DESIGN OF SIDAL - A DATA MANIPULATION LANGUAGE

THESIS

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science in the Graduate College of the University of Illinois at Urbana-Champaign, 1980

Urbana, Illinois
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

THE GRADUATE COLLEGE

MAY 1980

WE HEREBY RECOMMEND THAT THE THESIS BY

SOHAIB ABBASI

ENTITLED DESIGN OF SIDAL - A DATA MANIPULATION LANGUAGE

BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR

THE DEGREE OF MASTER OF SCIENCE

Committee on Final Examination†

Chairman

† Required for doctor's degree but not for master's.

O-517
FORMAT APPROVAL

To the Graduate College:

The format of the thesis submitted by Sohaib Abbasi
for the degree of Master of Science is acceptable to the
department of Computer Science.

4/30/80
Date
(Signed) Nicola Gilmore
Departmental Representative
TABLE OF CONTENTS

1 INTRODUCTION..................................................................................1
  1.1 Background..................................................................................2
  1.2 Formal Definition..........................................................................3
  1.3 System Overview..........................................................................4

2 LANGUAGE SUMMARY........................................................................5
  2.1 Operators and Built-in Functions...............................................5
  2.2 Data Structures...........................................................................8
  2.3 System's Relations......................................................................9
  2.4 Creating and Destroying Relations............................................11
  2.5 Updating Relations.....................................................................14
  2.6 Relational Queries.....................................................................16
  2.7 Combining Relations...................................................................18
  2.8 Subexpressions..........................................................................19
  2.9 Expanding and Contracting Relations.......................................20
  2.10 Abbreviations...........................................................................21
  2.11 Built-in Functions......................................................................22
  2.12 Relational Definitions...............................................................24
  2.13 Integrity Constraints.................................................................25

3 SIBYL................................................................................................26
  3.1 SIBYL Data Structures and Operators.........................................26
  3.2 SIBYL Control Structures...........................................................29
  3.3 Operations on Lists.....................................................................31
  3.4 Examples of Operations on Lists and Records...........................33
4 THE DATA SUBLANGUAGE

4.1 Formal Specification of the Semantics

4.1.1 Beck's Data Manipulation Primitives

4.1.2 Additional SIDAL Primitives

4.1.3 Additional notes

5 OPERATIONS ON RELATIONS

5.1 Construction

5.2 Constraint

5.3 Insertion

5.4 Deletion

5.5 Projection

5.6 Selection

5.7 Assignment

5.8 The Join Operator

5.9 The Union Operator

5.10 The Difference Operator

6 THE OVERSEER

6.1 Checks based on the Target Relation

6.2 Checks based on the Operator

7 COMPARISON WITH OTHER SUBLANGUAGES AND EXAMPLES

7.1 Retrieval Queries

7.2 Update Queries

REFERENCES

A GRAMMAR FOR SIDAL
B  SCENARIO OF A SIDAL SESSION..............................74
1 INTRODUCTION

SIDAL (Simple DAta Language) is a simple online data manipulation language designed to serve as the primary interface to a student-oriented relational database system. It includes several system commands and a query notation based on Codd’s Relational Algebra [6].

SIDAL is formally defined in Chapters 4 and 5, in terms of an extensible programming language, SIBYL [13].

The system is currently being implemented in PASCAL on a CYBER 175 computer by the author. The current implementation uses building blocks of the SIBYL interpreter [14].

This chapter explains the background, the formal definition approach, and contains a brief system overview. Chapter 2 contains a summary of the SIDAL language. Chapter 3 provides an overview of SIBYL. Chapter 4 contains a definition of the Process Definition Language [3, 4] used to define SIDAL. Chapter 5 defines the semantics of the SIDAL operators in terms of the Process Definition Language. An explanation of the Overseer which enforces the integrity and relation status constraints follows in Chapter 6. SIDAL is compared with other data sublanguages by means of examples in Chapter 7. The appendices contain a complete syntactic definition of SIDAL and a scenario of a SIDAL session.
1.1 Background

The project was undertaken by the author in February 1979 under the supervision of Dr. G. R. Kampen. The initial specifications were provided by Dr. Kampen. About the same time, Achim Stulpnagel started the implementation of the SIBYL interpreter.

The author under the supervision of Dr. Kampen completed this design document in December 1979. The SIBYL interpreter was completed in September 1979. In a separate project, the author extended the SIBYL interpreter to handle set operators.

During January and February 1980, the author integrated a number of projects geared towards improving the SIBYL interpreter. The implementation of the SIDAL operators was also started in February 1980 under the supervision of Dr. G. G. Belford. The implementation of SIDAL would be as an extension to the SIBYL interpreter. As the implementation uses building blocks of the SIBYL interpreter, it was felt that the documentation should be appended to the SIBYL Implementation Manual [14].

In March 1980, Dr. Belford after reading this document suggested ways to improve the document. These changes included modifying certain design decisions. These changes were made to this document.
1.2 Formal Definition

The semantics of SIDAL are defined by devolution in two stages. Figure 1.1 illustrates this approach.

```
<table>
<thead>
<tr>
<th>SIDAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>v</td>
</tr>
<tr>
<td>------</td>
</tr>
<tr>
<td>PDL'</td>
</tr>
<tr>
<td>------</td>
</tr>
<tr>
<td>v</td>
</tr>
<tr>
<td>------</td>
</tr>
<tr>
<td>SIBYL</td>
</tr>
</tbody>
</table>
```

Figure 1.1

The arrows in Figure 1.1 indicate the relation 'is-specified-by'.

The first stage is a mapping of SIDAL into a semantically equivalent program in PDL'. PDL' is a syntactically sugared extension of SIBYL that resembles the Process Definition Language proposed by Beck as a base language for relational sublanguages [3, 4]. It contains the primitive tuple manipulation operations proposed by Beck and an additional set of tuple operators based on the algebra of sets.

The second stage is a definition of PDL' by devolution into SIBYL.
1.3 System Overview

SIDAL is an on-line, relational data sublanguage. It allows the user to create, modify and display relations from a terminal. Figure 1.2 illustrates the implementation.

```
<table>
<thead>
<tr>
<th>text file</th>
</tr>
</thead>
<tbody>
<tr>
<td>V</td>
</tr>
<tr>
<td>-----------------------------</td>
</tr>
<tr>
<td>input -&gt;</td>
</tr>
<tr>
<td>SIBYL modules</td>
</tr>
<tr>
<td>-----------------------------</td>
</tr>
<tr>
<td>V</td>
</tr>
<tr>
<td>text file</td>
</tr>
</tbody>
</table>
```

Figure 1.2

The SIDAL modules implement the sublanguage operators. They interact with the existing SIBYL modules along with the Overseer. The Overseer enforces the integrity constraints and the restrictions imposed by the relation's status.

The current implementation of SIDAL is for an in-core database. The text file in Figure 1.2 is a workspace file containing all the standard operators, delimiters, brackets, names of built-in functions, and special symbols and names [14]. It will also contain the database.

Appendix B contains a scenario of a SIDAL session. It includes the details of how to access and use the system.
There are four system commands, similar to those in the existing Baran-Yueh database system [19]:

- **USE file;** <Fetch the database from a CYBER file>
- **COPY (file,newfile);** <Copy the file to another CYBER file>
- **LOAD (file,textfile);** <Convert a text file to internal form>
- **UNLOAD (file,textfile);** <Convert the file into a text file>

There are three SIDAL statements:

- **name := expression;**  <Set name to the value of expression>
- **PRINT operand;**      <Print the value of operand offline>
- **DISPLAY operand;**    <Display the value of operand online>

An expression is a freely formatted sequence of the operators, functions, and operands listed below. The value of an expression can be anything from an atomic value like 12 to an entire relation. Illegal expressions yield the value % (undefined).

### 2.1 Operators and Built-in Functions

Operators are listed in order of precedence from highest to lowest.

- **operator:** * / #  <join, projection, selection>
  + -  <union, difference>
  <= >= /= &:  <&: is set inclusion>
  ~  <NOT>
The last five operators provide abbreviations for the corresponding operations where the result is assigned to the first (left hand side) operand relation. For example, REL *= REL1 is an abbreviation for REL := REL * REL1.

The + symbol is used for three different operations. The actual operation depends on the second (right hand side) operand. If the second operand is a relation, it corresponds to the union operator. If the second operand is a tuple, it denotes tuple insertion. If a predicate is specified as the second operand, it corresponds to the constraint operation (i.e., adding a constraint). The symbol += also denotes three equivalent operations.

The - symbol denotes the difference or tuple deletion operation depending on whether the second operand is a relation or a tuple respectively. Similarly, the symbol -= denotes two equivalent operations.
<table>
<thead>
<tr>
<th>function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVG</td>
<td>Compute the average</td>
</tr>
<tr>
<td>MIN</td>
<td>Compute the minimum</td>
</tr>
<tr>
<td>MAX</td>
<td>Compute the maximum</td>
</tr>
<tr>
<td>SUM</td>
<td>Compute the sum</td>
</tr>
<tr>
<td>COUNT</td>
<td>Return the number of tuples</td>
</tr>
<tr>
<td>DOMAINS</td>
<td>Return the list of domains</td>
</tr>
<tr>
<td>KEYS</td>
<td>Return the list of keys</td>
</tr>
<tr>
<td>STATUS</td>
<td>Return the status</td>
</tr>
<tr>
<td>CONSTRAINT</td>
<td>Return the constraint</td>
</tr>
</tbody>
</table>
2.2 Data Structures

operand:
  subexpression: \((A \times B)\)

atom:
  integer: 15
  number (decimal): -35.003
  string: "OLD MACDONALD"
  name: X1 FRED
  undefined: %
  fill: ""

tuple: \{(56, 'M', FRED)\}
descriptor: \{(EMPNO), SEX, NAME\}
selector: \[EMPNO < 57 \land SEX = 'M'\]
domain:
  \{name: SEX, 
   description: 'SEX OF EMPLOYEE',
   type: STR, 
   length: 1, 
   predicate: \[X &: \{'M', 'F'\}\]}
relation:
  \{type: 'REL',
   doms: \{EMPNO, SEX, NAME\},
   keys: \{EMPNO\},
   status: 'P',
   last: \{58, 'M', TOM\},
   constraint: \[EMPNO < 60 \land SEX = 'M'\],
   numtuples: 3,
   tuples: \{(56, 'M', FRED),
              \{57, 'M', BILL\},
              \{58, 'M', TOM\}\}}

Figure 2.1

All operands other than domains and relations are written, stored, and printed out as shown in the examples above. To create a tuple, for example, we simply
type a list of atoms enclosed in braces (curly brackets). In the case of
domains and relations, we have shown their internal formats as record struc-
tures. They are represented in SICAL by expressions composed from atoms,
tuples, descriptors, and constraints, and they are printed or displayed in yet
another format.

Each relation has a descriptor associated with it. It consists of a list of
attributes. The key attributes are enclosed in braces (curly brackets). The
tuples of the relation are sorted on these key fields. The order of the
attributes in the key is from left to right, i.e., the leftmost key is used
for the major sort, then the next attribute to the right, and so on.

Each domain used in a relation has to be defined prior to such use. Domain
definitions are represented as tuples in the System's relation, DD (Domain
Dictionary).

2.3 System's Relations

The database system maintains two relations -- DD (Domain Dictionary), and RD
(Relations Directory). These relations store information about the contents
of the database. A part of the database manager called the Overseer accesses
these relations in order to perform its functions of insuring integrity and
status constraints. The Overseer's function is elaborated in Chapter 6.

Relation DD contains the domain definitions. The schema of this relation is
as follows.

{ (DOMAIN_NAME), DESCRIPTION, TYPT, LENGTH, DOM_PREDICATE }
The fields of the relation are explained below.

**DOMAIN_NAME**: the name of the domain; this is the key field of the relation.

**DESCRIPTION**: contains a description of the domain.

**TYPE**: contains the "type" of values in the domain; can be one of the following: STR (characters enclosed in quotes; a quote symbol is represented in a string as a pair of quote symbols), NAME (first character must be alphabetic; may contain alphanumeric characters along with embedded underline characters), INT (integer), NUM (decimal number), and PRD (predicate enclosed in square brackets).

**LENGTH**: the maximum length of the values in the domain.

**DOM_PREDICATE**: a predicate that all values of the domain must satisfy.

The system also maintains a Relations Directory, RD, which has the following schema.

```plaintext
{ {RELATION_NAME}, STATUS }
```

The fields of this relation are described below.

**RELATION_NAME**: the name of the relation; this is the key field of the RD relation.

**STATUS**: contains the status of the relation; the possible values are R (Read-only), S (Stable), P (Permanent), and T (Temporary).

The status of relation RD is "R". Such a relation can only be displayed. The relation cannot be altered in any manner. However the system may insert new
tuples into it. The status of relation DD is "S". Such a relation can be displayed. In addition, the user may insert tuples to such a relation. However, tuple modifications and deletions are not permitted. Also the user is not allowed to either delete, or change the schema of, the relation.

The other status types are "P" and "T". Status "P" relations allow all tuple operations. However operations that delete, or change the structure of, the relation require reconfirmation from the user. All operations are allowed on "T" type relations.

Relations with status "R", "S", and "P" are saved at the end of the session. "T" type relations are not saved for later use.

2.4 Creating and Destroying Relations

To create the domain shown in Figure 2.1 and assign it to the variable SEX, we would enter the SIAL statement below:

```
DD += {SEX, 'EMPLOYEE SEX', STR, 1, [SEX &: {'M', 'F'}]};
```

Notice that as the status of relation DD is "S", tuples may only be inserted one at a time by the += command. This would allow the system to insure that the status restrictions are not violated.

The above command essentially inserts the domain definition for "SEX" as a tuple in the system's relation DD (Domain Dictionary). All domains to be used in relations must be defined prior to creation of the relations in which they appear.
To create the relation shown in Figure 2.1 and assign it to the variable EMP, we enter:

\[
\text{EMP} := \text{P} \{ \{\text{EMPNO}\}, \text{SEX}, \text{NAME} \} + \{ \text{EMPNO} < 60 \land \text{SEX} = 'M' \}
\]
\[
+ (56, 'M', \text{FRED})
\]
\[
+ (57, 'M', \text{BILL})
\]
\[
+ (58, 'M', \text{TOM});
\]

In the definition of relation EMP, the key field EMPNO is indicated by enclosure in braces. The P indicates that this is a permanent relation, to be retained when the database is saved at the end of the session. The constraint contains a predicate that must be satisfied by every tuple in the relation. As a side effect, the system inserts a tuple for the new relation in the System's Relation RD.

Both definitions could have been considerably simpler:

\[
\text{DD} += \{ \text{SEX}, '\text{EMPLOYEE SEX}', \text{STR}, 1, [1] \};
\]

\[
\text{EMP} := \text{P} \{ \{\text{EMPNO}\}, \text{SEX}, \text{NAME} \};
\]

The statements above create a domain with a predicate field which always returns a "True" value, i.e., an empty predicate, and a relation with an empty constraint field and no data.

To insert a pair of tuples, print the resulting relation, and then delete EMP from the database, we would type:

\[
\text{EMP} := \text{EMP} + \{58, 'M', \text{TOM}\}
\]
\[
+ \{53, '', \text{JOE}\};
\]
PRINT EMP;

EMP := %;

Note the use of " to save rewriting the atom 'M'. When a tuple is inserted, fields containing the fill atom " take their value from the corresponding field of the last tuple inserted, modified, or deleted. This tuple will always be found in field LAST of the record that implements the relation. Aside from its use as a source of fill data, it can be helpful in reminding a user where to resume after an interruption in a data entry session.

The result of the PRINT statement is the following printed output:

EMP
{(EMPNO), SEX, NAME}
   53 'M' JOE
   58 'M' TOM

Note that the tuples have been sorted in ascending order on the key of the relation.

As each tuple is inserted, it is placed in the proper order and checked against the relation constraint and the domain predicates. This ensures that the relation always satisfies the following integrity and other restrictions:

1. The set of keys is a (possibly empty) subset of the set of domains.

2. Tuples of the relation are sorted on their key fields in ascending order, using the leftmost key for the major sort, then the next key to the right, and so on. If the key set is empty, tuples appear in their order of insertion.
3. Every tuple is uniquely identified by its key fields. If the key set is empty, duplicate tuples are permitted; otherwise, every tuple is distinct.

4. If a constraint is present, the constraint predicate is true for every tuple in the relation.

5. For every domain name to which a domain value has been assigned, the corresponding field of every tuple has the same type. If the domain has a predicate, the predicate is true for every tuple value belonging to that domain. An illegal tuple value will be set to \( \sim (\text{undefined}) \), which is always a legal value.

2.5 Updating Relations

It is an easy matter to update an existing relation using a sequence of + and - operators. For example, to insert tuples for Ted and Bob into the existing relation for EMP, change Joe's name to Jay, and delete the tuple Tom, we would type:

\[
\text{EMP} := \text{EMP} + \{50, 'T', \text{TED}\} \\
+ \{51, '', \text{BOB}\} \\
+ \{53, 'J', \text{JAY}\} \\
- \{58\};
\]

The last tuple in the above command specified only the key field. SICAL allows such abbreviation for the insertion and deletion commands. In case of insertion, the system will set the unspecified attributes to null values. However, the entire key must be specified. A tuple with an unspecified key
field is flagged as an error.

The new value of EMP is:

EMP

{{EMPNO}, SEX, NAME}

50 'M' TED
51 'M' BOB
53 'M' JAY

When an existing tuple is to be modified, only the new values need to be entered. The symbol % is used as a placeholder for fields that will not be updated. If we wanted to replace 'M' with % (to indicate a partly successful sex-change operation, perhaps, or to evade the male chauvinist relation selector) we would need to delete and then reinsert the tuple in question:

EMP := EMP - {53} + {53, %, JAY};

We can use the same technique to change the value of a key field:

EMP := EMP - {53} + {54, 'M', JAY};

Or, to avoid having to copy the data in the original tuple:

EMP := EMP - {53} + {54, '', ''};
The result of executing all of the statements above would look like this:

EMP
{(EMPNO), SEX, NAME}
  50 "M" TED
  51 "M" BOB
  54 "M" JAY

2.6 Relational Queries

The operators / (project) and # (select) provide a means of selecting and/or rearranging the columns of a relational table or selecting a set of rows. The projection operator selects the columns named in the descriptor and combines them to form a new relation whose domains and keys are obtained from the descriptor. If none of the domains in the descriptor are enclosed in braces, tuples of the resultant relation are ordered just as they were in the original; if some domains are indicated as keys by enclosure in braces, the result will be sorted on these domains and tuples with duplicate keys will be eliminated. The selection operator selects those tuples that satisfy the selector predicate and forms a new relation with the same descriptor as the original.

For example, suppose we type:

A := EMP;

B := A # [ EMPNO < 52 ];

C := A # [ EMPNO >= 52 ];
D := A \{ EMPNO, SEX \};

E := A \{ EMPNO, NAME \};

F := A \# [ EMPNO \geq 52 ] \{ EMPNO, NAME \};

Given the relation EMP produced by the updates of the previous section, the resulting relations would be as shown below.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>{(EMPNO), SEX, NAME}</td>
<td>{(EMPNO), SEX, NAME}</td>
</tr>
<tr>
<td>50 'M' TED</td>
<td>50 'M' TED</td>
</tr>
<tr>
<td>51 'M' BOB</td>
<td>51 'M' BOB</td>
</tr>
<tr>
<td>54 'M' JAY</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>{(EMPNO), SEX, NAME}</td>
</tr>
<tr>
<td>54 'M' JAY</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>{(EMPNO, SEX)}</td>
<td>{(EMPNO, NAME)}</td>
<td>{(EMPNO, NAME)}</td>
</tr>
<tr>
<td>50 'M'</td>
<td>50 TED</td>
<td>54 JAY</td>
</tr>
<tr>
<td>51 'M'</td>
<td>51 BOB</td>
<td></td>
</tr>
<tr>
<td>54 'M'</td>
<td>54 JAY</td>
<td></td>
</tr>
</tbody>
</table>

In this example we would have obtained the same result if we had done the projection first and then the restriction. In general, however, it is safer to do the restriction first to avoid the possibility that domains required by the selector are deleted by the projection.
2.7 Combining Relations

Just as the / and $ operators permit us to take relations apart, the operators described in this section enable us to put relations together. The operator $ forms the join of two relations by combining all pairs of tuples that match on their common domains. The + and - operators form a new relation by taking the first operand and inserting into it or deleting from it the tuples in the second operand. When the descriptors of both relations are the same, operator $ is set intersection, + is set union, and - is set difference.

If the relation A were erased and we wished to reconstruct it, either of the statements below would suffice.

A := B + C;
A := D $ E;

If relation B were erased we could reconstruct it by typing:

B := A - C;

When the descriptors of the operands are not equal, various special cases arise. If the sets of domains are disjoint, $ forms the Cartesian product of its operands. When the common domains are exactly the key set of one of the operands, $ is a cross-reference operator. In other cases + is a merge operation and $ and - are used to select subsets of a relation.
2.8 Subexpressions

SQL AL expressions may include relational, logical, and arithmetic subexpressions among their operands. For example, if we are given only the relations D and E from the previous example and we wish to print the names of all employees whose sex is undetermined, we can obtain the results we want by typing:

```
PRINT D * E # [SEX = %] / {NAME};
```

The evaluation of an expression is from left to right for operators with the same precedence. The precedence can be changed by enclosure in parentheses which have the highest precedence. Parentheses are used for altering the order of evaluation of an expression. A subexpression refers to any expression enclosed in parentheses.

As another example, suppose we wish to add two additional tuples to EMP. We could use the method of the previous section:

```
EMP := EMP + {55, %, JOE}
    + {56, %, JIM};
```

But there's another way:

```
EMP := EMP + ( T{{EMPNO}, NAME}
    +( 55, JOE)
    +( 56, JIM});
```

By creating a relation and then adding it to EMP we can limit our tuples to just those fields that contain data, and moreover we can arrange the fields in any order we choose. The use of a subexpression in this case is absolutely necessary.
2.9 Expanding and Contracting Relations

SIDAL allows contraction and expansion of a relation's degree. However such operations have to be compatible with the relation's status. Specifically, such operations are permissible for relations with status 'P' and 'T'. For status 'P' relations, the user is asked for reconfirmation prior to the execution of these commands.

To expand the degree of a relation by adding one or more attributes, the "join" operator is used. As in relation creation, the new domains have to be defined prior to their use. Given the relation EMP : ({EMPNO}, SEX, NAME),

\[ \text{EMP} := \text{EMP} \ast (T\{\text{ADDRESS}\} + \{\%\}); \]

would include the attribute ADDRESS in the relation EMP. In addition, it would initialize this attribute to \% (null) in each tuple. The following abbreviation will have the same effect.

\[ \text{EMP} \ast= (T\{\text{ADDRESS}\} + \{\%\}); \]

To contract the degree of a relation by deleting one or more attributes, the "projection" operator is used.

\[ \text{EMP} := \text{EMP} / ({\{\text{EMPNO}\}}, \text{NAME}); \]

would eliminate the attribute SEX from the original relation. In general, the attributes that are to be kept are specified in the projection descriptor. The following abbreviation will have the same effect.

\[ \text{EMP} /= ({\{\text{EMPNO}\}}, \text{NAME}); \]
2.10 Abbreviations

Because update expressions like the one above are so common, SIDAL includes the abbreviation

\[
\text{name op= expression}
\]

for statements of the form

\[
\text{name := name op (expression)}
\]

Thus the last update statement above could have been written as follows:

\[
\text{EMP += ( T\{\text{EMPNO}, \text{NAME}\} +\{ 55, \text{JOE}\} +\{ 56, \text{JIM}\});}
\]

The abbreviation makes it easy to add a new column to a relation:

\[
\text{EMP += (T\{AGE\} + \{\%\});}
\]

The single-element tuple \{\%\} initializes the new field in each row to the null value.

The abbreviation provides a means of inserting tuples one at a time. This is the only means of inserting tuples in "S" type relations which do not allow tuple updates or tuple deletions.

\[
\text{EMP += (59, \text{M'}, \text{JACK});}
\]

A similar abbreviation exists for the constraint operator.

\[
\text{EMP += [EMPNO < 55];}
\]
Difference and tuple deletion can also be abbreviated using the $-$ operator.

The symbol $\neq$ can be used to contract the degree of a relation. The symbol $\#=$ is used to shrink the number of tuples of a relation by means of a selection operation.

2.11 Built-in Functions

SIDAL provides a number of built-in functions. These include aggregate functions along with functions that return certain characteristics of the given relation.

The arguments to these functions are relation-names or expressions that yield a relation. The syntax of the function-call is as follows.

\[
\text{function-name ( relation-name | expression )}
\]

i.e., the function-name followed by a relation-name or an expression evaluating to a relation.

The following aggregate functions are provided. It should be noted that these functions take as their arguments relations with only numeric attributes.

1) **AVG** : Computes the average values of the attributes of the given relation.

2) **MIN** : Computes the minimum values of the attributes of the given relation.

3) **MAX** : Computes the maximum values of the attributes of the given relation.
4) SUM : Computes the sums of the values of each attribute of the given relation.

The following functions are available to inquire about a relation's characteristics.

1) COUNT : Returns the count of the number of tuples in the given relation.

2) DOMAINS : Returns the list of attributes of the given relation.

3) KEYS : Returns the list of the attributes that constitute the "key" of the given relation.

4) STATUS : Returns the status of the given relation.

5) CONSTRAINT : Returns the constraint predicate that the tuples of the given relation must satisfy.

The function CONSTRAINT may appear wherever a predicate can. For instance, it may be used to specify the predicate of a selection operator.

DISPLAY SUP # (CONSTRAINT SUP1);

would display those tuples of relation SUP that satisfy the constraint predicate of the relation SUP1.

The functions DOMAINS and KEYS may appear wherever descriptors are allowed. Such a case is as a substitute for a domain-list of a projection operator.

DISPLAY SUP / (DOMAINS SUP1);
would display those attributes of relation SUP that appear in relation SUP1.

In the above examples, the attributes of relation SUP1 must be a subset of the attributes of relation SUP. Care must be taken when using the functions CONSTRAINT, DOMAINS and KEYS in the above manner to insure that the projection or selection does not specify attributes not present in the operand relation.

2.12 Relational Definitions

SIDAL contains facilities for relational definitions. These definitions allow the creation of "dynamic" views of the database. All the queries discussed so far provided a snapshot view of the database. SIDAL provides a mechanism to define relations whose tuples reflect any changes made to the database since their definitions. The actual evaluation of the definition takes place when such a relation appears in an expression. The following example will illustrate this facility.

RDS := [ EMP # ( EMPNO > 50 ) ];

Relational definitions are enclosed in square brackets. In CODASYL DBTG terminology, this would correspond to a "virtual" result as opposed to an "actual" result.

The relation RDS may be used in any expression and will be evaluated at that point. Any modifications made to relation EMP will be reflected in relation RDS.
2.13 Integrity Constraints

Integrity constraints in SIDAL are at two levels: relations and domains. Every relation has one integrity constraint that each tuple in it must satisfy. In addition, every domain has one integrity constraint associated with it which every value in that domain must satisfy. These constraints are represented as predicates enclosed in square brackets. The predicates consist of arithmetic comparison operators and the set inclusion operator along with logical connectives.

The Overseer is responsible for maintaining the integrity constraints. This role is elaborated in Chapter 6.
The semantics of SIDAL operators are formally specified in SIBYL in Chapters 4 and 5. In addition, SIDAL operators are currently being implemented as an extension to SIBYL. This chapter gives a brief and informal discussion of SIBYL.

SIBYL is a self-extensible interactive programming language [13, 14]. It is a dynamically typed language and provides a rich collection of data and control structures.

3.1 SIBYL Data Structures and Operators

1) Integer and decimal numbers and the arithmetic operators +, -, *, / together with an edit operator #.

2) Character strings and the operators & (concatenate), hd (head), tl (tail), and len (length).

3) Boolean values (represented by the integers 0 and 1) and the operators \~ (not), |\ (and), and |\/ (or).

4) Lists and the operators &/ (compress), &* (intersection), &<< (membership), &+ (union), &- (difference), & (concatenate), hd (head), tl (tail), sortu (sort in ascending order), sortd (sort in descending order), and len (length). A list is represented by a sequence of values (which may themselves be lists) separated by commas and enclosed in braces. Arrays, matrices, and sets are special cases of the list structure. These operators are explained
5) Records and the operators & (concatenate), : (bind), dom (domain), and rge (range). A record is represented by a nonempty sequence of fields separated by commas and enclosed in braces; each field consists of a name and a value separated by a colon.

6) Procedures and the operators ! (execute), and & (concatenate). A procedure is represented by an expression enclosed in square brackets.

7) Variables and the operators $ (declare), ? (fetch), -> and <- (assign), rd (read), ->> (write), and ! (subscript).

Additional operators and built-in functions include the following.

8) Relations <, =, >, <=, >=, and /= (not equal) on numbers and strings. Relation == compares any pair of objects, including variables and procedures.

9) Relations &<= (subset), &< (proper subset), &= (equality), &!! (disjoint), and &: (set inclusion) on lists. These relations are explained in Section 3.3.

10) Operators ( and ) that enclose subexpressions and operator , that separates list elements.

11) Transfer functions that convert an object into its character string representation and vice versa.

12) Built-in functions for type-checking and other purposes.

13) Operators on atoms (names, numbers, and character strings) distribute themselves over lists as in APL. Such distribution of an operator can be
obtained by appending a dot as a prefix or suffix or both to the operator.

14) SIBYL maintains an activation record for the block or procedure currently being activated. It is a SIBYL record denoted by the special name env, and consists of pairs of variables and their values. Since the user can access env like any other variable, he can display the local environment or link it to other environments to simulate various scoping rules.

SIBYL operators are listed below in order of precedence from highest to lowest.

```
(  
!  
~ ! =
* / #
+ -
< > <= >= /=
~ ..
/\ 
\ /
&* &/
&+ &- &-> ->>
&<< &<= &< &>= &! !
<- :
$ ; => @ //
,
)
{ } [ ]
```
3.2 SIBYL Control Structures

1) A series is an expression of the form \( e_1; e_2; \ldots; e_n \). The component expressions are executed from left to right, and the value of the series is the value of the last expression \( e_n \).

2) A conditional is a subexpression of the form \( (p \Rightarrow e) \) or \( (p \Rightarrow e, s) \) where \( p \) is a predicate. The former corresponds to an IF-THEN and the latter to an IF-THEN-ELSE statement.

3) A loop is a subexpression of the form \( (n @ s) \) or \( (i e @ s) \) or \( (l e @ s) \) or \( (s e @ s) \) where \( i e, l e, s e \) are expressions that evaluate to an integer, a list, and a string respectively.

Variations of the first form include:

\[
\begin{align*}
(@ p & \Rightarrow e) & \quad \text{while } p \text{ do } e \text{ endwhile} \\
(@ e & ; p \Rightarrow ,) & \quad \text{repeat } e \text{ until } p \text{ endrepeat} \\
(@ e, p & \Rightarrow e_1, e_2) & \quad \text{perform } e \text{ when } p \text{ return } e_1 \text{ else } e_2 \text{ endperform}
\end{align*}
\]

The second form is defined by

\[
(n @ s) = \begin{cases} 
(s; n-1 @ s) & \text{when } n > 0 \\
() & \text{otherwise}
\end{cases}
\]

The following combinations of the first two forms are allowed.

\[
\begin{align*}
(n @ s) & \quad \text{perform } s \text{ } n \text{ times} \\
(n @ p & \Rightarrow e) & \quad \text{perform } e \text{ } n \text{ times while } p \\
(n @ e & ; p \Rightarrow ,) & \quad \text{perform } e \text{ } n \text{ times until } p
\end{align*}
\]

The remaining two loop formats are defined by

\[
(x @ s) = (\text{hd } x \text{ s; tl } x @ s) \quad \text{when } \text{len } x > 0
\]
\[
= () \quad \text{when } \text{len } x = 0
\]

\(\text{hd}\) and \(\text{tl}\) are built-in functions that return the first and last elements of the argument list (or string) respectively.

The expression \(\text{le}\) or \(\text{se}\) are evaluated to yield a list or a string \(x\), and then successive list elements or characters from \(x\) are appended to successive instances of \(s\) and the result is evaluated.

4) A case statement is a subexpression of the form \((e // e_1, e_2, \ldots)\) where \(e_1, e_2, \ldots\) are expressions. A case statement is defined by

\[
(v // e_1, e_2, \ldots) = (v e_1, v e_2, \ldots)
\]

where \(v\) denotes the value of the expression \(e\).

By combining the case and the conditional statements, we obtain a form of case statement equivalent to that appearing in other languages:

\[
(i // =1 \Rightarrow e_1, \quad \text{case } i \text{ of } 1: e_1,
=2 \Rightarrow e_2, \quad \text{2 : } e_2,
; e_3) \
\]
\[
\text{else } e_3 \text{ endcase}
\]
3.3 Operations on Lists

The author extended SIBYL to include certain APL and Set operators. These operators are defined for lists whose elements are primitive, i.e., numbers, names, and strings. Numbers may be either integer or decimal. Strings consist of characters enclosed in quotes. Names consist of an alphabetic character followed by alphanumeric characters. Names may contain embedded underline characters. These operators include the following:

1) Intersection (&\*) : generates a list whose elements appear in both the operand lists.

2) Compress (&/) : this is equivalent to the APL Compress operator. The left hand side operand is a list consisting of the integers one and zero. Compress forms a list by selecting those elements of the right hand side list for which the corresponding elements of the other list equal the integer one.

3) Union (&+) : generates a list whose elements appear in either of the operand lists.

4) Difference (&-) : generates a list whose elements appear in the left hand side list but do not appear in the other list.

5) Membership (&<<) : this is equivalent to the APL Membership operator. It forms a list that contains the integer one if the corresponding element of the left hand side list appears in the right hand side list and the integer zero if it does not.

The extension included the following relations on lists.

6) Subset (&<=) : returns the integer one if the elements of the left hand
side list appear in the right hand side list. It returns the integer zero otherwise.

7) Proper Subset (&<) : returns the integer one if all the elements of the left hand side list appear in the right hand side list and the right list contains an element which is not in the left list. It returns the integer zero otherwise.

8) Equality (&=) : returns the integer one if all the elements of each operand list also appear in the other operand list. It returns the integer zero otherwise.

9) Disjoint (&!!) : returns the integer one if none of the elements of the right hand side list appear in the left hand side list. It returns the integer zero otherwise.

10) Set Inclusion (&:) : returns the integer one if the right hand side operand appears as an element in the right hand side list. It returns the integer zero otherwise.

In addition, two utility functions are also provided.

11) Sort in ascending order (sortu) : sorts the elements of the argument list in ascending order. The order between different element types is : first numbers (numerical ordering) then names (collating sequence) followed by strings (collating sequence).

12) Sort in descending order (sortd) : sorts the elements of the argument list in descending order.
3.4 Examples of Operations on Lists and Records

The following operators take lists as arguments. An example for each operator is also given.

\[
\begin{align*}
&\text{t \& t2 \rightarrow t3} \quad <\text{concatenate}> \quad \{A,B\} \& \{C,A\} \rightarrow \{A,B,C,A\} \\
&\text{t \&+ t2 \rightarrow t3} \quad <\text{union}> \quad \{A,B\} \&+ \{C,A\} \rightarrow \{A,B,C\} \\
&\text{t \&- t2 \rightarrow t3} \quad <\text{difference}> \quad \{A,B\} \&- \{C,A\} \rightarrow \{B\} \\
&\text{t \&* t2 \rightarrow t3} \quad <\text{intersection}> \quad \{A,B\} \&* \{B,A\} \rightarrow \{A,B\} \\
&\text{t \&/ t2 \rightarrow t3} \quad <\text{compress}> \quad \{1,0\} \&/ \{A,B\} \rightarrow \{A\} \\
&\text{t \&<< t2 \rightarrow t3} \quad <\text{membership}> \quad \{A,B\} \&<< \{C,A\} \rightarrow \{1,0\} \\
&\text{t \&<= t2 \rightarrow b} \quad <\text{subset}> \quad \{A,B\} \&<= \{A,B,C\} \rightarrow 1 \\
&\text{t \&< t2 \rightarrow b} \quad <\text{proper subset}> \quad \{A,B\} \&< \{A,B\} \rightarrow 0 \\
&\text{t \&= t2 \rightarrow b} \quad <\text{equal}> \quad \{A,B\} \&= \{A,B,C\} \rightarrow 0 \\
&\text{t \&!! t2 \rightarrow b} \quad <\text{disjoint}> \quad \{A,B\} \&!! \{C,D\} \rightarrow 1 \\
&\text{v \&: t \rightarrow b} \quad <\text{inclusion}> \quad A \&: \{A,B\} \rightarrow 1 \\
&\text{hd t \rightarrow v} \quad <\text{head}> \quad \text{hd} \{A,B,C\} \rightarrow A \\
&\text{tl t \rightarrow t2} \quad <\text{tail}> \quad \text{tl} \{A,B,C\} \rightarrow \{B,C\} \\
&\text{len t \rightarrow i} \quad <\text{length}> \quad \text{len} \{A,B,C\} \rightarrow 3 \\
&\text{sortu t \rightarrow t2} \quad <\text{sort up}> \quad \text{sortu} \{B,C,A\} \rightarrow \{A,B,C\} \\
&\text{sortd t \rightarrow t2} \quad <\text{sort down}> \quad \text{sortd} \{A,C,B\} \rightarrow \{C,B,A\}
\end{align*}
\]

The following examples illustrate the set of operators on records.

\[
\begin{align*}
&\text{r \& r2 \rightarrow r3} \quad <\text{concatenation}> \quad \{A:7\}& \{B:8\} \rightarrow \{A:7,B:8\} \\
&\text{t .: t2 \rightarrow r} \quad <\text{binding}> \quad \{A,B\} .: \{7,8\} \rightarrow \{A:7,B:8\} \\
&\text{rge r \rightarrow t} \quad <\text{range}> \quad \text{rge} \{A:7,B:8\} \rightarrow \{7,8\} \\
&\text{dom r \rightarrow t} \quad <\text{domain}> \quad \text{dom} \{A:7,B:8\} \rightarrow \{A,B\}
\end{align*}
\]
The ! operator causes expression e to be evaluated in the environment r, which is always a record consisting of name-value pairs. The activation record that contains the name-value bindings of the current environment of execution is always referenced by the special variable env.
The data sublanguage used to define SIDAL is described in this chapter. The statements that compose the language are listed below, grouped so as to indicate their provenance: The first four statements are the data manipulation primitives proposed by Beck [3, 4]; the next three statements are additional primitives required for SIDAL in particular; and the last group contains the usual conditional, serial, and assignment statements provided by languages like Algol and Pascal. Expressions may include various operations on tuples and data records as well as the usual arithmetic, relational, and logical operators; these new operations are summarized at the end of this chapter.

add r to R1
for each r in R1 do s rof
for each r in R1 with p do s rof
if r in R1 with p then s else s fi
form R1 from D
verify r in R1
update t from t2 ignoring a
if p then s else s2 fi
if p then s fi
s ; s2
v := e

We now proceed to define each of the statements above by means of a semantically equivalent SIBYL expression accompanied by an informal description.
It should be noted that the use of the symbols &< and &= in this chapter is
different from that explained in Chapter 3. In the following discussion, &< is the "lexicographic less than" operator. Symbol &= denotes "lexicographic
equality". The operator &< compares two tuples by comparing the first pair of
arguments that differ. These are converted to strings if necessary before the
final comparison is made. Thus {0,F} is less than {0,G} since 'F' is less
than 'G'. The evaluation of the &= is similar to that of &<.

4.1 Formal Specification of the Semantics

4.1.1 Beck's Data Manipulation Primitives

add r to RI

\[ \text{rge } r \rightarrow \text{RI.tuples;} \]

\[ \text{(RI.keys } /= \text{ } \{\} \rightarrow \text{sort(RI.tuples,[x(RI.keys) } \&< \text{ y(RI.keys)])}) \]

The tuple described by the data record r is inserted into relation RI,
preserving the sort of RI if RI has keys. If, for example, \(r = \{\text{AGE: 12, WT:}
120\})\), the tuple \(\{12,120\}\) will be inserted.

for each r in RI do s rof

\[ \text{(RI.tuples } @ \rightarrow r; \text{RI.doms } : r \rightarrow r; s) \]

The operator .: constructs a record by binding each element of the first list
with the corresponding element of the second list. This results in name-value
pairs. Each such pair represents a field of the record. For each tuple of
Rl, the corresponding data record is constructed and assigned to r, and statement s is executed.

for each r in Rl with p do s rof

(Rl.tuples @ -> r; Rl.doms : r -> r; (p => s))

For each tuple of Rl that satisfies predicate p, the corresponding data record is constructed and assigned to r, and statement s is executed.

if r in Rl with p then s else s2 fi

((Rl.tuples @ -> r; Rl.doms : r -> r; p => 1) = 1 => s , s2)

If there is a tuple in Rl that satisfies predicate p, the data record for one such tuple is constructed and assigned to r, and statement s is executed; otherwise statement s2 is executed.

4.1.2 Additional SIDAL Primitives

form Rl from D

{type:'REL',doms:{},keys:{},status:'T',last:{},constraint:[1],
 numtuples:0,
 tuples:[]} -> Rl;

extract(D) -> Rl(doms,keys)

Extract from descriptor D a list of domains and a list of keys and use these to define an otherwise empty temporary relation Rl whose constraint always returns 1 (true).
verify $r$ in $R_l$

$$(\text{member}(r,R_l\text{.doms});\ (r \ & \ \text{env})!R_l\text{.constraint} \Rightarrow \% \ \Rightarrow \ r)$$

Successive components of the tuple described by data record $r$ are examined and set to $\%$ if they fail to meet the requirements of the corresponding domain. If the resulting tuple fails to satisfy the constraint for relation $R_l$, it is flagged as an error and ignored. Also if a key field of the tuple becomes $\%$, it is flagged as an error and ignored.

update $t$ from $t_2$ ignoring $a$

$\text{update}(t,t_2,a)$

Those components of tuple $t_2$ that are different from atom $a$ are used to replace the corresponding components of tuple $t$. Tuples of $t$ that correspond to a components of $t_2$ are not changed.

NOTE: The description above defines a mapping from our data sublanguage into SIBYL that preserves meaning but increases complexity. This is fine as long as the mapping is only used as a semantic definition. If we wanted to map our sublanguage into SIBYL with little decrease in readability, we would define each sublanguage statement with a SIBYL procedure call. For example, we could replace $\text{exp}$ in the definition

$$\text{for each } r \text{ in } R_l \text{ do s rof} \ \text{exp}$$
by the procedure call

```
foreach ( r, Rl, [s] )
```

where procedure foreach is defined by

```
foreach: [ (r,Rl,s) & (en:env) $ <exp with en\!s in place of s> ]
```

Like the statement s in this example, predicates and expressions must be enclosed in square brackets in the procedure call (to prevent their premature evaluation) and preceded by en! in the procedure definition. Otherwise the construction of a procedure is straightforward.

### 4.1.3 Additional notes

Of the remaining statements, only assignment requires further explanation. In the assignment statement \( v := e \), \( e \) is an expression and \( v \) a variable. A variable can be a name like \( r \), a qualified name like \( r.a \), or a compound variable like \( (r.a, r.b) \) or its equivalent \( r(a,b) \). Compound variables can be used to move selected subfields from one record to another using a single assignment statement, as in

```
r(x,y,z) := r2(a,b,c)
```

In the definitions above, we have referred to predicates \( p \) and expressions \( e \) without further explanation. Since a predicate is just an expression that returns the Boolean value 1 (true) or 0 (false), it is enough to describe the class of expressions:

An expression \( e \) is a sequence of operators and operands: variables, literal
atoms, tuples, domain lists, records, and subexpressions of the various forms shown below.

\[(e) \rightarrow v\]  \text{\textless\text{subexpression}\textgreater}

\[(e, e_2, \ldots) \rightarrow (v, v_2, \ldots)\]  \text{\textless\text{list expression}\textgreater}

\[(\text{name}\::e, \ldots) \rightarrow (\text{name}\::v, \ldots)\]  \text{\textless\text{record expression}\textgreater}

Here \(e, e_2, \ldots\) are expressions and \(v, v_2, \ldots\) are their values.

All combinations of operands and operators are permitted by the syntax of SIBYL; those expressions that have no semantic meaning return the null value \(\%\).
5 OPERATIONS ON RELATIONS

In this chapter we give a formal description of the semantics of the relational operators introduced in Chapter 2. The description is based on a data sublanguage proposed by Beck [3, 4], augmented by a set of operators on tuples and descriptors. This language is described in chapter 4, and is itself formally defined by devolution into the experimental programming language SIBYL. SIBYL has been used to provide semantic definitions for several other existing relational query languages [10]. Since SIBYL is described (and formally defined) elsewhere [13], we will limit ourselves here to a brief informal commentary. Note that since both the intermediate data sublanguage and SIDAL itself are defined as extensions of SIBYL, the full power of the SIBYL language can be carried up into SIDAL simply by relaxing the restrictions of the SIDAL grammar.

We will first summarize the relational operators described in this section and then give a more detailed description of each of them. In our description, we will adopt the following conventions:

R1, R2, R3 denotes a relation.
D denotes a descriptor, eg. {{COST}, CODE}.
S denotes a constraint, eg. [COST < 1].
st denotes a status variable, e.g., P.
t denotes a tuple, eg. {5,'AB'}. 
r denotes a data record, eg. {COST:5, CODE:'AB'}. 
d denotes a domain list, eg. {COST, CODE}.

The relational operators are listed below. The range of each operator is
given at the right.

(1) D + st -> R1 <Construct a relation>
(2) R1 + S -> R2 <Constrain its range of values>
(3) R1 + t -> R2 <Insert a tuple>
(4) R1 - t -> R2 <Delete a tuple>
(5) R1 / D -> R2 <Project a relation>
(6) R1 # S -> R2 <Select a subset of a relation>
(7) R1 := R2 -> R3 <Assign a relation to a variable>
(8) R1 * R2 -> R3 <Join two relations together>
(9) R1 + R2 -> R3 <Form the set union of two relations>
(10) R1 - R2 -> R3 <Form the set difference>

Each SIDAL operator is defined in the following sections. The operators are defined by means of devolution into the Process Definition Language explained in Chapter 4. An informal description of the Process Definition Language code is also given for each operator.

It should be noted that the symbol := appearing in the Process Definition Language code is different from the assignment operator, :=, of SIDAL. The former was defined in Section 4.1.3.
5.1 Construction

\[ \text{st } D \rightarrow R_l \]

\[
\begin{align*}
\text{form } R_l \text{ from } D; \\
R_l.\text{status} & := \text{st}
\end{align*}
\]

To construct a relation, the descriptor and the status need to be specified. The descriptor contains the domain-names in the order in which they will appear in the relation. The key-domains are bracketed in the descriptor. If the key consists of more than one domain, the order of these domains in the descriptor is significant. Possible status types are R (Read-only), S (Stable), P (Permanent) and T (Temporary).

eg., \[ SP := P \{ \{SN, PN\}, QTY \}; \]

The above statement would construct a relation, \( SP \), with domains \( SN, PN \) and \( QTY \). The key is composed of \( SN \) and \( PN \). This relation would be a Permanent one.
5.2 Constraint

R1 + S -> R2

\begin{align*}
R2 & := R; \\
R2\text{.constraint} & := S
\end{align*}

To impose integrity restrictions on the tuples in a relation, a constraint can be specified. The constraint can have restrictions on any or all the domains of the relation. This constraint is used to validate the effect of tuple-operations, specifically to verify a tuple prior to insertion in the target relation. If a tuple does not satisfy the constraint predicate it is discarded.

eg., \( SP := SP + [ \text{SN} > 'S0' \lor \text{SN} <= 'S9' ] \);

The above constraint insures that all tuples in relation \( SP \) have \( SN \) values ranging from \( S1 \) through \( S9 \).
5.3 Insertion

R1 + t -> R2

\[
\begin{align*}
R2.\text{doms} & := R1.\text{doms}; \\
R2.\text{keys} & := R1.\text{keys}; \\
R2.\text{status} & := R1.\text{status}; \\
R2.\text{constraint} & := R1.\text{constraint}; \\
R2.\text{tuples} & := \{\}; \\
\text{doms} & := R2.\text{doms}; \\
\text{keys} & := R2.\text{keys}; \\
\text{update} & R2.\text{last} \text{ from } t \text{ ignoring } ";
\end{align*}
\]

r2 := doms .: R2.\text{last};

if keys = {} then

R2.\text{tuples} := R1.\text{tuples};

\text{verify r2 in R2};

if r2 /= % then add r2 to R2 fi

else

for r in R1 with NOT r(\text{keys}) \&= r2(\text{keys}) do

\text{add r to R2}

\text{rnof};

if r in R1 with r(\text{keys}) \&= r2(\text{keys}) then

tuple := r(\text{doms});

\text{update tuple from r2(\text{doms}) ignoring %};

r2 := doms .: tuple;

\text{verify r2 in R2};

if r2 /= % then add r2 to R2

\text{else add r to R2 fi}

\text{else verify r2 in R2;}

if r2 /= % then add r2 to R2 fi

fi

When a tuple is inserted into an existing relation, it is extended with null atoms (%) or truncated on the right to fit the relation, verified with respect to the relation constraint and the definitions of its domains, and then inserted into the relation. If the key set is empty, the tuple is simply appended to the existing list of tuples. If the relation is ordered by a nonempty set of keys, the tuple is inserted in the proper position, and if there is an existing tuple with the same key values, it is updated by the new tuple. If the verification test fails for the updated tuple, the original tuple is retained.
5.4 Deletion

R1 - t -> R2

R2.doms := R1.doms;
R2.keys := R1.keys;
R2.status := R1.status;
R2.constraint := R1.constraint;
R2.tuples := {};
update R2.last from t ignoring ";
r2 := R2.doms .: R2.last;
keys := R2.keys;
if keys = {} then R2.tuples := R1.tuples
else
  for each r in R1 do
    if r(keys) &= r2(keys) then
      R2.last := r
    else
      add r to R2
  fi
fi

Tuple-deletion starts with the formation of a target relation. The tuple operand may have " (fill) specified for some of its fields. These fields are assigned values from the "last"-tuple of the relation which contains a copy of the last tuple which was modified, deleted or inserted. A dummy tuple is set up which contains only non-fill values.

Each tuple of the source relation is compared with the dummy tuple. If the key-value of the two tuples match, the tuple is ignored; otherwise the tuple is added to the target relation.

Note that tuples may only be deleted from relations which have keys specified. Also, the key-value of the tuple to be deleted must be specified. This may be done either explicitly or implicitly by knowledge of the "last"-tuple from which the fields with fill-values assume their actual values.

The deletion operation initializes the "last"-tuple of the relation with a copy of the tuple deleted to reflect the occurrence of the above operation.
5.5 Projection

R1 / D -> R2

\[
\begin{align*}
&\text{form } R2 \text{ from } D; \\
&\text{doms} := R2.\text{doms}; \\
&r2 := \text{doms} \cdot %; \\
&\text{for each } r \text{ in } R1 \text{ do} \\
&\quad r2(\text{doms}) := r(\text{doms}); \\
&\quad \text{add } r2 \text{ to } R2 \\
\end{align*}
\]

The Projection operation is used to obtain a subset of the attributes of a relation. The target relation is first constructed from the descriptor specified in the projection operator. Next, tuples are formed from the tuples of the source relation containing only those attributes which appear in the target relation. These "projected" tuples are inserted into the target relation. Note that if a key is desired in the target relation, its component fields are enclosed in braces within the projection descriptor.

e.g. \( SPX := SP / \{\text{PN, QTY}\}; \)

The above operation will form a relation \( SPX \) with fields \( \text{PN} \) and \( \text{QTY} \) and insert into it the projected tuples of \( SP \). In addition, the above relation will have no key, so that duplicate tuples are allowed.

The Projection operator is also utilised to contract the degree of a relation. This can be done by specifying the same relation-name as the source and target relations.

e.g. \( SP := SP / \{\text{SN, PN}\}; \)

The above statement will contract the degree of the relation \( SP \) to include only the \( \text{SN} \) and \( \text{PN} \) fields both of which will constitute the key of the contracted relation.
5.6 Selection

R1 # S -> R2

R2.doms := R1.doms;
R2.keys := R1.keys;
R2.constraint := [1];
R2.status := 'T';
for each r in R1 with (r & env)!S do
    add r to R2
rof

Here env is the current SIBYL activation record.

The Selection operation is used to obtain a subset of the tuples of a relation. The target relation is constructed with the same descriptor as the source relation. Each tuple of the source relation is tested to see if it satisfies the selection predicate. If it does, it is inserted into the target relation; otherwise it is ignored.

e.g. SPS1 := SP # [SN = 'S1'];

The above statement would create a relation SPS1 which will have those tuples of SP which have S1 as the value for the SN field.
5.7 Assignment

R1 := R2 -> R3

    if R1.status = 'R' or R1.status = 'S' then error
    else if R1.status = 'P' then
        if R2.doms = R1.doms AND R2.keys = R1.keys then R1 := R2
        else request user reconfirmation
            if user response = 'yes' then R1 := R2 fi
        R1.status := 'P'
    fi
    else R1 := R2
    fi

R3 := R1

Assignment is carried out by the overseer, and for 'T' type relations involves nothing more than the replacement of the value of the destination variable by the value of the source expression. If, however, the status of the destination variable R1 is 'R' or 'S', the system issues an error message and ignores the command. It should be noted that the abbreviated command += may be used for 'S' type relations to insert new tuples. If the status of the destination relation is 'P', the overseer checks to see if the domains and keys of the operand relations are the same. If they are, the operation is carried out. Otherwise the system prompts the user for reconfirmation. Only if the user response is affirmative is the operation carried out.
5.8 The Join Operator

\[ R_1 \times R_2 \rightarrow R_3 \]

\[
\text{form } R_3 \text{ from } \{ \};  \\
R_3.\text{doms} := R_1.\text{doms} \cup R_2.\text{doms};  \\
R_3.\text{keys} := R_1.\text{keys} \cup R_2.\text{keys};  \\
\text{common} := R_2.\text{doms} \cap R_1.\text{doms};  \\
\text{unique} := R_2.\text{doms} \cap R_1.\text{doms};  \\
r_3 := (\text{common} \cup R_1.\text{doms}) \cup \%
\]

for each \( r \) in \( R_1 \) do

\[
\text{for each } r_2 \text{ in } R_2 \text{ with } r(\text{common}) \cap r_2(\text{common}) \text{ do}
\]

\[
r_3 := r(R_1.\text{doms}) \cup r_2(\text{unique});  \\
\text{add } r_3 \text{ to } R_3
\]

rof

The statement, form \( R_3 \) from \( \{ \} \), will create a relation with "empty" domains and keys fields. These are assigned values in the next two statements.

A join operation is used to obtain the cross-product or subset thereof, of two relations. Tuples of the operand-relations which have identical values for the matching fields are concatenated and inserted into the target relations. In case of non-overlapping descriptors, a cross-product of the tuples of the two operand-relations is obtained.

A target relation is constructed with the union of the fields of the operand-relations as its fields. Its key-set is the union of the key-sets of the operands. The overlapping fields of the operands are determined and referred to as "common". Next, the fields of the second operand-relation that do not appear in the first are found and labelled "unique".

The join operation proceeds by comparing the "common" fields of each tuple of the first operand to the corresponding fields of the second operand-relation's tuples. For each such match, a tuple is formed with values of all the fields of the first operand's tuple and the "unique" fields of the second operand's
tuple. This insures that the "common" fields appear only once in the resultant tuple. The tuple thus formed is inserted into the target relation.

Note that the join operation is based on the set of all "common" fields. A join on a subset of such fields can be obtained by first eliminating the unwanted "common" fields from one of the operand-relation by means of a projection operation. A join of the projected relation with the other operand-relation would give the desired result.
5.9 The Union Operator

R1 + R2 -> R3

form R3 from {};
R3.doms := R1.doms &+ R2.doms;
R3.keys := R1.keys &+ R2.keys;
doms := R3.doms;
keys := R3.keys;
for each r in R1 do
  tuple := r(doms);
  if r2 in R2 with r2(keys) &= r(keys) then
    update tuple from r2(doms) ignoring %
  fi;
  add doms .: tuple to R3
rof;
for each r2 in R2 do
  tuple := r2(doms);
  if r in R1 with r(keys) &= r2(keys) then %
  else add doms .: tuple to R3
  fi
rof

The union operation is used to form the union of the tuples of two relations.

A target relation is constructed with the union of the attributes of the two operand-relations' attributes as its attributes. Its key-set is the union of the key-sets of the operand-relations. This key-set is labelled "keys" and is later used for the comparison of tuples of the two operand-relations.

The operation is handled in two steps: transfer of the tuples of the first operand-relation with or without tuple-modifications; transfer of a subset of the tuples of the second operand-relation.

In the first step, each tuple of the first operand-relation is assigned to a dummy tuple. The tuple is then compared to the tuples of the second operand-relation. If a match based on the values of the "keys"-fields is found, the dummy tuple is modified with the tuple of the second operand-relation and then inserted into the target relation. However, if such a match is not found
dummy tuple is simply inserted without any modifications.

In the second step, each tuple of the second operand-relation is assigned to a dummy tuple. The second operand-relation's tuple is compared to the tuples of the first. If a match based on the "keys"-fields is found, the dummy tuple is ignored otherwise it is inserted into the target relation.

It should be noted that this operation is more general than that proposed by Codd in his Relational Algebra [6]. This generality allows the user to specify tuples with a subset of the relation's attributes. The system automatically assigns null values to the undefined attributes.
5.10 The Difference Operator

R1 - R2 -> R3

form R3 from { };  
R3.doms := R1.doms;  
R3.keys := R1.keys;  
keys := R1.keys &* R2.keys;  
for each r in R1 do  
  if r2 in R2 with r(keys) &= r2(keys) then %  
  else add r to R3  
  fi  
rof

The Difference operation is used to obtain a subset of the tuples of a relation. This subset consists of those tuples of the first operand-relation that do not appear in the second operand-relation.

A target relation is constructed with the same descriptor as the first operand-relation. Next, the intersection of the operand-relations' key-sets is found and labelled "keys".

The difference operation proceeds by comparing each tuple of the first operand-relation with those of the second. If a match based on the value of the "keys"-fields is found, the tuple is ignored otherwise the tuple (of the first operand-relation) is inserted into the target relation.

It should be noted that the result of a difference operation between relations whose key-sets are non-overlapping is the first operand-relation. In case of identical key-sets, the operation is essentially the same as set-difference.
6 THE OVERSEER

In this chapter, a description of the function of the Overseer is given. The Overseer is responsible for the integrity of the database. To perform this task, the Overseer has access to two System's relations which store information about the contents of the database. These relations contain integrity restrictions on domains and relations that appear in the database. These two relations are:

\[
\begin{align*}
DD &: \{ \text{DOMAIN\_NAME}, \text{DESCRIPTION}, \text{TYPE}, \text{LENGTH}, \text{DOM\_PREDICATE} \} \\
RD &: \{ \text{RELATION\_NAME}, \text{STATUS} \}
\end{align*}
\]

These relations are explained in Section 2.3.

In addition to the above two relations, the overseer may access the database to check the validity of certain queries, such as the existence of the domains specified in a projection, before their execution.

The Overseer carries out a number of run-time checks. Such tests can be broken down into two parts: checks based on the "target relation" in an assignment statement and checks based on the operator. The former checks are to insure the relation-status restrictions and the latter are to insure the integrity of the contents of the database. The latter checks also detect certain errors in query formulation prior to its execution.
6.1 Checks based on the Target Relation

These checks are to maintain the relation-status restrictions. These checks also insure the privacy of the restricted-access relations. The operators that invoke this Overseer function are :=, *=, /=, #=, +=, and -=.

The Overseer accesses the RD (Relations Directory) relation to determine if the relation-name specified as the target relation already exists. If it does not, a new relation is created with status 'T' and whose domains are determined by the sequence of operands and operators specified by the expression it is assigned. If however the relation already exists, the Overseer determines its status and then proceeds to check the validity of the specified operations. The status of a relation places restrictions on the operations that are permitted on it.

The following are the different types of relation-status along with descriptions of the restrictions on the operations on them:

Status 'R': "Read-only". Such a relation may not appear as the target relation in an assignment statement. The user may only view its contents by means of a DISPLAY or PRINT command.

Status 'S': "Stable". Relations with status 'S' permit only tuple-insertions. No tuple-updates or tuple-deletions are allowed. If a tuple with key-value identical to an existing tuple is specified for insertion, the system will flag it as an error and ignore the new tuple. It should be noted that tuples may only be inserted one at a time by the += command. Any attempts to delete the relation or change its structure by means of contracting or expanding its degree will also be treated as errors and flagged as
such.

Status 'P' : "Permanent". All tuple-operations are allowed. Operations to contract or expand the degree of, or delete, the relation would be preceded by a reconfirmation from the user. The user will be asked a response (Y or N) prior to execution.

Status 'T' : "Temporary". All tuple-operations, contract, expand and delete commands are allowed.

One other difference between T-type and the others is that the former are not saved at the end of the terminal-session whereas the latter are saved for later use.

6.2 Checks based on the Operator

These checks insure the integrity of the database. This integrity information is stored as domain-predicates and relation-constraints. Furthermore, these checks allow the system to detect certain errors in the query formulation prior to actual execution. The following checks depend on the operator and are described for each operator.

Tuple-deletion : If the user specifies a tuple which does not exist in the relation, the system issues a message.

Tuple-insertion : The Overseer checks the tuple to see if it satisfies the relation-constraint and the domain-predicates prior to insertion. These tests are to guarantee the integrity of the database. If a tuple does not satisfy the relation-constraint or its key-fields do not satisfy the corresponding
domain-predicates, the tuple is flagged as being in error and ignored. However, if a tuple does satisfy the above constraints but does not satisfy some other domain-predicate (for a non-key attribute), it will be inserted with an undefined value (%) for that attribute.

Projection: The Overseer checks to see if the domain-names specified in the projection operator are attributes of the specified relation.

Selection: The Overseer checks to see if the domain-names that appear in the selection-predicate are attributes of the specified relation.

Join: none.

Union: If the result is to be stored in an already existing relation, the Overseer insures that the tuples actually transferred to the target relation satisfy its constraint-clause.

Difference: none.

Contraction: Same as for "projection". Also, the Overseer checks to see if the proposed contraction affects the relations constraint-clause by leaving out an attribute which appears in the constraint. If such is the case, the system will issue an error message and ignore the command.

Expansion: The Overseer checks to see if the attributes to be added have already been defined, i.e., they exist in the DD relation.

Relation-creation: The domain-names that appear in the descriptor must have been previously defined, i.e., they must appear in the DD relation.

Statistical-function calls: The Overseer checks to make sure that the attri-
butes of the argument relation are all of numerical type, i.e., either 'int' or 'num'.

In addition to the above checks, the Overseer also checks for the existence of all operand-relations that appear in a statement. This check is done by consulting the RD relation. If a specified relation-name (unless it is the target relation) does not exist, the Overseer issues an error-message to the user.
A set of example queries are solved in this chapter to provide a comparison of SIDAL with other Relational Data Sublanguages. Some of these queries have been taken from Date [9] and Schubert [16]. These examples illustrate both the syntax and "power" of the different Data Sublanguages. The languages compared with SIDAL are Codd's ALPHA [7, 9, 17], SEQUEL [2, 9, 17], SQUARE [5,17], and QBE (Query By Example) [9, 17, 20].

The queries illustrate both the retrieval and update facilities of the languages. The queries access a simple database whose schema consists of three relations.

1) S (Supplier) with attributes SN (supplier number), SNAME (supplier name), STATUS (status of the supplier), and CITY (location of supplier).

2) P (Parts) with attributes PN (part number), PNAME (name of part), COLOR (color of part), and WEIGHT (weight of part).

3) SP (Supplier-Part Relationship) with attributes SN (supplier number), PN (part number), and QTY (quantity of part supplied by the supplier).

The key attributes of the relations are starred.
7.1 Retrieval Queries

1. Retrieve from S the supplier number and status of suppliers in London. Label the result W.

SIDAL  \[ W = S \{ \text{CITY} = \text{'LONDON'} \} / \{ \text{SN, STATUS} \}; \]

ALPHA  \[ \text{GET} W(\text{S.SN, S.STATUS}): \text{CITY='LONDON'} \]

SEQUEL  \[ W: \text{SELECT SN, STATUS} \]
\[ \text{FROM } S \]
\[ \text{WHERE CITY='LONDON'} \]

SQUARE  \[ W<- S ('LONDON') \]
SN, STATUS CITY

QBE

<table>
<thead>
<tr>
<th>S</th>
<th>SN</th>
<th>SNAME</th>
<th>STATUS</th>
<th>CITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>SX</td>
<td></td>
<td></td>
<td>LONDON</td>
</tr>
</tbody>
</table>

2. Retrieve all of S

SIDAL  \[ \text{DISPLAY S;} \]

ALPHA  \[ \text{GET} W(\text{S}) \]
\[ \text{or} \]
\[ \text{GET} W(\text{S.SN, S.SNAME, S.STATUS, S.CITY}) \]

SEQUEL  \[ \text{SELECT S} \]

SQUARE  S
SN, SNAME, STATUS, CITY

QBE

<table>
<thead>
<tr>
<th>S</th>
<th>SN</th>
<th>SNAME</th>
<th>STATUS</th>
<th>CITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

3. Retrieve supplier numbers of all suppliers in Paris whose status is greater than 20.

SIDAL  \[ \text{DISPLAY S # [CITY='PARIS' \(\backslash\) STATUS>20] / \{SN\};} \]

ALPHA  \[ \text{GET} W(\text{S.SN}): \text{S.CITY='PARIS' \(\backslash\) S.STATUS>20} \]

SEQUEL  \[ \text{SELECT SN} \]
\[ \text{FROM S} \]
\[ \text{WHERE CITIES'PARIS'} \]
\[ \text{AND STATUS>20} \]

SQUARE  S ('PARIS',>20)
SN CITY, STATUS
4. Retrieve supplier number and status of suppliers in Paris, in ascending order of STATUS.

**SIDAL**
DISPLAY S # [CITY=’PARIS’] / { (STATUS, SN) };

**ALPHA**
GET W(S.SN,STATUS): S.CITY=’PARIS’
UP S.STATUS

**SEQUEL**
SELECT SN,STATUS
FROM S
WHERE CITY=’PARIS’
ORDER BY STATUS ASC

5. Retrieve names of suppliers who supply part P2.

**SIDAL**
W1 = SP # [ PN=’P2’ ];
DISPLAY W1 * S / { SNAME };

**ALPHA**
RANGE SP X
GET W(S.SNAME): \[ X.(SN=S.SN \lor X.PN=’P2’) \]

**SEQUEL**
SELECT SNAME
FROM S
WHERE SN = SELECT SN
FROM SP
WHERE PN=’P2’

**SQUARE**
S * SP (’P2’)
SNAME SN SN PN

6. Retrieve names of suppliers who supply red parts.

**SIDAL**
X1 = P # [ COLOR=’RED’ ] / { PN };
X2 = X1 * SP / { SN };
DISPLAY X2 * S / { SNAME };

**ALPHA**
RANGE P X
RANGE SP Y
GET W(S.SNAME):
[ X Y(S.SN=Y.SN \lor Y.PN=X.PN \lor X.COLOR=’RED’) ]
7. Retrieve the names of the suppliers who supply at least one part supplied by S2.

SIDAL
\[ Y_1 = |SP \# [SN='S2']] / \{PN}\];
\[ Y_2 = Y_1 * SP / \{SN}\];
DISPLAY Y2 * S / \{SNAME\};

8. Retrieve the supplier numbers of those suppliers who supply at least all those parts supplied by S2.

SIDAL
\[ T_1 := SP # [SN = 'S2'] / \{PN}\];
T2 := SP / {SN};
T := T1 * T2;
T3 := SP / {SN, PN};
T4 := T - T3;
T5 := T4 / {SN};
DISPLAY T2 - T5;

ALPHA
RANGE P X
RANGE SP Y
RANGE SP Z
GET W(SP, SN) : \forall X \exists Y (Y\cdot SN = 'S2' \land Y\cdot PN = X\cdot PN)
-> \exists Z (Z\cdot SN = S\cdot SN \land X\cdot PN = X\cdot PN))

SEQUEL
SELECT SN
FROM SP
WHERE PN ALL >= ALL
SELECT PN
FROM SP
WHERE SN = 'S2'

SQUARE impossible

QBE
<table>
<thead>
<tr>
<th>SP</th>
<th>SN</th>
<th>PN</th>
<th>QTY</th>
</tr>
</thead>
<tbody>
<tr>
<td>P.X</td>
<td>ALL</td>
<td>PX</td>
<td></td>
</tr>
<tr>
<td>S2</td>
<td>ALL</td>
<td>PX</td>
<td></td>
</tr>
</tbody>
</table>

7.2 Update Queries

9. Change the color of part P2 to yellow.

SIDAL
P = P - {P2} + {P2, "", 'YELLOW', "};

ALPHA
HOLD W(P\cdot PN, P\cdot COLOR) : P\cdot PN = P2
W\cdot COLOR = YELLOW
UPDATE W

SQUARE
-> P ('P2', 'YELLOW')
PN; COLOR

SEQUEL
UPDATE P
SET COLOR = 'YELLOW'
WHERE PN = 'P2'

QBE
<table>
<thead>
<tr>
<th>P</th>
<th>PN</th>
<th>PNAME</th>
<th>COLOR</th>
<th>WEIGHT</th>
</tr>
</thead>
<tbody>
<tr>
<td>P2</td>
<td>RED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>update</td>
<td>P2</td>
<td></td>
<td>YELLOW</td>
<td></td>
</tr>
</tbody>
</table>
10. Increase by 1 the quantity of each part supplied by Sl.

SIDAL impossible.

ALPHA HOLD W(QTY) : SP.SN='Sl'
    W.QTY=W.QTY+1
UPDATE W

SEQUEL UPDATE SP
    SET QTY = QTY+1
    WHERE SN='Sl'

SQUARE -> SP ('Sl',1)
    SN;QTY+

QBE

```
SP | SN | PN | QTY
---|----|----|----
S1 | Q  | Q+1
```

11. Multiply by 2 the quantity of each part supplied by Sl and make sure this number does not exceed 5; if it does, set it equal to 5.

SIDAL impossible.

ALPHA HOLD W(SP.QTY):SP.SN='Sl'
    W.QTY=W.QTY*2
UPDATE W
    HOLD W(SP.QTY) : SP.SN='Sl' \ QTY>5
    W.QTY= 5
END

SEQUEL impossible

SQUARE -> SP ('Sl',2)
    SN;QTY*
    -> SP ('Sl',>5,5)
    SN, QTY;QTY

QBE impossible

12. Delete from S all suppliers in London.

SIDAL S = S - S \ [CITY='LONDON'];

ALPHA HOLD W(S):CITY='LONDON'
UPDATE W

SEQUEL DELETE S
    WHERE CITY='LONDON'
13. Destroy the entire relational table P.

SIDAL  

\[ P = \% ; \]

ALPHA  

HOLD W(P) 

DELETE W

SEQUEL  

DELETE P

SQUARE  

P

QBE  

<table>
<thead>
<tr>
<th>P</th>
<th>PN</th>
<th>PNAME</th>
<th>COLOR</th>
<th>WEIGHT</th>
</tr>
</thead>
<tbody>
<tr>
<td>DELETE</td>
<td>PX</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

14. Remove the attribute COLOR from P.

SIDAL  

\[ P := P / \{\{PN\}, PNAME, WEIGHT\}; \]

ALPHA  

impossible

SEQUEL  

impossible

SQUARE  

impossible

QBE  

impossible

15. Constrain the relation S to suppliers with supplier number between S1 and S10.

SIDAL  

\[ S := S + [\text{SN} \geq 'S1' \land \text{SN} \leq 'S10']; \]

ALPHA  

impossible

SEQUEL  

ASSERT Al ON S : SN BETWEEN S1 AND S10

SQUARE  

impossible

QBE  

<table>
<thead>
<tr>
<th>S</th>
<th>SN</th>
<th>SNAME</th>
<th>STATUS</th>
<th>CITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>I.CONSTR(I.*,U.,)I.</td>
<td>SX &gt;= S1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SX &lt;= S10</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
REFERENCES


A GRAMMAR FOR SIDAL

The following is a complete syntactic specification of the SIDAL data language. Anything suffixed with * may be repeated 0 or more times; anything suffixed with + may be repeated 1 or more times; anything suffixed with ? is optional; the terminal symbols are underlined.

```
session -> command* HALT RUN;

command -> USE filename;
          | COPY ( filename , filename );
          | LOAD ( filename , filename );
          | UNLOAD ( filename , filename );
          | PRINT ( term | relation-function relation-name );
          | DISPLAY ( term | relation-function relation-name );
          | abbrev-assign-command
          | assign-command

filename -> letter (digit | letter)*

relation-function -> DOMAINS | KEYS | STATUS | CONSTRAINT

abbrev-assign-command -> contract | expand | abbrev-constrain
                       | insert-tuple | delete-tuple
                       | shrink | extend | reduce

contract -> relation-name /= descriptor ;
expand -> relation-name *= expression ;
abbrev-constrain -> relation-name += predicate ;
insert-tuple -> relation-name += tuple ;
delete-tuple -> relation-name -= tuple ;
```
shrink -> relation-name #= predicate;

extend -> relation-name += exp2;

reduce -> relation-name -= exp2;

assign-command -> relation-name := (definition | expression | %);

definition -> [ expression ]

expression -> constrain | insert | delete | union |

  | difference | exp2

constrain -> expression + predicate

insert -> expression + tuple

delete -> expression - tuple

union -> expression + exp2

difference -> expression - exp2

exp2 -> project | select | join | term

  project -> exp2 / descriptor

  select -> exp2 # predicate

  join -> exp2 * term

term -> constructor descriptor |

  | aggregate-function ( expression )

  | ( expression )

  | relation-name

constructor -> R | S | P | T

aggregatetime -> MAX | MIN | AVG | SUM | COUNT
predicate -> [ boolean-expression ]
  | CONSTRAINT relation-name
--------
boolean-expression -> 1 | 0
  | boolean-term (logical-op boolean-term)*

boolean-term -> negation? (comparitional | membership)
comparitional -> domain-name comp-op domain-value
membership -> domain-name &: list

comp-op -> < | <= | = | >= | > | /=
list -> { domain-value (, domain-value)* }
logical-op -> /\ | \/
negation -> ~

tuple -> { domain-value (, domain-value)* }
domain-value -> int | dec | str | name | %
int -> digit+
dec -> digit+.digit+
str -> 'char+'

descriptor -> DOMAINS relation-name
  |
| { ( { domain-list } , )* domain-list }
domain-list -> domain-name (, domain-name )*
domain-name -> name
relation-name -> name
name -> letter (underline | digit | letter)*

underline -> _
digit -> 0 | ... | 9
letter -> A | ... | Z
char -> all ASCII characters
B SCENARIO OF A SIDAL SESSION

This document provides a scenario of a SIDAL session in which the user defines domains, creates relations and inserts, updates and retrieves tuples.

It is assumed that the physical database file initially contains only the predefined system relations DD (Domain Dictionary) and RD (Relations Directory). The structure of these relations is as follows.

DD : { (DOMAIN_NAME), DESCRIPTION, TYPE, LENGTH, DOM_PREDICATE }

RD : { (RELATION_NAME), STATUS }

The key attributes are enclosed in the inner level braces.

Note : Lines following the symbol > or the symbol / are entered by the user.

Scenario :

The user logs in on an on-line CYBER terminal :

TERMINAL : 323
SIGNON : 123456789
PASSWORD
XXXXXXXX
TERMINAL : 323, TTY
RECOVER/CHARGE : BILL,DCS,PS0000

$CHARGE,DCS,PS0000.
.
.
.

Next the user makes the SIBYL program locally available by issuing a GET command. SIDAL is essentially a subset of SIBYL.
The user enters the system. The system responds with an information header.

/-SIBYL(WORKIN=<file1>,WORKOUT=<file2>)

SIBYL Interpreter Version 1.1 AUGUST 79
system initialized

The parameter WORKIN specifies the initial SIBYL workspace file which includes the database. If any changes are to be made to the database, parameter WORKOUT specifies the file which will contain the modified database. The latter file should be saved after exiting SIBYL. To use the modified database, the user should specify the modified file as the WORKIN file.

The system provides a default workspace and database file. To use this file, the user should not specify any parameters. The user types only -SIBYL.

The user commences the creation of the database file. The domain definitions are stored in the system relation, DD. The user may view its contents in order to avoid redefinition of domains. The names of domains have to be distinct. The system insures that the name of a domain defined by the user is not already defined in DD. In case it is, the system will issue an error message to the user. To view the contents of DD the user need only issue a DISPLAY command.
The symbol &: is the set inclusion operator.

The domain TYPE in DD includes STR (strings), NAME (names), INT (integers), NUM (decimal numbers), and PRD (predicates enclosed in brackets).

The user may wish to view the relations-directory. This is the relation RD.

Another type of relation-status is 'P' (Permanent). The user is allowed to delete or update the structure of such a relation. The system would prompt the user for reconfirmation prior to execution of such commands. Tuple insertions, updates and deletions are allowed. Yet another type of status is 'T'
(Temporary): the user may specify the status to be 'T' which would insure the relation's existence for the duration of the terminal session; it will not be saved for later use. Such relations are automatically generated when the user assigns a relational expression to a previously undefined relation-name. All updates of the structure and contents are permissible. Only redefinition requires reconfirmation from the user.

In contrast to 'T' type relations, relations with status 'R' or 'S' or 'P' are saved after the terminal session for later use.

The user now inserts tuples in the relation DD to define domains that will be used in the relations that he wishes to create. Tuples are inserted in a "S" type relation one at a time by the += command.

> DD += {SN,'SUPPLIER-NAME',STR,[SN > 'SO' \ SN <= 'S9']};
> DD += {SNAME,'SUPPLIER-NAME',STR,10,[1]};
> DD += {STATUS,'SUPPLIER-STATUS',INT,2,[1]};

** ERROR : DOMAIN 'STATUS' ALREADY DEFINED; NEW DEFINITION IGNORED. **

The user now reenters this domain with a new name:

> DD += {SSTATUS,'SUPPLIER-NAME',INT,2,[1]};
> DD += {CITY,'LOCATION',STR,10,[1]};

The user can now create relations using the above defined domains. As an example, the user creates a "Supplier" relation, SUP, with domains SN, SNAME, SSTATUS and CITY. The user also specifies the primary key of the relation
which in this case is the supplier-number SN. This relation is to be a "permanent" one and no constraint on the tuples is desired. This would be done as follows:

> SUP := P{{SN}, SNAME, SSTATUS, CITY};

The prefix 'P' of the expression is the status of the relation created and assigned the name SUP. The relation-names must be unique.

Once the relation is created, tuples can be inserted as follows. For "P" and "T" type relations a number of tuples can be specified for insertion using a single command. A command is terminated by a semi-colon.

> SUP := SUP + {'S1', 'SMITH', 20, 'LONDON'}
> + {'S2', 'JONES', 10, 'PARIS'}
> + {'S3', 'BLAKE', 30, 'PARIS'}
> + {'S4', 'CLARK', 20, 'LONDON'};

The above step of "tuple-insertions" could have been incorporated within the relation-creation step and is shown here separately for the sake of clarity.

The relation SUP can be displayed to see if the tuples entered were as desired.

> DISPLAY SUP;

<table>
<thead>
<tr>
<th>LINE #</th>
<th>SN</th>
<th>SNAME</th>
<th>STATUS</th>
<th>CITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>S1</td>
<td>SMITH</td>
<td>20</td>
<td>LONDON</td>
</tr>
<tr>
<td>2</td>
<td>S2</td>
<td>JONES</td>
<td>10</td>
<td>PARIS</td>
</tr>
<tr>
<td>3</td>
<td>S3</td>
<td>BLAKE</td>
<td>30</td>
<td>PARIS</td>
</tr>
<tr>
<td>4</td>
<td>S4</td>
<td>CLARK</td>
<td>20</td>
<td>LONDON</td>
</tr>
</tbody>
</table>

Suppose the location of supplier S3 was actually Athens. As the field CITY is not a key-field, this modification can be specified by the following command.
> SUP := SUP + {`S3', `BLAKE', 30, `ATHENS'};

Note that as the other fields are not being modified, the tuple can be more simply represented by use of the placeholder symbol, %, as shown below.

> SUP := SUP + {`S3', %, %, `ATHENS'};

The key-field(s) must always be specified.

The above modification involved only a non-key field. If the key-field(s) needs to be modified, the update is more involved. It essentially calls for the deletion of the erroneous tuple followed by the insertion of the correct tuple. Suppose the key-field of the above tuple, SN, was actually S5 instead of S3. This modification can be specified as follows.

> SUP := SUP - {`S3', `BLAKE', 30, `ATHENS'}  
>          + {`S5', `BLAKE', 30, `ATHENS'};

To delete a tuple only the key-field(s) needs to be specified. This would lead to the following simplification.

> SUP := SUP - {`S3'}  
>          + {`S5', `BLAKE', 30, `ATHENS'};

The use of the fill-atom, ",", leads to the following simplification.

> SUP := SUP - {`S3'}  
>          + {`S5', ",", ","};

The fields with fill-atoms assume their values from the last tuple modified, deleted or inserted. In this case, the tuple that is added will get the field-values from the tuple that is deleted. Hence the above command would
obtain the desired result.

The user can now proceed to define further domains and create relations using the defined domains.

> PARTS := P{(PN),PNAME,COLOR,WEIGHT};

**ERROR : UNDEFINED DOMAINS **

> DD += {PN,'PART-NUMBER',STR,2,[PN > 'P0' /\ PN <= 'P9']};
> DD += {PNAME,'PART-NAME',STR,10,[1]};
> DD += {COLOR,'PART-COLOR',STR,5,[1]};
> DD += {WEIGHT,'PART-WEIGHT',INT,2,[WEIGHT > 0]};
> DD += {QTY,'QUANTITY',INT,2,[QTY > 0]};
> DD += {PRICE,'UNIT-PRICE',NUM,5,[PRICE > 0.00]};

The user now creates the "parts" relation. The first attempt shown above resulted in an error as the domains had not been previously defined. The following command combines the steps of relation-creation and tuple-insertions.

> PARTS := P{(PN),PNAME,COLOR,WEIGHT} + [WEIGHT < 50] +
>     + {'P1', 'NUT', 'RED', 12}
> + { 'P2', 'BOLT', 'GREEN', 17}
> + { 'P3', 'SCREW', 'BLUE', 17}
> + { 'P4', 'SCREW', 'RED', 14}
> + { 'P5', 'CAM', 'BLUE', 12} ;

Now, suppose the user enters a tuple with an erroneous value for WEIGHT which violates the relation-constraint :
A tuple which violates the relation-constraint or whose key-value violates a domain-predicate is ignored.

A domain-value that violates the domain-predicate is replaced by the null-symbol (%) before the tuple is inserted. If the domain is a key-domain, the tuple will not be inserted.

The user next proceeds to create the parts-supplier relation, SP.

```sql
> SP := P{ {SN, PN}, QTY} + [PN<='P5' \ (PN>'P9' \ PN <= 'P9')]
>       + {'S1', 'P1', 3}
>       + {'S1', 'P2', 2}
>       + {'S1', 'P3', 4}
>       + {'S1', 'P4', 2}
>       + {'S1', 'P5', 1}
>       + {'S2', 'P1', 3}
>       + {'S2', 'P2', 4}
>       + {'S3', 'P3', 4}
>       + {'S3', 'P5', 2}
>       + {'S4', 'P4', 3}
```
There is another way of specifying insertions in which the order of the domains in the tuples is not necessarily the same as that in the relation definition. However, this requires the user to explicitly state the order of the domains.

```plaintext
> SP := SP + ( T({PN, SN}, QTY)
> + {'P2', 'S4', 2}
> + {'P5', 'S4', 4} );
```

Note that in the above example the order of PN and SN is interchanged. The system would do the required reordering prior to insertion. Also the above facility would allow initialization of a subset of the domains. The system will then set the other domains to % (undefined).

SIDAL also provides powerful and simple operators to expand or contract the degree of a relation. The following examples provide illustration of such actions by first adding a domain PRICE to SP relation and then removing it. The first operation also initializes the new domain in each tuple to "undefined". As the relation SP is a permanent one, the system will ask the user for his intent prior to carrying out the action.

```plaintext
> SP := SP * ( T(AGE) + {} );

** DO YOU WISH TO CHANGE THE STRUCTURE OF THE RELATION SP ? **

** ANSWER Y OR N **

? Y
```
To remove a domain, a projection over the desired domains is required.

? SP := SP / {{SN, PN}, QTY};

** DO YOU WISH TO CHANGE THE STRUCTURE OF THE RELATION SP ? **

** ANSWER Y OR N **

> Y

SIDAL provides an abbreviation for such commands:

> SP *:= T{AGE} + {%};

This command is equivalent to the example presented before. Such an abbreviation can be used in any command where an operation is performed on the "target" relation.

One other type of modification is "deletion". Suppose the user decides to delete the relation SP. As SP is a permanent relation, the system will ask the user his intent before executing the command.

> SP := % ;

** DO YOU WISH TO DELETE THE RELATION SP ? **

** ANSWER Y OR N **

> Y

In addition to the above operations, SIDAL provides facilities for relational "definitions". Such a definition allows "dynamic" relations whose tuples reflect the changes made to the database. This is in contrast to simple assignments which give a snapshot view of the database. The actual evaluation of the definition takes place when such a relation appears in an expression.
The following example illustrates this facility:

```plaintext
> RDS := [ S # [ SN > 'S2' ] ];
```

The relation RDS may be used in any expression and will be evaluated at that point. Whenever RDS is displayed, all the tuples currently in S that satisfy the predicate [SN > 'S2'] will be displayed.

The user may continue to define more domains, create relations or formulate relational queries using the existing relations. A query using a nonexistent relation results in an error message. For example, if the user specifies the relation SP in a query, it will be flagged as an error as the relation has been deleted and no longer exists.

```plaintext
> DISPLAY SP / {SN, PN};
**ERROR : RELATION SP DOES NOT EXIST**
```

To terminate the session the user enters the HALT RUN command.

```plaintext
> HALT RUN;
Workspace saved
exiting SIBYL
```

Before logging-off the CYBER system, if the user altered the database file, the modified version should be saved for later use.

```
/SAVE,<file2>.
/BYE.
```