The 3rd Year CS Project

Interfacing EDEN with ORACLE

Part I

The Relational Interpreter

Son V Truong

January 1996

Contents

1. Introduction

This project has been divided into two parts. This is the first part, and it is concerned with the specification, design and implementation of the relational translator, forming the front-end to EDEN.

EDDI (EDEN Database Definition Interpreter) is a front-end translator for EDEN. The language accepted by EDDI is a partial relational language based on some of the syntax of a relational language called ISBL.

To introduce you to the fundamental properties of relational algebra, Section 2 describes the Relational Model which encompasses the relational abstraction and the relational algebra.

The main sections are the later ones: Section 3 describes the specification and design of the EDDI language for database manipulations and Section 4 describes its implementation using a lexical analyser (Flex) and a parser (Bison) as the translator for relational statements into EDEN program code.

The final objective of the project is use the ORACLE database facility, so this version of EDDI will be called the pseudo translator EDDI/P. This initial, simple implementation of EDDI only translates the relational code into EDEN manipulation code. The real translator will be called EDDI/R which is described more fully in Part II of this project and is introduced in Section 5. This discusses how EDDI's relational language relates to SQL, and some preliminary thoughts on interfacing EDEN with Oracle, which will be the subject of Part II of this project.

2. The Relational Model

This section contains edited extracts from C.J. Date's "An Introduction to Database Systems" which introduces some fundamental concepts of relations, their properties and the operations which can be performed on them.

2.1. Relational Abstraction

2.1.1. Terminology

Formal Relational Term	Informal Equivalent
relation	table
tuple	row or record
cardinality	number of rows
attribute	column or field
degree	number of column
primary key	unique identifier
domain	pool of legal values

2.1.2. Relations

- **Definition**

A relation R on a collection of domains D_1, D_2, ... , D_n consists of two parts, a heading and a body. (The heading is the row of column heading and the body is the set of data rows.)

The heading consists of a fixed set of attributes, or attribute domain pairs:

$$\{(A_1:D_1),\ (A_2:D_2),\ \ldots\ ,\ (A_n:D_n)\}$$

The A_j's must all be distinct and correspond to exactly one of the underlying domains D_j. The body consists of a time-varying set of tuples, where each tuple in turn consists of a set of attribute-value pairs:

$$\{(A_1:v_{i1}),\ (A_2:v_{i2}),\ \ldots\ ,\ (A_n:v_{in})\}$$

i = 1, ... , m, where m is the number of tuples in the set. In each tuple there is one attribute-value pair $(A_j:v_{ij})$ for each attribute A_j in the heading. n is the cardinality, and m is the degree of the relation R.

- **Kinds of Relation**

Base Relations (real relations) correspond to what SQL calls base tables. **Views (virtual relations)** are named, derived relations that are represented within the system purely by its definition in terms of other named relations - it does not have any separate, distinguishable stored data of its own (unlike a base relation).

2.2. Relational Algebra

The relational algebra is the manipulative part of the relational model. This has two parts, the set of operators and the assignment operation.

2.2.1. The Operators (the Relational Algebra)

SELECT[1]	Extract specified tuples from a specified relation (i.e. restricts the specified relation to just those tuples satisfying a specified condition)
PROJECT	Extracts specified attributes from a specified relation
PRODUCT[2]	Builds a relation from two specified relations consisting of all possible combinations of tuples, one from each of the two relations
UNION	Builds a relation consisting of all tuples appearing in either or both of two relations
INTERSECT	Builds a relation consisting of all tuples appearing in both of two specified relations
DIFFERENCE	Builds a relation consisting of all tuples appearing in the first but not in the second of two relations
JOIN	Builds a relation from two specified relations consisting of all possible combinations of tuples, one from each of the two relation, such as the two tuples contributing to any given combination satisfy some condition
DIVIDE	Takes two relations, one binary on unary, and builds a relation consisting of all values of one attribute of the binary relation that match (in the other attribute) all values in the unary relation

- **Closure**

The closure of all the operations mean that the output from each of the operations is another relation. Since the output of any operation is the same type of object as the input then it can become the input to another operation. In other words, it is possible to write nested relational expressions - that is expressions in which the operands are themselves represented by expression, instead of just by names.

- **Completeness**

The set {select, project, union, difference, product} of relational operators is a *complete set*. Any other relational algebra operations can be expressed as a sequence of operations from this set. For example, intersect can be expressed using union and difference, join can be expressed using product and select, natural join can be expressed as select and project. However, the operations in the examples are so commonly required that they are provided.

- **Union-Compatibility**

Two relation are *union-compatible* if:

(a) they each have the same set of attribute names, and
(b) corresponding attributes.

Union, intersection, and difference all require their operands to be union-compatible. Cartesian product, however, has no such a requirement.

[1] Called RESTRICT in C.J. Date's "An Introduction to Database Systems"
[2] Cartesian Product

- **Set Operations**

Union
The union of two union-compatible relations A and B, A UNION B, is a relation with the same heading as each of A and B and with a body consisting of all tuples t belonging to either A or B (or both).

Intersection
The intersection of two union-compatible relations A and B, A INTERSECT B, is a relation with the same heading as each of A and B and with a body consisting of all tuples t belonging to both A and B.

Difference
The difference between two union-compatible relations A and B, A MINUS B, is a relation with the same heading as each of A and B and with a body consisting of the set of tuples t belonging to A and not to B.

Examples:

relation A

S#	Sname	Status	City
s1	Smith	20	London
s4	Clark	20	London

relation B

S#	Sname	Status	City
s1	Smith	20	London
s2	Jones	10	Paris

A UNION B

S#	Sname	Status	City
s1	Smith	20	London
s4	Clark	20	London
s2	Jones	10	Paris

A INTERSECT B

S#	Sname	Status	City
s1	Smith	20	London

A MINUS B

S#	Sname	Status	City
s4	Clark	20	London

B MINUS A

S#	Sname	Status	City
s2	Jones	10	Paris

- **(Extended) Cartesian Product**
The relational algebra version of the Cartesian product is an extended form of the mathematical operation, in which order pair of tuples is replaced by the single tuple that is the union (set theory sense) of the two relations operated on. That is, given two tuples:

$(A_1:a_1, A_2:a_2, \ldots, A_m:a_m)$ and $(B_1:b_1, B_2:b_2, \ldots, B_n:b_n)$

then the extended Cartesian product of the two, is the single tuple:

$(A_1:a_1, A_2:a_2, \ldots, A_m:a_m, B_1:b_1, B_2:b_2, \ldots, B_n:b_n)$

Also, two relations are product-compatible if and only if their headings are disjoint (having no attribute names in common).

Example:

relation A	S#	relation B	P#	A TIMES B	S#	P#
	s1		p1		s1	p1
	s2		p2		s1	p2
			p3		s1	p3
					s2	p1
					s2	p2
					s2	p3

Although the cartesian product of two relations is not important, it provides an intermediate step for the (theta-)join operation, which is.

- **Associatively and Commutativity**

UNION is associative. If A, B and C are arbitrary "projections" then the expression:

```
(A UNION B) UNION C
```
is equivalent to
```
A UNION (B UNION C)
```

From this property a sequence of unions can be written without parentheses. This is also the case for INTERSECT (intersection) and TIMES (product). However, this is not true for MINUS (difference).

UNION, INTERSECT, and TIMES (but not MINUS) are commutative, that is, the expression:

```
A UNION B
```
is equivalent to
```
B UNION A
```

- **Special Relational Operations**

Selection

Let theta represent any simple scalar comparison operator (for example, =, <>, >, >=, etc). The theta-select of relation A on attributes X and Y:

```
A WHERE X theta Y
```

is a relation with the same heading as A and with a body consisting of the set of all tuples t of A such that the comparison "X theta Y" is true for that tuple t.

A literal value may be specified in place of either attribute X or Y (or both), for example

```
A WHERE X theta literal
```

It is possible to extend the definition unambiguously to a form where the conditional expression in the WHERE clause consists of an arbitrary Boolean combination of such simple comparisons.

```
A WHERE c1 AND c2   is equivalent to   (A WHERE c1) INTERSECT (A WHERE c2)
A WHERE c1 OR c2    is equivalent to   (A WHERE c1) UNION (A WHERE c2)
A WHERE NOT c       is equivalent to    A MINUS (A WHERE c)
```

Examples:

S WHERE CITY = 'London'

S#	SNAME	STATUS	CITY
s1	Smith	20	London
s2	Clark	20	London

P WHERE WEIGHT < 14

P#	COLOR	WEIGHT	CITY
p1	Red	12	London
p2	Blue	12	London

SP WHERE S# = 's1' AND P# = 'p1'

S#	P#	QTY
s1	p1	300

Projection

The projection of relation A on attributes X, Y, ..., Z:

```
A [X,Y, ... ,Z]
```

is a relation with the same heading (X,Y, ... ,Z) and a body consisting of the set of all tuples (X:x, Y:y, ... , Z:z) such that a tuple t appears in A with X-value x, Y-value y, ... , Z-value z.

Example:

S (CITY)

CITY
London
Paris
Athens

P (COLOR,CITY)

COLOR	CITY
Red	London
Green	Paris
Blue	Rome
Blue	Paris

(S WHERE CITY = 'Paris') (S#)

S#
s2
s3

Natural Join

Let relations A have headings (X_1,X_2, ... , X_m, Y_1, Y_2, ... Y_n) and B have headings (Y_1, Y_2, ... , Y_n, Z_1, Z_2, ... , Z_p). Attributes Y1, Y2, ... , Ym are the only common to the two relations, then the natural join of A and B:

```
A JOIN B
```

is a relation with the heading (X,Y,Z). If A and B have no names in common, then A JOIN B is equivalent to A TIMES B. Natural join is both associative and commutative.

Theta Join

Theta-join is intended for those occasions where we need to join two relations on the basis of some condition other than equality.

Let relations A and B be union-compatible, then theta-join on relations A and B is defined to be the result of evaluating the expression:

```
(A TIMES B) WHERE X theta Y
```

It is the relation with the same heading as the Cartesian product of A and B, with a body consisting of the set of all tuples t such that t belongs to the Cartesian product and the condition "X theta Y" is true for that tuple t.

2.2.2. The Assignment Operation

The purpose of the assignment operation is to make it possible to "remember" the value of some algebraic expression, and thereby to change the state of the database. The assignment operation could be used as the basis of such finer-precision operations.

2.3. Characteristics Of Relations[3]

- **Ordering of Tuples in a Relation**

A relation is defined as a set of tuples. Elements of a set have no order, hence tuples in a relation do not have a particular order. A relation definition attempts to represent facts at a logical or abstract level.

- **Ordering of Values within Tuples**

An n-tuple is an ordered list of n values, so ordering of values in a tuple is important.

- **Values in Tuples**

Each value in a tuple is an atomic value. Hence composite multi-valued attributes are not allowed. The value within a particular tuple may be unknown or may not apply to this particular tuple. A special value, called **null** is used for these cases.

- **Key Attributes of a Relation**

A relation is defined as a set of tuples. By definition, all elements of a set are distinct, hence, all tuples in a relation must also be distinct. This means that no two tuples can have the same combination of values for all their attributes.

Usually, there are subsets of attributes of a relation R with the property that no two tuples in any relation instance r of R should have the same combination of values for these attributes. Any such set of attributes is a **superkey** of the relation R.

A **key** K of a relation R is a superkey of R with the additional property that removing any attribute A from K leaves a set of attributes K' that is not a superkey of R, i.e. a key is a minimal set of a superkey.

The value of a key attribute can be used to identify uniquely the tuple in the relation. A key is determined from the meaning of the attributes in the relation. Hence, the property is a time invariant; it must hold when we update operations (see below) on the relation.

No key of a relation can be null. This is because we use the key value to identify individual tuples in a relation; having null values for keys implies that we can not identify some tuples.

- **Update Operations on Relations**

There are three basic update operations on relations: insert, delete, and modify. When ever we apply update operations, we must check that the integrity constraints specified on the relational database are not violated.

[3] Excerpts from Elmasri & Navathe's "Fundamentals of Database Systems"

3. The EDDI Relational Language

3.1. EDDI and ISBL

The interpreter, called **EDDI** (**E**DEN **D**atabase **D**efinition **I**nterpreter), will have a similar syntax to ISBL (Information System Base Language[4]). The correspondence between relational algebra, ISBL and EDDI is given in the table below, where R and S can be any relational expressions, and F a Boolean formula.

Operation	Relational Algebra	ISBL/EDDI
1. Union	$R \cup S$	R + S
2. Difference	$R - S$	R - S
3. Intersection	$R \cap S$	R . S
4. Selection[5]	$\sigma_F(R)$	R : F
5. Projection	$\pi A_1,...,A_n(R)$	R % $A_1,...,A_n$
6. Natural Join	$R \triangleright\triangleleft S$	R * S

The six relational operations above, form the basis of the EDDI language. These six operators, union, difference, intersect, select, project and join are the standard operators of relational algebra (see Section 2 above).

3.1.1. EDDI Relational Algebra Syntax

- **Union** `relation_name + relation_name`
- **Difference** `relation_name - relation_name`
- **Intersection** `relation_name . relation_name`
- **Selection** `relation_name : boolean_formula`
- **Projection** `relation_name % attribute <, attribute>`
- **Natural Join** `relation_name * relation_name`

3.1.2. Assignment and Delayed Evaluation

EDDI will retain the ISBL assignment operator '='. To assign the value of an expression E to a relation named R, we write R = E. ISBL has a delayed evaluation operator N! which causes the binding of relations to names in an expression, until the name on the left of the of the assignment is used, to be delayed. EDDI will not use this operator since this delayed evaluation mechanism is a part of the EDEN programming environment. Instead, and in keeping with the EDEN programming philosophy, EDDI will employ the definition operator 'is'. In EDDI we can define a relation R to be some relational expression RE by writing R is RE. The dependency of the relation R on the relational expression RE is maintained by EDEN, and R remains a definition until it is redefined. When the value of R is needed EDEN will automatically evaluate the expression RE and assigns this to R. Unlike ISBL however, the strength of EDEN programming means that the relation R will always be maintained and is updated with values belonging to the expression RE if changes occurs which affects R.

- **Assignment** relation_name **=** relation_expression
- **Definition** relation_name **is** relation_expression

[4] Developed at IBM UK Science Center and closely approximates relational algebra for query languages
[5] Selection is properly known as restriction. Confusion can arise when using the SQL Select statement.

3.1.2. Closure Property

The closure property of relational algebra is retained in EDDI programming. Operations performed on relations by any of the six relational operators will result in a relation. So compound relational operations can be performed. For example:

```
E is (A + B) . (C % C1, C2);
```

The above statement defines the relation E to be the intersection of the union of relation A and B, and relation resulting in the selection of columns C1 and C2 from a relation C. In other words, we have three tables A, B and C. The table C has columns called C1 and C2. We wish to obtain all the records which are both common to the combined tables A and B and the columns which we have selected from table C.

3.2. EDDI Requirements

For EDDI to be of any use we must have other expressions and operators other than the basic relational expressions and operators. In this section we describe some additional EDDI language requirements.

3.2.1. Relation Declaration and Key Attributes

- **Relation Declaration**

We must be able to define relations to be able to work with them. The syntax for declaring a relation is:

```
relation_name (attribute attribute_type
                    <, attribute attribute_type>)
```

where **attribute** is the name of the column and the **attribute_type** defines the type of data the column will be able to hold (either characters, integers or floating points numbers).

For example, we wish to set up a table called CUSTOMERS with fields CUST_ID and NAME. The type of CUST_ID will be numeric and of NAME will be characters. An EDDI statement for declaring this relation is:

```
CUSTOMERS (CUST_ID int, NAME char);                    (1)
```

This statement will initially set up our desired table with no records. EDDI expression for inserting records or data into tables are described in 2.2.2.

- **Key Atrributes**

We can impose an integrity constraint of a relation when declaring it. We can specify one or more attributes of the relation to form a key of the relation. We do this by the word ‘key’ after the appropriate column and type., e.g.

```
CUSTOMERS (CUST_ID int key, NAME char);                (2)
```

The actual syntax for declaring a relation is then:

```
relation_name (attribute attribute_type [key]
                    <, attribute attribute_type [key]>)
```

3.2.2. Tuple Declaration and Manipulation

A tuple of values is, in a sense, a record of our table. EDDI provides the facility to declare tuples for inserting and removing tuples from tables.

- **Declaring Tuples**

The syntax for declaring a tuple is as follows:

```
tuple_name [attribute_value <, attribute_value >];
```

For example, if we wish to declare three tuples to insert into our relation CUSTOMERS, then we write:

```
smith [1000, "Smith"];                    (3)
jones [2000, "Jones"];                    (4)
brown [3000, "Brown"];                    (5)
```

- **Manipulating Tuples**

There are two main manipulations of tuples in terms of relations; we can either insert a tuple into, or delete a tuple from, a relation.

Insertion

The syntax for inserting a tuple into a relation is:

```
relation_name + { tuple };
```

For example, if we have the relation CUSTOMERS and the tuple **smith**, we would insert it as follows:

```
CUSTOMERS + { smith };                    (6)
```

Or we can insert a tuple by declaring the tuple values directly by writing:

```
CUSTOMERS + {[4000, "White"]};            (7)
```

Deletion

The syntax for deleting a tuple from a relation is:

```
relation_name - { tuple };
```

For example, if the tuple **jones** was in our table CUSTOMERS and we wish to delete this record, we write:

```
CUSTOMERS - { jones };                    (8)
```

EDDI provides the facility to insert or delete multiple tuples. To insert a set of tuples into a relation we write:

```
relation_name + { tuple <, tuple > };
```

and to delete, we write:

```
relation_name - { tuple <, tuple > };
```

Example:

```
CUSTOMERS + { smith, jones, [5000, "Brown"] };        (9)
CUSTOMERS - { smith, jones, [8000, "Black"] };        (10)
```

- **Tuple Manipulation, Assignment and Definition**

The above expressions, when interpreted by EDDI will not be remembered; EDDI will just show (output to screen) the new relation - either with a new tuple inserted or removed. To be able to remember the new relation, tuple manipulations would have to be used with the assignment operator or the definition operator.

For example if we wish the CUSTOMERS relation to be as it is currently, together with the tuple smith inserted then we write:

```
CUSTOMERS = CUSTOMERS + { smith };                    (11)
```

Or CUSTOMERS as the same relational definition together with the new tuple, then we write:

```
CUSTOMERS is CUSTOMERS + { smith };                   (12)
```

- **Creating Relations and Tuple Manipulation**

The problem with the tuple operators, described above, is that type checking is very difficult for EDDI to implement efficiently. This is because the tuple operations can be performed within expressions which, for complex expressions, can be very inefficient for the translator to type check. For example, an expression such as:

```
R = (A + B) + {z,x,y};                                (13)
```

we need to evaluate the union of relation A and B, check to see that all the tuples have the same number of attributes as A union B, check that the values in each tuple has the same type as the attributes, check that none of the tuples is already present in the union of A and B. For the definition:

```
R is (A + B) + {z,x,y};                               (14)
```

this is not possible because A + B has no current value which can be used to check for the tuple insertion. Therefore, you must take care when manipulating tuples using the + and - operators, just because you do not see any warning or error messages does not mean the tuple operation has been successful. However, if there are errors these will be displayed at run time when EDEN interprets the translated instructions.

The safer way to insert and delete tuples into and from a relation is to use the insert (<<) and delete (!!) operators. An errors here will be detected at compilation time by EDDI. The syntax for these two operations are as follows.

Insert `relation_name << tuple <, tuple>;`

Delete `relation_name !! tuple <, tuple>;`

Examples:

Suppose we have a relation R and tuples z and x, then insert is:

```
R << z, x;                                                    (15)
```

and delete is:

```
R !! z, x;                                                    (16)
```

The advantage of using these 'direct' insert/delete operators is that type checking can be easily done since we have only one relation to reference and its current state is readily available.

3.3. EDEN Requirements

For EDDI to work in conjuction with EDEN we will need some additional manipulations on the relations we define. These manipulations are mainly operations which transform relations into data structures which EDEN can understand and manipulate.

3.3.1. Relations into Lists

The end result of any EDDI expression is a relation. The structure of a relation is somehow defined internally, and it is such that the user need not be concerned. However once the user has produced a final relation of which he needs to manipulate in EDEN, then we need to inform him of the structure in which his data is stored.

EDDI provides an operation which extracts the tuples in a relation into an ordered list. The syntax is:

```
relation_name || eden_list_name;
```

The tuples within the relations are sublists in **eden_list_name**, ordered by their key attributes.

Example:

```
> ? CUSTOMERS;

CUST_ID         NAME
1000            Smith
2000            Jones
3000            Brown

>CUSTOMERS || cust;
>%eden
>writeln(cust);

[[1000,"Smith"],[2000,"Jones"],[3000,"Brown"]]
```

3.4. The Relational Language: EDDI

This section describes the relational language accepted by the EDDI translator.
The syntax notation used in this chapter are as follows: syntactic categories are indicated *by italic times roman* type, literal words and characters in `courier` type, and keywords in **bold** type. Alternative categories are listed on separate lines. An optional terminal or non-terminal symbol is indicated by the subscript "opt".

3.4.1. Comments

Comments are arbitrary strings of symbols placed between the delimiters `/*` and `*/`. The whole sequence is equivalent to a white space. Note that /* */ can be nested. It is useful to comment a block of program with comments in it. A whole line can be commented by using // at the begin on the line.

```
/* This is a comment */
/* --- Begin of comment ---
    /* This is a nested comment */
--- End of comment --- */
// This is a line of comment
```

3.4.2. Identifiers

An *identifier* (*name*) consists of a sequence of letters and digits - the first letter must be a letter. An EDDI keyword cannot be used as an identifier. Identifiers are not case-sensitive, for example the identifier 'hello' is the same as 'HELLO' and 'HellO'.
Examples of identifiers are:

hello FOO bAR IF var1 var2

3.4.3. Keywords

The following keywords are reserved for EDDI and may not be used otherwise:

```
char    int     real    null    key     is
and     or      not     %eden   %eddi
```

3.4.4. Data Structures

There are only two data structures that EDDI will recognise: strings and numbers.

Strings (**char**) are sequences of characters, enclosed in quotes:

```
"this is a string"
```

Numbers can be integers (**int**) or floating points (**real**):

```
123 1.23    0.23    1.0 456     4
```

There is a null value, which is denoted by **null.**

3.4.5. Program, Declarations and Statements

An EDDI program is made up of relational declarations and statements. EDDI allows the user to embed EDEN code in the EDDI program. To do this the keyword **%eden** must be written at a beginning of a single line followed by nothing else. Any text that follow this keyword is denoted as EDEN code and will not be parsed by EDDI, but merely passed to EDEN (or standard output). To return to writing EDDI code, the keyword **%eddi** must be written at a beginning of a single line followed by nothing else.

EDDI program:

eddi_program:
    %eddi *eddi_code*
    %eden *eden_code*

eddi_code:
    eddi_declaration
    eddi_statement
    eddi_declaration; eddi_program
    eddi_statement; eddi_program

A declaration is of the form:

eddi_declaration:
    relation_declaration
    tuple_declaration
    relation_declaration; eddi_declaration
    tuple_declaration; eddi_declaration

A statement is of the form:

eddi_statement:
    definition_statement
    assignment_statement
    relational_statement
    definition_statement; eddi_statement
    assignment_statement; eddi_statement
    relational_statement; eddi_statement

3.3.6. Relation and Tuple Declarations

Declaring a relation is equivalent to creating a table. Once a relation is declared it cannot be re-declared. Tuples and tuple declarations behave similarly as relation declarations.

A relation is declared using the following syntax:

relational_declaration:
    identifier (attribute_list)

attribute_list:
    identifier type ***key***$_{opt}$
    identifier type ***key***$_{opt}$, *attribute_list*

type:

char
int
real

A tuple is declared using the following:

tuple_declaration:
identifier [*value_list*]

value_list:
string_value
number_value
null
string_value, value-list
number_value, value_list
null*, value_list*

3.3.7. Insertions and Deletions

Once a relation has been declared then values (tuples) can be inserted into the relation using the following syntax:

insertion_statement:
identifier << *tuple_list;*

tuple_list:
[value_list]
[value_list], tuple_list

Once a relation has been declared, and there are tuples within it then tuple deletion can be performed. The syntax for this is:

deletion_statement:
identifier !! tuple_list;

3.3.8. Definitions and Assignments

An EDDI relational definition has the form:

definition_statement:
identifier ***is*** *relational_expression*

An EDDI relational assignment has the form:

assignment_statement:
identifier = *relational_expression*

3.3.9. Relational Statements

A Relational statement has the form:

relational_statement:
? relational_expression

In other words a relational statement, preceeded by a '?', is a relational expression which is immediately evaluated and the result sent to standard output.

3.3.10. Expressions and Operators

Relational expressions are formed by using identifiers and the relational operators: union (+), difference (-), intersection (.), selection (:), projection (%), and natural join (*). There are also two additional operators which perform tuple insertion (+) and deletion (-) on relations within statements and expressions. A relational expression is of the form:

relational_expression:
- *term*
- *term + relational_expression*
- *term - relational_expression*
- *term . relational_expression*
- *term * relational_expression*

term:
- *identifier*
- *(relational_expression)*
- *term : selection_condition*
- *term % projection_attribute_list*
- *term + { tuple_list }*
- *term - { tuple_list }*

tuple_list:
- *tuple*
- *tuple, tuple_list*

tuple:
- *identifier*
- *[value_list]*

3.3.11. Selection and Projection

These relational operators are unary operators. With a selection expression you need to provide a condition for which the selection of tuples is true. With a projection you must provide the attributes (columns) for the operation to project.

The condition for a selection is of the form:

selection_condition:
- *s_col == s_value*
- *s_col != s_value*
- *s_col < s_value*
- *s_col > s_value*
- *s_col <= s_value*
- *s_col >= s_value*

s_col:
- *identifier*

s_value:
- *string*
- *integer*
- *float*
- ***null***

The attributes for a projection is of the form:

projection_attribute_list:
identifier
identifier, projection_attribute_list

3.3.12. Natural Join

The natural join operation can be performed on any two relations. The join condition is the equivalence condition on the attributes which have the same name in both relations. Once the join has taken place, then one of the duplicate columns is removed. When the two relations have no attributes in common then the result of the natural join is a cartesian product.

relation A

a	**b**	**c**
1	2	3
3	4	2
4	2	3

relation B

a	**d**	**e**
1	3	5
3	7	4
4	2	2

where the underlined attributes of both A and B are the key attributes, then:

`A * B` results in the relation

a	**b**	**c**	**d**	**e**
1	2	3	3	5
3	4	2	7	4
4	2	3	2	2

The cartesian product results when the two relation have no attributes in common:

relation C

a	**b**	**c**
1	2	3
3	4	5
4	5	6

relation D

x	**y**	**z**
7	5	5
8	4	4

then `C * D` is the relation

a	**b**	**c**	**x**	**y**	**z**
1	2	3	7	5	5
1	2	3	8	4	4
3	4	5	7	5	5
3	4	5	8	4	4
4	5	6	7	5	5
4	5	6	8	4	4

Notice in all cases of a natural join the key attributes of the individual tables are retained in the resulting relation.

4. The Implementation of EDDI/P

This is Part I of the **Interfacing EDEN with ORACLE** project and, as explained in the Project Specification document, the first part only involves translating the relational language of EDDI into EDEN program code. In other words this initial EDDI translator is a pseudo translator which does not yet interface with ORACLE, it merely simulates the behaviour of a database through EDEN.

The main task of EDDI/P (the pseudo translator) is to split the input program into tokens, and then parse the input translating it into EDEN code which simulates the real database manipulations.

The task of analysing the input program is implemented using the gnu lexical analyser tool **Flex**. Flex takes a *Lex* specification which is a set of instructions that defines tokens (lexemes) for a given an input string and then creates the C program which does the job. The syntactic analysis is implemented using the gnu tool, **Bison**. Bison takes a *Yacc* specification and creates a C code program for the syntax analysis. Some semantic analysis, error checking, and code translation are done using supporting C routines, called within the action parts of the *Yacc* specification. The *Lex* and *Yacc* specifications and supporting C code are given in Section 4.1. below.

4.1. Lex and Yacc Specifications

The front-end of the EDDI translator is written using *Lex* and *Yacc* specifications for lexical processing of the input program and then parsed using bottom-up LALR(1) grammar rules, respectively. The supporting C functions are called within the *Yacc* specification action parts. These maintain symbol tables, build expressions trees, and perform error checking as well as sending the translated code to standard output. For the *Lex* & *Yacc* Specification see Appendix I-A, and for the supporting C code see Appendix I-B.

4.2. The EDDI/P Interpreter

EDDI/P is an interpreter, so it is used interactively in conjuction with EDEN. However, pre-written EDDI programs can be piped into EDDI/P using the unix commands:

```
tcsh> cat prog.ed | eddip > prog.e
```

where **prog.ed** is the EDDI program file and the translated EDEN code is to be placed in the file **prog.e**. The translation code is sent directly to standard output.

To work with EDDI/P interactively with EDEN the commands:

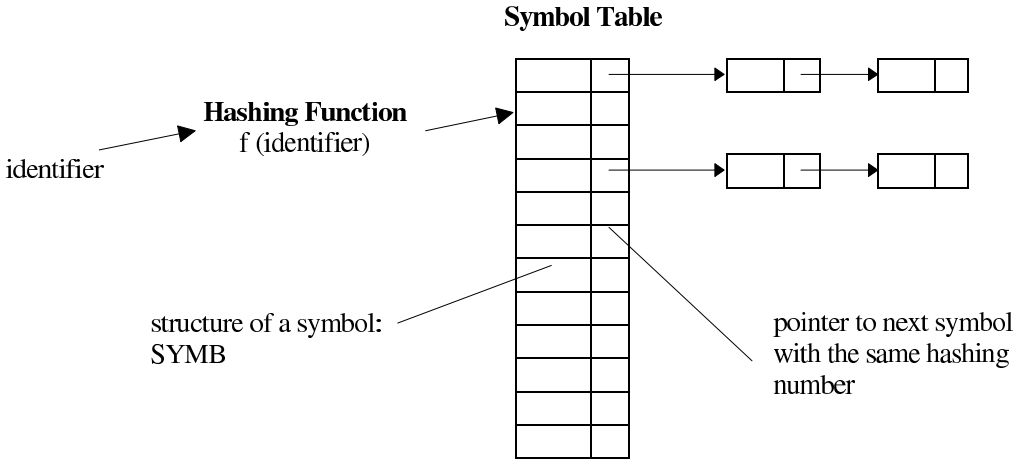
```
tcsh> cat -u prog.ed - | eddip | eden -n
```

are used, where **prog.ed** can be an empty file. It is unlikely that **prog.ed** will be empty because you need to send the EDEN pseudo code to EDEN before the translated EDDI code can be used (see Section 4.3). To exit EDDI/P (a hence end as well) a ctrl-D needs to be used.

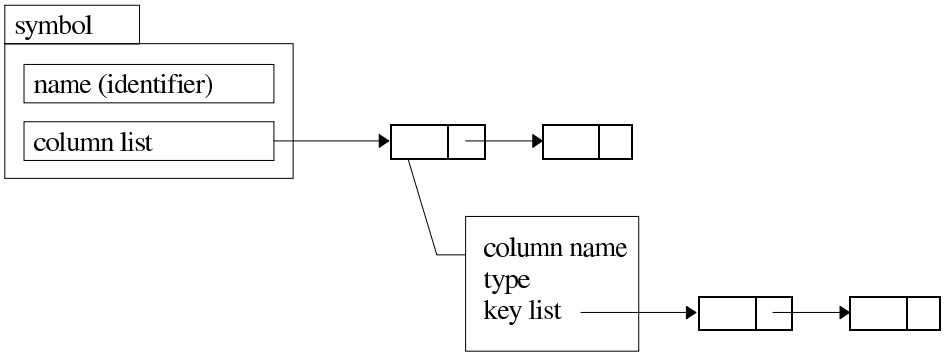
4.3. Symbol Tables and Expression Trees

4.3.1 Symbol Tables

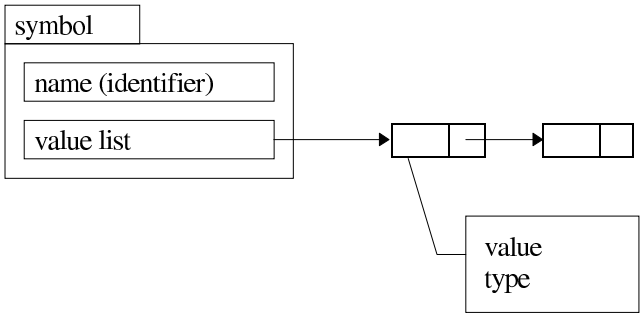
As the EDDI code is parsed, the supporting C code of the *Yacc* specification maintains two symbol tables: one for the relation identifiers and one for the tuple identifiers. The identifiers are placed at a position in the table via a hashing function.



A relational declaration will cause the program to store the identifier of that relation in the relation symbol table. The data kept in the table for a relation is as follows :



A tuple declaration will cause the program to store the identifier of that tuple in the tuple symbol table. The data kept in the table for a tuple is as follows :

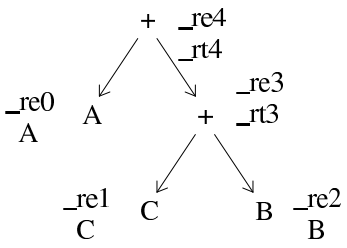


4.3.2 Expression Trees

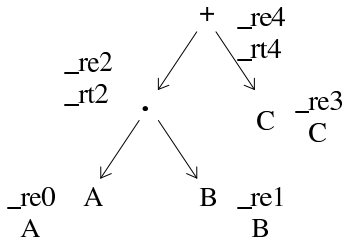
With all relational expressions, an expression tree is built, and relational checks are made, such as union compatibility, and when inserting or deleting tuples, the type and number of the values are checked to see if the action is valid. When a complete expression has been parsed, EDDI will then translate the expression tree into EDEN code. The expression tree is built with nodes of the tree containing the operation to be performed and the leaves containing the names of the relation to be combined. Sub-trees containing sub-expressions are evaluated and the name for an EDEN action procedure is stored in the root node of that sub-tree. The name at the root of the whole expression tree is the name of the final EDEN action procedure.

Examples:

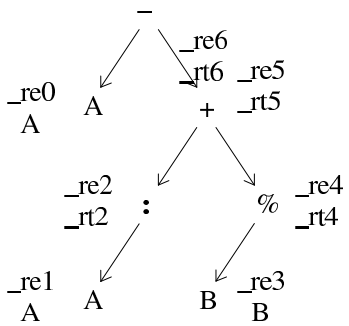
(i) `A + B + C;`



(ii) `(A.B) + C;`



(iii) `A - A : x, y, z + B % x, z;`



4.4. EDDI-EDEN Translation Code

The output of EDDI/P is the EDEN program code which equivalently manipulates data and strutures specified (in Section 3). The code sent to EDEN calls dedicated functions which accesses the ORACLE database facilities, when needed. Otherwise, the translated code uses the facilities provided by EDEN to manipulate data in its own way (mainly to maintain dependency relationships and delay evaluations until the data is needed). This is a transparent task of EDEN with which the user should not be concerned. Using the relational language facility provided by EDDI should be enough for users who want to write programs in an abstractive relational way.

4.4.1. Translation

A program written in the EDDI relational language is piped to EDDI/P which will translate the program into EDEN code. This code is sent to standard output, and then should be piped to EDEN for execution.

```
tcsh> cat -u prog.ed -| eddip | eden -n
```

where **prog.ed** is an EDDI program file.

4.4.2. EDDI-EDEN Functions

An example of EDDI-EDEN translation code is as follows.

```
tcsh> cat fruits.ed

FRUITS (name char key, price num);
FRUITS << [“banana”,0.30];
apple1 [“cox”,0.12];
apple2 [“granny”,0.15];
FRUITS << {apple1,apple2};

tcsh> cat fruits.ed | eddip

FRUITS = create(“name”,”#”,”price”,””);
FRUITS = addvals(FRUITS,[ “banana”,0.30]);
apple1 = [“cox”,0.12];
apple2 = [“granny”,0.15];
FRUITS = addvals(FRUITS,[“cox”,0.12],[“granny”,0.15]);
```

This relatively simple example illustrates the dedicated functions which need to be integrated with EDEN to allow the ORACLE manipulations. From the example above, we can see that functions such as **create** and **addvals** are likely to directly access the ORACLE database. However, simple data manipulation such as declaring tuples can be easily done in EDEN.

The subject of implementing functions such as **create** and **addvals** will be explained in Part II, and an initial discussion is found in Section 5 below. At the moment, and with the initial implementation, we are only concerned with writing some EDEN code which imitate the relational behaviour.

4.4.3. EDEN Programming

Most translated code from EDDI/P to EDEN will define action procedures which calls the dedicated database functions described above. Action procedure is the EDEN programming method of using the dependency maintainer to delay evaluations. The

translation of EDDI expressions into EDEN action procedures is done using the expression tree built while parsing the input program. EDDI keeps a count on the number of nodes in an expression tree it builds and generates names for action procedures. In this way an operation performed on relations in EDDI/P translate to an action definition in EDEN.

The following is an example translation which creates action procedures:

```
tcsh> cat example.ed

R (a num key, b char);
S (a num key, b char);
r [3,"cat"];
s [4,"dog"];

T is (R + {r}) + (S + {s});

tcsh> cat example.ed | eddip

R = create("a","#","b","");
S = create("a","#","b","");
r = [3,"cat"];
s = [4,"dog"];
proc _rt1: R
{
    _re1 = addvals(R, [3,"cat"]);
}
proc _rt2: S
{
    _re2 = addvals(S,[4,"dog"]);
}
proc _rt3: _re1, _re2
{
    _re3 = union(_re1,_re2);
}
T is _re3;
```

4.4. EDDI-EDEN Pseudo Code

With the Part I implementation of the project, the translator EDDI/P is a pseudo translator which does not actually use ORACLE for it database manipulation. Instead these manipulations are simulated using EDEN functions. These functions provide the equivalent activity that ORACLE will need to performed when the actual interfacing implemenation takes place in Part II of this project.

The pseudo database functions are implemented using the list data structure of EDEN to simulate the attributes and tuples of a relational database table. However, this simple implementation of a relational database in the EDEN language provide minimal integrity checks. For example there is no type checking performed when inserting records into tables because no information is held about and attribute's domain. The implemented pseudo functions are the necessary functions for relational expressions, declarations and insertion & deletion. Other forms of data manipulations can be done using the facilities provided by the EDEN language.

The psuedo database functions are not integrated with the EDDI/P translator. They are written in a separate file called **eddipf.e**. When you write an EDDI program, you must

insert these functions into your EDDI program file. Since these functions are written in EDEN they must be surrounded by the special delimiters **%eden**, **%eddi**. For example:

```
%eden
/* eddipf.e file inserted here */

%eddi
/* eddi code follow here */
```

For the EDEN Pseudo Code see Appendix I-C.

5. EDDI and ORACLE

This section introduces some preliminary design thoughts on how to interface EDEN with ORACLE.

The ORACLE DBMS provides an interactive manipulation of tables and records through the use of SQL. This is done through an interpreter called SQL*Plus. This powerful interactive facility is of no use to us when we wish to interface a program we have written with ORACLE. However, like many other DBMS, ORACLE provides various ways of database programming (as opposed to database interaction) such as the OCI (ORACLE Command Interface) commands and the standard techniques of using embedded SQL.

For this project, the technique of using embedded SQL to link EDEN with ORACLE will be used. The main reason for using this technique over OCI commands is that we can translate our required actions (functions) directly into SQL and let the ORACLE pre-compiler do the work of translating the statements into our desired language (in this case C - the language in which EDEN is written).

5.1. EDDI Language & SQL

Both of the EDDI Relational Language and SQL are based on the Relational Model. So initial thoughts of linking EDDI with ORACLE is merely a simple and direct translation. However, since EDDI is based on the Relational Model at a higher level of abstraction than SQL, we will encounter some problems. For example in EDDI we have the natural join operation which SQL does not provide. The advantages of SQL, with respect to EDDI, is that since SQL restrict operations which do not conform to the integrity rules, we can use this facility for error trapping and handling.

*5.2. Pre-Compiler Pro*C*

ORACLE provides several pre-compilers for translating embedded SQL into target languages (ORACLE call these host-languages). The target language we desire is C, hence we use the Pro*C compiler for the translation process.

5.3. Embedded & Dynamic SQL

When using embedded SQL we can freely intermix SQL statements with our host-language statements and use host-language variables in SQL statements. However ORACLE requires the keywords EXEC SQL to begin SQL statements in our host program and end statements with the keyword SQL.

The main weakness of using embedded SQL statements is that they must be a static form of SQL. The static implies that manipulations or transactions are pre-determined before they can be executed. This means that we must know the makeup of each SQL statement that we are going to embed before run time. This requires that we know which SQL commands will be issued and which database tables will be used or modified when a user writes an EDDI program.

If this is the case then the power of EDDI will be very limited. We need to be able to accept and process any valid EDDI statements (defined in Section 3) through EDEN accessing ORACLE. This means that we must be able to manipulate ORACLE at run time. This can be done using dynamic SQL statements. This programming facility is also provided by the ORACLE DBMS, and it works in a similar way, using the pre-compiler, as embedded SQL.

5.4. EDDI, EDEN and ORACLE

We have seen how EDDI/P translates the relational language into EDEN code in this first part of the project. In Part II we use the translated code to access the ORACLE database directly and use some of its facility to do most of the data manipulative work. As mentioned in 4.3.2. we have to write functions such as **create** and **addvals** which manipulate ORACLE database facilities to perform the equivalent actions. We do this using the database programming facility of embedded and dynamic SQL statements, which are translated into C code for us by the pre-compiler. Once we have the C code we need to integrate them with EDEN. These translated C functions effectively replace the pseudo functions (see Appendix I-B) but behave in the same (if not in a very similar) way. The discussion of interfacing EDEN with ORACLE will be examined in detail in Part II: **The EDEN-ORACLE Interface**.

Appendix I-A

Lex & *Yacc* Specifications of EDDI/P (see Section 4.1).

```
/*
        File:           eddip.l
        Program:        EDEN DATABASE DEFINTIION INTERPRETER
                        - Lex Specification
        Author:         Son V Truong
        Date:           5/11/95
        Last Modified:  18/1/96
*/

%{
#define YY_INPUT(buf,result,max_size) \
        { \
         int c = getchar(); \
         result = (c == EOF) ? YY_NULL : (buf[0] = c, 1); \
        }

#include <stdlib.h>

int lines = 1; /* counts number of lines read */
int tokpos = 1; /* counts the token position */
int eof = 0;

%}
a [aA]
b [bB]
c [cC]
d [dD]
e [eE]
f [fF]
g [gG]
h [hH]
i [iI]
j [jJ]
k [kK]
l [lL]
m [mM]
n [nN]
o [oO]
p [pP]
q [qQ]
r [rR]
s [sS]
t [tT]
u [uU]
v [vV]
w [wW]
x [xX]
y [yY]
z [zZ]

space        [ \t]*
nspace       [ \t]+
letter       [a-zA-Z]
digit        [0-9]
ident        {letter}({letter}|{digit})*
string       \"[^\"\n]*\"
litstr        \"[^\"\n ]*\"
int          {digit}+
float        {digit}+"."{digit}+(e("+"|"-")?{digit}+)?
eden         ("%eden"|"%EDEN")
eddi         ("%eddi"|"%EDDI")
icom         "//".*\n
```

```
%x comment edencode eddicode

%%

\n                      { ++lines; tokpos = 0; return(NEWLINE);}

{eden}[ \t]*            { BEGIN(edencode);
                          ++lines; tokpos = 0; return(EDEN);}
{space}"/*"               BEGIN(comment);

<comment>[^*\n]*
<comment>"*"+[^*/\n]*
<comment>\n             { ++lines; tokpos = 0;}
<comment>"*"+"/"          BEGIN(INITIAL);

<edencode>{eddi}[ \t]*  { ++lines; BEGIN (INITIAL); return(EDDI);}
<edencode>\n            { ++lines; ECHO;}
<edencode>.*            { ECHO;}

"(" { ++tokpos; return(LPAREN);}
")" { ++tokpos; return(RPAREN);}
"[" { ++tokpos; return(SLPAR);}
"]" { ++tokpos; return(SRPAR);}
"{" { ++tokpos; return(CLPAR);}
"}" { ++tokpos; return(CRPAR);}

"=" { ++tokpos; return(BEQUAL);}
";" { ++tokpos; return(SCOLON);}
"," { ++tokpos; return(COMMA);}

"*" { ++tokpos; return(JOIN);}
"/" { ++tokpos; return(DIV);}
"+" { ++tokpos; return(UNION);}
"%" { ++tokpos; return(PROJECT);}
":" { ++tokpos; return(SELECT);}
"." { ++tokpos; return(INTER);}
"-" { ++tokpos; return(DIFF);}

"!!" { ++tokpos; return(DELETE);}
"<<" { ++tokpos; return(INSERT);}

"==" { ++tokpos; return(ISEQUAL);}
"<>" { ++tokpos; return(NEQUAL);}
"!=" { ++tokpos; return(NEQUAL);}
"<"  { ++tokpos; return(LESS);}
">"  { ++tokpos; return(GREAT);}
"<=" { ++tokpos; return(LESSET);}
">=" { ++tokpos; return(GREATET);}

"@" { ++tokpos; return(NULL);}
"?" { ++tokpos; return(SHOW);}

{k}{e}{y}    { ++tokpos; return(KEY);}
{i}{s}       { ++tokpos; return(IS);}
{n}{o}{t}    { ++tokpos; return(NOT);}
{a}{n}{d}    { ++tokpos; return(AND);}
{o}{r}       { ++tokpos; return(OR);}
{c}{h}{a}{r} { ++tokpos; return(CHAR);}
{i}{n}{t}    { ++tokpos; return(NUM);}
{r}{e}{a}{l} { ++tokpos; return(REAL);}

{string}     { yylval.s = newstring(yytext);
               ++tokpos; return(STRING);}
{ident}      { yylval.s = newstring(yytext);
               ++tokpos; return(IDEN);}
{int}        { yylval.s = newstring(yytext);
               ++tokpos; return(INT);}
{float}      { yylval.s = newstring(yytext);
               ++tokpos; return(FLOAT);}
{space}
{icom}       { ++lines; tokpos = 0;}
```

```
/*
     File:          eddip.y
     Program:       EDEN DATABASE DEFINITION INTERPRETER
                    - Yacc Specification
     Author:        Son V Truong
     Date:          5/11/95
     Last Modified: 8/1/96
*/

%{
#include <stdio.h>
#include <string.h>
#include "eddipf.h"

%}

%union {
     char      *s;
     int       i;
     TREE      *t;
     CLIST     *c;
     SYMB      *h;
     VAL       *v;
     TLIST     *p;
}

%token <i> JOIN DIV UNION PROJECT SELECT INTER DIFF
%token <i> INSERT DELETE SHOW EPIPE
%token <i> IS BEQUAL NEWLINE SCOLON
%token <i> LPAREN RPAREN SLPAR SRPAR CLPAR CRPAR COMMA
%token <i> ISEQUAL NEQUAL LESS GREAT LESSET GREATET
%token <i> AND OR NOT EDDI EDEN QUIT
%token <i> CHAR NUM REAL DATE KEY NULL ERROR
%token <s> STRING IDEN INT FLOAT EDENCODE

%type <s> relation_decl define_decl assign_decl epipe_decl
%type <s> tuple_decl insert_decl delete_decl
%type <s> relation_stmt ecode eddip
%type <t> term relation_expr
%type <c> attr_list column_list
%type <v> value_list
%type <p> indel_list tuple_list
%type <s> comp_term comp_op comp_expr comparison
%type <i> type

%start ecode
%%
ecode:
                              { ;}
     | ecode eddip            { printf("%s\n",$2);}
     | ecode error NEWLINE    { yyclearin; yyerrok;}

eddip:

       relation_decl    SCOLON    NEWLINE   { $$ = $1;}
     | tuple_decl       SCOLON    NEWLINE   { $$ = $1;}
     | define_decl      SCOLON    NEWLINE   { $$ = $1;}
     | assign_decl      SCOLON    NEWLINE   { $$ = $1;}
     | relation_stmt    SCOLON    NEWLINE   { $$ = $1;}
     | insert_decl      SCOLON    NEWLINE   { $$ = $1;}
     | delete_decl      SCOLON    NEWLINE   { $$ = $1;}
     | epipe_decl       SCOLON    NEWLINE   { $$ = $1;}
     |                  SCOLON    NEWLINE   { $$ = "";}
     | EDEN EDDI        NEWLINE             { $$ = "";}
     |                  NEWLINE             { $$ = "";}
     ;

epipe_decl:

       IDEN EPIPE IDEN  { $$ = epipe($1,$3);}
     ;

insert_decl:

       IDEN INSERT indel_list              { $$ = rinsert($1,$3);}
     | IDEN INSERT CLPAR tuple_list CRPAR  { $$ = rinsert($1,$4);}
     ;
```

```
delete_decl:

       IDEN DELETE indel_list              { $$ = rdelete($1,$3);}
     | IDEN DELETE CLPAR tuple_list CRPAR  { $$ = rdelete($1,$4);}
     ;

relation_decl:

       IDEN LPAREN attr_list RPAREN
          {
            if (checkiden($1))
            {
               if (!checkcols($3))
                    ederror("duplicate column names");
               else
               {
                    rstinsert($1,$3);
                    $$ = relation($1);
               }
            }
            else
               ederror("relation already defined or
                             identifier is used by a tuple");
          }

     | IDEN LPAREN attr_list RPAREN INSERT indel_list
          {
            if (checkiden($1))
            {
               if (!checkcols($3))
                    ederror("duplicate column names");
               else
               {
                    rstinsert($1,$3);
                    $$ = append(relation($1),insert($1,$6));
               }
            }
            else
               ederror("relation already defined or
                             identifier is used by a tuple");
          }
     ;

attr_list:

       IDEN type                     { $$ = mkclist($1,$2,"n",0);}
     | IDEN type COMMA attr_list     { $$ = mkclist($1,$2,"n",$4);}
     | IDEN type KEY                 { $$ = mkclist($1,$2,"k",0);}
     | IDEN type KEY COMMA attr_list { $$ = mkclist($1,$2,"k",$5);}
        ;

type:

       CHAR    { $$ = 0;}
     | NUM     { $$ = 1;}
     | REAL    { $$ = 2;}
     ;

tuple_decl:

       IDEN SLPAR value_list SRPAR
          {
            tstinsert($1,$3);
            $$ = tuple($1,$3);
          }
     ;

tuple_list:

       IDEN                  { $$ = gettuple($1);}
     | IDEN COMMA tuple_list { $$ = linktuple($1,$3);}
     | indel_list            { $$ = $1;}
     ;

indel_list:

       SLPAR value_list SRPAR     { $$ = mktlist($2,0);}
     | SLPAR value_list SRPAR COMMA indel_list
                                  { $$ = mktlist($2,$5);}
     ;
```

```
value_list:

       NULL                  { $$ = mkval(0,0,0);}
     | STRING                { $$ = mkval($1,0,0);}
     | INT                   { $$ = mkval($1,1,0);}
     | FLOAT                 { $$ = mkval($1,2,0);}
     | NULL COMMA value_list { $$ = mkval(0,0,$3);}
     | STRING COMMA value_list    { $$ = mkval($1,0,$3);}
     | INT COMMA value_list       { $$ = mkval($1,1,$3);}
     | FLOAT COMMA value_list     { $$ = mkval($1,2,$3);}
     ;

assign_decl:

       IDEN BEQUAL relation_expr
          { $$ = assign($1,$3);}
     ;

define_decl:

       IDEN IS relation_expr
          { $$ = append(exprtree($3,""),define($1,$3));}
     ;

relation_stmt:

       SHOW relation_expr    { $$ = showrel($2);}
     ;

relation_expr:

       term    { $$ = $1;}
     | term UNION relation_expr
             { $$ = mktree(UNI,0,0,0,$1,$3);}
     | term INTER relation_expr
             { $$ = mktree(INS,0,0,0,$1,$3);}
     | term DIFF relation_expr
             { $$ = mktree(DIF,0,0,0,$1,$3);}
     | term JOIN relation_expr
             { $$ = mktree(JOI,0,0,0,$1,$3);}
     ;

term:
       IDEN    { $$ = mkleaf($1);}
     | LPAREN relation_expr RPAREN
             { $$ = $2;}
     | term PROJECT column_list
             { $$ = mktree(PRO,$3,0,0,$1,0);}
     | term SELECT comparison
             { $$ = mktree(SEL,0,$3,0,$1,0);}

     | term UNION CLPAR tuple_list CRPAR
             { $$ = mktree(TIN,0,0,$4,$1,0);}
     | term DIFF CLPAR tuple_list CRPAR
             { $$ = mktree(TDE,0,0,$4,$1,0);}
     ;

column_list:

       IDEN                       { $$ = mkclist($1,-1,"",0);}
     | IDEN COMMA column_list     { $$ = mkclist($1,-1,"",$3);}
     ;

comparison:

       comp_term comp_op comp_expr
          { $$ = append($1,append(",",append($2,append(",",$3))));}

/*   negation not implemented
     | NOT LPAREN comparison RPAREN */
     ;

comp_expr:
```

```
       comp_term   { $$ = $1;}

/*   logical connectives not implelemted
     | comp_term AND comparison
     | comp_term OR comparison */
     ;



comp_term:

       STRING      { $$ = $1;}
     | INT         { $$ = $1;}
     | FLOAT       { $$ = $1;}
     | IDEN        { $$ = append("\"",append($1,"\""));}
     | NULL        { $$ = "@";}
     ;

comp_op:

       ISEQUAL     { $$ = "\"==\"";}
     | NEQUAL      { $$ = "\"!=\"";}
     | LESS        { $$ = "\"<\"";}
     | GREAT       { $$ = "\">\"";}
     | LESSET      { $$ = "\"<=\"";}
     | GREATET     { $$ = "\">=\"";}
     ;

%%

#include "eddip.lex.c"
#include "eddipf.c"
```

Appendix I-B

Supporting C Code of EDDI/P (see Section 4.1).

```
/*
        File:           eddip.h
        Program:        EDEN DATABASE DEFINTION INTERPRETER

        Author:         Son V Truong
        Date:           5/11/95
        Last Modified:  21/1/96
*/

#define TRUE 1
#define FALSE 0
#define EOS 0
#define HASHSIZE 97

#define REL 0
#define PRO 1
#define SEL 2
#define UNI 3
#define DIF 4
#define INS 5
#define JOI 6
#define TIN 7
#define TDE 8

typedef struct clist
/* column identifier list */
{
     struct clist *next;
     char *ciden;
     int type;
     char *key;
} CLIST;

typedef struct val
/* value list */
{
     struct val *next;
     int type;
     char *value;
} VAL;

typedef struct klist
/* column-value list for keys */
{
     struct klist *next;
     char *column;
     int type;
     VAL *valist;
} KLIST;

typedef struct tlist
/* tuple list */
{
     struct tlist *next;
     VAL *vals;
} TLIST;

typedef struct symb
/* symbol */
{
     struct symb *next;
     char *iden;
     CLIST *cols;
     KLIST *keys;
     VAL *vals;
} SYMB;

typedef struct tree
/* expression tree node */
{
     struct tree *left;
     struct tree *right;
     int optype;
     int eno;
     char *ename;
```

```
    char *rname;
    CLIST *cols;
    TLIST *tup;
    char *comp;
} TREE;

extern void ederror(char *emess);
extern void ierror(char *emess);
extern void *safe_malloc(int n);
extern char* newstring(char *s);
extern char* append(char *s, char *t);

extern SYMB *getsymb(void);
extern void freesymb(SYMB *s);
extern int hash(char *s);
extern void rstinsert(char *rel, CLIST* cols);
extern void tstinsert(char *tuple, VAL* cols);
extern SYMB *rstlookup(char *rel);
extern SYMB *tstlookup(char *tuple);
extern int checkiden(char *rel);

extern CLIST *mkclist(char *ciden, int type, char *key, CLIST *next);
extern VAL *mkval(char *value, int type, VAL *next);
extern VAL *mkvalist(VAL *list1, VAL *list2);
extern TLIST *mktlist(VAL *valist, TLIST *next);
extern int cntvals(VAL *vals);
extern int cntcols(CLIST *cols);
extern VAL *findkeyval(KLIST *keys, char *column);
extern void addkey(VAL *valist, char *value, int type);

extern char *insert(char *rel, TLIST *tuplist);
extern char *rinsert(char *rel, TLIST *tuplist);
extern char *rdelete(char *rel, TLIST *tuplist);

extern char *showrel(TREE *t);
extern char *epipe(char *rel, char *eval);
extern int compare(CLIST *rclist, CLIST *pclist);

extern char *tuple(char *tup, VAL *vals);
extern TLIST *gettuple(char *t);
extern TLIST *linktuple(char *t, TLIST *tlist);

extern char *relation(char *rel);
extern char *eden(char *edencode);
extern TREE *mkleaf(char *rel);
extern char *int2str(int value);
extern char *name(TREE *t);
extern TREE *mktree(int op,CLIST *cols,char *comp,
                    TLIST *tup,TREE *left,TREE *right);

extern char *assign(char *rel, TREE *t);
extern char *define(char *rel, TREE *t);
extern char *exprtree(TREE *t, char *ecs);
extern char *exprtree2(TREE *t, char *ecs);

extern char *edbase(char *edbstr);

SYMB *rst[HASHSIZE]; /* Relation Symbol Table */
SYMB *tst[HASHSIZE]; /* Tuple Symbol Table */
SYMB *symblist;

char *linebuff; /* line buffer for error reporting */
int nameno; /* for naming nodes in expression trees */
```

```
/*
   File:           eddipf.c
   Program:        EDEN DATABASE DEFINITION INTERPRETER

   Author:         Son V Truong
   Date:           5/11/95
   Last Modified:  21/1/96
*/

int main()
     /* main - redefined from the standard main of bison
     nameno is a counter for the expression tree names */
{
  setbuf(stdin, 0);
  setbuf(stdout, 0);
  nameno = 0;
  yyparse();
}

int yyerror(char *s)
     /* yyerror - redefined from the standard yyerror function of bison */
{
  fprintf(stderr,"EDDI/P: PARSE ERROR (at line %i:%i).\n",lines,tokpos);
}

void debug(char *s)
     /* for tracing errors - simply writes to screen a message */
{
  printf("\n");
  printf("%s",s);
  printf("\n");
}

void ederror(char *emess)
     /* simple error handler - prints error message and exits immediately */
{
  fprintf(stderr,"EDDI/P: ERROR (line %i): %s.\n",lines,emess);
}

void error(char *emess)
     /* possible error warning */
{
  printf("EDDI/P: WARNING (line %i): %s.\n",lines,emess);
}

void ierror(char *mess)
     /* interactive error handler */
{
  fprintf(stderr,"EDDI/P: %s (%i:%i).\n",mess,lines,tokpos);
}

void *safe_malloc(int n)
     /* a safe memory allocation function */
{
  void *t = (void*)malloc(n);
  if (t == 0)
    ederror("memory allocation failed");
  return t;
}

char *newstring(char *s)
     /* creates and allocates space for a string */
{
  char *p = (char*)safe_malloc(strlen(s)+1);
  strcpy(p,s);
  return p;
}

char *nullstring(int len)
     /* creates and allocates and empty string give a length size */
{
  int i;
  char *p = (char*)safe_malloc(len);
  for(i=0;i<=len;i++)
    p[i] = '\0';
  return p;
}
```

```
char *append(char *s, char *t)
     /* creates and return the concatenation of strings s and t */
{
  char *temp;
  int length;
  length = strlen(s) + strlen(t) + 1;
  temp = (char*)safe_malloc(length);
  strcpy(temp,s);
  strcat(temp,t);
  cfree(s);
  return temp;
}

SYMB *getsymb(void)
     /* memory saving function which finds a free symbol space before
     allocating a symbol for use */
{
  SYMB *t;
  if (symblist != 0)
  {
    t = symblist;
    symblist = symblist->next;
  }
  else
    t = (SYMB*)safe_malloc(sizeof(SYMB));
  return t;
}

void freesymb(SYMB *s)
     /* allows space of no longer used symbols to used for other symbols */
{
  s->next = symblist;
  symblist = s;
}

int hash(char *s)
     /* hash function to find the storage space in the symbol table using
     the name of the symbol */
{
  int hv = 0;
  int i;
  for (i=0; s[i] != EOS; i++)
  {
    int v = (hv >> 28) ^ (s[i] & 0xf);
    hv = (hv << 4) | v;
  }
  hv = hv & 0x7fffffff;
  return hv % HASHSIZE;
}

void rstinsert(char *rel, CLIST *cols)
     /* inserts a relational identifier into the relational symbol table
     together with its attribute names */
{
  SYMB *s;
  int hv;
  s = getsymb();
  s->iden = rel;
  s->cols = cols;
  s->keys = 0;
  s->vals = 0;
  hv = hash(s->iden);
  s->next = rst[hv];
  rst[hv] = s;
}

void tstinsert(char *tup, VAL *vals)
     /* inserts a tuple identifier into the tuple symbol table together
     with its values    */
{
  SYMB *s;
  int hv;
  s = getsymb();
  s->iden = tup;
  s->vals = vals;
  hv = hash(s->iden);
  s->next = tst[hv];
  tst[hv] = s;
}

SYMB *rstlookup(char *rel)
```

```
/*  finds and returns the symbol of a relation in the symbol table
    using the identifier of the relation */
{
  int hv = hash(rel);
  SYMB *r = rst[hv];
  while (r != 0)
    if (strcmp(r->iden,rel) == 0)
      break;
    else
      r = r->next;
  return r;
}

SYMB *tstlookup(char *tuple)
     /*  finds and returns the symbol of a tuple in the symbol table
      using the idenifier of the tuple */
{
  int hv = hash(tuple);
  SYMB *t = tst[hv];
  while (t != 0)
    if (strcmp(t->iden,tuple) == 0)
      break;
    else
      t = t->next;
  return t;
}

int checkiden(char *rel)
     /* checks to see if identifier for a relation is not already in the symbol
     table (if it is already in the table the relation is already defined */
{
  SYMB *r, *t;
  r = rstlookup(rel);
  t = tstlookup(rel);
  if (t != 0 || r != 0)
    return 0;
  else return 1;
}

CLIST *mkclist(char *ciden, int type, char *key, CLIST *next)
     /* creates and returns a column list node */
{
  CLIST *c = (CLIST*)safe_malloc(sizeof(CLIST));
  c->ciden = newstring(ciden);
  c->type = type;
  c->key = newstring(key);
  c->next = next;
  return c;
}

TLIST *mktlist(VAL *valist, TLIST *next)
     /* creates and returns a tuple list node */
{
  TLIST *p = (TLIST*)safe_malloc(sizeof(TLIST));
  p->next = next;
  p->vals = valist;
  return p;
}

VAL *mkval(char *value, int type, VAL *next)
     /* creates and returns a value list node */
{
  VAL *v;
  v = (VAL*)safe_malloc(sizeof(VAL));
  v->value = newstring(value);
  v->type = type;
  v->next = next;
  return v;
}

VAL *mkvalist(VAL *list1, VAL *list2)
     /* joins two value lists together */
{
  VAL *v1;
  v1 = list1;
  if (v1 != 0)
  {
    while (v1->next != 0)
      v1 = v1->next;
    v1->next = list2;
    return list1;
```

```
  }
  else
    return list2;
}

int cntvals(VAL *vals)
     /* counts the number of values nodes in a value list */
{
  int i;
  VAL *v;
  v = vals;
  i = 0;
  while (v != 0)
  {
    i++;
    v = v->next;
  }
  return i;
}

int cntcols(CLIST *cols)
     /* counts the number of attributes (columns) in a column list */
{
  int i;
  CLIST *c;
  i = 0;
  c = cols;
  while (c != 0)
  {
    i++;
    c = c->next;
  }
  return i;
}

int checkcols(CLIST *cols)
     /* check a column list to see that no two columns have the same name */
{
  CLIST *c1, *c2;
  c1 = cols;
    c2 = cols;
  if (cntcols(cols) > 1)
    while (c1 != 0)
    {
      while (c2 != 0)
      {
     if (!(c1 == c2) && (strcmp(c1->ciden,c2->ciden) == 0))
       return 0;
     c2 = c2->next;
      }
      c1 = c1->next;
      c2 = cols;
    }
  return 1;
}

int cntkeys(KLIST *keys)
     /* counts the number of key attributes in a column list */
{
  int i;
  KLIST *k;
  i = 0;
  k = keys;
  while (k != 0)
  {
    i++;
    k = k->next;
  }
  return i;
}

KLIST *findkey(KLIST *keys, char *column)
/* finds and returns the key value list using the key column name */
{
  KLIST *k;
  k = keys;
  while (k != 0)
  {
    if (strcmp(k->column,column) == 0)
      return (k);
    k = k->next;
```

```
  }
  ederror("key value list missing");
  return 0;
}

VAL *getvals(KLIST *keys)
     /* returns the value list given the key value list */
{
  return (keys->valist);
}

void addkey(VAL *valist, char *value, int type)
/* adds a key value to the key value list */
{
  VAL *x,*y;
  x = valist;
  y = (VAL*)safe_malloc(sizeof(VAL));
  y->value = value;
  y->type = type;
  y->next = 0;
  if (x != 0)
  {
    while (x->next != 0)
         x = x->next;
    x = y;
  }
  else valist = y;
}

char *tuple(char *tup, VAL *valist)
     /* output to eden the code to create a tuple */
{
  char *ecs;
  SYMB *t;
  VAL *v;
  t = tstlookup(tup);
  v = t->vals;
  ecs = newstring(tup);
  if (v != 0)
  {
    ecs = append(ecs," = [");
    while (v->next != 0)
    {
      ecs = append(ecs,v->value);
      ecs = append(ecs,",");
      v = v->next;
    }
    ecs = append(ecs,v->value);
    ecs = append(ecs,"]");
  }
  else ecs = append(ecs," = []");
  ecs = append(ecs,";\n");
  return ecs;
}

char *relation(char *rel)
     /* output to eden the code to create a relation */
{
  char *ecs;
  SYMB *t;
  CLIST *c;
  KLIST *k,*j;
  t = rstlookup(rel);
  c = t->cols;
  j = t->keys;
  ecs = newstring(rel);
  ecs = append(ecs," = create(");
  while (c->next != 0)
  {
    if (strcmp(c->key,"k") == 0)
    {
      k = (KLIST*)safe_malloc(sizeof(KLIST));
      k->column = c->ciden;
      k->type = c->type;
      k->next = 0;
      k->valist = 0;
      if (j == 0)
     t->keys = k;
      else
      {
     while(j->next != 0)
```

```
      j = j->next;
     j->next = k;
      }
    }
    j = t->keys;
    ecs = append(ecs,"\"");
    ecs = append(ecs,c->ciden);
    ecs = append(ecs,"\"");
    ecs = append(ecs,",");
    if (strcmp(c->key,"k") == 0)
      ecs = append(ecs,"\"#\"");
    else
      ecs = append(ecs,"\"\"");
    ecs = append(ecs,",");
    c = c->next;
  }
  if (strcmp(c->key,"k") == 0)
  {
    k = (KLIST*)safe_malloc(sizeof(KLIST));
    k->column = c->ciden;
    k->type = c->type;
    k->next = 0;
    k->valist = 0;
    if (j == 0)
      t->keys = k;
    else
    {
      while(j->next != 0)
     j = j->next;
      j->next = k;
    }
  }
  ecs = append(ecs,"\"");
  ecs = append(ecs,c->ciden);
  ecs = append(ecs,"\"");
  ecs = append(ecs,",");
  if (strcmp(c->key,"k") == 0)
    ecs = append(ecs,"\"#\"");
  else
    ecs = append(ecs,"\"\"");
  ecs = append(ecs,");\n");
  return ecs;
}

void showkeys(SYMB *rel)
     /* output a relation's keys to screen */
{
  SYMB *trel;
  KLIST *keys;
  trel = rel;
  keys = trel->keys;
  while(keys != 0)
  {
    while(keys->valist != 0)
    {
      keys->valist = keys->valist->next;
    }
    keys = keys->next;
  }
}

char *insert(char *rel, TLIST *tuplist)
     /* insert a tuple (list) into a relation and output to eden the code */
{
  SYMB *trel;
  TLIST *tup;
  CLIST *cols;
  KLIST *keys, *kl;
  VAL *vals, *y;
  VAL *keyvals;
  char *ecs;
  int nkeys, ncols, nvals;
  int thatkey, thiskey, samekey, onlykey;
  thatkey = 0;
  thiskey = 0;
  samekey = 0;
  onlykey = 0;
  trel = rstlookup(rel);
  if (trel == 0)
  {
    ederror("relation does not exist");
```

```
  return "";
}
tup = tuplist;
cols = trel->cols;
keys = trel->keys;
if (tup != 0)
{
  vals = tup->vals;
  if (cntvals(vals) != cntcols(cols))
  {
    ederror("incorrect number of values");
    return "";
  }
  ecs = newstring("");
  ecs = append(ecs,rel);
  ecs = append(ecs," = addvals(");
  ecs = append(ecs,rel);
}
else
  return (";\n");
while (tup != 0)
{
  ecs = append(ecs,",[");
  if (cntkeys(keys) > 0)
    onlykey = 1;
  while(vals->next != 0)
  {
    if (cols == 0)
   cols = trel->cols;
    if (cols->type == vals->type)
   if (strcmp(cols->key,"k") == 0)
   {
     kl = findkey(keys,cols->ciden);
     keyvals = getvals(kl);
     if (keyvals != 0)
     {
       if (strcmp(keyvals->value,vals->value) == 0)
         thiskey = 1;
       else
       {
         while (keyvals->next != 0)
         {
        keyvals = keyvals->next;
        if (strcmp(keyvals->value,vals->value) == 0)
        {
          thiskey = 1;
          break;
        }
        else
          thiskey = 0;
         }
       }
       y = (VAL*)safe_malloc(sizeof(VAL));
       y->value = vals->value;
       y->type = vals->type;
       y->next = 0;
       keys = trel->keys;
       kl = findkey(keys,cols->ciden);
       keyvals = getvals(kl);
       while (keyvals->next != 0)
         keyvals = keyvals->next;
       keyvals->next = y;
     }
     else
     {
       y = (VAL*)safe_malloc(sizeof(VAL));
       y->value = vals->value;
       y->type = vals->type;
       y->next = 0;
       kl->valist = y;
     }
     ecs = append(ecs,vals->value);
     ecs = append(ecs,",");
   }
   else
   {
     ecs = append(ecs,vals->value);
     ecs = append(ecs,",");
   }
    else
    {
```

```
 ederror("incompatible types");
 return "";
  }
  samekey = (thatkey || onlykey) && thiskey;
  thatkey = thiskey;
  cols = cols->next;
  vals = vals->next;
  keys = trel->keys;
}
if (cols == 0)
  cols = trel->cols;
if (cols->type == vals->type)
{
  if (strcmp(cols->key,"k") == 0)
  {
 kl = findkey(keys,cols->ciden);
 keyvals = getvals(kl);
 if (keyvals != 0)
 {
   if (strcmp(keyvals->value,vals->value) == 0 && (onlykey || samekey))
   {
     ederror("non-unique values in key columns");
     return "";
   }
   while (keyvals->next != 0)
   {
     keyvals = keyvals->next;
     if (strcmp(keyvals->value,vals->value) == 0 && (onlykey || samekey))
     {
       ederror("non-unique values in key columns");
       return "";
     }
   }
   y = (VAL*)safe_malloc(sizeof(VAL));
   y->value = vals->value;
   y->type = vals->type;
   y->next = 0;
   keys = trel->keys;
   kl = findkey(keys,cols->ciden);
   keyvals = getvals(kl);
   while (keyvals->next != 0)
     keyvals = keyvals->next;
   keyvals->next = y;
 }
 else
 {
   y = (VAL*)safe_malloc(sizeof(VAL));
   y->value = vals->value;
   y->type = vals->type;
   y->next = 0;
   kl->valist = y;
 }
  }
  else
 if (samekey)
 {
   ederror("non-unique values in key columns");
   return "";
 }
}
else
{
  ederror("incompatible types");
  return "";
}
ecs = append(ecs,vals->value);
ecs = append(ecs,"]");
if (tup->next != 0)
{
  tup = tup->next;
  vals = tup->vals;
  cols = trel->cols;
  keys = trel->keys;

  if (cntvals(vals) != cntcols(cols))
  {
 ederror("incorrect number of values");
 return "";
  }
}
else
```

```
    {
      tup = 0;
      ecs = append(ecs,");\n");
    }
  }
  return ecs;
}

char *rinsert(char *rel, TLIST *tuplist)
     /* wrapper for insert function */
{
  return insert(rel,tuplist);
}

char *rdelete(char *rel, TLIST *tuplist)
     /* delete a tuple (list) from a relation and output to eden the code */
{
  SYMB *trel;
  TLIST *tup;
  CLIST *cols;
  KLIST *keys, *kl;
  VAL *vals, *y;
  VAL *keyvals, *prevals;
  char *ecs, *tcs;
  int allmatched, nomatch;
  trel = rstlookup(rel);
  if (trel == 0)
  {
    ederror("relation does not exist");
    return "";
  }
  tup = tuplist;
  cols = trel->cols;
  keys = trel->keys;
  ecs = newstring("");
  ecs = append(ecs,rel);
  ecs = append(ecs," = delvals(");
  ecs = append(ecs,rel);
  ecs = append(ecs,",");
  while (tup->next != 0)
  {
    vals = tup->vals;
    if (cntvals(vals) < cntkeys(keys))
    {
      ederror("incorrect number of (key) values");
      return "";
    }
    tcs = newstring("");
    allmatched = TRUE;
    nomatch = FALSE;
    while (vals->next != 0)
    {
      if (strcmp(cols->key,"k") == 0)
      {
     kl = findkey(keys,cols->ciden);
     keyvals = getvals(kl);
     prevals = 0;
     while (keyvals != 0)
     {
       if (strcmp(keyvals->value,vals->value) == 0)
       {
         tcs = append(tcs,vals->value);
         tcs = append(tcs,",");
         nomatch = FALSE;
         if (prevals == 0)
         {
           kl->valist = keyvals->next;
           cfree(keyvals);
           keyvals = getvals(kl);
         }
         else
         {
           prevals->next = keyvals->next;
           cfree(keyvals);
         }
         break;
       }
       else
         nomatch = TRUE;
       prevals = keyvals;
       keyvals = keyvals->next;
```

```
    }
     }
     else
     {
    tcs = append(tcs,vals->value);
    tcs = append(tcs,",");
    nomatch = FALSE;
     }
     if (nomatch)
     {
    error("tuple does not exist");
    allmatched = FALSE;
     }
     vals = vals->next;
     cols = cols->next;
   }
   if (strcmp(cols->key,"k") == 0)
   {
     kl = findkey(keys,cols->ciden);
     keyvals = getvals(kl);
     prevals = 0;
     while (keyvals != 0)
     {
    if (strcmp(keyvals->value,vals->value) == 0)
    {
      tcs = append(tcs,vals->value);
      nomatch = FALSE;
      if (prevals == 0)
      {
        kl->valist = keyvals->next;
        cfree(keyvals);
        keyvals = getvals(kl);
      }
      else
      {
        prevals->next = keyvals->next;
        cfree(keyvals);
      }
      break;
    }
    else
      nomatch = TRUE;
    prevals = keyvals;
    keyvals = keyvals->next;
     }
   }
   else
   {
     tcs = append(tcs,vals->value);
     nomatch = FALSE;
   }
   if (nomatch)
   {
     error("tuple does not exist");
     allmatched = FALSE;
   }
   if (allmatched)
   {
     ecs = append(ecs,"[");
     ecs = append(ecs,tcs);
     ecs = append(ecs,"],");
   }
   tup = tup->next;
   cols = trel->cols;
 }
 vals = tup->vals;
 cols = trel->cols;
 keys = trel->keys;
 if (cntvals(vals) < cntkeys(keys))
 {
   ederror("incorrect number of (key) values");
   return "";
 }
 tcs = newstring("");
 allmatched = TRUE;
 nomatch = FALSE;
 while (vals->next != 0)
 {
   if (strcmp(cols->key,"k") == 0)
   {
     kl = findkey(keys,cols->ciden);
```

```
    keyvals = getvals(kl);
    prevals = 0;
    while (keyvals != 0)
    {
   if (strcmp(keyvals->value,vals->value) == 0)
   {
     tcs = append(tcs,vals->value);
     tcs = append(tcs,",");
     nomatch = FALSE;
     if (prevals == 0)
     {
       kl->valist = keyvals->next;
       cfree(keyvals);
       keyvals = getvals(kl);
     }
     else
     {
       prevals->next = keyvals->next;
       cfree(keyvals);
     }
     break;
   }
   else
     nomatch = TRUE;
   prevals = keyvals;
   keyvals = keyvals->next;
    }
  }
  else
  {
    tcs = append(tcs,vals->value);
    tcs = append(tcs,",");
    nomatch = FALSE;
  }
  if (nomatch)
  {
    error("tuple does not exist");
    allmatched = FALSE;
  }
  cols = cols->next;
  vals = vals->next;
}
if (strcmp(cols->key,"k") == 0)
{
  kl = findkey(keys,cols->ciden);
  keyvals = getvals(kl);
  prevals = 0;
  while (keyvals != 0)
  {
    if (strcmp(keyvals->value,vals->value) == 0)
    {
   tcs = append(tcs,vals->value);
   nomatch = FALSE;
   if (prevals == 0)
   {
     kl->valist = keyvals->next;
     cfree(keyvals);
     keyvals = getvals(kl);
   }
   else
   {
     prevals->next = keyvals->next;
     cfree(keyvals);
   }
   break;
    }
    else
   nomatch = TRUE;
    prevals = keyvals;
    keyvals = keyvals->next;
  }
}
else
{
  tcs = append(tcs,vals->value);
  nomatch = FALSE;
}
if (nomatch)
{
  error("tuple does not exist");
  allmatched = FALSE;
```

```
  }
  if (allmatched)
  {
    ecs = append(ecs,"[");
    ecs = append(ecs,tcs);
    ecs = append(ecs,"]);\n");
  }
  return ecs;
}

char *showrel(TREE *t)
     /* output to eden code for direct relational statements */
{
  char *ecs;
  ecs = newstring("");
  ecs = append(ecs,exprtree(t,""));
  ecs = append(ecs,"showrel(");
  ecs = append(ecs,t->rname);
  ecs = append(ecs,");\n");
  return ecs;
}

int compare(CLIST *rclist, CLIST *pclist)
     /* compares two attribute lists to see if there are the same */
{
  CLIST *i, *j;
  int found1, found2;
  found1 = FALSE;
  found2 = FALSE;
  j = rclist;
  i = pclist;
  while (i != 0)
  {
    while (j != 0)
      if (strcmp(i->ciden,j->ciden) != 0)
     j = j->next;
      else
      {
     found2 = TRUE;
     break;
      }
    if (found2)
    {
      found1 = TRUE;
      found2 = FALSE;
      i = i->next;
      j = pclist;
    }
    else
    {
      found1 = FALSE;
      break;
    }
  }
  return found1;
}

char *define(char *rel, TREE *t)
     /* output eden code for a relational definition */
{
  char *ecs;
  ecs = newstring(rel);
  ecs = append(ecs," is ");
  ecs = append(ecs,t->rname);
  ecs = append(ecs,";\n");
  return ecs;
}

char *assign(char *rel, TREE *t)
/* output eden code for a relational assignment */
{
    char *ecs, *ecs2;
    ecs = newstring(rel);
    ecs = append(ecs," = ");
    ecs2 = exprtree2(t,"");
    ecs = append(ecs,ecs2);
    ecs = append(ecs,";\n");
    return ecs;
}
/*
char *assign(char *rel, TREE *t)
```

```
      output eden code for a relational assignment
{
  char *ecs;
  ecs = newstring(rel);
  ecs = append(ecs," = ");
  ecs = append(ecs,t->rname);
  ecs = append(ecs,";\n");
  return ecs;
}
*/
char *eden(char *s)
     /* output eden embedded codes */
{
  int i;
  char *t;
  t = (char*)safe_malloc(strlen(s)-5);
  for (i=3; i<=strlen(s)-4; i++)
    t[i-3] = s[i];
  t[i-3] = '\0';
  return t;
}

char *int2str(int value)
     /* convert the counter of the expression tree to a string */
{
  int x;
  char cx;
  char *sx;
  char *numstr;
  numstr = newstring("");
  sx = nullstring(2);
  sx[1] = '\0';
  while(value > 9)
  {
    x = value%10;
    value = value/10;
    cx = (char)(x + 48);
    sx[0] = cx;
    numstr = append(sx,numstr);
    sx = nullstring(2);
    sx[1] = '\0';
  }
  cx = (char)(value + 48);
  sx[0] = cx;
  numstr = append(sx,numstr);
  return numstr;
}

char *name(TREE *t)
     /* returns the identifier of the root node of the expression tree */
{
  return (newstring(t->rname));
}

TREE *mkleaf(char *rel)
     /* creates and returns a leaf node for the expression tree */
{
  TREE *t;
  char *name;
  t = (TREE*)safe_malloc(sizeof(TREE));
  t->left = 0;
  t->right = 0;
  t->optype = 0;
  t->eno = nameno++;
  t->ename = 0;
  t->rname = newstring(rel);
  t->cols = 0;
  t->comp = 0;
  t->tup = 0;
  return t;
}

TREE *mktree(int op,CLIST*cols,char *comp,TLIST *tup,TREE *left,TREE *right)
     /* creates and return a node in the expression tree */
{
  TREE *t;
  char *name;
  t = (TREE*)safe_malloc(sizeof(TREE));
  t->left = left;
  t->right = right;
  t->optype = op;
```

```
  t->eno = nameno++;
  name = int2str(t->eno);
  t->ename = append("_rt",name);
  t->rname = append("_re",name);
  t->cols = cols;
  t->comp = comp;
  t->tup = tup;
  return t;
}

void showtree(TREE *t)
     /* outputs the names of all the nodes in the expression tree */
{
  if (t->left != 0)
    showtree(t->left);
  if (t->right != 0)
    showtree(t->right);
  printf("%s %i\n",t->rname,t->optype);
}

char *exprtree(TREE *t, char *ecs)
     /* outputs eden code for a relational expression */
{

  char *necs, *pecs;
  necs = newstring("");
  pecs = newstring("");
  if (t != 0)
  {
    if (t->left != 0)
      necs = exprtree(t->left,ecs);
    if (t->right != 0)
      pecs = exprtree(t->right,ecs);
    if (t->left != 0 && t->right == 0)
    {
      ecs = append("proc ",t->ename);
      ecs = append(ecs,": ");
      ecs = append(ecs,t->left->rname);
      ecs = append(ecs,"\n{\n\t");
      ecs = append(ecs,t->rname);
      ecs = append(ecs," = ");
      switch(t->optype)
      {
      case PRO:
     ecs = append(ecs,"project(");
     ecs = append(ecs,t->left->rname);
     ecs = append(ecs,",[");
     if (t->cols != 0)
       while (t->cols->next != 0)
       {
         ecs = append(ecs,"\"");
         ecs = append(ecs,t->cols->ciden);
         ecs = append(ecs,"\"");
         ecs = append(ecs,",");
         t->cols = t->cols->next;
       }
     ecs = append(ecs,"\"");
     ecs = append(ecs,t->cols->ciden);
     ecs = append(ecs,"\"");
     ecs = append(ecs,"]");
     break;
      case SEL:
     ecs = append(ecs,"select(");
     ecs = append(ecs,t->left->rname);
     ecs = append(ecs,",");
     ecs = append(ecs,t->comp);
     break;
      case TIN:
     ecs = append(ecs,"addvals(");
     ecs = append(ecs,t->left->rname);
     ecs = append(ecs,",");
     while (t->tup->next != 0)
     {
       ecs = append(ecs,"[");
       while (t->tup->vals->next != 0)
       {
         ecs = append(ecs,t->tup->vals->value);
         ecs = append(ecs,",");
         t->tup->vals = t->tup->vals->next;
       }
       ecs = append(ecs,t->tup->vals->value);
```

```
      ecs = append(ecs,"],");
      t->tup = t->tup->next;
    }
    ecs = append(ecs,"[");
    while (t->tup->vals->next != 0)
    {
      ecs = append(ecs,t->tup->vals->value);
      ecs = append(ecs,",");
      t->tup->vals = t->tup->vals->next;
    }
    ecs = append(ecs,t->tup->vals->value);
    ecs = append(ecs,"]");
    break;
     case TDE:
    ecs = append(ecs,"delvals(");
    ecs = append(ecs,t->left->rname);
    ecs = append(ecs,",");
    while (t->tup->next != 0)
    {
      while (t->tup->vals->next != 0)
      {
        ecs = append(ecs,"[");
        ecs = append(ecs,t->tup->vals->value);
        ecs = append(ecs,",");
        t->tup->vals = t->tup->vals->next;
      }
      ecs = append(ecs,t->tup->vals->value);
      ecs = append(ecs,"],");
      t->tup = t->tup->next;
    }
    while (t->tup->vals->next != 0)
    {
      ecs = append(ecs,"[");
      ecs = append(ecs,t->tup->vals->value);
      ecs = append(ecs,",");
      t->tup->vals = t->tup->vals->next;
    }
    ecs = append(ecs,t->tup->vals->value);
    ecs = append(ecs,"]");
    break;
     }
     ecs = append(ecs,");\n};\n");
   }
   if (t->left != 0 && t->right != 0)
   {
     ecs = append("proc ",t->ename);
     ecs = append(ecs,": ");
     ecs = append(ecs,t->left->rname);
     ecs = append(ecs,",");
     ecs = append(ecs,t->right->rname);
     ecs = append(ecs,"\n{\n\t");
     ecs = append(ecs,t->rname);
     ecs = append(ecs," = ");
     switch(t->optype)
     {
     case UNI:
    ecs = append(ecs,"union(");
    break;
     case INS:
    ecs = append(ecs,"inter(");
    break;
     case DIF:
    ecs = append(ecs,"diff(");
    break;
     case JOI:
    ecs = append(ecs,"njoin(");
    break;
     }
     ecs = append(ecs,t->left->rname);
     ecs = append(ecs,",");
     ecs = append(ecs,t->right->rname);
     ecs = append(ecs,");\n};\n");
   }
  }
  necs = append(necs,pecs);
  return append(necs,ecs);
}

char *exprtree2(TREE *t, char *ecs)
     /* outputs eden code for a relational expression */
{
```

```
char *necs, *pecs, *x;
necs = newstring("");
pecs = newstring("");

if (t != 0)
{
  if (t->right != 0)
    pecs = exprtree2(t->right,ecs);
  if (t->left != 0)
    necs = exprtree2(t->left,ecs);
  if (t->left != 0 && t->right == 0)
  {
    switch(t->optype)
    {
    case PRO:
   x = newstring("project(");
   x = append(x,t->left->rname);
   x = append(x,",[");
   if (t->cols != 0)
     while (t->cols->next != 0)
     {
       x = append(x,"\"");
       x = append(x,t->cols->ciden);
       x = append(x,"\"");
       x = append(x,",");
       t->cols = t->cols->next;
     }
   x = append(x,"\"");
   x = append(x,t->cols->ciden);
   x = append(x,"\"");
   x = append(x,"]");
   break;
    case SEL:
   x = newstring("select(");
   x = append(x,t->left->rname);
   x = append(x,",");
   x = append(x,t->comp);
   break;
    case TIN:
   x = newstring("addvals(");
   x = append(x,t->left->rname);
   x = append(x,",");
   while (t->tup->next != 0)
   {
     x = append(x,"[");
     while (t->tup->vals->next != 0)
     {
       x = append(x,t->tup->vals->value);
       x = append(x,",");
       t->tup->vals = t->tup->vals->next;
     }
     x = append(x,t->tup->vals->value);
     x = append(x,"],");
     t->tup = t->tup->next;
   }
   x = append(x,"[");
   while (t->tup->vals->next != 0)
   {
     x = append(x,t->tup->vals->value);
     x = append(x,",");
     t->tup->vals = t->tup->vals->next;
   }
   x = append(x,t->tup->vals->value);
   break;
    case TDE:
   x = newstring("delvals(");
   x = append(x,t->left->rname);
   x = append(x,",");
   while (t->tup->next != 0)
   {
     while (t->tup->vals->next != 0)
     {
       x = append(x,"[");
       x = append(x,t->tup->vals->value);
       x = append(x,",");
       t->tup->vals = t->tup->vals->next;
     }
     x = append(x,t->tup->vals->value);
     x = append(x,"],");
     t->tup = t->tup->next;
```

```
     }
     while (t->tup->vals->next != 0)
     {
       x = append(x,"[");
       x = append(x,t->tup->vals->value);
       x = append(x,",");
       t->tup->vals = t->tup->vals->next;
     }
     x = append(x,t->tup->vals->value);
     break;
      }
      x = append(x,"])");
      t->rname = x;
    }
    if (t->left != 0 && t->right != 0)
    {
      switch(t->optype)
      {
      case UNI:
     x = newstring("union(");
     break;
      case INS:
     x = newstring("inter(");
     break;
      case DIF:
     x = newstring("diff(");
     break;
      case JOI:
     x = newstring("njoin(");
     break;
      }
      x = append(x,t->left->rname);
      x = append(x,",");
      x = append(x,t->right->rname);
      x = append(x,")");
      t->rname = x;
    }
  }
  return (t->rname);
}

TLIST *gettuple(char *t)
     /* returns a tuple list by retrieving it from the tuple symbol table */
{
  SYMB *s;
  TLIST *tl;
  s = tstlookup(t);
  if (s->vals != 0)
  {
    tl = mktlist(s->vals,0);
    return tl;
  }
  else
  {
    ederror("tuple does not exist");
     return 0;
    }
}

TLIST *linktuple(char *t, TLIST *tlist)
     /* returns a tuple list by retrieving it from the tuple symbol table
     then joining it with another tuple list    */
{
  SYMB *s;
  TLIST *tl, *ttl;
  s = tstlookup(t);
  ttl = tlist;
  if (s->vals != 0)
  {
    tl = mktlist(s->vals,tlist);
    return tl;
  }
  else
  {
    ederror("tuple does not exist");
    return 0;
  }
}

char *epipe(char *rel, char *eval)
{
```

```
  char *ecs;
  ecs = newstring(eval);
  ecs = append(ecs," = epipe(");
  ecs = append(ecs,rel);
  ecs = append(ecs,");");
  return ecs;
}

char *edbase(char *edbstr)
{
  int i,j,k,len,diff;
  char *dname;
  len = strlen(edbstr);
  for(i=0;i<len;i++)
    if(edbstr[i]=='"')
    {
      i++;
      break;
    }
  if(i!=len)
    for(j=i+1;j<len;j++)
      if(edbstr[j]=='"')
     break;
  diff = j-i;
     dname = nullstring(diff+1);
  for(k=i;k<j;k++)
    dname[k-i] = edbstr[k];

  return dname;
}
```

Appendix I-C

EDDI Pseudo Functions (see 4.4.1).

```
%eden
/*
     File:          eddipf.e
     Program:  EDEN functions for EDDI/P
     Date:          6/2/96
     Author:        Son V Truong
*/


func create
/* table = (c1,k/n,c2,k/n,...,cN,k/n) */
{
     auto i, tlist;
     tlist = [[[]]];
     for (i=1; i<=$#; i+=2)
     {
          append tlist[1],$[i];
          if ($[i+1] == "#")
               append tlist[1][1],1;
          else
               append tlist[1][1],0;
     }
     return tlist;
}

func ucompat
/* bool = ucompat(t1,t2) */
{
     return ($1[1][1] == $2[1][1]);
}

func notin
/* bool = notin(t,v) */
{
     auto i,j;
     for (i=1; i<=$1#; i++)
     {
          if ($1[i] == $2)
               return 0;
     }
     return 1;
}

func showrel
/* showrel(t) */
{
     auto i,j;
     if ($1 != [])
     {
          for (i=2; i<=$1[1]#; i++)
                {
                        write("----------------");
                }
          writeln("");
          for (i=2; i<=$1[1]#-1; i++)
          {
               write($1[1][i]);
               write("\t\t");
          }
          writeln($1[1][i]);
          for (i=2; i<=$1[1]#; i++)
                {
                        write("----------------");
                }
          writeln("");
          for (j=2; j<=$1#; j++)
          {
               for (i=1; i<=$1[1]#-2; i++)
                  {
                          write($1[j][i]);
                          write("\t\t");
                   }
                 writeln($1[j][i]);
          }
          for (i=2; i<=$1[1]#; i++)
```

```
                {
                        write("----------------");
                }
          writeln("");
     }
     else
          writeln("EDEN: table is empty.");

}

func addvals
/* table2 = addvals(t1,tup1,tup2,...,tupN) */
{
     auto i;
        for (i=2; i<=$#; i++)
     {
          if ($1[1]#-1 == $[i]#)
          {
               if (notin($1,$[i])==1)
                    append $1,$[i];
               else
                    writeln("EDDI/EDEN ERROR: tuple exists already.");
          }
          else
               writeln("EDEN: ERROR: incorrect number of values.");
     }
     return $1;
}

func delvals
/* table2 = delvals(t1,v1,v2,...,vN) */
{
     auto i,j;
     for (i=2; i<=$#; i++)
          if ($1[1]#-1 == $[i]#)
          {
               for (j=2; j<=$1#; j++)
                    if ($1[j] == $[i])
                         delete $1,j;
          }
          else
          {
               writeln("EDEN: ERROR: incorrect number of values.");
               return $1;
          }
     return $1;
}

func union
/* table3 = union(table1,table2) */
{
     auto i;
     if (ucompat($1,$2) != 1)
     {
          writeln("EDEN: ERROR: tables are not union-compatible.");
          return [[[]]];
     }
     else
     {
          for (i=2; i<=$2#; i++)
               append $1,$2[i];
          return $1;
     }

}

func getindex
/* index = getindex(t) */
{
     return $1[1][1];
}

func getkeys
/* keys = getkeys(t) */
{
     auto i,j, index;
     key = [];
     keylist = [];
     index = getindex($1);
     for (j=2; j<=$1#; j++)
     {
```

```
          for (i=1; i<=index#; i++)
               if (index[i] == 1)
                    append key, $1[j][i];
          append keylist, key;
          key = [];
     }
     return keylist;
}

func diff
/* t3 = diff(t1,t2) */
{
     auto i, j, k;
     tlist = $1;
     dlist = [];
        if (ucompat($1,$2) != 1)
        {
                writeln("EDEN: ERROR: tables are not union-compatible.");
                return [[[]]];
        }
     else
     {
          for (j=2; j<=$1#; j++)
               for (k=2; k<=$2#; k++)
                    if ($1[j] == $2[k])
                         append dlist, $1[j];
          for(i=1;i<=dlist#;i++)
               tlist = delvals(tlist,dlist[i]);
          return tlist;
     }
}

func inter
/* t3 = inter(t1,t2) */
{
        auto i, j, k;
        tlist = [];
        posint = [];
        dv = [];
        if (ucompat($1,$2) != 1)
        {
                writeln("EDEN: ERROR: tables are not union-compatible.");
                return [[[]]];
        }
     else
     {
          append tlist, $1[1];
             for (j=2; j<=$1#; j++)
                  for (k=2; k<=$2#; k++)
                            if (($1[j] == $2[k]) && (j != k))
                                     append tlist, $1[j];
             return tlist;
     }
}

func getcolumn
/* c_number = getcolumn(t,c) */
{
     auto i,j;
     j = 0;
     for (i=2; i<=$1[1]#; i++)
          if ($1[1][i] == $2)
          {
               j = i;
               break;
          }
     if (j != 0)
          return j-1;
     else
     {
          writeln("EDEN: ERROR: column not found.");
          return 0;
     }
}

func project
/* t2 = project(t1,c_list) */
{
     auto i,j,k;
     tlist = [[[]]];
     for (i=1; i<=$2#; i++)
```

```
          append tlist[1],$2[i];
        for (i=2; i<=$1#; i++)
              append tlist,[];
     for (i=1; i<=$2#; i++)
     {
          addcol = getcolumn($1,$2[i]);
          append tlist[1][1], $1[1][1][addcol];
          for (j=2; j<=$1#; j++)
               append tlist[j], $1[j][addcol];
     }
     return tlist;
}

func getcolval
{
     auto i,j;
     tlist = [];
     for (i=2; i<=$1#; i++)
          append tlist,$1[i][$2];
     return tlist;
}

func select
/* t2 = select(t1,c,c_op,c_val) */
{
     auto i,j,k,addcol;
     tlist1 = [];
     append tlist1, $1[1];
     keys = getcolval($1,getcolumn($1,$2));
     switch($3)
     {
          case "==" :
          {
               for (i=1; i<=keys#; i++)
               {
                    if (keys[i] == $4)
                         append tlist1,$1[i+1];
               }
               break;
          }
          case "!=" :
          {
               for (i=1; i<=keys#; i++)
               {
                    if (keys[i] != $4)
                         append tlist1,$1[i+1];
               }
               break;
          }
          case "<" :
          {
               for (i=1; i<=keys#; i++)
               {
                    if (keys[i] < $4)
                         append tlist1,$1[i+1];
               }
               break;
          }
          case "<=" :
          {
               for (i=1; i<=keys#; i++)
               {
                    if (keys[i] <= $4)
                         append tlist1,$1[i+1];
               }
               break;
          }
          case ">" :
          {
               for (i=1; i<=keys#; i++)
               {
                    if (keys[i] > $4)
                         append tlist1,$1[i+1];
               }
               break;
          }
          case ">=" :
          {
               for (i=1; i<=keys#; i++)
               {
                    if (keys[i] >= $4)
```

```
                    append tlist1,$1[i+1];
               }
               break;
          }
          default : writeln("EDDI/P Error: not a comparison.");
     }
     return tlist1;
}

func getkcols
{
     auto i;
     tlist = [];
     for (i=1; i<=$1[1][1]#; i++)
          if ($1[1][1][i] == 1)
               append tlist, $1[1][i+1];
     return tlist;
}

func rdupcols
{
     auto i,j,k;
     tlist = [[[]]];
     colnos = [2];
     delcols = [$1[1][2]];
     for (i=3; i<=$1[1]#; i++)
          if (notin(delcols,$1[1][i]))
          {
               append colnos, i;
               append delcols, $1[1][i];
          }
     for (j=1; j<=colnos#; j++)
     {
          append tlist[1][1], $1[1][1][colnos[j]-1];
          append tlist[1], $1[1][colnos[j]];
     }
     for (k=2; k<=$1#; k++)
     {
          append tlist, [];
          for (j=1; j<=colnos#; j++)
               append tlist[k], $1[k][colnos[j]-1];
     }
     return tlist;
}

func matchcols
/* [[mct1],[mct2]] = matchcols(t1,t2) */
{
     auto i,j;
     mct1 = [];
     mct2 = [];
     for(i=2;i<=$1[1]#;i++)
          for(j=2;j<=$2[1]#;j++)
          {
               if ($1[1][i] == $2[1][j])
               {
                    append mct1,i;
                    append mct2,j;
               }
          }
     return [mct1,mct2];
}

func njoin
{
        auto i,j,k,x,y,z;
        tlist1 = [];
     vals1 = [];
     vals2 = [];
     nvals1 = [];
     nvals2 = [];
        append tlist1, $1[1];
     matcols = matchcols($1,$2);
     if (matcols != [[],[]])
     {
          for (i=1; i<=matcols[1]#; i++)
               append vals1, getcolval($1,matcols[1][i]-1);
          for (i=1; i<=matcols[2]#; i++)
               append vals2, getcolval($2,matcols[2][i]-1);
          for (j=1; j<=vals1[1]#; j++)
          {
```

```
               append nvals1, [];
               for (k=1; k<=vals1#; k++)
                    append nvals1[j], vals1[k][j];
          }
          for (j=1; j<=vals2[1]#; j++)
             {
                     append nvals2, [];
                for (k=1; k<=vals2#; k++)
                          append nvals2[j], vals2[k][j];
          }
          attr = $1[1];
          attr[1] = attr[1] // $2[1][1];
          for (i=2; i<=$2[1]#; i++)
               append attr, $2[1][i];
          tlist = [attr];
          z = 1;
          for (x=1; x<=nvals1#; x++)
               for (y=1; y<=nvals2#; y++)
                    if (nvals1[x] == nvals2[y])
                    {
                         append tlist, $1[x+1];
                         z++;
                         for (i=1; i<=$2[y+1]#; i++)
                              append tlist[z], $2[y+1][i];
                    }
          return rdupcols(tlist);
     }
     else
     {
          attr = $1[1];
          attr[1] = attr[1] // $2[1][1];
          for (i=2; i<=$2[1]#; i++)
                         append attr, $2[1][i];
            tlist = [attr];
          for (x=2;x<=$1#;x++)
               for (y=2; y<=$2#; y++)
               {
                    append tlist, ($1[x] // $2[y]);
               }
          return tlist;
     }
}

func epipe
{
     auto i;
     tlist = [];
     for(i=2;i<=$1#;i++)
          append tlist,$1[i];
     return tlist;
}

%eddi
```