followed either by an ASCII character, that is not a carriage return/linefeed or a numeric constant, followed by a closing single quote.

String constants consist of a string of zero or more ASCII characters (except carriage return/linefeed) enclosed in double quotes.

4.4. Blanks

Blanks and tabs are ignored by the assembler except where required to separate adjacent constants or identifiers.

4.5. Comments

The character ";" produces a comment. The assembler ignores all further characters on the line up to the terminating carriage return/linefeed.

4.6. Delimiters

All other characters found in the input stream are treated as delimiters.

5. Statements

A source program is composed of a sequence of statements, one statement per line. There are 3 kinds of statements: directives, instructions and null.

Instructions and null statements may be preceded by a label. Directives may (in some cases, must) be preceded by an identifier.

5.1. Labels & Identifiers

A label consists of an identifier followed by a colon ":". When the assembler encounters a label, the effect is to assign the current value of the location counter to the name.

An identifier preceding a directive is assigned a value whose type depends upon the directive. For instance, the equate

directive assigns a typed value to an identifier, while the define storage directive assigns the current value of the location counter.

Neither labels nor identifiers may be redefined within a single source file.

5.2. Null Statements

A null statement is an empty statement. Although ignored the assembler, null statements may be preceded by a label.

5.3. Directive Statements

A directive is a command to the assembler to perform some sort of operation which does not involve emitting an executable instruction. Typical directives (also known as "pseudo ops" or "pseudo instructions") allocate storage for variables, make names within the current module visible to other modules and set the location counter. Directives also produce instructions for the AM linker and loader.

Directives consist of a keyword followed by zero or more arguments, depending upon the context. Directives and their syntax are described in more detail in Section 12.

5.4. Instruction Statements

Instruction statements produce the code which is ultimately executed by AM. An instruction may be preceded by a label, and consists of a keyword followed by zero or more arguments, depending upon context.

The AM instruction set and its syntax will be described in detail in Section 14.

6. The Machine

Because AM differs from conventional machines in a number of important ways, some discussion is necessary before introducing the instruction set. Outwardly similar to a number of well-known examples, AM instructions form an unconventional set of primitive operations which implement a formally specified semantics. The reasons for this are described below.

AM uses a tagged architecture. Thus, each data element contains, within it, information which uniquely identifies a finite set of legal operations which may be performed upon it, as well as a range of legal values it may take on. This set of operations and values is known formally as a data type. AM supports a number of data types. An element of a particular data type will be referred to throughout the rest of this manual as an atom.

AM physical resources are partitioned into segments. There are several types of segments, and these together form a conventional overall model of the familiar stored program computer. There are memory segments (primary storage), register segments (high-speed memory), display register segments (bit-mapped display memory), stacks, a queue, a monitor (display terminal attributes) and file segments (secondary storage). Segments are further partitioned into discrete, addressable elements (alternatively, "cells") which will contain atoms during the execution of a program. These elements will be referred to repeatedly as typed values. The reason for the distinction between atoms and values will become more clear shortly.

AM is the finite implementation of a formal specification. As such, data elements and the operations which can be applied to them must reflect a mathematical consistency not required by conventional architectures. Since all operations which affect the state of the machine must be able to "communicate" with each other during the execution of a AM program, they must do so using a common object. This object is a value. The memory, registers, display registers, stack, queue, and files all hold values. Store, fetch, execute, read, write--any operations which change the state of the machine--all operate on values (i.e.,storage cells). All other operations, such as "add," "multiply," "and," and "or," work on atoms. Atomic operations in AM correspond to those which take place in the temporary registers of the arithmetic and logic unit of a conventional processor.

6.1 Configuration

A unique feature of AM is the ease with which it is possible to reconfigure the machine by partitioning the physical resources in different ways. A typical configuration would be something like this:

```
 2 memory segments
 1 register segment (with a useful number of registers)
 1 display register segment (with one or two registers)
 1 stack
 1 queue
 1 monitor (only one is permitted)
 1 database (one or more are possible)
16 files
```

The configuration chosen should provide a good indication of the types of programs AM is intended to execute.

Note that, in conventional machines, stacks are implemented in primary storage. This constitutes an overloading of data structures which obscures the intent of the user of these structures. It also creates a semantic nightmare for the

specification writer. In AM, stacks and queue take their rightful places as separate entities with easy to understand properties.

In addition to the resources listed above, AM has a conventional program counter.

6.1.1. Memory

AM memory is partitioned into segments which may be of unequal but fixed length. A program and its data will reside in memory segments. It is not necessary that code and data share the same segment, nor is it required that code and data be contiguous. The loader will determine from the origin directive where to load code and data values.

The AM heap is implemented as a set of operations which allocate and deallocate memory segments.

AM has a rich set of addressing modes which interact with a powerful move instruction which allows the programmer to move a value from "anywhere to anywhere."

61.2. Registers

AM registers form the high-speed storage into which operands are placed.

All atomic operations, such as add, divide and poffst, require operands to be in registers. Form operations are an exception. Their operands may be in either a register or a display register.

6.1.3. Display Registers

The form is the atomic data type that represents an image. Like any other atomic data type, it may be placed in any memory, register, stack or file cell. A form can not be "viewed" by the monitor unless it is in a display register.

Display registers may only contain form values. Each display register has its own window which is fixed in size but with a variable origin. The display window determines what part of the form is "viewed" by the monitor.

In general, display registers may be partitioned into multiple segments. However, the hardware on most machines will only support one segment of one or two registers. A segment of two display registers is equivalent to the idea of a "front" and "back" plane.

6.1.4. Monitor

The monitor represents a set of terminal attributes which are part of the "state of the machine." The attributes: vertical and horizontal number of pixels, vertical and horizontal screen dimensions, intensity capability and color planes are fixed for any terminal. The background color and display register selection attributes are programmable.

6.1.5. Stack

The AM stack is conventional in every respect except that it is impossible to access any value except the top. Thus, frames are implemented on the heap, not the stack.

AM has a typical set of push and pop instructions for operating on stacks.

6.1.6. Files

Input/output is implemented rather arbitrarily along the lines of system calls to an operating system and should not be considered part of AM itself.

Instructions are provided to open, close, read to and write from a file.

6.1.7. Queue

Primarily, the queue acts as a buffer for 'objects' being retrieved from the database during a select operation. It is implemented in the same way as the stack to prevent the accessing of any value except the one residing in the front position. This method ensures that the order of the values defining an object will be kept.

A set of write and read instructions is provided for operating on the queue.

6.1.8. Database

The database consists of two major parts: the data representing the information and a set of commands to perform the defined operations on it. These commands can only be applied to data that have explicitly been specified as a database and meet its structural requirements. In principle, the data need to be arranged as ordered pairs of lists. For database operations all resources of AM may be used, with exception of the display registers and the monitor.

Instructions are provided to open and close the database.

7.0 Atoms

An atom is a component of a data type. The assembler recognizes the following type of atoms:

```
file address
pidlist
pidsetlist
vallist
valsetlist
proplist
propsetlist
pvallist
pvalsetlist
objlist
classlist
dblist
qaddress
```

As operands to instruction mnemonics, these atoms form the familiar set of literal and symbolic constants found in typical assembly language programs.

With certain exceptions, atoms may appear in the form of literal constants:

```
100
$d0f1
'a'
"this is a string atom"
```

They may also appear as symbols which take on the value of the atom in some other part of the source program. With few exceptions, anywhere a literal constant may be used, a symbolic constant of the appropriate type may also be used.

The assembler distinguishes between types of atoms using syntax and context. The syntax is described below.

7.1. Boolean

A boolean atom has only two values, true and false. These values are represented to the assembler by the decimal or hexadecimal constants for 1 and 0, respectively.

```
0
1
$1
$0
```

are legal boolean atoms.

7.2. Natural

This type represents as the name implies, the natural (unsigned) numbers. Legal values range from zero to positive

infinity. Natural numbers are represented to the assembler as decimal or hexadecimal constants whose values are greater than or equal to zero.

```
0
$2f5
240
```

are legal natural atoms.

7.3. Integer

Integers range from negative to positive infinity, and are specified as hexadecimal or signed or unsigned decimal constants.

```
-250
0
$ed67f
+10
```

are legal integer atoms.

7.4. Character

Character atoms may take values defined by the ASCII character set. They are represented to the assembler as literal character constants.

```
'a'
'r'
```

are legal character atoms.

7.5. String

String atoms are composed of zero or more concatenated ACIII characters. They are specified as literal strings.

```
"this is a legal string atom"
""
```

are both legal string atoms.

7.6. Intensity

An intensity atom ranges from 0 to 199 decimal. It is represented as a unsigned decimal or hexadecimal constant preceded with the character "&." "@" represents the null intensity which is used to construct the null color.

```
&@
&0
&89
&199
```

are legal intensity atoms.

7.7. Color

A color atom is a composite of a red, green and blue intensity. It is represented as an ordered triple of unsigned decimal or hexadecimal constants separated by commas ",", enclosed within parenthese "("")" and preceded with "&". The nullcolor provides the concept of background and transparency. It is represented as the "@" enclosed within parentheses and preceded with "&".

```
&(0,0,0)
&(70,0,190)
&@
```

are legal color atoms.

7.8. Point

Points are composed of integer pairs. The x and y coordinates correspond to the first and second integers respectively. Increasing integer values represents positions shifted right and up. A point is represented as an ordered pair of decimal or hexadecimal constants separated by a comma ","and enclosed within parentheses "("")".

```
(0,0)
(4,1047)
(-8,25)
(-50677,-293399)
```

are legal point atoms.

7.9. Rectangles

Rectangles are composed of a pair of points which represent the opposing corners. A rectangle is represented as an unordered pair of points separated by a colon ":" and enclosed within square brackets "["]".

```
[(0,0):(0,0)]
[(0,0):(50,45)]
[(50,45):(0,0)]
[(-20,-20000):(30,59)]
```

are legal rectangle atoms.

7.10. Form

A form atom is a composite structure. It has a two dimensional size and a color map which is an array of colors with each color corresponding to a point in its area. The form atom has no literal constant representation. It is created using the operator, newfrm, and modified using other operators.

7.11. Font

A font atom is an array of forms. The font atom has no literal constant representation. It is constructed from the operator, newfnt, and modified using other operators.

7.12. Ptblt

A ptblt atom is a composite of three rectangles and a natural which represents a copy rule. The ptblt atom has no literal constant representation. It is constructed from the operator, newblt, and is modified using other operators.

7.13. Memory Address

Memory address atoms consist of two components: a segment address,and an element address. Memory addresses are represented as an ordered pair of unsigned decimal or hexadecimal constants, separated by a colon ":" and enclosed within parentheses "("")".

```
(0:100)
```

represents memory segment 0, element 100.

```
(2:$10)
```

represents segment 2, element 16.

Segment and element addresses start at 0. The number and size of available memory segments depends upon the current configuration of AM.

Labels are considered memory address atoms, as are names which appear to the left of the define storage and define constant directives.

7.14. Register Address

Register address atoms have a syntax identical to that of memory addresses except that a lower case "r" is prepended to the address.

```
r(0:3)
```

refers to register segment 0, register 3.

Segment and element addresses start, as with memory addresses, at 0. The number of register segments, and the number of registers within each segment, varies as determined by the current AM configuration.

7.15. Display Register Address

Display register address atoms have a syntax identical to that of register addresses except that the lower case "r" is replaced with a lower case "d".

```
d(0:1)
```

refers to display register segment 0, register 1.

Segment and element addresses start at 0. The number of display register segments, and the number of display registers within each segment, varies as determined by the current AM configuration.

7.16. Monitor Attribute

The monitor consists of eight attributes values which are:

```
x--represents number of horizontal pixels (natural)
y--represents number of vertical pixels (natural)
v--represents screen height in inches (natural)
h--represents screen width in inches (natural)
i--represents intensity capability (natural)
c--represents number of color planes (natural)
b--current background color (color)
d--selected display register to view
       (display register address)
```

A monitor attribute is represented by a dash "-" followed by one of the above characters for the indicated attribute.

```
-x
-y
-b
```

are all legal monitor attribute atoms.

7.17. Stack Address

A stack address has only one component: the segment address. Stack addresses are specified by prepending a lower case "s" to an unsigned decimal or hexadecimal constant enclosed within parentheses.

```
s(2)
```

refers to stack segment 2.

Stack addresses begin at 0. The number of stacks depends upon AM's configuration.

7.18. File Addresses

File address atoms may not appear in a program except within typed values. File address atoms are represented as unsigned integer or hexadecimal constants.

File addresses start at 0. The number of files which may be open at one time is determined by the current AM configuration. The first three file addresses (0,1,2) are normally opened automatically by AM when a program is loaded.

7.19. Pidlist

Pidlist atoms are composed of one or more concatenated ASCII characters and form single strings that must not be empty. They are surrounded by angle brackets.

```
<name>
<id>
<grade_1>
```

are all legal pidlist atoms.

7.20. Pidsetlist

This type represents a number of zero or more pidlists, separated by commas "," and enclosed within a set of angle brackets.

```
<name,age,grade>
<>
```

are both legal pidsetlist atoms.

7.21. Vallist

Vallists are represented like pidlists as strings of one or more concatenated ASCII characters. Type distinction is made in accordance with the context in which they appear. Arithmetic operations on vallist atoms are not possible since they are treated as characters.

```
<123>
<A>
```

are legal vallist atoms.

7.22. Valsetlist

Analogous to the pidsetlist, valsetlist atoms are composed of zero or more vallists, separated by commas "," and enclosed within a set of angle brackets. Since a valsetlist atom is actually used to define a certain domain of values, it most likely will be of the following form:

```
<1,2,3,4,17,123>
<A,B,C,D,E,F>
<Monterey,San_Diego>
<>
```

but

```
<John,Cindy,Monterey>
```

would also be a legal valsetlist atom.

7.23. Proplist

Proplists are composed of ordered pairs that consist of a pidlist and a valsetlist, additionally enclosed within angle brackets and separated by a comma ",".

```
< <name>,<John,Cindy,Mark> >
< <grade>,<A,A-,B+> >
```

are legal proplist atoms.

7.24. Propsetlist

Propsetlist atoms are represented by zero or more proplists, additionally enclosed within angle brackets and separated by commas ",". Since a proplist consists of the ordered pair pidlist and valsetlist, a propsetlist atom also contains a number of ordered pairs.

```
< < <name>,<John,Cindy,Mary> >,
  < <age>,<20,10,30,17,65>   > >
```

is a legal propsetlist atom.

7.25. Pvallist

This type is composed of the ordered pair pidlist and vallist separated by a comma "," and additionally enclosed within angle brackets.

```
< <name>,<John> >
< <name>,<Cindy> >
< <age>,<17> >
```

are legal pvallist atoms.

7.26. Pvalsetlist

A pvalsetlist atom consists of zero or more pvallists, separated by commas "," and additionally enclosed within angle brackets. It is arranged as a number of ordered pairs.

```
< < <name>,<John> >,< <age>,<25> >, < <city> <Monterey> > >
<<>>
```

are both legal pvalsetlist atoms.

7.27. Objlist

Objlists are composed of zero or more different pvallists and can be considered as particular pvalsetlists. An objlist consists of a number of ordered pairs that like a pvalsetlist, are enclosed within an additional set of angle brackets and separated by commas ",". It can be empty, although this would not be meaningful.

```
< < <name>,<John> > >
< < <name>,<Cindy> >,< <sex>,<female> >,< <age>,<20> > >
<<>>
```

are legal objlist atoms.

7.28. Classlist

This type is represented by zero or more objlists, additionally enclosed within angle brackets and separated by commas ",". It is mandatory that all objlists belonging to the same classlist are equally structured. That is, their pidlist atoms must be identical.

```
< < < <name>,<John > >,< <age>,<25> > >,
  < < <name>,<Cindy> >,< <age>,<19> > >,
  < < <name>,<Paul > >,< <age>,<20> > > >
```

is a legal classlist atom.

7.29. Dblist

The dblist is composed of zero or more classlists which are additionally enclosed within angle brackets and separated by commas ",". An objlist atom can only be contained in the dblist if it is part of a classlist that itself must be contained in the dblist. Since the structure is top-down, a pidlist not included in any classlist may be comprised in the dblist, but never the reverse.

The following shows a legal dblist atom:

```
< < < < <name>,<John> >,< <score>,<375> > >,
    < < <name>,<Mary> >,< <score>,<380> > > >,

  < < < <course>,<CS4600> >,< <room>,<13> >,< <hours>,<4.0> >
    < < <course>,<OR3333> >,< <room>,<42> >,< <hours>,<4.5> >

  < < < <ID>,<ab> >,< <est>,<r19> > >,
    < < <ID>,<xy> >,< <est>,<o23> > >,
    < < <ID>,<vw> >,< <est>,<a95> > > > >
```

The database structure is simple and can easily be disclosed. The first list of this structure always corresponds to the first object class, while the first list of an object class is equivalent to its first object. Then the first list of an object represents its first property value which itself contains the property_id as first element and the corresponding value as its second.

7.30. Qaddress

The only component of a queue address is the segment address. Queue addresses are specified by prepending a lower case "q" to

an unsigned decimal or hexadecimal constant enclosed within parentheses.

```
q(1)
```

refers to queue segment 1.

8. Typed Values

Some of the atomic types may also appear as typed values in certain instructions and directives. A typed (immediate) value is represented as an ordered pair consisting of a keyword representing the type, and the atom itself, separated by a comma "," and enclosed within curly braces "{""}".

```
{int.100}
```

represents the integer value 100.

```
{addr.(1:100)}
```

represents memory address value (1:100).

A list of the types which may be used as immediate values alongside the corresponding keywords appears below:

```
bool--boolean
nat--natural
int--integer
char--character
string--character string
intens--intensity
color--color
pnt--point
rct--rectangle
addr--memory address
file--file address
pidlist--property id list
pidsetlist--property_idset list
vallist--value list
valsetlist--valueset list
proplist--property list
propsetlist--propertyset list
pvallist--propertyvalue list
pvalsetlist--propertyvalueset list
objlist--object list
classlist objectclass list
dblist--database list
```

Immediate values are used, as in conventional assembly languages, for loading constants into cells, initializing storage, pushing parameters to subroutines on the stack, and so on.

A special syntax may be applied when expressing typed values for the define storage and define directives. The type

keyword may be followed by a list of atoms of the appropriate type, separated by commas.

```
{int,1,2,3,4,5,6,7,8}
```

shows an example of this.

9. Expressions

An expression may be substituted anywhere an integer or natural atom is called for. The expression must be a sequence of integer/natural atoms (and symbolic constants equated to integer/natural atoms) separated by operators and grouping symbols which evaluates to an atom of the type called for where the expression is used.

9.1. Expression Operators

Legal operators are (in order of increasing precedence):

```
|      - or
&      - and
+-     - addition and subtraction
*/%    - multiplication, division, and modulus
-      - unary minus
```

Expressions may be grouped using parentheses "("")".

10. Notation

Throughout the rest of this manual, the following notational conventions will be used to describe the syntax of directives and instructions.

```
A - atom
V - typed value
N - natural atom
I - integer atom
M - memory address atom
R - register address atom
D - display register address atom
C - either a display or a high speed register address atom
T - monitor attribute atom
S - stack address atom
< > - items enclosed within angle brackets are arguments
[] - items enclosed in square brackets are optional
<ea> - effective address
<ev> - effective value
Q - queue address atom
```

11. Data Format

AMASM emits object code and directives using AM I/O modules. The object module is, thus, directly readable by AM. A linker and loader may be written either in a high level language, or AM assembler.

The data and object module formats described below are a direct reflection of AM's tagged architecture. The following conventions will apply:

- All numbers shown are in hexadecimal.
- The letter "H" is a place holder signifying any 4-bit value.
- The letter "D" is a place holder signifying any 32-bit value.
- The letter "P" is a place holder signifying a 32-bit pointer.
- The general form of a typed value is

  |tag| |val|

  where "tag" is a 16-bit type field, and "val" is either an 8 to 32-bit value or a 32-bit pointer.

Note the following:

- Character string atoms and values have a 16-bit size field inserted after the type field which indicates the number of characters in the value field (including the terminating null). This size field is omitted in memory (since it is not needed) and replaced by a pointer to the string.
- Instruction values have a 32-bit pointer following the type field, which points to an array of values. The first value is the opcode followed by the operands. The number of operands is encoded in the opcode.
- Form values have a 32-bit pointer to a form header. The header contains the form's rectangle and a pointer to the cmap which is an array of colors. The length of the cmap is determined from the form's rectangle.
- Font values have a 32-bit pointer to a font header. The header contains the font' rectangle and a 128 member array of cmap pointers.
- All list atoms and values with the exception of the dblist type have a 16-bit size field that is inserted after the type field and indicates the number of characters contained in the value field. Similar to the string type, this size field is replaced in memory by a pointer to the corresponding list.
- Dblist atoms and values have a 32-bit size field inserted after the type field which indicates the number

of characters in the value field and represents the total number of characters contained in the database. In memory the size field is replaced by a 32-bit pointer to the dblist.

A number of the formats listed below are not described elsewhere in this manual since they are either not accessible to the programmer, or are implied by context.

11.1. Atom Formats

boolean - |0001| |HH|
natural - |0002| |HHHH|
integer - |0003| |HHHH|
character - |0004| |HH|
character string - |0005| |P| |HH...00|
intensity - |0006| |HH|
color - |0007| |HH HH HH|
point - |0008| |P| |DD|
rectangle - |0009| |P| |DDDD|
form - |000A| |P| |DD DD| |P| |-cmap array-|
font - |000B| |P| |DD DD| |-128 P's-|
ptblt - |000C| |P| |DDDD DDDD DDDD HH|
memory address - |0030| |D|
register address - |0031| |D|
display register address - |0032| |D|
monitor attribute - |0033| |HH|
stack address - |0034| |D|
file address - |0035| |HHHH|
monadic operator - |0040| |HHHH|
dyadic operator - |0041| |HHHH|
triadic operator - |0042| |HHHH|
quadadic operator - |0043| |HHHH|
sexadic operator - |0044| |HHHH|
octadic operator - |0045| |HHHH|
relational operator - |0046| |HHHH|
boolean comparator - |0047| |HHHH|
pidlist - |000D| |P| |HH...00|
pidsetlist - |000E| |P| |HH...00|

vallist - |000F| |P| |HH...00|
valsetlist - |0010| |P| |HH...00|
proplist - |0011| |P| |HH...00|
propsetlist - |0012| |P| |HH...00|
pvallist - |0013| |P| |HH...00|
pvalsetlist - |0014| |P| |HH...00|
objlist - |0015| |P| |HH...00|
classlist - |0016| |P| |HH...00|
dblist - |0017| |P| |D...00|
qaddress - |0036| |D|

11.2. Value Formats

boolean - |0201| |HH|
natural - |0202| |HHHH|
integer - |0203| |HHHH|
character - |0204| |HH|
character string - |0205| |P| |HH...00|
intensity - |0206| |HH|
color - |0207| |HH HH HH|
point - |0208| |P| |DD|
rectangle - |0209| |P| |DDDD|
form - |020A| |P| |DD DD| |P| |-cmap array-|
font - |020B| |P| |DD DD| |-128P's-|
ptblt - |020C| |P| |DDDD DDDD DDDD HH|
memory address - |0230| |D|
register address - |0231| |D|
display register address - |0232| |D|
monitor attribute - |0233| |HH|
stack address - |0234| |D|
file address - |0235| |HHHH|
instruction - |0250| |P| |HHHH| |zero or more operand atoms|
pidlist - |020D| |P| |HH...00|
pidsetlist - |020E| |P| |HH...00|
vallist - |020F| |P| |HH...00|
valsetlist - |0210| |P| |HH...00|

proplist - |0211| |P| |HH...00|
propsetlist - |0212| |P| |HH...00|
pvallist - |0213| |P| |HH...00|
pvalsetlist - |0214| |P| |HH...00|
objlist - |0215| |P| |HH...00|
classlist - |0216| |P| |HH...00|
dblist - |0217| |P| |D...00|
qaddress - |0236| |D|

11.3. Object Module Format

The structure of an object module is very simple. The only object always found is a leading org directive. Next, if any symbols were declared global or external in the source module, a pseudo instruction will be emitted for each such symbol. The rest of the file contains executable and pseudo instructions emitted as they occur in the source.

12. Assembler Directives

AMASM recognizes the following directives:

```
equ    - equate
org    - absolute origin
rorg   - relative origin
extern - external symbol
globl  - global symbol
trace  - trace execution
ds     - define stroage
dc     - define constant
```

Directives do not produce code which will be executed by AM, but they may cause linker/loader instructions to be emitted. The meaning and syntax of each directive is described in the following pages.

Syntax:

```
<name> equ <equivalence>
```

where:
<name> is any legal identifier
<equivalence> is any atom or typed value

Description:

The symbol <name> is assigned the value of <equivalence>. Elsewhere in the source module, the symbol may be used in place of a literal value of the same type as <equivalence> using the following syntax:

- If the symbol represents a memory address atom, the symbol may be used directly.
- If the symbol represents a typed (immediate) value, it must be enclosed in curly braces "{""}".
- If the symbol represents an integer or natural atom, it must be preceded by a pound sign "#".

Example:

```
propseg    equ  (0:0)
dataseg    equ  (1:100)
offset     equ  10
datafile   equ   file,3

           org  progseg
           move {addr,data},r(0:0)
           move {int,100},r(0:0)@#offset

           push {string,"test.dat"}s(0)
           push {datafile},s(0)
           push {int,0},s(0)
           push {int,0},s(0)
           open s(0)
           stop

           org  dataseg
data       ds   100
```

"progseg" and "dataseg" are equated to memory address atoms.

"offset" is equated to the integer atom 10.

"datafile" is equated to the file address value {file,3}.

Format:

equ does not cause an emission.

Syntax:

```
org [M]
```

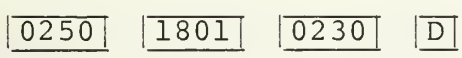
Description:

The location counter is reset to M, if specified; otherwise it remains unchanged. All memory addresses and labels specified after an org directive up to the next org or rorg directive not explicitly expressed as displacements are treated as absolute addresses. Code generated after an org directive up to the next org or rorg directive is not relocatable.

Example:

```
                org
                move    (0:0),r(0:0)

                org     (1:0)
data            ds      {int,100},{nat,0}
```

Format:



Syntax:

rorg [M[

Description:

The location counter is reset to M, if specified; otherwise it remains unchanged. All memory addresses and labels specified after a rorg directive up to the next org or rorg directive are computer as displacements. Code generated after a rorg directive up to the next org or rorg directive is relocatable (program counter independent).

Example:

```
            rorg


            move {int,100},data
            jsr  stuff
            stop


data        ds   10
```

In the above example, the move would be emitted using destination program counter relative addressing.

Format:

|0250| |1801| |0230| |D|

Syntax:

```
extern <name>...
```

where:
<name> is any legal identifier

Description:

The list of symbols is made visible to the current module and is assumed to be defined elsewhere. An error is flagged if a symbol in the list is not referenced somewhere within the current module. It is also an error for any symbol in the list to be defined within the current module.

Example:

```
        extern expon

        push  {int,100},s(0)
        jsr   expon,s(0)
```

Format:

For each symbol decalred external, an extern pseudo op is emitted, followed by a string containing the symbol.

Syntax:

globl <name>...

where:
<name> is any legal identifier

Description:

The list of symbols is made visible to external modules. Each name in the list must be defined as a memory address somewhere within the current module.

Example:

```
            globl  test,data

test:

            move   (0:0),r(0:0)
            stop
data        ds     10
```

"test" and "data" are made visible to other modules.

Format:

For each symbol declared global, a globl pseudo op is emitted, followed by a string containing the symbol, followed by a memory address representing the value of the symbol.

Syntax:

trace <flag>,<toggle>

where:
<flag> is "-t" for normal trace and "-x" is for extended trace
<toggle> is "+" for on and "-" for off

Description:

A trace of the programs execution is available in two modes, normal and extended. The normal mode traces the main function calls and the major paths through them. The extend mode includes the normal trace plus memory allocation calls and creation of temporary values. The trace directive may be selected in the command line when AM is invoked, or embedded in the source code to enable trace over selected portions of the program.

Example:

```
progseg         equ   (0:0)
          org   progseg
          move  {addr,data},r(0:0)
          trace -t,+
          move  {int,100},r(0:0)@
          trace -t,-
          push  {int,0},s(0)
          stop
data      ds    100
```

Format:

| 0250 | 3800 | 0204 | HH | 0203 | HHHH |

Syntax:

```
[<name>] ds N[V...]
[<name>] ds [N] V...
```

where:
<name> is an optional identifier

ds permits a list of atoms to follow the type keyword of each value.

Description:

ds allocates storage for values starting at the current value of the location counter.

- If N is specified and N is greater than or equal to the number of values in the list, space for N values is allocated and the location counter is incremented by N.

- If N is specified and N is less than the number of values in the list, N is ignored.

- If N is not specified, the amount of storage allocated is equal to the number of values in the list. The location counter is incremented by this number.

- If a value list is specified, the allocated cells will be initialized to those values, beginning with the first.

- Cells allocated but not initialized are considered to hold undefined values. It is an error to attempt to read an undefined value.

Example:

```
data 1      ds    10
data2       ds    10{int,100},{nat,0,20,40}
data3       ds   {char,'a','b'}
            ds   {string,"this is a string value"}
```

The first ds allocates 10 values and leaves them undefined. "data1" may be used to index into those values.

The second also allocates 10 values, but initializes the first to the integer 100, and the next 3 to the naturals 0, 20, and 40. The last 6 values are left undefined.

The third ds shown allocates 2 character values.

The fourth allocates a single string value. No identifier was specified.

Format:

A typed value is emitted for each value in the list. In addition, ds will emit an org pseudo op (see org) whenever the number of values in the value list is less than N.

Syntax:

[<name>] dc V...

where:
<name> is an optional identifier

dc permits a list of atoms to follow the type keyword of each value.

Description:

dc allocates and initializes storage from a list of values starting at the current value of the location counter.

Example:

```
data3       dc    {char,'a','b'}
            dc    {string,"this is a string value"}
```

The first ds shown allocates 2 character values.

The second allocates a single string value. No identifier was specified.

Format:

A typed value is emitted for each value in the list.

13. Addressing Modes

AM supports 11 addressing modes:

```
d    - display register direct
r    - register direct
ri   - register indirect
rid  - register indirect with displacement
ridn - n-level register indirect with displacement
m    - memory absolute
mi   - memory indirect
pcr  - program counter relative
i    - immediate value
a    - immediate atom
s    - stack direct
q    - queue direct
```

Like other more familiar processors, not all AM instructions can use all of the addressing modes.

In addition, AMASM supports address expressions, which provides a rudimentary indexing capability.

13.1. Display Register Direct

The form operand is in the display register.

Syntax: D

Format:

```
|0232| |D|
```

13.2. Register Direct

The operand is in a register.

Syntax: R

Format:

```
|0231| |D|
```

13.3. Register Indirect

The address of the operand is in a register.

Syntax: R@

R - holds the operand address

Format:

```
|0231| |D|
```

13.4. Register Indirect with Displacement

The address of the operand is the sum of the address in a register and an integer displacement.

Syntax: R@I

R - holds a base address
I - an integer displacement

Format:

13.5. N-level Register Indirect with Displacement

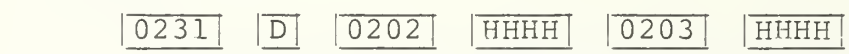
The address of the operand is the sum of the address obtained from the nth link in a chain of dynamic links and an integer displacement.

Syntax: RN@I

R - holds the current frame pointer
N - a non-negative frame reference
I - an integer frame displacement

(RN@I) is equivelent to R@I)

Format:

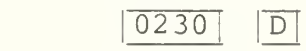


13.6. Memory Absolute

Syntax: M

M - the operand address

Format:



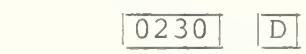
13.7. Memory Indirect

The address of the operand is in a memory cell.

Syntax: M@

M - a pointer to the operand address

Format:



13.8. Program Counter Relative

The address of the operand is the sum of the program counter and an integer displacement.

Syntax: M

M - the operand address

The specified address must be in the same module as the instruction. The assembler automatically computes the displacement. Program counter relative is specified for a block by placing a rorg directive at the top of the block.

Format:

| 0203 | | D |

13.9. Immediate Value

The operand is an immediate value.

Syntax: V

V - any typed value

Format:

| tag | | valu |

13.11. Stack Direct

The operand is a stack.

Syntax: S

Format:

| 0234 | | D |

13.12. Queue Direct

The operand is a queue.

Syntax: Q

Format:

| 0236 | | D |

14. Instruction Set

The AM instruction set is simple but powerful. The rigid data types make it meaningless to specify operations like shift and mask, thus removing some of the programmer's freedom to

muck with data in arbitrary ways. The tagged architecture will detect errors like jumping to data, or accessing instructions as data, as well as the more common bounds checking performed by runtime libraries.

141. Machine Errors

The following errors are detected by AM during loading and execution:

- attempt to execute a non-instruction
- attempt to execute an illegal instruction
- memory segment not defined
- memory segment overflow
- memory segment underflow
- register segment not defined
- register segment underflow
- register segment underflow
- display register segment not defined
- stack segment not defined
- undefined monitor attribute
- <file> contains unresolved references
- attempt to convert negative int to nat
- no predecessor to zeronat
- no predecessor to minintens
- no successor to maxintens
- addition illegal with nullintens
- subtraction illegal with nullintens
- gtintens illegal with nullintens
- ltintens illegal with nullintens
- geintens illegal with nullintens
- leintens illegal with nullintens
- illegal color definition
- form is not correct size for font
- icon is undefined
- unknown operator to applymop
- unknown operator to applydop
- unknown operator to applytop
- unknown operator to applyqop
- unknown operator to applysop
- unknown operator to applyoop
- unknown operator to applyrop
- unknown operator to applybop
- type error - GT
- type error - GE
- type error - LT
- type error - LE
- no more segment available
- attempt to free invalid memory segment
- attempt to free non-allocated segment
- stack empty
- stack overflow
- stack underflow

- file already open
- unable to close file
- unable to open <file>
- file already closed
- file not open
- file not open for reading
- file not open for writing
- reading file, type not recognized
- error reading file
- writing file, type not recognized
- invalid memory segment
- memory segment not allocated
- invalid memory address
- invalid register segment
- invalid register address
- invalid stack segment
- invalid file descriptor
- attempt to return head of null string
- value not of type bool
- atom not of type bool
- value not of type int
- atom not of type int
- value not of type nat
- atom not of type nat
- value not of type char
- atom not of type char
- value not of type string
- atom not of type string
- value not of type ilev
- atom not of type ilev
- value not of tyep colr
- atom not of type colr
- value not of type pnt
- atom not of type pnt
- value not of type rct
- atom not of type rct
- value not of type form
- atom not of type form
- value not of type font
- atom not type font
- value not of type ptblt
- atom not of type ptblt
- value not of type mad
- atom not of type mad
- value not of type rad
- atom not of type rad
- value not of type dad
- atom not of type dad
- value not of type mattribute
- atom not of type mattribute
- value not of type sad

- atom not of type sad
- value not of type file
- atom not of type file
- value not of type mop
- atom not of type mop
- value not of type dop
- atom not of type dop
- value not of type top
- atom not of type top
- value not of type qop
- atom not of type qop
- value not of type sop
- atom not of type sop
- value not of type oop
- atom not of type oop
- value not of type rop
- atom not of type rop
- value not of type bop
- atom not of type bop
- value not of type instr
- atom not of type instr
- type error
- queue segment not defined
- queue empty
- queue overflow
- queue underflow
- db already open
- unable to close db
- unable to open <db>
- db already closed
- db not open
- illegal object insertion
- object not contained in class
- invalid queue segment
- atom not of type pid
- value not of type pid
- atom not of type val
- value not of type val
- atom not of type pval
- value not of type pval
- atom not of type obj
- value not of type obj
- atom not of type db
- value not of type db
- atom not of type lst
- value not of type lst

All machine errors are fatal.

14.2. Assembler Errors

AMASM will detect and report the following errors:

- symbol not an address
- symbol defined locally
- <symbol> does not match declared type
- relative memory indirect not permitted
- symbol not a value
- symbol not an integer
- intensity value exceeds range
- symbols declared but not referenced
- displacement from external addresses not permitted
- relative addressing not permitted between segments
- out of symbol space
- symbol declared external
- symbol already defined
- symbol not of same type
- impossible value for given type
- syntax error

Assembler errors are not fatal, but will prevent the creation of the object module and, usually, the cross-reference file.

14.3. AM Operations

AM supports a useful set of monadic, dyadic, triadic, quadadic, sexadic, octadic, relational and test operators. These operators are to be used with the monad, dyad, triad, quad,sexad, octad, if and test instructions. The mnemonics/ symbols for each operator along with the data types to which each may be applied are described below.

14.3.1. Monadic Operators (MOP's)

not - boolean negation

not accepts a boolean argument and returns its negation

abs - absolute value

abs accepts an integer argument and returns is absolute value

ntoi - natural to integer

ntoi accepts a natural argument and converts it to an integer

iton - integer to natural

iton accepts an integer argument and converts it to a natural

len - string length

len accepts a string argument and returns its length as a natural number.

make - make a string

This operator accepts a character argument and returns a string of length 1.

head - the head of a string

This operator accepts a string argument and returns the character at its head. It is an error to take the head of an empty string.

tail - the rest of a string

tail accepts a string argument and returns a string containing all but the first character. The tail of an empty string is the empty string.

remp,gcmp,bcmp - color components

remp, gcmp and bcmp accept a color argument and return the respective red, green, or blue component of the color.

xcord,ycord - point coordinate

xcord and ycord accept a point argument and return the respective coordinate integer.

origin,corner - rectangle corner points

These operators accept a rectangle argument and return a corner point. Origin returns the lower left and corner the upper right.

xdim,ydim - rectangle dimensions

xdim and ydim accept a rectangle argument and return the respective dimension integer.

newfrm - new form

newfrm accepts a rectangle argument and returns a new blank form whose rectangle is the same as the input rectangle.

farea - form area

farea accepts a form argument and returns its rectangle.

gblts,gbltd,gbltc - get ptblt rectangles

These operators accept a ptblt argument and return the specified rectangle. gblts returns the source, gbltd returns the destination, and gbltc returns the clipping rectangle.

gbltr - get ptblt rule

gbltr accepts a ptblt argument and returns the natural that represents the copy rule.

newfnt - new font

newfnt accepts a rectangle argument and returns an empty font whose icon rectangles are the same as the input rectangle.

rctfnt - rectangle of font

rctfnt accepts a font argument and returns its rectangle.

lenfnt - length of font

lenfnt accepts a font argument and returns the number of icons in it as a natural.

makenew - make a list

This operator accepts a list argument and returns a new list.

first - the first list

first accepts a list argument and returns the first list contained in it. It is an error to take the first of an empty list.

rest - the rest of a list

rest accepts a list argument and returns a list containing all but the first list. It is an error to apply this operator to the empty list.

sofirst - set of first lists

This operator accepts a list and returns the set of all first lists contained in it. Applying sofirst to a list which does not contain at least two sublists results in an error.

14.3.2. Dyadic Operators (DOP's)

and,or

and and or accept two boolean arguments and return a boolean result.

add,sub,mul,div,mod - computational operators

These operators accept integer, natural or intensity arguments (both of the same type) and return a result of that type. Divide by zero returns an error. div discards any remainder. mod returns the remainder. mul, div and mod do not apply to intensity arguments.

cat - string concatenation

cat accepts two string arguments and returns the concatenation of the first onto the second.

loc - point location

loc accepts two integer arguments and returns the defined point.

Usage - loc(x,y) where x is the x coordinate integer and y is the y coordinate integer.

area - rectangle definition

area accepts two unordered point arguments and returns the defined rectangle.

inrct - point in rectangle

inrct accepts a point and a rectangle argument, checks if the point is inside the area of the rectangle, and returns the boolean result.

Usage - inrct(p,r) where p is a point and r is a rectangle.

intrct - rectangle intersection

intrct accepts two rectangle arguments and returns the intersection rectangle.

putrct - put rectangle at

putrct accepts a point and a rectangle argument and returns the rectangle with the same area as the input and its origin at the point argument.

Usage - putrct(p,r) where p is a point and r is a rectangle.

mapsp,mapps - conversion operators

These operators convert points between point coordinates and font spot coordinates. They accept a point and a font argument and return a point. mapsp takes a spot coordinate and based on the font size returns its origin point, e.g., the origin point of spot (2,3) for a 10 by 10 font is point (20,30). mapps takes a point and returns the font spot that it falls inside, e.g., the point (21,31) for a 10 by 10 font is in spot (2,3).

Usage:

- mapsp(f,p) where f is a font and p is a point,
- mapps(f,p) where f is a font and p is a point.

gcolor - get color

gcolor accepts a point and a form argument and returns the form's color at that point.

Usage - gcolor (p,f) where p is a point and f is a font.

fill - fill the form

fill accepts a color and a form and returns the from with all its points set to the color argument.

Usage - fill(c,f) where c is a color and f is a font.

sblts,sbltd,sbltc - set ptblt rectangles

These operators accept a rectangle and a ptblt argument and return the ptblt with the specified rectangle set to the rectangle argument.

sblts sets the source, sbltd sets the detaintion, and sbltc sets the clipping rectangle.

Usage - sblt_(r,b) where r is a rectangle and b is ptblt.

sbltr - set ptblt rule

sbltr accepts a natural and a ptblt argument and returns the ptblt with copy rule set to the natural argument.

Usage - sbltr(n,b) where n is a natural and b is ptblt.

infnt - is icon in font

infnt accepts a natural and a font argument and returns a boolean result based on whether the icon indexed by the nautral argument is defined.

Usage - infnt(n,f) where n is a natural and f is a font.

dfnt - delete icon

dfnt accepts a natural and a font argument and returns the font with the indexed icon deleted.

Usage - dfnt(n,f) where n is a natural and f is a font.

gfnt - get icon

gfnt accepts a natural and a font argument and returns the form of the icon indexed.

Usage - gfnt(n,f) where n is a natural and f is a font.

un - union of lists

un accepts two lists as arguments and returns the union of both.

int - intersection of lists

int accepts two list arguments and returns the intersection of both.

cat - list concatenation

This operator accepts two list arguments and returns the concatenation of the first list onto the second.

get - get a list

get accepts two list arguments and returns the list from the first argument that corresponds with the second. If any of the two arguments is the empty list the operation results in an error.

de - delete a list

This operator accepts two list arguments and returns a list that is equal to the second argument but reduced by the list indicated by the first argument. It is an error to apply de to an empty list or to specify a first argument that is not contained in the second.

retobj - retrieve an object

retobj accepts two list arguments and returns the list that corresponds to the second argument. If the second argument is the empty list the result will also be the empty list.

14.3.3. Triadic Operators (TOP's)

dcolor - define color

dcolor accepts three intensity arguments and returns the defined color.

Usage - dcolor(r,g,b) where r is the red intensity, g is the green intensity, and b is the blue intensity.

poffst

poffst accepts a point and two integer arguments and returns the point that is offset from the point argument by the integer arguments.

Usage - poffst(x,y,p) where x and y are the offset integers and p is the reference point.

sftrct - shift rectangle

sftrct accepts a rectangle and two integer arguments and returns the rectangle formed by offsettting its origin by the integer arguments.

Usage - sftrct(x,y,r) where x and y are the offset integers and r is the reference rectangle.

scolor - set color

scolor accepts a color, a point and a form argument and returns the form with its point argument set to the color argument.

Usage - scolor(p,c,f) where p is the point, c is the color, and f is the font.

invfrm - inverse form

invfrm accepts a form and two color arguments and returns the form with its fore and background colors inversed by the color arguments.

Usage - invfrm(fg,bg,frm) where fg is the new foreground color, bg is the new background color, and frm is the form to be inversed.

sfnt - set font

sfnt accepts a natural, a form, and a font and returns the font with the new icon inserted that is defined by the form and natural arguments.

Usage - sfnt(frm,n,fnt) where frm is the icon form, n is the index, and fnt is the font.

mod - modify list

mod accepts a dblist, an objlist, and a pvallist and returns the dblist with the new pvallist inserted into the appropriate position of the objlist identified by the corresponding 'pid.' It is an error to apply a pvallist to an object for which it is not defined.

14.3.4. Quadadic Operators (QOP's)

foffst - font offset

foffst accepts two integer arguments as an offset, a point argument and a font argument. It returns the spot origin point based on the spot coordinate offset from the point argument, e.g., a font size of 10 by 10 which is offset 2,3 from point (5,5) returns the spot origin point at (25,35).

Usage - foffst(x,y,fnt,p) where x and y are the offset integers, fnt is the basis font, and, p is the reference point.

cpfrm - form copyblt

cpfrm merges a source and a mask form with a destination form using the parameters in ptblt. It accepts a ptblt and three form arguments and returns the resultant form.

Usage - cpfrm(pb,s,m,d) where pb is the governing ptblt, s is the source form, m is the mask form, and d is the destination form.

14.3.5. Sexadic Operators (SOP's)

drawln - draw line

drawln draws a line from point y to point z on the destination form, using the specified brush and mask forms. It accepts two point arguments, three form arguments and a ptblt argument and returns a form.

Usage - drawln(x,y,pb,b,m,d) where y is the start point, z is the end point, pb is the ptblt, b is the brush form, m is the mask form, and d is the destination form.

cpfnt - copy font

cpfnt copies a font icon to a designated point on the destination form. It accepts a natural and a font argument which defines the source form, a point argument for the target location, two form arguments and a ptblt argument and returns the resultant form.

Usage - cpfnt(p,pb,n,fnt,m,d) where p is the target location, pb is the ptblt, n is the font index; fnt is the font, m is the mask form, and d is the destination form.

14.3.6. Octadic Operators (OOP's)

invfnt - inverse font

invfnt performs the same operation as cpfnt except that the font icon is combined with inverse coloring. It accepts the same arguments plus two color arguments and returns the resultant form.

Usage - invfnt(fg,bg,p,pb,n,fnt,m,d) where fg is the new foreground color, bg is the new background color, p is the target location, pb is the ptblt, n is the font index, fnt is the font, m is the mask form, and d is the destination form.

14.3.7. Relational Operators (ROP's)

The relational operators are:

== - equality
> - greater than

```
>=  - greater than or equal to
<   - less than
<=  - less than or equal to
!=  - not equal to
```

They may be applied to int, nat, char, string, intens, pnt, and lst.

If == or != are applied to arguments of different types, == returns false, != returns true. This applies also to types not listed above. >,>=,< and <= return an error if their arguments are not of the same type.

Relational operators return a boolean result.

14.3.8. Test Operators (BOP's)

These operators permit the programmer to test a cell for type before attempting to access it. These are necessary because AM considers it a fatal error to read from an undefined cell or apply an operator of one type on data of another. The test operators are the same as the type mnemonics, plus a mnemonic for testing undefined values:

```
bool
nat
int
char
string
intens
color
pnt
rct
form
font
ptblt
instr
addr
file
undef
pid
val
pval
obj
db
lst
```

Test operators accept a typed value and return true if the value is of the specified type, false otherwise. undef returns true if a value is undefined, false otherwise.

Syntax:

offset I,R

R must contain a memory address value

Operation:

R + I --> R

Description:

The sum of I and the address in R is stored in R.

Example:

```
offset  20,r(0:0)
```

Addressing Modes:

I: a

R: r

Format:

| 0250 | P | 3810 | operands |

Syntax:

```
link R,N
```

Operation:

```
R@ --> address@
address --> R
```

Description:

A segment of N cells is allocated from the heap. The value stored in R is save at the base address of the segment. The segment base address is returned in R.

This instruction is designed to create dynamic links for local environments.

Example:

```
proc:     link   r(0:5),1
          move   r(0:5)2@4,r(0:0)
          add    {int,100},r(0:0)
          move   r(0:0),r(0:5)2@4
          unlink r(0:5)
          rts
```

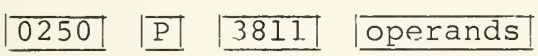
Above is an example of uplevel addressing.

Addressing Modes:

R: r

N: a

Format:



Syntax:

```
unlink R
```

Operation:

```
R@ --> R
```

Description:

The value in the base address of the segment pointed to by R is returned in R. The segment is freed.

Example:

```
proc:   link   r(0:5),1
        move   r(0:5)2@4,r(0:0)
        add    {int,100},r(0:0)
        move   r(0:0),r(0:5)2@4
        unlink r(0:5)
        rts
```

Addressing Modes:

R: r

Format:

0250	P	2812	operand

Syntax:

gdwin D,R

Operation:

D --> R

Description:

The value of the display window origin point at D is stored in R.

Example:

```
gdwin d(0:0),r(0:0)
```

Addressing Modes:

R: r

Format:

0250	P	3813	operands

Syntax:

sdwin R,D

R must contain a point value

Operation:

R --> D

Description:

The display window origin point at D is set to the point value in R.

Example:

```
sdwin r(0:0),d(0:0)
```

Addressing Modes:

R: r

Format:

| 0250 | P | 3814 | operands |

Syntax:

gmtr T,R

Operation:

T --> R

Description:

The T value is stored in R.

Example:

```
gmtr  -b,r(0:0)
```

Addressing Modes:

R: r

Format:

|0250| |P| { |2815| ... |281C| } |operand|

Syntax:

smtr R,T

R must contain a value appropriate for the selected attribute.

Operation:

R --> T

Description:

The T value is set to the value in R

Example:

```
smtr  r(0:0),-d
```

Addressing Modes:

R: r

Format:

Syntax:

<mop> C

where:
<mop> is a monadic operator

Operation:

<mop> C --> C

Description:

The operator corresponding to mop is applied to C and the result stored in C.

Example:

```
not   r(0:0)
```

Addressing Modes:

C: r,d

Format:

| 0250 | P | 3830 | operand |

Syntax:

<mop> Cx,Cy

where:
<mop> is a monadic operator

Operation:

<mop> Cx --> Cy

Description:

The operator corresponding to <mop> is applied to Cx and the result stored in Cy.

Example:

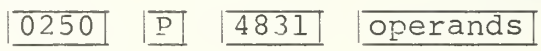
```
not    r(0:0),r(1:0)
farea  d(0:0),r(0:0)
```

Addressing Modes:

Cx: r,d

Cy: r,d

Format:



Syntax:

<mop> V,C

where:
<mop> is a monadic operator

Operation:

<mop> V --> C

Description:

The operator corresponding to <mop> is applied to the immediate value V and the result stored in C.

Example:

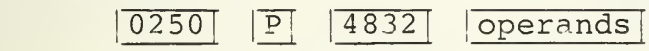
```
not       {addr,flag},r(1:0)
newfrm    {addr,rctsize},d(0:0)
```

Addressing Modes:

V: i

C: r,d

Format:



Syntax:

<dop> Cx,Cy

where:
<dop> is a dyadic operator

Operation:

Cy <dop> Cs --> Cy

Description:

The operation corresponding to <dop> is applied to the operands and the result stored in Cy.

Example:

```
and   r(0:0),r(0:1)
fill  r(0:0),d(0:0)
```

Addressing Modes:

Cx: r,d

Cy: r,d

Format:

| 0250 | P | 4833 | operands |

Syntax:

<dop> V,C

where:
<dop> is a dyadic operator

Operation:

C <dop> V --> C

Description:

The operation corresponding to <dop> is applied to the operands and the result stored in C.

Example:

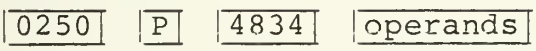
```
sub   {int,100},r(0:1)
fill  {color,&(10,10,10)},d(0:0)
```

Addressing Modes:

V: i

C: r,d

Format:



Syntax:

<dop> Cx,Cy,Cz

where:
<dop> is a dyadic operator

Operation:

Cy <dop> Cx --> Cz

Description:

The operation corresponding to <dop> is applied to Cx and Cy and the result stored in Cz.

Example:

```
add    r(0:0),r(0:1),r(0:3)
gcolor r(0:0),d(0:0),r(0:1)
```

<dop> Cx,Cy,Cy is equivalent to <dop> Cx,Cy

Addressing Modes:

Cx: r,d

Cy: r,d

Cz: r,d

Format:

| 0250 | P | 4835 | operands |

Syntax:

<dop> V,Cx,Cy

where:
<dop> is a dyadic operator

Operation:

Cx <dop> V --> Cy

Description:

The operation corresponding to <dop> is applied to V and Cx and the result stored in Cy.

Example:

```
add    {int,100},r(0:0),r(0:1)
gcolor {pnt,(0,0)},d(0:0),r(0:0)
```

<dop> V,Cx,Cx is equivalent to <dop> V,Cx

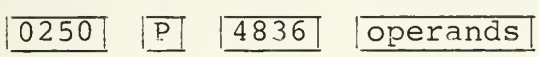
Addressing Modes:

V: i

Cx: r,d

Cy: r,d

Format:



Syntax:

<top> Cx,Cy,Cz

where:
<top> is a triadic operator

Operation:

<top> Cx,Cy,Cz --> Cz

Description:

The operation corresponding to <top> is applied to the operands and the result stored in Cz.

Example:

```
sfnt   r(0:0),r(0:1),r(0:2)
```

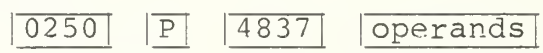
Addressing Modes:

Cx: r,d

Cy: r,d

Cz: r,d

Format:



Syntax:

<top> Cw,Cx,Cy,Cz

where:
<top> is a triadic operator

Operation:

<top> Cw,Cx,Cy --> Cz

Description:

The operation corresponding to <top> is applied to the operands and the result stored in Cz.

Example:

```
scolor r(0:0),r(0:1),d(0:2),r(0:3)
```

Addressing Modes:

Cw: r,d

Cx: r,d

Cy: r,d

Cz: r,d

Format:

| 0250 | P | 4838 | operands |

Syntax:

<qop> Cw,Cx,Cy,Cz

where:
<qop> is a quadadic operator

Operation:

<qop> Cw,Cx,Cy,Cz --> Cz

Description:

The operation corresponding to <qop> is applied to the operands and the result stored in Cz.

Example:

```
cpfrm r(0:0),d(0:0),r(0:1),d(0:1)
```

Addressing Modes:

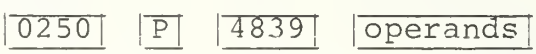
Cw: r,d

Cx: r,d

Cy: r,d

Cz: r,d

Format:



Syntax:

<qop> Cv,Cw,Cx,Cy,Cz

where:
<qop> is a quadadic operator

Operation:

<qop> Cv,Cw,Cx,Cy --> Cz

Description:

The operation corresponding to <qop> is applied to the operands and the result stored in Cz.

Example:

```
cpfrm r(0:0),r(0:1),r(0:2),r(0:3),d(0:0)
```

Addressing Modes:

Cv: r,d

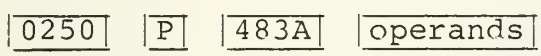
Cw: r,d

Cx: r,d

Cy: r,d

Cz: r,d

Format:



Syntax:

<sop> Cu,Cv,Cw,Cx,Cy,Cz

where:
<sop> is a sexadic operator

Operation:

<sop> Cu,Cv,Cw,Cx,Cy,Cz --> Cz

Description:

The operation corresponding to <sop> is applied to the operands and the result stored in Cz.

Example:

```
drawln    r(0:0),r(0:1),r(0:2),r(0:3),r(0:4),d(0:0)
```

Addressing Modes:

Cu: r,d

Cv: r,d

Cw: r,d

Cx: r,d

Cy: r,d

Cz: r,d

Format:

| 0250 | P | 483B | operands |

Syntax:

<sop> Ct,Cu,Cv,Cw,Cx,Cy,Cz

where:
<sop> is a sexadic operator

Operation:

<sop> Ct,Cu,Cv,Cw,Cx,Cy --> Cz

Description:

The operation corresponding to <sop> is applied to the operands and the result stored in Cz.

Example:

```
cpfnt  r(0:0),r(0:1),r(0:2),r(0:3),r(0:4),
       d(0:0),d(0:1)
```

Addressing Modes:

Ct: r,d

Cu: r,d

Cv: r,d

Cw: r,d

Cx: r,d

Cy: r,d

Cz: r,d

Format:

| 0250 | P | 483C | operands |

Syntax:

<oop> Cs,Ct,Cu,Cv,Cw,Cx,Cy,Cz

where:
<oop> is a octadic operator

Operation:

<oop> Cs,Ct,Cu,Cv,Cw,Cx,Cy,Cz --> Cz

Description:

The operation corresponding to <oop> is applied to the operands and the result stored in Cz.

Example:

```
invfnt r(0:0),r(0:1),r(0:2),r(0:3),r(0:4),r(0:5),
       r(0:6),d(0:1)
```

Addressing Modes:

Cs: r,d

Ct: r,d

Cu: r,d

Cv: r,d

Cw: r,d

Cx: r,d

Cy: r,d

Cz: r,d

Format:

| 0250 | P | 483D | operands |

Syntax:

<oop> Cr,Cs,Ct,Cu,Cv,Cw,Cx,Cy,Cz

where:
<oop> is a octadic operator

Operation:

<oop> Cr,Cs,Ct,Cu,Cv,Cw,Cx,Cy --> Cz

Description:

The operation corresponding to <oop> is applied to the operands and the result stored in Cz.

Example:

```
        invfnt  r(0:0),r(0:1),r(0:2),r(0:3),r(0:4),r(0:5),
                r(0:6),d(0:0),d(0:1)
```

Addressing Modes:

Cr: r,d

Cs: r,d

Ct: r,d

Cu: r,d

Cv: r,d

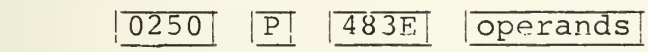
Cw: r,d

Cx: r,d

Cy: r,d

Cz: r,d

Format:



Syntax:

move <ea1>,<ea2>

where:
<ea> must be one of the addressing modes listed below

Operation:

source --> destination

Description:

The value found at the source address is copied into the destination address.

Example:

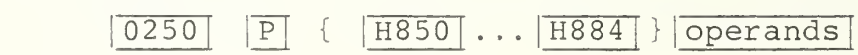
```
            move r(0:0),d(0:0)
            move d(0:1),r(0:4)
            move r(0:0),data
            move {addr,data},r(0:20)
            move {int,100},r(0:20)@
            move r(0:20)@10,r(0:10)

data:       ds 100
```

Addressing Modes:

<ea1>: d,r,ri,rid,ridn,m,pcr,i

<ea2>: d,r,ri,rid,ridn,m,pcr

Format:



Syntax:

push <ea>,S

where:
<ea> is one of the addressing modes listed below

Operation:

source --> S

Description:

The source value is pushed onto stack S. The programmer has no access to the stack pointer.

Example:

```
push {int,100},s(0)
push r(0:10),s(1)
push d(0:0),s(1)
```

Addressing Modes:

<ea>: d,m,pcr,r,ri,rid,ridn,i

S: s

Format:

Syntax:

pop S,<ea>

where:
<ea> is one of the addressing modes listed below

Operation:

S --> destination

Description:

The source value is popped off stack S and stored at <ea>. The programmer has no access to the stack pointer.

It is an error to attempt to pop a value from an empty stack.

Example:

```
            pop s(0),r(0:1)
            pop s(0),data
            pop s(1),d(0:0)

data:       ds 1
```

Addressing Modes:

S: s

<ea>: d,m,pcr,r,ri,rid,ridn

Format:

|0250| |P| { |H889| ... |H88f| } |operands|

Syntax:

popx S

Operation:

S -->

Description:

The top value of stack S is removed.

It is an error to attempt to remove the top of an empty stack.

Example:

popx s(0)

Addressing Modes:

S, s

Format:

| 0250 | P | 2888 | operands |

Syntax:

nop

Operation:

-

Description:

Does nothing.

Addressing Modes:

-

Format:

0250	P	18A0

Syntax:

stop

Operation:

-

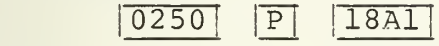
Description:

Execution is terminated.

Addressing Modes:

-

Format:



Syntax:

jmp <ea>

where
<ea> is one of the addressing modes listed below

Operation:

<ea> --> PC

Description:

Execution resumes at <ea>.

If jmp follows a rorg directive, a jump to memory absolute is converted to a branch.

Example:

```
            jmp here
            jmp r(0:0)
here:       jmp (1:150)@
```

Addressing Modes:

<ea>: m,r,mi,pcr

Format:

|0250| |P| {|H8A2|...|H85A4|}|operands|

Syntax:

bra <ev>

where:
<ev> is one of the addressing modes listed below

Operation:

PC + <ev> --> PC

Description:

Execution resumes at the sum of the program counter and the effective value.

Example:

```
bra 100
```

Addressing Modes:

<ev>: a,r

Format:

|0250| |P| |H8A5|...|H8A6| } |operands|

Syntax:

```
if R <rop> <ev>,M
if <bop> <ea>,M
```

where:
<rop> is a relational operator
<bop> is a test operator
<ea> and <ev> are one of the addressing modes listed below

Operation:

```
if R <rop> <ev> then
       M --> PC

if <bop> <ea> then
       M --> PC
```

Description:

If the comparison is true, execution resumes at M; otherwise,with the next instruction.

Example:

```
            move {int,10},r(0:0)
loop:       if   r(0:0) < {int,1},done
            sub  {int,1},r(0:0)
            jmp  loop
done:       if   int data,loop

data        ds   1
```

Addressing Modes:

R: r

<ev>: r,i

<ea>: r,m

M: m,pcr

Format:

|0250| |P|

{ |58A7|,|58A8|,|58AB|,|58AC|,|48AF|,|48B0|,|48B3|,|48B4| }

|operands|

IFTE If-Then-Else: Conditional Jump/Branch IFTE

Syntax:

```
if R <rop> <ev>,Mx,My
if <bop> <ea>,Mx,My
```

where:
<rop> is a relational operator
<bop> is a test operator
<ea> and <ev> are one of the addressing modes listed below

Operation:

```
if R <rop> <ev> then
        Mx --> PC

else

        My --> PC

if <bop> <ea> then
        Mx --> PC
else
        My --> PC
```

Description:

If the comparison is true, execution resumes at Mx; otherwise, at My.

Example:

```
            if    r(0:0) > r(0:1),case1,case2
stuff:      move  r(0:0),data
case1:      jsr   first,s(0)
            if    int r(0:0),case1
            stop
case2:      jsr   second,s(0)
            stop
```

Addressing Modes:

R: r

<ev>: r,i

<ea>: r,m

Mx: m,pcr

My: m,pcr

Format:

```
|0250| |P|
{ |68A9|,|68AA|,|68AD|,|68AE|,|58B1|,|58B2|,|58B5|,|58B6| }
|operands|
```

Syntax:

jsr <ea>,S

where:
<ea> is one of the addressing modes listed below

Operation:

PC --> S
<ea> --> PC

Description:

The program counter is pushed onto stack S, and execution resumes at <ea>.

Following a rorg directive, memory absolute is converted automatically to program counter relative.

Example:

```
        jsr   incr,s(0)
```

Addressing Modes:

<ea>: m,mi,r,pcr,S: s

Format:

|0250| |P| { |H8B7| ... |H8B9| } |operands|

Syntax:

bsr <ev>,S

where:
<ev> is one of the addressing modes listed below

Operation:

PC --> S
PC + <ev> --> PC

Description:

The program counter is pushed onto stack S, and execution resumes at the sum of the program counter and <ev>.

Example:

```
bsr   r(1:0),s(0)
```

Addressing Modes:

<ev>: r,a S: s

Format:

Syntax:

```
rts S
```

Operation:

```
S --> PC
```

Description:

Execution resumes at the address popped from stack S.

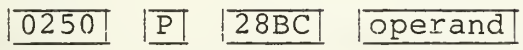
Example:

```
incre:      add   {int,1},r(0:0)
            rts   s(0)
```

Addressing Modes:

```
S: s
```

Format:



Syntax:

```
open S
```

Operation:

```
S -->
```

Description:

To open a file, four file parameters must be pushed on the stack, in proper order, before the open instruction is invoked. These attributes are: a string atom for the filename, a file descriptor atom, an integer atom for the access mode, and an integer atom for the data type (raw or AM typed values). The open instruction pops these parameters off the stack and opens the file. All future file operations are referenced by the file descriptor.

Example:

```
datafile    equ  {file,3}
            push {string,"filename"},s(0)
            push {datafile},s(0)
            push {int,0},s(0)
            push {int,0},s(0)
            open s(0)
```

Addressing Modes:

```
S: s
```

Format:

```
|0250| |P| |28C0| |operand|
```

Syntax:

close S

Operation:

S -->

Description:

The file descriptor atom must first be pushed on the stack. The close instruction pops the stack and closes the file.

Example:

```
datafile    equ   {file,3}
            push  {datafile},s(0)
            close s(0)
```

Addressing Modes:

S: s

Format:

| 0250 | P | 28C1 | operand |

Syntax:

read S

Operation:

S -->

Description:

The file descriptor atom must first be pushed on the stack. The memory address atom for the destination buffer cell is pushed next. The read instruction pops these parameters off the stack and puts the next file cell in the destination buffer.

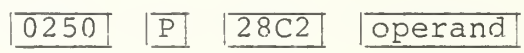
Example:

```
datafile    equ   {file,3}
            push  {datafile},s(0)
            push  {addr,data},s(0)
            read  s(0)
data        ds    100
```

Addressing Modes:

S: s

Format:



Syntax:

write S

Operation:

S -->

Description:

The file descriptor atom must first be pushed on the stack. The memory address atom for the source buffer cell is pushed next. The write instruction pops these parameters off the stack and puts the contents of the source buffer cell into the next file cell.

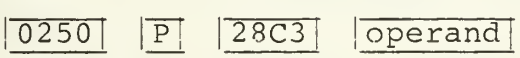
Example:

```
datafile    equ   {file,3}
            push  {datafile},s(0)
            push  {addr,data},s(0)
            write s(0)
data        dc    {string,"hello world"}
```

Addressing Modes:

S: s

Format:



Syntax:

write<ea>,Q

where:
<ea> is one of the addressing modes listed below

Operation:

Source --> Q

Description:

The source value is written onto queue Q. The programmer has no access to the queue pointer.

Example:

```
write {nat.17},q(0)
write r(0:10), q(1)
```

Addressing Modes:

<ea> : m,r,i,q

Q: q

Format:

| 0250 | | P | { | H890 | ... | H897 | } | operands |

Syntax:

read Q,<ea>

where
<ea> is one of the addressing modes listed below

Operation:

Q --> destination

Description:

The source value is read from the queue Q and stored at <ea>. The programmer has no access to the queue pointer. It is an error to attempt to read a value from an empty queue.

Example:

```
            read  q(0),r(0:1)
            read  q(0),data
data:        dsl
```

Addressing Modes:

Q: q

<ea> : m,r

Format:

|0250| |P| { |H899| ... |H89f| } |operands|

DELETE Delete the Front Value of the Queue DELETE

Syntax:

delete Q

Operation:

Q -->

Description:

The front value of queue Q is removed. It is an error to attempt to remove the front value of an empty queue.

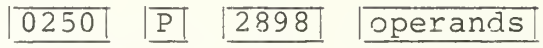
Example:

```
delete  q(0)
```

Addressing Modes:

Q: q

Format:



Syntax:

open D

Operation:

D -->

Description:

A database identifier is required to open the databse. All future operations are referenced by this identifier.

Example:

open {database,databaseid}

Addressing Modes:

D: i

Format:

| 0250 | P | 28C4 | operand |

Syntax:

close D

Operation:

D -->

Description:

The database identifier is required to close the database.

Example:

close{database,databaseid}

Addressing Modes:

D: i

Format:

| 0250 | P | 28C5 | operand |

LIST OF REFERENCES

1. Davis, D., "Research on Portability and Reusability," in Bits 'n' Bytes, Vol. 3, No. 1, Naval Postgraduate School, Monterey, California, 1985, pp. 13-15.

2. Yurchak, J., The Formal Specification of an Abstract Machine: Design and Implementation, Master's Thesis, Naval Postgraduate School, Monterey, California, December 1984.

3. Hunter, J.E., The Formal Specification of a Visual Display Device: Design and Implementation, Master's Thesis, Naval Postgraduate School, Monterey, California, June 1985.

4. Myers, G.J., Advances in Computer Architecture, Wiley, New York, 1982.

5. Guttag, J.V., "Notes on Type Abstraction," IEEE Transactions on Software Engineering, January 1980.

6. Goguen, J.A., J.W. Thatcher, E.G. Wagner, "An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types," Current Trends in Programming Methodology IV, Data Structuring, R.T. Yeh (ed.), Prentice-Hall, Englewood Cliffs, New Jersey, 1978, pp. 80-97.

7. Naval Postgraduate School, NPS52-84-002, A Formal Method for Specifying Computer Resources in an Implementation Independent Manner, by D. Davis, Monterey, California, November 1984.

8. Clocksin, W.F., Mellish, C.S., Programming in Prolog, Springer-Verlag, 1984.

9. MacLennan, B.J., Functional Programming Methodology, Naval Postgraduate School, Monterey, California, 1985.

10. Fairley, R., Software Engineering Concepts, McGraw-Hill Book Company, 1985.

11. Bjoerner, D., "Formalization of Data Base Models," in Bjoerner, D. (ed.): Abstract Software Specification, Lecture Notes in Computer Science, No. 86, Springer Verlag, 1980.

12. Booch, G., Software Engineering with Ada, The Benjamin/ Cummings Publishing Co., Inc., 1983.

13. Deen, S.M., Fundamentals of Database Systems, Hayden Book Co., 1977, pp. 36-52, 80-88.

14. Hsiao, D.K., Database Computers--A Tutorial and Review, Naval Postgraduate School, Monterey, California, April 1982.

15. Kroenke, D., Database Processing: Fundamentals, Design, Implementation, Science Research Associates, Inc., 1983.

16. Hull, R., Yap, C.K., "The Formal Model: A Theory of Database Organization," in Journal of the Association for Computing Machinery, Vol. 31, No. 3, July 1984, pp. 518-536.

17. Smith, J.M., Smith, D.C.P., "Database Abstractions: Aggregation and Generalization," in ACM Transactions on Database Systems, Vol. 2, No. 2, June 1977, pp. 105-133.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93943-5100	2
3.	Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93943-5100	1
4.	Computer Technology Programs, Code 37 Naval Postgraduate School Monterey, California 93943-5100	1
5.	Daniel L. Davis, Code 52Vv Department of Computer Science Naval Postgraduate School Monterey, California 93943-5100	5
6.	David K. Hsiao, Code 52Hq Department of Computer Science Naval Postgraduate School Monterey, California 93943-5100	1
7.	Kommando Marinefuehrungssysteme Heppenser Groden 2940 Wilhelmshaven W. Germany	1
8.	Harald Zang Karl-Tuerk-Str. 20 8630 Coburg W. Germany	5

215837

Thesis
Z2421 Zang
c.1 The formal specification of an abstract database: design and implementation.

14 AUG 87 33480

215837

Thesis
Z2421 Zang
c.1 The formal specification of an abstract database: design and implementation.

thesZ2421
The formal specification of an abstract
3 2768 000 69023 4
DUDLEY KNOX LIBRARY