BEL00050
L A BELADY
IBM T J WATSON RESEARCH CENTER
PO BOX 218
YORKTOWN HEIGHTS NY 10598 USA

~~[illegible]~~

[illegible]

DATA BASE INSTANCES, ABSTRACT DATA TYPES AND DATA BASE SPECIFICATION*

by

T.S.E. Maibaum

RESEARCH REPORT CS-80-14

University of Waterloo
Computer Science Dept.
Waterloo, Ontario
CANADA N2L 3G1

March 1980

*This research was partially supported by a grant from the Natural Sciences and Engineering Research Council of Canada.

ABSTRACT

We present in this report a method of data base specification which overcomes a severe drawback of most data base models and specification techniques - namely the inability to formalise the concept of data base instance (state). The lack of formalisation of this concept in a model results in the inability to present a uniform treatment of queries and updates. We indicate why previous models and specification techniques are inadequate in this respect and then develop a modelling technique to overcome this situation. The use of algebraic specification tools in conjunction with this technique is illustrated in the specification of an example data base.

Keywords: Data base models, algebraic specification, abstract data types, data base instance.

1. Introduction

There have been in recent years numerous attempts at using the algebraic theory of abstract data types to specify data bases (as, for example, in [15, 9,39,30,31]). The procedure adopted is to choose one of the standard models of data bases (relational, hierarchical, network, etc.) and formalise this model as an abstract data type. That this can be done in principle is not in doubt here. We would just point out that none of these specifications introduces a new modelling concept for databases. We intend in this report to introduce a new modelling concept for data bases which arose from the study of abstract data types.

Central to any model of data bases is the concept of data base instance. A data base instance may be described as a set of data values (structured in some well specified way) at some particular point in time. The fact that a data base may contain different sets of values at different times is of the essence in the concept of data base. It is a sad fact that none of the conventional models of data bases contends in a straightforward manner with the formalisation of data base instance.

For example, consider the case of the relational model ([12,13,14]). Discussion of this model is divided into two quite distinct areas. Firstly, there is the theory of relational algebras (and the related concept of relational calculus) as described for example, in [13,18,39,6].
This deals with the theory of (all possible) relations (defined on some possibly infinite set of attributes) and the operations defined on them (such as join, division, restriction as well as the more conventional operations of union, intersection, difference, etc.). The special set of relations defining a particular data base plays no role in this theory. Secondly, we have the theory of updates (through the concepts of functional

dependencies, multi-valued dependencies, normal forms, etc.) as described, for example, in [7,17,28,41]. This deals only with the relations which constitute the data base and ignores all other relations and how these relate to the special data base relations. Thus the theory of queries (on a particular data base instance) is unrelated to the theory of instances. That this is a serious shortcoming becomes even more evident when the algebraic specification method is applied to formally axiomatise any such model. For example, the axiomatisations outlined in [7, 28] ignore completely the fact that some relations are special - namely, the data base relations. All relations are treated in the same way. Thus it is hard to describe in the object specified a consistent theory of instances. (This is because many different expressions in the algebra of relations defined by the axioms can denote "the same" database instance. However, most of these are based on relations which are not the data base relations. Thus one would have to eliminate these "misleading" expressions from the algebra - something which one is not allowed to do in the theory of data types.)

A somewhat different use of algebraic specification techniques is made in [30,31] where the need to be able to specify data base instances is recognised. This is done by a so-called "extension" of the algebraic theory of data types. In order to understand the shortcomings of this attempt, we must delve a bit more deeply into the concepts underlying the (algebraic) theory of abstract data types. This theory was formalised in [2] although various precursors ([44,29,22]) pointed in the appropriate direction.

A data type is a set of operations and tests (regarded as boolean valued operations) together with a family of sets of different kinds of data over which the operations are defined. That is, a data type is a many-sorted algebra. A data type is said to be <u>abstract</u> if the operations, tests and sets of data can be specified implicitly - without any reference to any representation. One such implicit method is the use of equational axioms to specify the properties of the operations of the type. This method is called the algebraic specification method. However, the specification defines a whole class of algebras having these properties some of which are trivial and obviously not intended to be the representations of the type while for other algebras, the problem of deciding whether it is a representation becomes more difficult.

In [2], the so-called ADJ group suggested that a unique isomorphism class of algebras can be defined to act as representations of the type. This was done using the concept of initiality - an algebra is initial in a class of algebras if it is in the class and if for every algebra in the class there is a unique structure preserving mapping (homomorphism) from the initial algebra to the given algebra. It so happens that the class of algebras defined by a set of equational axioms always has an initial algebra. This initial algebra has the property that those and only those things are true of the algebra which are logical implications of the axioms. Thus the properties of the algebra are <u>completely</u> specified by the axioms. In addition to this nice property, initiality gives rise to many proof methods which are helpful in proving that purported implementations really are implementations. Among these proof methods are generator induction ([23]) and possible proofs of inequality.

Now, to get back to the proposals outlined in [30,31]. The authors feel that algebraic specification techniques have great potential for application to data base system design. However, from this observation they proceed to note a number of shortcomings of the method which they perceive and hope to overcome. To cope with the problem of instances, in [31] the suggestion seems to be that some sorts of a type not be specified and that different instances of the database can then be realised by "concretising" ([31]) the unspecified sorts in different ways. It is also noted in both [30] and [31] (presumably due to the inability to formally encode the concept of instance in the type) that the specification technique cannot overcome the following problems without "extension":

(i) preconditions on certain operations - some operations require certain conditions to be true in some previous instance(s);

(ii) coincident conditions on certain operations - some operations require conditions or actions on data objects (in the present instance) which are <u>not</u> arguments of the operation;

(iii) postconditions on certain operations - some operations require that they be immediately followed by other operations with arguments which may include some which are <u>not</u> arguments of the first.

To contend with these problems, these reports suggest that they can be overcome using extra conditions on implementations.

The objection to this proposal is that the concept of "concretisation" and the use of extra conditions on implementations seems to destroy completely the underlying principle in the now widely accepted theory of abstract data types - axioms define a unique isomorphism class of algebras, namely the initial algebras. "Concretisation" and extra implementation conditions ensure that the implementation is anything but the initial algebra in the class specified by the axioms. In fact, generally the implementation will define an algebra which is not even in the class specified by the axioms. Thus the mathematical characterisations, proof techniques, measures of what is and what is not an implementation suddenly disappear. As the authors note in [31], "... the axiomatic approach and its extensions in this paper, while intuitively appealing, still lack a complete theoretical foundation".

However, all this effort is based on a mistaken assumption - namely, the concept of instance cannot be encoded in the algebraic theory of data types as outlined in [2,22,26]. In the next section, we give a semi-formal description of a refutation of this negative assumption. In section 3 we outline the mathematical theory underlying the work on abstract data types and present an axiomatisation of a relational data base for keeping university records. In the final section we discuss the implications of this work and outline possible future directions for developing these ideas. Throughout the paper, we will use the relational model as the basis for our discussion.

2. Environments and Data Base Instances

It has been recognised for a long time in treatments of operational semantics of programming languages that the concept of environment is a very necessary part of the formalism. This is necessitated because of the different values which the program associates with program variables at different times during the execution of the program. For example, consider the following fragment of program:

```
        ⋮
x := A[i];   (1)
A[i]:= e;    (2)
y:= A[i];    (3)
        ⋮
```

A is some array and we assume that i is a valid index. Associated with A at the point (1) in the program is a function from the index set of the array to the values "stored" in the array. Thus the "query" A[i] returns some value which is then associated with x by (1). In (2), the value associated with A is changed by changing the value of the function associated with A to produce the value of e at index i. The "query" A[i] at (3) then returns the (possibly) different value to be associated with y. The mathematical explanation for this is simply that the function associated with A has an implicit second argument - the current environment. Thus, although the same "query" is being done at (1) and (3), the hidden or implicit argument - the current environment - has been changed by (2) (as well as (1) and (3)).

We note another important feature of the above programming example. The array A retains an identity throughout the program in spite of the fact that the value associated with A changes. This is because the <u>name</u> A is associated with these different values. This is to the point in our discussion of data bases since one of the important characteristics of any axiomatisation will have to be this property of associating different values with the same name - namely, that of each basic "relation" used to define the data base.

So, let us suppose that the data base is defined in terms of relations named $R_1,\ldots,R_n$ where R_i is defined in terms of attributes $A_{i,1},\ldots,A_{i,n_i}$ for $1 \leq i \leq n$. We will use the symbol σ (σ',σ_1,σ_2, etc.) to denote database instances. Now the relation associated with R_i depends on the instance σ in which R is to be evaluated. Thus we can formalise this by thinking of R_i as a function from instances to relations (sets of tuples). So the expression $R_i(\sigma)$ denotes the relation associated with R_i in the instance σ. Consider now some basic operation of the relational algebra, say projection. We want to evaluate, for example, project(R_i,S) (usually written R_i[S]). Clearly something is missing since we do not know what R_i is without specifying the instance in which it is to be evaluated. Thus project is really a function of three arguments: an expression denoting a relation, a set of attributes, and an instance in which the first argument is to be evaluated. Thus we should write project(R_i,S,σ). (In fact, we can retain the flavour of the previous notation by thinking of project (R_i,S) as denoting a function from instances to relations. Thus to evaluate project(R_i,S) in σ we would write project(R_i,S)(σ). There is a formal equivalence between these two "notations" for the same idea but we will not state it here.)

For join, we must write join(R_i,S,S',R_j,σ) to indicate the instance in which the join is to be performed. Similar changes must be made in the other operations. We note in passing that a user would not have to explicitly indicate the environment σ in expressions denoting relations (or in the update operations to be discussed shortly) just as in our programming example, the program does not contain explicit references to environments. It is only the formal model which has explicit references to instances.

Consider now some update operation such as add which is intended to add a new tuple to the specified basic relation. Thus we see add as a function of three arguments: a relation name, an expression denoting a tuple, and an instance. Thus, $add(R_i,t,\sigma)$, for example. The difference between this operation and previous operations, however, is that whereas above the operations denoted a relation, $add(R_i,t,\sigma)$ denotes a new instance. To recover the relation denoted by R_i we must write $R_i(add(R_i,t,\sigma))$ and, presumably, but not necessarily we would have $R_i(add(R_i,t,\sigma)) \neq R_i(\sigma)$. Before we proceed to a formalisation of these ideas, see the appendix for an example of an instance of our example data base (obtained from [21]).

3. Mathematical Preliminaries

Since our model is defined by means of the algebraic specification method, we take a "time out" to quickly summarise the underlying mathematical concepts. The reader is referred to [2,22,26] for details and motivation.

A data type is viewed as a many-sorted algebra. Discussion of data types as many-sorted algebras can be found in [2,22,26]. An algebra of one sort is roughly-speaking a set of objects and a family of operators on the set. The set is called the carrier of the algebra. Many-sorted algebras extend this notion by allowing the carrier of the algebra to consist of many disjoint sets. Each of these sets is said to have a sort. The operators are sorted or typed, but must be closed with respect to the carrier. For example, if A,B,C are three sets in the carrier of an algebra, then

$$+ : A \times B \rightarrow C$$

could be an operator of type $< ab,c >$, arity ab and sort c, where a,b and c are distinguished names (the sorts) of A, B and C respectively. A data type is then a many-sorted algebra, while a data structure is an element of the carrier of a data type.

Let S be a set whose elements are called sorts. An S-sorted operator domain Σ is a family $\{\Sigma_{\omega,s}\}$ of sets of symbols, for $s \in S$ and $\omega \in S^*$ where S^* is the free monoid on S. $\Sigma_{\omega,s}$ is the set of operator symbols of type $<\omega,s>$, arity ω and sort s.

A Σ-algebra A consists of a family $\{A_s\}_{s \in S}$ of sets called the carrier of A, and for each $<\omega,s> \in S^* \times S$ and each $f \in \Sigma_{\omega,s}$, a function

$$f_A: A_{s_1} \times A_{s_2} \times \ldots \times A_{s_n} \to A_s$$

(where $\omega = s_1 s_2 \ldots s_n$) called the operation of A named by f.

Here $\{x_s\}_{s \in S}$ denotes a family of objects x_s indexed by s, such that there is exactly one object x_s for each $s \in S$. The subscript $s \in S$ will be omitted when the index set S can be determined from the context. For $f \in \Sigma_{\lambda,s}$ where λ is the empty string, $f_A \in A_s$ (also written $f_A : \to A_s$). These operators are called constants of A of sort s.

Example 1: Let Σ be defined by: $S = \{i,s\}$ and $\Sigma_{\lambda,i} = \{0\}$; $\Sigma_{i,i} = \{succ\}$; $\Sigma_{\lambda,s} = \{\Lambda\}$; $\Sigma_{s,s} = \{pop\}$; $\Sigma_{s,i} = \{top\}$; $\Sigma_{is,s} = \{push\}$. (Thus Λ is a constant of sort (stack) (denoting what we will interpet as an empty stack).)

Let A_i be the set of natural numbers and $A_s = A_i^*$ (the set of finite strings on A_i). Then 0_A is the natural number zero, Λ_A is the empty string, $succ_A$ is successor, and pop_A, top_A, $push_A$ have the obvious interpretations (using one end of a string as the "top" of the stack).

□

If $\omega = s_1 s_2 \ldots s_n$, then let A^ω denote $A_{s_1} \times \ldots \times A_{s_n}$.

If A and A' are both Σ-algebras, then a Σ-homomorphism $h: A \to A'$ is a family of functions

$$\{h_s: A_s \to A'_s\}_{s \in S}$$

such that if $f \in \Sigma_{\omega,s}$ and $\langle a_1,\ldots,a_n\rangle \in A^\omega$ then

$$h_s(f_A(a_1,\ldots,a_n)) = f_{A'}(h_{s_1}(a_1),\ldots,h_{s_n}(a_n)).$$

A Σ-algebra A in a class $\underline{\underline{C}}$ of Σ-algebras is said to be initial in $\underline{\underline{C}}$ iff for every B in $\underline{\underline{C}}$ there exists a unique homomorphism

$$h: A \to B.$$

THEOREM 1 The class of all Σ-algebras has an initial algebra called T_Σ. $\square$ (T_Σ is sometimes called the free Σ-algebra.) This theorem, as well as the others given in this section, is proved in [2] . T_Σ can be viewed intuitively as the algebra of well formed expressions over Σ.

Variables can be included in terms in T_Σ in the following way. Let

$$X_s = \{ x_s^{(n)} | n \in N\}$$

where N is the set of natural numbers. The elements

$$x_s^{(i)} \in X_s$$

are symbols called variables of sort s. Suppose Σ is an S-sorted operator domain. Let

$$X = \{X_s\}_{s \in S}$$

where X_s is a family of variables of sort s for each $s \in S$. We say that X is an S-indexed family of variables.

Then let $\Sigma(X)$ be the S-sorted operator domain defined as follows:

$$\Sigma(X)_{\lambda,s} = \Sigma_{\lambda,s} \cup X_s$$

$$\Sigma(X)_{\omega,s} = \Sigma_{\omega,s} \quad \text{for} \quad \omega \neq \lambda.$$

Thus variables are being treated as nullaries or constants. Clearly $T_{\Sigma(X)}$ is an initial $\Sigma(X)$-algebra. We define $T_\Sigma(X)$ as the algebra with the same carrier as $T_{\Sigma(X)}$, but with operations Σ. We say that $T_\Sigma(X)$ is the Σ-algebra freely generated by X.

For any S-sorted Σ-algebra A and an S-indexed family X of variables, if

$$\theta: X \to A$$

denotes a family of functions

$$\{ \theta_s: X_s \to A_s \}_{s \in S}$$

then θ is called an interpretation or assignment of values of sort s in A to variables of sort s in X.

THEOREM 2 Let A be a Σ-algebra and $\theta: X \to A$ an assignment. Then there exists a unique homomorphism

$$\overline{\theta}: T_\Sigma(X) \to A$$

that extends θ in the sense that $\overline{\theta}_s(x) = \theta_s(x)$ for all $s \in S$ and $x \in X$. □

A Σ-equation is a pair $e = \langle L,R \rangle$ where $L,R \in T_{\Sigma(X),s}$ (the carrier of $T_\Sigma(X)$ of sort s). A Σ-algebra A satisfies e if

$$\overline{\theta}(L) = \overline{\theta}(R)$$

for all assignments $\theta: X \to A$. If ε is a set of Σ-equations, then A satisfies ε iff A satisfies each $e \in \varepsilon$. Thus a set of equations ε can be viewed as a set of axioms whose free variables are implicitly universally quantified. The class of Σ-algebras which satisfy ε is denoted $\underline{\underline{\mathrm{Alg}}}_{\Sigma,\varepsilon}$.

Example 2: Let Σ and S be as in example 1 and consider the following equations:

$$\mathrm{pop}(\mathrm{push}(n,st)) = st$$

$$\mathrm{top}(\mathrm{push}(n,st)) = n$$

Then A_Σ as defined in example 1 satisfies these equations.

□

An <u>equational specification</u> is a triple $< S,\Sigma,\varepsilon >$ where Σ is an S-sorted operator domain and ε is a set of Σ-equations (called the <u>type axioms</u>).

Let $w = s_1 s_2 \ldots s_n$ and $a_i, a_i' \in A_{s_i}$ for $1 \leq i \leq n$. Then a <u>Σ-congruence</u> $\equiv$ on a Σ-algebra A is a family $\{\equiv_s\}_{s \in S}$ of equivalence relations $\equiv_s$ on A_s such that if $f \in \Sigma_{\omega,s}$ and if $a_i \equiv_{s_i} a_i'$ for $1 \leq i \leq n$, then

$$f_A(a_1,\ldots,a_n) \equiv_s f_A(a_1',\ldots,a_n').$$

If A is a Σ-algebra and $\equiv$ is a Σ-congruence on A, let $A/\equiv\, = \{A_s/\equiv_s\}_{s \in S}$ be the set of $\equiv_s$-equivalence classes of A_s. For $a \in A_s$ let $[a]_s$ denote the $\equiv_s$-class containing a. It is possible to make $A/\equiv$ into a Σ-algebra by defining the operations $f_{A/\equiv}$ as follows:

(i) If $f \in \Sigma_{\lambda,s}$ then $f_{A/\equiv} = [f_A]_s$

(ii) If $f \in \Sigma_{s_1 \ldots s_n,s}$ and $[a_i]_{s_i} \in A_{s_i}/\equiv_{s_i}$ for $1 \leq i \leq n$

then

$$f_{A/\equiv}([a_1]_{s_i},\ldots,[a_n]_{s_i}) = [f_A(a_1,\ldots,a_n)]_s$$

Then it can be shown that $A/\equiv$ is a Σ-algebra called the <u>quotient</u> of A by $\equiv$. (The property of $\equiv$ being a congruence ensures that $\sigma_{A/\equiv}$ is well defined.)

A set of Σ-equations $\varepsilon = \{< L,R > \mid L,R \in T_\Sigma(X)\}$ generates a binary relation $R \subseteq A \times A$ on any algebra A. This relation is the set of all pairs $\{< \overline{\theta}(L),\overline{\theta}(R) > \mid \theta$ is an assignment$\}$. (Intuitively, consider all pairs with variables in L and R replaced by terms in T_Σ and the resulting expressions then being evaluated in A_Σ.) Now any relation R on algebra A generates a congruence q on A which is the least congruence on A containing R.

THEOREM 3 If ε is a set of Σ-equations generating a congruence q on T_Σ, then T_Σ/q is initial in $\underline{\underline{\text{Alg}}}_{\Sigma,\varepsilon}$.

□

Example 3: It can in fact be shown that A_Σ of example 1 is initial in $\underline{\underline{\text{Alg}}}_{\Sigma,\varepsilon}$ (where ε is defined in example 2) and so is isomorphic to T_Σ/q where q is the least congruence generated by ε.

□

In the data types, that we define in the sequel, we make a number of assumptions. Firstly, we assume that a sort bool is always a sort of the type. bool has constants TRUE and FALSE and the "propositional" operations $\neg$, $\vee$ and $\wedge$ appropriately axiomatised. Moreover, we assume that for each sort s of the type being defined we have an operation of type $\langle \underline{\text{bool}}_{ss,s}\rangle$ called if ... then ... else ... which takes three arguments (the first of sort bool and the other two of sort s). Associated with this operation are the axioms

$$\begin{aligned} &\underline{\text{if}}\ \text{TRUE}\ \underline{\text{then}}\ t\ \underline{\text{else}}\ t' = t \\ &\text{if}\ \text{FALSE}\ \underline{\text{then}}\ t\ \underline{\text{else}}\ t' = t'. \end{aligned} \qquad (I)$$

Secondly, we assume that each sort s of a type being defined contains a distinguished constant $\underline{\text{error}}_s$ (or error if the s is obvious from the context). This value will be used to indicate the result of an operation which would otherwise yield "undefined" or some other inappropriate value. For example, in the case of stacks if we try to pop the empty stack, we would like to return error. The introduction of this value into each sort causes us to condition the "normal" equations we write with the condition that none of the variables are error. We also add so-called error equations which state that if any argument of an operation is error, then the result is error. For a full discussion of the role of error values see [2,19]. For discussion of conditioned equations, see [2].

Example 4: Under the above assumptions we must add the sort bool to S of example 1 together with the following operations:

$$
\begin{aligned}
\text{error}_i &: \ \to i \\
\text{error}_s &: \ \to s \\
\text{error}_{\text{bool}} &: \ \to \underline{\text{bool}} \\
\underline{\text{if}}\ \underline{\text{then}}\ \underline{\text{else}} &: \ \underline{\text{bool}} \times a \times a \to a \text{ for } a \in S \\
\neg &: \ \underline{\text{bool}} \to \underline{\text{bool}} \\
\vee, \wedge &: \ \underline{\text{bool}} \times \underline{\text{bool}} \to \underline{\text{bool}} \\
\text{TRUE}, \text{FALSE} &: \ \to \underline{\text{bool}}.
\end{aligned}
$$

We also condition the equations in ε so that the arguments are not error values and add to ε conditioned versions of the following:

$$\neg \text{TRUE} = \text{FALSE} \qquad \neg\text{FALSE} = \text{TRUE}$$

$$b \wedge b' = b' \wedge b \qquad b \vee b' = b' \vee b$$

$$\text{TRUE} \wedge b' = b' \qquad \text{TRUE} \vee b = \text{TRUE}$$

$$\left.\begin{aligned} &\underline{\text{if}}\ \text{TRUE}\ \underline{\text{then}}\ v\ \underline{\text{else}}\ v' = v \\ &\underline{\text{if}}\ \text{FALSE}\ \underline{\text{then}}\ v\ \underline{\text{else}}\ v' = v' \end{aligned}\right\} \text{ for } v, v' \in a \text{ for each } a \text{ in } S.$$

We also add equations for each operation to indicate that if any argument is an error value, then the result is an error value.

To complete the axiomatisation, we should add the following axioms to handle "exceptional conditions" for stacks.

$$\text{pop}(\Lambda) = \text{error}_s$$

$$\text{top}(\Lambda) = \text{error}_i \ .$$

□

4. An Algebraic Model of Data Bases

We will now define our model of a relational data base. Relations will be "pairs" of a set of attributes and a set of tuples each of which is defined over the attributes forming the first argument. Thus we need to define the types *set of [elements]* (a so-called parameterised type which can be made into a normal type by substituting for *element* some appropriate type such as *attribute* or *tuple*), *attribute*, and *tuple*. The notation we use for defining types is illustrated by the definition of the type *set of [element]*.

```
type set of [element] with:
    sorts set of [element], element, bool;
    syntax
    EMPTY:   → set of [element]
   INSERT:  set of [element] × element → set of [element]
   DELETE:  set of [element] × element → set of [element]
     CONT:  element × set of [element] → bool
   SUBSET:  set of [element] × set of [element] → bool
    EQUIV:  set of [element] × set of [element] → bool
      ADD:  set of [element] × set of [element] → set of [element]
     SUBT:  set of [element] × set of [element] → set of [element]
      INT:  set of [element] × set of [element] → set of [element]
     CART:  set of [element] × set of [element] → set of [element]
    semantics with s, s':  set of [element], v, v':  element
    INSERT(INSERT(s,v),v') = INSERT(INSERT(s,v'),v)
    CONT(v,EMPTY) =  FALSE
    CONT(v,INSERT(s,v')) = if EQ(v,v') then TRUE
                                       else CONT(v,s)
```

```
SUBSET(EMPTY,s) = TRUE
SUBSET(INSERT(s',v),s) = if CONT(v,s) then SUBSET(s',s)
                                      else FALSE
EQUIV(s,s') = SUBSET(s,s') ∧ SUBSET(s',s)
ADD(s,EMPTY) = s
ADD(s,INSERT(s',v)) = ADD(INSERT(s,v),s')
SUBT(EMPTY,s) = EMPTY
SUBT(INSERT(s,v),s') = if CONT(v,s') then SUBT(s,s')
                                     else INSERT(SUBT(s,s'),v)
INT(s,s') = SUBT(s,SUBT(s,s'))
DELETE(EMPTY,v) = error
DELETE(INSERT(s,v),v') = if EQ(v,v') then s
                                     else INSERT(DELETE(s,v'),v)
```

(Here EQ is the equality defined on the sort element (presumably by some data type which is used to define element.)

type tuple with

sorts tuple, attribute, set of [attribute], set of [tuple], value_a for each a $\in$ {DEPT, COURSE#, TITLE, P-DEPT, P-COURSE#, SECTION#, INSTRUCTOR, INSTR-STATUS, DAY, TIME, BLDG, RM}, value.

(Here each value_a is the set of allowed values for the sort defined by a as obtained from some data type defining the set. Elements of value can then be thought of as value-attribute pairs. See the "hidden" operations defined below.)

syntax

```
       NEW:  set of [attribute] → tuple
     STORE:  tuple × attribute × value → tuple
   COLUMNS:  tuple → set of [attribute]
      READ:  tuple × attribute → value
     PIECE:  tuple × set of [attribute] → tuple
  CATENATE:  tuple × tuple → tuple
   COMPOSE:  set of [tuple] × tuple → set of [tuple]
     MATCH:  tuple × tuple × set of [attribute] → bool
```

hidden (These are operations which are used in the axiomatisation but which are not allowed to be "used" in manipulating the objects of the type.)

```
       EQ:  value × value → bool
```
IN_a: $\text{value}_a \to \text{value}$

ATT: value → attribute

VAL: value → value_a for each a

semantics with t,t': tuple; a,a': attribute; A,A': set of [attribute];

s,s': set of [tuple]; v,v': value; v_a: value_a for each a

```
COLUMNS(NEW(A)) = A

COLUMNS(STORE(t,a,v)) = if CONT(a,A) ∧ ATT(v) ≡ a
                        then COLUMNS(t)
                        else error

STORE(STORE(t,a,v),a',v') =
    if CONT(a,COLUMNS(t)) ∧ CONT(a',COLUMNS(t)) ∧ ATT(v) ≡ a ∧ ATT(v') ≡ a'
    then if a' < a then STORE(STORE(t,a,v),a',v')
         else if a < a' then STORE(STORE(t,a',v'),a,v)
         else STORE(t,a',v')
    else error
```

```
READ(NEW(A),a) = error
READ(STORE(t,a',v),a) =
     if a ≡ a'
     then if ATT(v) ≡ a' then v else error
     else READ(t,a)
PIECE(NEW(A),A') = if SUBSET(A',A) then NEW(A') else error
PIECE(STORE(t,a,v),A) =
     if CONT(a,A)
     then STORE(PIECE(t,A),a,v)
     else PIECE(t,A)
CATENATE(NEW(A),NEW(A')) =
     if EQUIV(INT(A,A'),EMPTY) then NEW(ADD(A,A'))else error
CATENATE(NEW(A), STORE(t,a,v)) = STORE(CATENATE(NEW(A),t),a,v)
CATENATE(STORE(t,a,v), STORE(t',a',v')) =
     if a < a' then STORE(CATENATE(t,STORE(t',a',v')),a,v)
               else STORE(CATENATE(STORE(t,a,v),t'),a',v')
COMPOSE(EMPTY,t) = EMPTY
COMPOSE(INSERT(s,t),t') = INSERT(COMPOSE(s,t),CATENATE(t,t'))
MATCH(t,t',EMPTY) = TRUE
MATCH(t,t',INSERT(A,a) =
     if SUBSET(INSERT(A,a),COLUMNS(t)) ∧ SUBSET(INSERT(A,a),COLUMNS(t'))
     then if READ(t,a) EQ READ(t',a)
          then MATCH(t,t',A)
          else FALSE
     else error
vEQv' = if ATT(v) ≡ ATT(v') ∧ VAL(v) $=_{ATT(v)}$ VAL(v')
        then TRUE
        else FALSE
```

(Here $=_a$ is the equality defined on values of attribute a for each a used in the example.)

$ATT(IN_a(v_a)) = a$ for each a

$VAL(IN_a(v_a)) = v_a$ for each a

type attribute with

sorts attribute;

syntax

DEPT: $\rightarrow$ attribute

COURSE#: $\rightarrow$ attribute

TITLE: $\rightarrow$ attribute

P-DEPT: $\rightarrow$ attribute

P-COURSE#: $\rightarrow$ attribute

SECTION#: $\rightarrow$ attribute

INSTRUCTOR: $\rightarrow$ attribute

INST-STATUS: $\rightarrow$ attribute

DAY: $\rightarrow$ attribute

TIME: $\rightarrow$ attribute

BLDG: $\rightarrow$ attribute

RM: $\rightarrow$ attribute

$\equiv$: attribute $\times$ attribute $\rightarrow$ bool

semantics with i,j: attribute

$\equiv(i,i)$ = TRUE for all $i \in$ attribute

$\equiv(i,j)$ = FALSE for all $i \neq j$ in attribute

In the following definition of the type database we use the notation $\{a_1,\ldots,a_n\}$ as a short form for $INSERT(INSERT(\ldots(INSERT(EMPTY,a_1),a_2),\ldots,),a_n)$.

```
type database with
    sorts relation, tuple, set of [tuple], attribute, set of [attribute],
    relnames, dbi, restrictor
    syntax
              φ:  → dbi (the empty data base instance)
       ADDTUPLE:  relnames × tuple × dbi → dbi
                        (an update operation to add a tuple to one of the
                        basic data base relations)
       DELTUPLE:  relnames × tuple × dbi → dbi
                        (an update operation to delete a tuple from one of
                        the basic data base relations)
         CREATE:  set of [attribute] → relation
                        (creates the empty relation over some set of attributes)
        ATTRIBS:  relation → set of [attribute]
                        (ATTRIBS is used to obtain the set of attributes
                        over which a relation is defined)
           TUPS:  relation → set of [tuple]
                        (TUPS is used to obtain the set of tuples constituting
                        a relation)
           KEYS:  relnames → set of [attribute]
      CARTESIAN:  relation × relation → relation
          UNION:  relation × relation → relation
      INTERSECT:  relation × relation → relation
     DIFFERENCE:  relation × relation → relation
                        (The above four operations are the usual operations of
                         cartesian product, union, intersection, and difference.)
```

PROJECT: relation × set of [attribute] → relation

(This is the usual projection operation.)

RESTRICT: relation × set of [attribute] × restrictor × set of [attribute] → relation

(This is the restriction operation.)

JOIN: relation × set of [attribute] × relation → relation

(This is the usual join operation.)

DIVIDE: relation × set of [attribute] × set of [attribute] × relation → relation

(This is the usual division operation.)

COURSE: dbi → relation

COURSE: → relnames

P-REQ: dbi → relation

P-REQ: → relnames

SECTION: dbi → relation

SECTION: → relname

SCHEDULE: dbi → relation

SCHEDULE: → relnames

INSTR-INFO: dbi → relation

INSTR-INFO: → relnames

(Each of the above is the name of one of the basic data base relations and is also a function from data base instances to relations. Thus the relation denoted by say, COURSE in σ is COURSE(σ).)

hidden

DEL: relation × tuple → relation

(DEL is used to remove a tuple from a relation but is not an update operation. It is just used to facilitate the axiomatisation of several operations.)

TUPMATCH: tuple × set of [attribute] × set of [tuple] → set of [tuple]

(Used in the definition of JOIN.)

TUPJOIN: set of [tuple] × set of [attribute] × set of [tuple]

→ set of [tuple]

(Used in the definition of JOIN.)

semantics with r,r': relation; t,t': tuple; s,s': set of [tuple];

a,a': attribute; A,A': set of [attribute]; R,R': relnames;

σ,σ': dbi

ATTRIBS(CREATE(A)) = A

TUPS(CREATE(A)) = EMPTY

(The empty relation with attributes A is made up of the set of attributes A and the empty set of tuples.)

ATTRIBS(COURSE(σ)) = {DEPT, COURSE#, TITLE}

ATTRIBS(P-REQ(σ)) = {DEPT, COURSE#, P-DEPT, P-COURSE#}

ATTRIBS(SECTION(σ)) = {DEPT, COURSE#, SECTION#, INSTRUCTOR}

ATTRIBS (SCHEDULE(σ)) = {DEPT, COURSE#, SECTION#, DAY, TIME, BLDG, RM}

ATTRIBS(INSTR-INFO(σ)) = {INSTRUCTOR, INSTR-STATUS}

(The above defines the attributes of the basic data base relations.)

KEYS(COURSE) = {DEPT, COURSE#}

KEYS(P-REQ) = {DEPT, COURSE#, P-DEPT, P-COURSE#}

KEYS(SECTION) = {DEPT, COURSE#, SECTION#}

KEYS(SCHEDULE) = {DEPT, COURSE#, SECTION#, DAY}

KEYS(INSTR-INFO) = {INSTRUCTOR}

(The above defines the keys of the basic data base relations.)

```
ATTRIBS(CARTESIAN(r,r')) =
    if EQUIV(INT(ATTRIBS(r),ATTRIBS(r')),EMPTY)
    then ADD(ATTRIBS(r),ATTRIBS(r'))
    else error
TUPS(CARTESIAN(r,r')) =
    if EQUIV(INT(ATTRIBS(r),ATTRIBS(r')),EMPTY)
    then CART(TUPS(r),TUPS(r'))
    else error
ATTRIBS(UNION(r,r')) =
    if EQUIV(ATTRIBS(r),ATTRIBS(r'))
    then ADD(ATTRIBS(r),ATTRIBS(r'))
    else error
TUPS(UNION(r,r')) =
    if EQUIV(ATTRIBS(r),ATTRIBS(r'))
    then ADD(TUPS(r),TUPS(r'))
    else error
```

The axioms for INTERSECT and DIFFERENCE are analogous to the above two for UNION.

```
ATTRIBS(PROJECT(r,A)) =
    if SUBSET(A,ATTRIBS(r))
    then SUBT(ATTRIBS(r),A)
    else error
TUPS(PROJECT(r,A)) =
    if SUBSET(A,ATTRIBS(r))
    then if EQUIV(TUPS(r),INSERT(s,t))
         then INSERT(TUPS(PROJECT(DEL(r,t),A)),PIECE(t,A))
    else error
```

```
ATTRIBS(JOIN(r,A,r') =
     if EQUIV(INT(SUBT(ATTRIBS(r),A),SUBT(ATTRIBS(r'),A)),EMPTY)
     then ADD(ATTRIBS(r),ATTRIBS(r'))
     else error
TUPS(JOIN(r,A,CREATE(A')) =
     if SUBSET(A,A') ∧ SUBSET(A,ATTRIBS(r))
     then CREATE(ADD(A',ATTRIBS(r))
     else error
TUPS(JOIN(r,A,r')) =
     if EQUIV(TUPS(r),INSERT(s,t))
     then TUPJOIN(TUPS(r),A,TUPS(r'))
     else TUPS(JOIN(r',A,CREATE(ATTRIBS(r))))
TUPMATCH(t,A,INSERT(s,t')) =
     if SUBSET(A,COLUMNS(t)) ∧ SUBSET(A,COLUMNS(t'))
     then INSERT(TUPMATCH(t,A,s),CATENATE(PIECE(t,SUBT(COLUMNS(t),A)),t'))
     else error
TUPJOIN(INSERT(s,t),A,s') =
     ADD(TUPJOIN(s,A,s'),TUPMATCH(t,A,s'))
```

We have omitted the axioms for division and restriction as these are rather complicated and do not add to the impact of the example.

```
ATTRIBS(R(ADDTUPLE(R',t ,σ))) =
     if EQUIV(COLUMNS(t),ATTRIBS(R'(σ)))
     then ATTRIBS(R)
     else error
ATTRIBS(R(DELTUPLE(R,t,σ))) =
     if EQUIV(COLUMNS(t),ATTRIBS(r(σ)))
     then ATTRIBS(R)
     else error
```

```
TUPS(R(ADDTUPLE(R',t,σ))) = if ≡ (R,R') then
     if EQUIV(COLUMNS(t),ATTRIBS(R(σ)))
     then if  CONT(PIECE(t,KEYS(R)),PROJECT(R(σ),KEYS(R)))
          then error
          else INSERT(TUPS(R (σ)),t)
     else error
                                                      else TUPS(R(σ))

TUPS(R(DELTUPLE(R',t,σ))) = if ≡ (R,R') then
     if  EQUIV(COLUMNS(t),ATTRIBS(R(σ)))
     then if CONT(t,TUPS(R(σ)))
          then DELETE(TUPS(R(σ)),t)
          else TUPS(R(σ))
     else error
                                                      else TUPS(R(σ))
```

Notes: (1) The axiomatisation of ADDTUPLE (one of the two update operations allowed in the data base) is such that the appropriate functional dependencies are "obeyed". Thus, for example, in the case of COURSE with the functional dependency {DEPT, COURSE#} $\rightarrow$ {TITLE} the axiom for ADDTUPLE indicates that ADDTUPLE(COURSE,t,σ) is defined if and only if

(i) the attributes of t are those of the relation instance COURSE(σ)(EQUIV(COLUMNS(t),ATTRIBS(R(σ)));

and (ii) there is no tuple in the relation COURSE(σ) which has DEPT and COURSE# values which are the same as those of the corresponding values of t (CONT(PIECE(t,KEYS(R)), PROJECT(R(σ),KEYS(R)))).

Furthermore, only the basic data base relations can be updated.

(2) The usual relational algebra operations (join, division, restriction, etc.) are defined over relations. Thus the only way in which the basic data base relations can appear in a relational algebraic expression is by being applied to some particular data base instance. However, different occurrences of basic data base relations can be applied to different data base instances in forming some relational algebra expression. Clearly such an expression does not make sense if one is just using the expression to answer a query. However, such an expression might be useful if one is trying to express "historical" properties of data base instances (as is the case in proofs of properties of the data base).

3. Consistency conditions which need to be applied in some situations can also be axiomatised. Suppose, for example, that we have some condition such as - If relation R is updated, then relation R' must be updated (in a well defined way). These are the kinds of conditions which caused the authors of [30] to abandon the algebraic theory which we use here. To illustrate this, consider our example axiomatisation and the following consistency condition.

   The basic relation SECTION cannot be updated with an insertion unless the INSTR-INFO relation contains an entry with the same instructor value as that in the tuple to be inserted.

   This is an example of a co-incidence condition as described above. To take this condition into account, we could modify the axiom with left hand side TUPS(R(ADDTUPLE(R',t,σ))) by replacing its right hand side, which we denote by X, by the following expression:

```
if ≡ (R',SECTION)
then if CONT(PIECE(t,{INSTRUCTOR}),PROJECT(TUPS(INSTR-STATUS
          (σ)),{INSTRUCTOR})
     then X
     else error
else X.
```

An example of a post-condition is the following:

If SECTION is updated by adding a tuple with a given INSTRUCTOR value, then the basic relation INSTR-INFO must be updated by adding the value 10 to the INSTR-STATUS value corresponding to the above INSTRUCTOR value.

This can be done by replacing the update axiom by:

```
TUPS(R(ADDTUPLE(R',t,σ))) =
     if ≡ (R',SECTION)
     then if ≡ (R,INSTR-INFO)
          then INSERT(DELETE(TUPS(R(σ)), < READ(t,INSTRUCTOR),y > )
                       , < READ(t,INSTRUCTOR),y + 10 >)
          else X
     else X
```

where X is as above, y is a variable ranging over INSTR-STATUS values and <w,w'> is an abbreviation for

```
STORE(STORE(NEW{INSTRUCTOR,INSTR-STATUS}),INSTRUCTOR,w)
     ,INSTR-STATUS,w')
```

(Note that although the INSTR-STATUS component of the tuple concerned is not known, only one value of y will satisfy the above equation - namely that associated with the INSTRUCTOR value in the tuple t.) Although we do not do so here, pre-conditons can be similarly expressed as well as combinations of such conditions. If there are many such consistency conditions, then the one axiom we have been discussing may have to be replaced with more specific axioms concerning each basic relation.

5. Conclusions

We have proposed in this report a technique available in the algebraic theory of abstract data types for overcoming deficiencies in earlier attempts at modelling data bases. The formalisation of the concept of data base instance (or environment) allowed us to treat "query" and update operations in the same setting and thus to completely axiomatise their expected interrelationship. We have attempted this axiomatisation for a relational database but there is no reason why the same techniques could not be used to treat the other normal models in current use.

Much work remains to be done in studying such models. For example, an attempt can be made to formalise mathematically the so-called ANSI/SPARC architecture. The conceptual schema could be an axiomatisation such as the one provided here. The internal schema is then an implementation in the technical sense used in the algebraic theory of abstract data types [2,26]. (In terms of our stack example, the algebra A defined in example 1 could be used to define an implementation for the "stack of natural numbers" axiomatised in example 4 because it is isomorphic to the initial algebra defined by the axioms.) The interface between the above schemas is the isomorphism defining the relationship between the two schemas.

An external schema must be definable in terms of the conceptual schema. In other words, the internal schema must provide an implementation language for any external schema. Consistency conditions on external schema could then be expressed as consistency conditions on the mappings which define their implementations.

6. References

[1] ADJ - J.A. Goguen, J.W. Thatcher, E.G. Wagner, J.B. Wright: Initial Algebra Semantics and Continuous Algebras, JACM, Vol. 24, No. 1, pp. 68-95, 1977.

[2] ADJ - J.A. Goguen, J.W. Thatcher, E.G. Wagner, J.B. Wright: An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types in "Current Trends in Programming Methodology, Vol. 4", ed. R.T. Yeh, Prentice Hall, 1978.

[3] ADJ - J.W. Thatcher, E.G. Wagner, J.B. Wright: Data Type Specification: Parameterization and the Power of Specification Techniques, Proc. of 10th SIGACT Symposium on Theory of Computing, 1978.

[4] J.R. Abrial: Data Semantics, in "Data Base Management", J.W. Klimbie and K.L. Koffeman (eds.), North-Holland Publishing Co., 1974, pp. 1-59.

[5] M. Adiba, M. Leonard, C. Delobel: An Unified Approach for Modelling Data in Logical Data Base Design, IFIP-TC-2 Working Conference on Modelling in Data Base Management Systems, Freudenstadt, 1976, pp. 634-665.

[6] A.V. Aho, C. Beeri, J.D. Ullman: The Theory of Joins in Relational Data Bases, ACM TODS, Vol. 4, No. 3, 1979.

[7] C. Beeri, P.A. Bernstein, N. Goodman: A Sophisticate's Introduction to Database Normalization Theory, Proc. of the 4th Int. Conf. on Very Large Data Bases, 1978.

[8] H. Biller, E.J. Neuhold: Semantics of Data Bases: the Semantics of Data Models, Information Systems, Vol. 3, 1978.

[9] M.L. Brodie, J. Schmidt: What is the Use of Abstract Data Types in Data Bases? Proc. of the 4th Int. Conf. on Very Large Data Bases, 1978.

[10] R.L. de Carvalho, T.S.E. Maibaum, T.H.C. Pequeno, A.A. Pereda Borquez, P.A.S. Veloso: A Model-Theoretic Approach to the Semantics of Data Types and Structures, Technical Report, DI-PUC/RJ, Rio de Janeiro, Brasil, 1979.

[11] P. P-S Chen: The Entity Relationship Model - Toward a Unified View of Data; ACM TODS, Vol. 1, No. 1, 1976, pp. 9-36.

[12] E.F. Codd: A Relational Model for Large Shared Data Banks, CACM, Vol. 13, No. 6, 1970, pp. 377-387.

[13] E.F. Codd: Relational Completeness of Data Base Sublanguages, in "Data Base Systems", Courant Computer Science Symposia Series, Vol. 6, Prentice-Hall, 1972.

[14] C.J. Date, E.F. Codd: The Relational and Network Approaches: Comparison of the Application Programming Interfaces, Proc. 1974 ACM-SIGFIDET.

[15] H. Ehrig, H.-J. Kreowski, H. Weber: Algebraic Specification Schemes for Database Systems, Proc. of 4th Int. Conf. on Very Large Data Bases, 1978.

[16] M.H. van Emden, T.S.E. Maibaum: Equations Compared with Clauses for Specification of Abstract Data Types, submitted for publication.

[17] R. Fagin: Relational Database Decomposition and Propositional Logic, IBM Res. Rept. RJ-1776, Apr. 1976.

[18] A.L. Furtado, L. Kerschberg: An Algebra of Quotient Relations, Proc. of SIGMOD Conference, 1977.

[19] J.A. Goguen: Abstract Errors for Abstract Data Types, Proc. of IFIP Working Conference on Formal Description of Programming Concepts, North Holland, 1977.

[20] J.A. Goguen: Some Design Principles and Theory for OBJ-O, A Language to Express and Execute Algebraic Specifications of Programs, Proc. of International Conference on Mathematical Studies of Information Processing, Kyoto, pp. 429-475, 1978.

[21] C.C. Gotlieb, L.R. Gotlieb, "Data Types and Structures", Prentice Hall, 1978.

[22] J.V. Guttag: Abstract Data Types and the Development of Data Structures, CACM, Vol. 20, No. 6, pp. 396-404, 1977.

[23] J.V. Guttag, E. Horowitz, D.R. Musser: Abstract Data Types and Software Validation, CACM, Vol. 21, No. 12, pp. 1048-1064, 1978.

[24] C.A.R. Hoare: Proof of Correctness of Data Representations, Acta Informatica, Vol. 1, No. 1, pp. 271-281, 1972.

[25] M.R. Levy: Verification of Programs with Data Referencing, Proc. of 3me Colloque international sur la programmation, Dunod, pp. 411-426, 1978.

[26] M.R. Levy: Data Types with Sharing and Circularity, Ph.D. Thesis, Department of Computer Science, University of Waterloo, 1978 (Also Technical Report CS-78-26.)

[27] M.R. Levy, T.S.E. Maibaum: Continuous Data Types, to appear in SIAM Journal on Computing.

[28] T.-W. Ling: Improving Data Base Integrity Based on Functional Dependencies, Ph.D. dissertation, University of Waterloo, 1978.

[29] B.H. Liskov, S.N. Zilles: Specification Techniques for Data Abstractions, IEEE TSE, SE-1, No. 1, pp. 7-18, 1975.

[30] P.C. Lockemann, H.C. Mayr, W.H. Weil, W.H. Wohlleber: Data Abstractions for Data Base Systems, ACM TODS, Vol. 4, No. 1, 1979.

[31] P.C. Lockemann, W.H. Wohlleber: Constraints and Transactions: Extensions to the Algebraic Specification Method, Technical Report, University of Karlsruhe, 1979.

[32] T.S.E. Maibaum, Mathematical Semantics and a Model for Data Bases, Proc. of IFIP Congress '77, Ed. B. Gilchrist, North Holland, 1977.

[33] T.S.E. Maibaum: Non-termination, Implicit Definitions, and Abstract Data Types, submitted for publication.

[34] T.S.E. Maibaum, C.J. Lucena: Higher Order Data Types, to appear in Int. Journal of Computer and Information Sciences, 1979.

[35] C. Pair: Formalization of the Notions of Data, Information and Information Structure, "Data Base Management", J.W. Klimbie and K.L. Koffeman (eds.), North Holland Publishing Co., 1974, pp. 149-168.

[36] M.E. Senko: DIAM as a Detailed Example of the ANSI SPARC Architecture, IFIP-TC-2 Working Conference on Modelling in Data Base Management Systems, Freudenstadt, 1976, pp. 170-195.

[37] J.M. Smith, D.C.P. Smith: Data Base Abstraction, ACM Conference on Data: Abstraction, Definition and Structure, 1976, Salt Lake City.

[38] SPARC Interim Report: ANSI Document No. 7514TS01, Washington, D.C., 1975.

[39] F.W. Tompa: A Practical Example of the Specification of Abstract Data Types, to appear in Acta Informatica, 1980.

[40] M. Wand: Final Algebra Semantics and Data Type Extensions, Technical Report 65, Dept. of Computer Science, Indiana University, 1978.

[41] H. Weber: A Semantic Model of Integrity Constraints on a Relational Data Base; IFIP-TC-2 Working Conference on Modelling in Data Base Management Systems, Freudenstadt, 1976, pp. 536-606.

[42] H. Weber: A Software Engineering View of Data Base Systems, Proc. of 4th Int. Conf. on Very Large Data Bases, 1978.

[43] B. Youmark: The ANSI/X3/SPARC/SGDBMS Architecture, Rand Corp., Santa Monica, Cal.

[44] S.N. Zilles: Algebraic Specification of Data Types, Project MAC Progress Report II, MIT, pp. 28-52, 1974.

APPENDIX

DEPT	COURSE #	TITLE
ENG	160	Creative Writing
ENG	220	Shakespeare
FRE	340	19th Cent Fiction

COURSE

DEPT	COURSE #	P-DEPT	P-COURSE #
ENG	160	ENG	20
ENG	160	ENG	51
ENG	220	ENG	120
ENG	220	ENG	121
FRE	340	FRE	100
FRE	340	FRE	240

P-REQ

DEPT	COURSE #	SECTION #	INSTRUCTOR
ENG	160	1	Andrews
ENG	160	2	Thomas
ENG	220	1	Andrews
ENG	220	2	Brown
FRE	340	1	Dubois
FRE	340	2	Armand

SECTION

DEPT	COURSE #	SECTION #	DAY	TIME	BLDG.	RM
ENG	160	1	Mon	10	UC	201
ENG	160	1	Thu	11	UC	202
ENG	160	2	Mon	10	NC	14
ENG	160	2	Wed	9	NC	20
ENG	220	1	Wed	1	TR	20
ENG	220	1	Fri	12	TR	21
ENG	220	2	Tue	10	NC	517
ENG	220	2	Thu	11	NC	517
FRE	340	1	Tue	2	UC	202
FRE	340	1	Thu	1	UC	201
FRE	340	2	Mon	3	VC	118
FRE	340	2	Thu	3	VC	120

SCHEDULE

INSTRUCTOR	INSTR-STATUS
Thomas	67
Andrews	18
Brown	56
Dubois	40
Armand	22

INSTR-INFO