Rosetta: A Generator of Data Language Compilers*

E. E. Villarreal
Computer Science Department
Cal Poly State University
San Luis Obispo, CA 93407

Don Batory
Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712

April 4, 1996

Abstract

A data language is a declarative language that enables database users to access and manipulate data. There are families of related data languages; each family member is targeted for a particular application. Unfortunately, building compilers for such languages is largely an ad hoc process; there are no tools and design methods that allow programmers to leverage the design and code of compilers for similar languages, or to simplify the evolution of existing languages to include more features.

Rosetta is a generator of relational data language compilers that demonstrates practical solutions to these problems. We explain how domain analysis identifies primitive building blocks of these compilers, and how grammar-based definitions (à la GenVoca) of the legal compositions of these blocks yields compact and easily-evolvable specifications of data languages. Rosetta automatically transforms such specifications into compilers. Experiences with Rosetta are discussed.

1 Introduction

Families of functionally and syntactically similar languages are prevalent today because languages have a history of development. The family or *domain* of non-object-oriented imperative programming languages (e.g., Pascal [JW78], Algol [ISO72], C [KR78], etc.) is one example; the domain of relational data languages (e.g., SQL [vdL89], Quel [Dat87], TQuel [Sno87]) is another.

Clearly, there will always be families of related languages, if only because no single language suits all situations. This is particularly evident in contemporary database research. To

*This work was supported in part by Texas Instruments, The University of Texas Applied Research Labs, Schlumberger, and Microsoft Research.

meet the demanding needs of geographic and temporal databases, for example, new data languages are frequently proposed. Few are wholly new: most extend common languages such as SQL or Quel. Fewer still are implemented. Database practitioners demand that researchers go beyond the proposal stage to have hands-on experience evaluating their languages. Tools and methodologies that enable building language families (or evolving existing languages) inexpensively and quickly are nonexistent and are sorely needed [Sal95]. Our system, Rosetta, is a generator of data language compilers that addresses precisely these needs of database researchers. In this paper, we explain the two software engineering concepts on which Rosetta is based.

First, there is a *backplane* or *virtual machine* of operators on relations (e.g., tuple selection, modification, deletion, etc.) that underlies the domain of data languages. Backplane operators are primitive units of computation that are shared by many or all languages in the domain. A well-designed backplane insulates operator implementation details from its client data languages. Thus, a variety of different implementations of operators can be explored without sacrificing the ability of the backplane to support families of diverse data languages.

Second, there exist primitive building-blocks for language construction. From our analysis, we have identified three classes of building-blocks: one class maps language syntax to backplane operators; another builds internal representations of backplane operator trees; and a third manages contextual information.

Rosetta exploits these two concepts to deliver data language extensibility through software building blocks. Rosetta is a *GenVoca generator* [BO92]; it relies on a grammar-based specification to define the legal combinations of building blocks. The syntax that is used to invoke a particular combination (and thus define the actions of a specific data language statement) is specified by instantiating parameters of these blocks. Rosetta generates a compiler for a target language from such specifications.

Although we believe the concepts of language extensibility on which Rosetta is based are domain-independent (and hence may lead to compiler generators for languages in other domains, such as the domain of object-oriented programming languages), in this paper we focus only on their applicability to relational data languages. We begin with a brief overview of the domain of relational data languages, and then progressively develop our model of Rosetta.

2 Overview of Relational Data Languages

A *data language* of a database management system (DBMS) is a declarative language that enables database users to access and manipulate data. Although early DBMSs featured proprietary data languages, most DBMSs today offer a dialect of SQL or Quel.

Contemporary data languages share much functionality. All provide ways for users to retrieve, update, insert, and delete tuples, but the manner in which these statements are expressed can differ markedly. The most obvious differences are syntactic. For example, to print the tuples of a relation R(a,b,c), only one SQL statement is needed, whereas Quel requires two. The SQL and Quel statements in Table 1a display all attributes of tuples that satisfy the predicate $R.a > R.b$. As another example, the SQL and Quel statements in Table 1b replace the original value of attribute $R.a$ with $R.a*10$ for tuples that satisfy the predicate $R.a > R.b$.

SELECT *	RANGE OF S IS R	UPDATE R	RANGE OF S IS R
FROM R		SET R.a = R.a * 10	
WHERE R.a > R.b	RETRIEVE S.a, S.b, S.c	WHERE R.a > R.b	REPLACE S (S.a = S.a * 10)
	WHERE S.a > S.b		WHERE S.a > S.b

(a) access in SQL and Quel (b) update in SQL and Quel

Table 1: Data Manipulation Statements in SQL vs. Quel

Nontrivial semantic differences also exist. For example, SQL and Quel differ in the options in pre-processing the inputs for aggregation operators (such as *avg()*, *max()*, etc.). The SQL SELECT statement may invoke multiple aggregations but all operate on the same set of tuples, possibly grouped on some set of attributes. Quel aggregation is more general: different aggregations operate on the same tuples, but each may specify its own grouping.

Non-traditional data languages are SQL and Quel with application-specific extensions. For example, the temporal data languages TQuel[Sno87] and TSQL2[Sno94] share several temporal data types such as events (points in time) and time intervals. There are functionalities that TQuel has that TSQL2 doesn't have (e.g., the *coalescing* of tuples with overlapping time intervals), and vice versa.

Data languages are often built from scratch. There are no tools or design methods that allow programmers to leverage previous designs and implementations. As a consequence, data language design and construction is ad hoc, often involving rote coding exercises that reinvent many basic concepts and functionalities. In the next section, we outline the Rosetta model which offers a fundamentally different approach to the construction of compilers for data languages.

3 The Rosetta Model

Generators of relational data language compilers are based on domain models. A *domain model* is the product of analyzing many languages to identify the fundamental data types and operators that underlie their implementation. By defining a *backplane* or *virtual machine* to consist of these types and operators, it is possible to largely separate the problems of backplane implementation from data language implementations (i.e., data language compilers) that instantiate backplane types and that invoke backplane operators)[BBG+88].

Our backplane is based on relational algebra. There are the usual selection and projection operators, but there are many others as well. Examples include operators on a wide range of primitive data types. Also, there are operators for tuple insertion, deletion, modification, and aggregation that reflect the state-based world of database updates.

Compilers for data languages are themselves generators. Each data language statement is transformed by a data language compiler into a program that performs the intended actions of that statement. Backplane operators are the basic units of language computation. A program that implements a data language statement is defined by an operator tree, i.e., a composition of backplane operators. To model *all* possible programs (and hence the actions of all possible data language statements) in terms of these operators, we create a grammar $\mathcal{G}$ that defines all syntactically correct operator trees.

$\mathcal{G}$ consists of a small number of compound productions called *catalogs*. A catalog $\mathcal{T}$ is the set of all backplane operators that produce a result of type $\mathcal{T}$. Catalog $\mathcal{T}$ is written as a production $\mathcal{T} \rightarrow f_1() \mid f_2() \mid \ldots;$ (see Table 2). That is, $\mathcal{T}$ is both a data type and a production name, operator $f_i()$ (which returns results of type $\mathcal{T}$) is its ith rewrite rule, and every operator parameter $\alpha{:}\mathcal{S}$ is a reference to the nonterminal whose production is defined by the catalog $\mathcal{S}$. All type-correct operator trees that can be constructed from backplane operators are defined by $\mathcal{G}$; the start symbol of $\mathcal{G}$ is the first catalog that is listed.

Not all operator trees (sentences) of $\mathcal{G}$ are meaningful or can be expressed by a data language. To model a particular data language $\mathcal{L}$, two refinements are needed. First, a subgrammar $\mathcal{G}_{\mathcal{L}}$ of $\mathcal{G}$ must be formed so that each statement in $\mathcal{L}$ corresponds to precisely one operator tree in $\mathcal{G}_{\mathcal{L}}$, and every operator tree of $\mathcal{G}_{\mathcal{L}}$ corresponds to at least one statement in $\mathcal{L}$ (i.e., it is legal for multiple statements of $\mathcal{L}$ to generate the same operator tree).[1] Second, the syntax of $\mathcal{L}$ is "grafted" onto $\mathcal{G}_{\mathcal{L}}$ to indicate when particular operator trees should be invoked. Thus, the specification $\mathcal{G}_{\mathcal{L}}$ of $\mathcal{L}$ defines the set of operator trees that can be expressed by the language and the syntax that invokes specific trees.

Our approach to specifying languages is different than that of typical syntax-directed compiler tools, such as lex/yacc [Joh86] and flex/bison [DS91]. Our approach is semantics-directed: we define the language's semantics (i.e., operator trees that can be produced) first and specify syntax last, rather than the reverse. The next section illustrates these ideas on the domain of elementary calculator languages. Later, we introduce additional Rosetta concepts for dealing with more complex languages.

3.1 Calculator Data Languages

Consider the domain of calculators that perform basic integer arithmetic and that can store results in named variables. Such calculators define elementary data languages. This domain can be modeled by a backplane (see Table 2a) that utilizes four data types: `Int` (integers),

[1] Data language compilers translate each statement s into a single operator tree t. Query optimizers can then rearrange and replace the operators of t to optimize t. However, the mapping from s to t must be unambiguous.

`Var` (variable references), `String`, and `Void`. The calculator backplane of Table 2a is arranged in Table 2b as a grammar $\mathcal{G}$ with three catalogs: `Void`, `Int`, and `Var` (no operators compute a result of type `String`, so there are only three catalogs).

assign(v:Var, y:Int):Void	assign value y to variable v	Void	$\rightarrow$	assign Var Int
clear():Void	discard active variables		$\mid$	clear
list():Void	list active variables and their values		$\mid$	list
defn_var(v:Var, i:Int):Void	define variable v with initial value i		$\mid$	defn_var Var Int
print(x:Int):Void	print the input value		$\mid$	print Int
		;		
add(x:Int, y:Int):Int	compute the sum x + y	Int	$\rightarrow$	add Int Int
sub(x:Int, y:Int):Int	compute the difference x - y		$\mid$	sub Int Int
eval(x:Int):Int	return the value of x		$\mid$	eval Int
refvar(v:Var):Int	return the value stored in variable v		$\mid$	refvar Var
str2int(s:String):Int	convert string s to an Int		$\mid$	str2int String
varcnt():Int	count the defined variables		$\mid$	varcnt
		;		
str2var(s:String):Var	interpret s as a variable	Var	$\rightarrow$	str2var String
		;		

(a) backplane functions (b) grammar $\mathcal{G}$

Table 2: Calculator Backplane Functions

To customize grammar $\mathcal{G}$ to obtain the grammar of a specific calculator, we must eliminate unwanted or meaningless operator trees. To do this, we create subsets of catalogs, called *subcatalogs*. For example, the subcatalog $\mathtt{S}_0 \rightarrow f_0() \mid f_1();$ contains the operators $f_0()$ and $f_1()$, while subcatalog $\mathtt{S}_1 \rightarrow \mathtt{S}_0 \mid f_2();$ contains operators $f_0()$, $f_1()$, and $f_2()$. By introducing subcatalogs and by specializing operator parameters from $\alpha : \mathcal{T}$ to $\alpha : \mathcal{S}_\tau$, where $\mathcal{S}_\tau$ is a subcatalog of $\mathcal{T}$, unwanted operator trees are eliminated.

For example, the `Int` catalog of the calculator backplane includes arithmetic operators and an administrative operator, *varcnt()*, which returns the number of variables that have been defined. It is meaningless to include *varcnt()* in any arithmetic computation (e.g., there is no significance to the computation *3*(varcnt()+1))*, so operator trees for arithmetic expressions should not contain *varcnt()*. This is easily accomplished by defining the $\mathtt{int}_1$ subcatalog (see Table 3) which defines legal arithmetic expressions to be type-correct compositions of the `add`, `sub`, `eval`, `refvar`, and `str2int` operators. On the other hand, we would like to print both the results of arithmetic expressions and values returned by *varcnt()*. The `int` subcatalog accomplishes this. By using `int` as the type of the formal parameter for the *print()* operator, any integer can be printed but only the operators included in the $\mathtt{int}_1$ subcatalog can appear in arithmetic computations. The refined grammar $\mathcal{G}_{calculator}$ is shown in Table 3.

void = assign(v:Var, x:int_1) | clear() | list() | defn_var(v:Var,i:int_1) | print(x:int)
int_1 = add(x:int_1, y:int_1) | sub(x:int_1, y:int_1) | eval(x:int_1) | refvar(v:Var) | str2int(s:String)
int = varcnt() | int_1
Var = str2var(s:String)

Table 3: Subcatalog Refinement of Calculator Grammar

The last step in defining a data language in Rosetta is specifying a customized syntax for each operator, called a *syntax signature*. The general form of a syntax signature is:

`f [` <*invoking condition*>`,` p_1`:`τ_1`,` p_2`:`τ_2`, ...]`

where *f()* names the backplane operator to be invoked, the <*invoking condition*> specifies a condition which must hold in order to invoke the backplane operator *f()*, and the p_1:τ_1, p_2:τ_2, ..., are the formal parameters of *f()*. There are three kinds of syntax signatures, each having a different invoking condition. Below we discuss the two signatures that consume lexical input, the third (and most complicated as it does not consume lexical input) is discussed in Section 3.2.

Parameterized signatures associate backplane operators with a specific syntax. A syntax specification is a sequence of keywords and parameters surrounded by quotes. (Parameters are differentiated from keywords by an underscore prefix). For example, suppose assignment statements for our calculator language are to have the syntactic form `''let _v = _x;''`. The keyword symbols are `let`, `=`, and `;`, while `v` and `x` are parameters[2]. When this syntax is recognized during the parse of a calculator statement, an operator tree is created with the *assign()* operator as the root. The arguments of *assign()* are the subtrees that evaluate the parameters `x` (the value to be assigned) and `v` (the variable to be assigned the value). The trees for `x` and `v` are derived from the `var` and int_1 subcatalogs, respectively. This entire specification is declared by a single parameterized signature: `assign[ ''let _v = _x;'', v:var, x:`int_1` ]`.

Conversion signatures are used exclusively with conversion operators, i.e., string-to-backplane-object mappings. The invoking condition is a regular expression enclosed in single quotes. For example, the syntax `''let myvar = 1304;''` assigns variable `myvar` the number `1304`.

[2] Blanks, tabs, etc. are treated by Rosetta as white space.

To parse the character string "1304" into an integer (which will then be value argument to the *assign()* operator) requires that we supply a regular expression that defines integer syntax and converts character strings matching this syntax into integer objects by the operator *str2int()*. This is accomplished by the syntax signature `str2int[ '[1-9][0-9]*' ]`. Notice that the `String` type has now disappeared completely; no operators compute a result of type `String` and all `String` parameters have been replaced by specific string constants.[3]

A Rosetta language specification is a file containing a set of subcatalogs whose constituent elements are syntax- and subcatalog-customized signatures. Tables 4 and 5 show definitions for two integer calculators, one in-fix and the other post-fix, which, despite different syntax, generate precisely the same set of operator trees.

```
action =print[" _x ", x:int]
       | assign[" let _v = _x ",v:var,x:int_1]
       | assign[" _v = _x ", v:var, x:int_1]
       | defn_var[" define _v = _x ",
               v:var, x:int_1]
       | list[" list "]
       | clear[" reset "]
-------------------------------------------------
int_1  =add [" _x + _y ", x:int_1, y:int_1]
       | sub [" _x - _y ", x:int_1, y:int_1]
       | eval[" ( _x ) ", x:int_1]
       | str2int [ ' [0-9]+ ' ]
       | refvar[" _v ", v:var]
-------------------------------------------------
int    =varcnt[" count variables "]
       | int_1
-------------------------------------------------
var    =str2var[ '[a-z][a-z0-9]*' ]
```

Table 4: In-Fix Calculator Definition

```
action =print[" _x ", x:int]
       | assign[" _v _x = ",v:var,x:int_1]
       | defn_var[" _v _x define ",
               v:var, x:int_1]
       | list[" showvar "]
       | clear[" reset "]
-------------------------------------------------
int_1  =add [" _x _y + ", x:int_1, y:int_1]
       | sub [" _x _y - ", x:int_1, y:int_1]
       | str2int [ ' [0-9]+ ' ]
       | refvar[" _v ", v:var]
-------------------------------------------------
int    =varcnt[" nvars "]
       | int_1
-------------------------------------------------
var    =str2var[ '[a-z][a-z0-9]*' ]
```

Table 5: Post-Fix Calculator Definition

3.2 Further Extensions of Rosetta

The central reason why Rosetta specifications for calculator data languages are so simple is that there is no distinction between *syntax trees* (operator trees that are produced directly by parsing) and *semantic trees* (operator trees that are to be executed). This fortuitous coincidence arises in elementary languages, but less often in complex languages. Rosetta provides additional features to support syntax-tree-to-semantic-tree mappings: directives, cycle signatures, and context variables.

Directives. Syntax trees can be transformed into semantic trees through the use of *directive operators,* distinct from backplane operators, which manipulate syntax trees. The distinction

[3]This example points out the optimization possibilities in constructing operator trees: the `str2int` operator could be applied directly to the input character string and the resulting integer could then replace the `str2int` operator in the tree.

between directive and backplane operators is that directive operators are used solely in *syntactic* modeling of languages whereas the backplane operators are applied only in the *semantic* modeling of languages; there is no overlap between the two.[4] Because new language requirements may outstrip any set of fixed capabilities, Rosetta allows the addition of new directive operators as well as new backplane operators.

To manipulate syntax trees, directive operators are grouped into *directive sections* which are sequences of directive operations delimited by curly braces, '{' and '}'. Two directive sections are added to signatures—a *pre-action*, placed before the signature, and a *post-action*, placed after the signature.

```
{   pre-action_directive0;   pre-action_directive1;   ...}
f [ < invoking condition >, p1:T1, p2:T2, ...]
{   post-action_directive0; post-action_directive1;  ...}
```

At data language statement *compile* time, the pre-action directives are evaluated sequentially, then the syntax pattern is matched, and finally the post-action directives are evaluated. Note that these are directives to the *data language compiler*, and need not introduce additional operators in an operator tree. The only requirement that is imposed on a directive function is that it must not cause type violations. In particular, an operator tree must compute a result of the same type before and after modification; e.g., an operator subtree that outputs strings cannot be replaced with an operator tree that outputs integers.

Cycle Signatures. The *cycle signature* is a variant of the syntax signature in which the invoking condition is a boolean expression. Unlike other signatures, cycle signatures do not consume additional syntactic input but instead re-process previously parsed input to conditionally introduce additional operators into an operator tree or to rearrange existing operators.

An example is converting syntax trees of the computational formula for variance, $\frac{1}{n}\sum x_i^2 - \overline{x}^2$, into (semantic) operator trees. The syntax tree for a variance expression is shown in Figure 1a. However, the backplane operators that evaluate this formula are based on processing streams of tuples: *compute()* and *aggregate()*. *compute()* performs simple arithmetic operations on one or more attributes of all tuples of an input stream of tuples. *aggregate()* maps a stream of tuples to a single tuple by aggregating (e.g., counting, finding the maximum, average, etc.) specific attribute values of each tuple of the tuple stream. Thus, to produce the corresponding (semantic) operator tree for Figure 1a requires nested calls to *compute()* and *aggregate()*: *compute()* evaluates x_i^2, *aggregate()* performs summation, count, and average aggregations on specific attributes of the tuples of the resulting stream, and finally *compute()* arithmetically combines the values of the tuple output by *aggregate()* into a single value (the variance). Figure 1b shows the desired operator tree.

To convert such syntax trees into their corresponding operator trees, we introduced the *rewrite()* directive, which makes node rearrangement explicit, and used it within a catalog

[4] *Syntactic modeling* is building a model that correctly parses the syntax of the target language. *Semantic modeling* defines what correct syntax means - i.e., what actions are to be taken in response to a legal syntactic phrase.

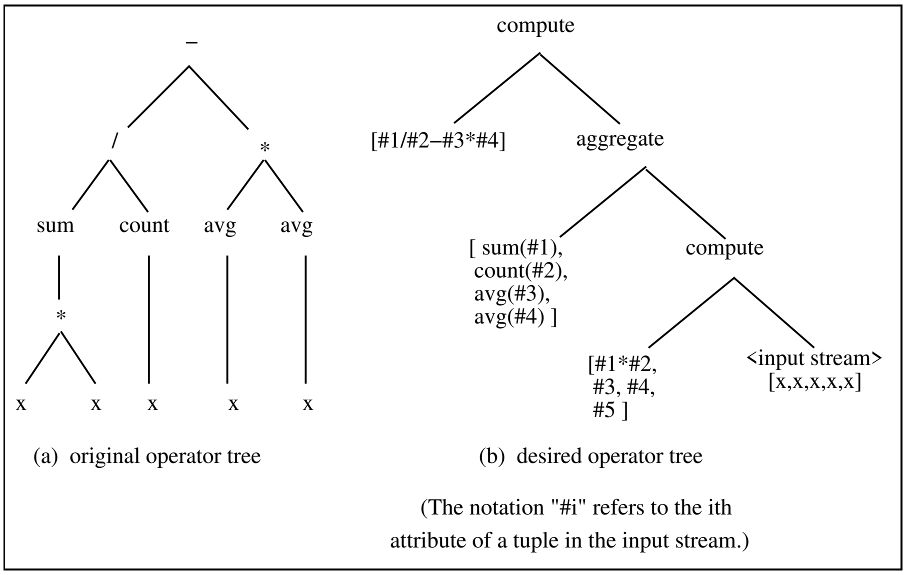

Figure 1: Computing variance: Original and Desired Operator Trees

of cycle signatures. Generally, cycle signatures are included in recursive subcatalogs (e.g., $\mathcal{T} \rightarrow f_1(\alpha : \mathcal{T}) \mid \ldots;$) which perform sophisticated conversions of syntax trees to operator trees. A full discussion of the *rewrite()* directive and cycle subcatalogs appears in [Vil94].

Context Variables. Often, values for operator parameters can be precisely determined from information that has already been parsed. Rather than redundantly specifying such information in data language statements, previously collected information can be placed in a named, global storage area called a *context*. Context variables store information that is globally needed in instantiating parameters of backplane operators.

Context definitions appear at the head of a Rosetta file and are delimited by `BEGIN CONTEXT` and `END CONTEXT` keywords (see Table 6). The operator *new_context()* allocates space for a context and makes it active (pushes it on top of the context stack). When the subcatalog that created that context is satisfied, that context is made inactive (it is popped off the context stack). However, the memory that was allocated to the deactivated context is not immediately garbage-collected. Because a context variable's value may be accumulated over the actions of multiple subcatalogs, references to it may be placed in the operator tree before the value is completely known; therefore, a copy of the value cannot always be made and memory reclamation could destroy values needed later, during query evaluation.

Consider the Quel `RANGE OF` statement: `RANGE OF T IS R`. Once this statement has been processed, the alias `T` is effective until it is redefined in another `RANGE OF` statement. Aliases must be maintained independently of other Quel statements; we use a context variable for this. The *decl* subcatalog of Table 6 processes the Quel `RANGE OF` statement. This example is interesting because the operator tree that is generated consists of a single operator, *noop()*. Thus, no computations on relations takes place. However, the parsing of the `RANGE OF` statement does update a context variable maintained by the Quel data language compiler. The identifier `a` (called an *alias*) is paired with the specified relation `r` and is added to an existing list (*rel_list*) of relation-alias pairs by the directive *new_alias(rel_list,[(a,r)])*. References to the alias `a` in subsequent Quel statements will be translated into references to relation `r`.

BEGIN CONTEXT C		
rel_list	:	List[*relation_alias*] – relations which may be accessed and their aliases.
proj_list	:	List[*Attribute*] – attributes to be retrieved.
xpr	:	List[expression] – expressions to be computed and displayed.
END CONTEXT C		

decl	=	noop[" RANGE OF _a IS _r ", NULL]
		{ a:*identifier*;
		r:*relation*;
		new_alias(rel_list,[(r,a)]);
		}

;		

Table 6: Partial Specification of Quel RANGE Statement

In addition to global variables, Rosetta supports a form of local variables. Like backplane parameter types, variable types are subcatalog names. Local variables are declared at the top of a directive section, e.g.,

$$\{\quad \alpha_0 : \tau_0 ;\ \alpha_1 : \tau_1 ;\ \ldots$$
$$\texttt{directive}_0 ;\ \texttt{directive}_1 ;\ \ldots$$
$$\}.$$

The scope of a local variable is limited to only its associated signature and the directive section where it is declared. Thus, a local variable defined in a pre-action can be referenced in the pre-action and the signature but not in the post-action; similarly for local variables declared in the post-action. Local variables are assigned values only by operators or as a result of parsing; there is no direct assignment operator. For example, the following fragment of the SQL specification parses the FROM clause[5]. The post-action local variable `r` is assigned its value during the parsing of the retrieve syntax `''FROM _r _y''` and is not available in the pre-action.

```
f_ret   =   {   is_attr_list(xpr); }
            retrieve['' FROM _r _y '', rel_list, xpr, y:where,NULL]
            {   r:relation_list;
                mergerelation(r,rel_list);
            }
;
```

[5] `xpr` and `rel_list` are context variables

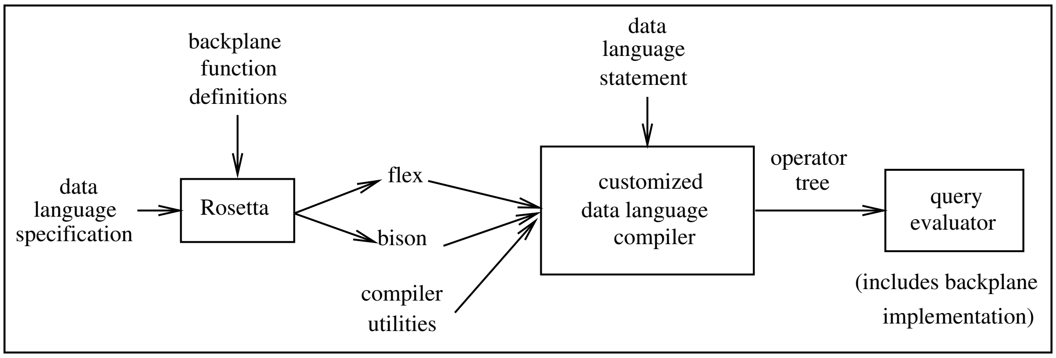

Figure 2: Architecture of Rosetta

4 The Rosetta Prototype

The Rosetta prototype (see Figure 2) was implemented in a mix of bison, flex, C, and Prolog. The input to Rosetta is a Rosetta language specification and definitions of all backplane operators. The generator type-checks the specification using these definitions and generates bison and flex files, which define the lexical analyzer and parser for the customized data language. When these files are compiled and linked with modules that include data structures and utilities that are common to all compilers produced by Rosetta, a compiler for the target data language is created. Input to the generated compiler is a statement of the data language; the compiler maps the statement to an operator tree which is executed by the query evaluator (written in Prolog).

The flex file that is produced by Rosetta is derived directly from the keywords of parameterized signatures and the regular expressions of conversion signatures of a Rosetta specification. They define the patterns of the flex rules and the tokens of the language.

Generating the bison file is more complicated. A bison rule consists of a name and a sequence of options; a bison rule option consists of a sequence of pattern elements[6] optionally interspersed with blocks of C code. Each Rosetta subcatalog maps to a bison rule with the same name, and each of its signatures maps to a bison rule option (see Figure 3).

Each Rosetta signature has a unique translation. Conversion signatures are the simplest. As the invoking syntax of a conversion signature is a regular expression, the unique token identifier generated to represent it in the flex file becomes the sole element of the pattern for the rule option. Code to add the associated backplane operation to the operator tree is generated after the pattern.

Mapping a parameterized signature to a bison rule option is more complicated. First, a block of C code consisting of calls to the directives in the pre-action is generated. Next, the invoking syntax is mapped: the keywords are translated to lexical tokens and the parameters are translated to subcatalog names (which also refer to bison rules). For example, the

[6] A bison rule pattern element may be either a reference to another bison rule or a token symbol returned from the lexical analyzer.

```
display =   union[" _x UNION _y ", x:display, y:display2]
            |   . . .
            |  select
;
```

(a) Rosetta specification

```
%type <nodeptr>  display
        . . .

%%

display:   display  UNION_RL  display2
            {
              $$ = alloc_node(FUNCTION,NULL);                       // allocate a node for a function
              $$->u.fn->fncode = str2FUNC("union");                // initialize function code
              $$->u.fn->argc = 2;                                   // initialize number of parameters
              $$->u.fn->argl = (node **) calloc(2,sizeof(node *));  // allocate array for parameters
              $$->u.fn->argl[0] = $1;                               // initialize 1rst parameter to display
              $$->u.fn->argl[1] = $3;                               // initialize 2nd parameter to display2
            }
         |  . . .
         |  select
            {
              $$ = $1;
            }
;
```

(b) generated bison code

Figure 3: Mapping a Rosetta Rule to a Bison Rule

parameters of the signature in Figure 3a are mapped to *display* and *display2*. Following the pattern elements, a final block of code is generated which performs three tasks: managing references to local and context variables; evaluating the post-action directives; and creating a node in the operator tree for the associated backplane operation. The code generated after the pattern assigns to local and context variables the values matched in the pattern. Post-action directive calls are then copied into the rule. Finally, code is added to allocate a node of the operator tree and to initialize it with the code number of the backplane operation to be called, the number of its parameters, and pointers to its parameters. Furthermore, every reference to a variable is resolved to a pointer and maintained in the local symbol table of the bison rule option.

Unlike the other subcatalogs, cycle subcatalogs do not consume additional input; instead, the signatures of a cycle subcatalog test condition operators. In the generated bison rule, these condition operators are mapped into a nested *if* statement and their post-actions become the conditional statements. Thus, the post-action of the first condition that is satisfied is evaluated. If the signature of that condition is recursive, condition testing begins anew with the first condition; otherwise, cycling terminates. Implementation of the cycle catalog is quite complex; for a full discussion, see [Vil94].

5 Results and Experiences

To validate Rosetta, we modeled five data languages. We selected SQL and Quel because these languages are historically significant and for their continuing popularity in industry and academia. The other three languages are derivatives of SQL and Quel which support alternative data models: SQL/NF[RKB88] is a data language for Non First Normal Form (¬1NF) relations while TSQL2[Sno94] and TQuel[Sno87] support temporal data models.

Statistics from our experiments are summarized in Table 7. We began by defining a Rosetta specification for SQL, and from there we evolved specifications for other data languages. The first row of the table shows the starting language (if any) from which we derived our specifications. The size of a specification (in numbers of lines) for each language is listed on line 2. Note that Rosetta specifications, even for rather complex languages such as TSQL2, are rather small.

While it is difficult to know precisely the number of lines of flex/bison code that would be written by hand to produce a comparable compiler, the number of lines of flex/bison that Rosetta generates (line 3) provides a first-order approximation. The ratio of the Rosetta specification size and the lines generated (called the *expansion ratio* — line 4) gives a crude estimate of the productivity gains that Rosetta offers. In general, we observed expansion ratios of 1:6, which supports our experience that Rosetta saves substantial time in software development.

		SQL	SQL/NF	TSQL2	Quel	TQuel
1	Starting Language	—	SQL	SQL	SQL	Quel
2	Specification Size	240	270	350	275	240
3	flex/bison files	1400	1440	2530	1500	1425
4	Expansion Ratio	1:7	1:6	1:7	1:6	1:6
5	Development Time	NA	3 wk.	3 wk.	4 wk.	1.5 wk.
6	Signature Reuse	NA	60%	87%	27%	44%

Table 7: Summary of Modeled Languages

Other statistics of Table 7 underscore the value of Rosetta. The development times (row 5) for individual languages ranged from one and one half weeks to four weeks. This is the time it took us to read and understand the literature pertaining to each language, and then to create the specification. (Rosetta was developed concurrently with the specification for SQL, so it was impossible for us to separate the language modeling time for SQL from the implementation time for Rosetta).

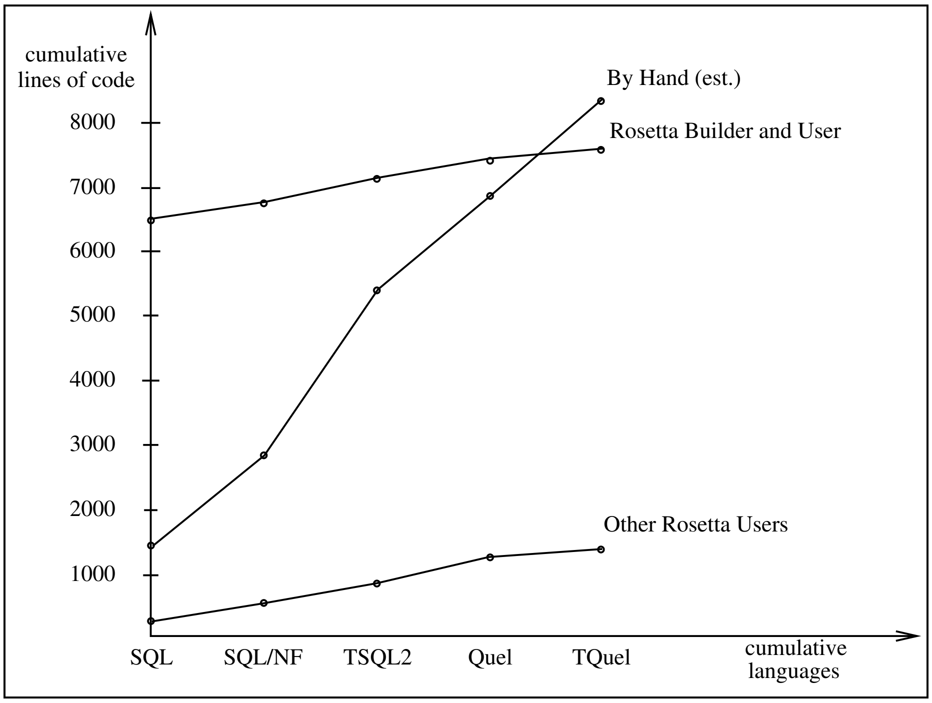

Figure 4: Compiler Generation: Cumulative Lines of Code

The last row of Table 7 gives some statistics on specification reuse and the similarities of different data languages. For example, what fraction of a new language is simply a direct copy of its parent language? Row 6 measures this as signature reuse. We counted only those signatures which were reused unchanged or with only minor modifications[7]. For example, the TSQL2 specification used 87% of the SQL specification unchanged, because TSQL2 was designed to be upward compatible with its parent language, SQL. In contrast, Quel used only 27% of our SQL specification unchanged. As these languages are quite different, this level of reuse is not unexpected. Most signature reuse comes from the primitive data types (int, float, string) and their operators that shared by the two languages.

Figure 4 shows the cumulative lines of code written during the specification of the five languages we modeled[8]. To model SQL using Rosetta, we wrote 6240 lines of code, comprising the generator and the SQL specification. A user building SQL without Rosetta would have written the flex and bison files by hand, for a total of about 1400 LOC. Clearly, writing Rosetta to generate only one language is not a winning proposition.

But writing a generator to generate one program is never a winning proposition. The utility of Rosetta is to generate multiple compilers, writing only a simple specification for each language. The graph shows that the development overhead for Rosetta becomes increasingly cost-effective (in terms of lines of model specification) with each additional language generated, until the fifth language, when using Rosetta is preferred. Finally, the bottom line of the graph shows the cumulative lines of code that a user who was *given* Rosetta would have

[7]Minor modifications include variable and subcatalog name changes and keyword changes.
[8]Overhead consists of lines of code for building Rosetta.

had to write, i.e., the cumulative count of lines in the specifications.

As the results in Table 7 suggest, it is not difficult to extend data languages that have been defined using Rosetta's building-blocks approach. The developer adds any backplane and directive operators that are not present in the Rosetta library and then integrates those operators into a Rosetta specification.

Rosetta has an additional advantage which is difficult to quantify: Rosetta's high-level specifications are fairly easy to understand and use. This enhances the ease with which a language and its compiler can evolve. For example, a language can be partially specified, generated, evaluated, modified, and then extended with additional operations as needed. Thus the compiler developer can make use of an evolutionary series of generated partial prototypes.

Our experiments also revealed limitations in both our prototype and our domain model. New data languages can often require generalizations of existing backplane operators (i.e., through the introduction of additional parameters or by requiring more general features). For example, aggregation operations were originally designed to be evaluated in parallel by a single controlling function, *aggregate()*. This was sufficient for SQL since aggregations in a query are evaluated over the same input stream. However, a Quel aggregate may include a nested grouping operation on its input stream, allowing the aggregated stream to *differ* for different aggregations in the same query. To resolve this problem we extended each aggregation operation with a stream parameter and re-designed the controlling function to duplicate the input stream for each aggregation and to collect the results into one output stream.

Error detection and recovery in Rosetta could be improved. Not all errors can be conveniently caught during operator tree construction. For example, one might generate an operator tree that sorts a tuple stream on a nonexistent attribute. Such errors are more easily caught by a pre-execution analysis of the tree. Each operator (when evaluated in this pre-execution phase) checks and outputs "metadata" that characterizes its results. Simple semantic checks (e.g., does the sort attribute correspond to one of the input attributes of tuples?) would then ensure that operator computations are semantically meaningful.

More work is needed on error recovery. Presently, Rosetta has no built-in error recovery mechanisms when syntactic errors are discovered. We believe that a general-purpose mechanism for error-handling can be introduced into the Rosetta framework without significantly impacting the results that we have reported in this section.

6 Related Work

Software Architectures and Software System Generators. Software architectures is the study of large-scale system design w.r.t. its underlying components and their methods of intercommunication. Software system generators is a subdiscipline of architectures, where components are designed to be plug-compatible and interoperable so that customized software systems can be produced quickly through component composition.

Rosetta is an example of a GenVoca generator [BO92]. A GenVoca domain model is a set of libraries (called *realms*) of plug-compatible components. A software system or program is defined as a composition of components, called a *type equation*. The set of all syntactically and semantically correct type equations (i.e., software systems that can be generated) is expressed as an attribute grammar [BG96].

As mentioned earlier, a data language compiler is itself a generator. It translates data language statements into programs that perform the intended actions of that statement. Rosetta catalogs are realms, and operator trees (i.e., data language programs) are type equations. What distinguishes Rosetta from other GenVoca generators is the manner in which type equations are specified. In other GenVoca generators, a template-like notation for composing components is used to declare type equations (e.g., a[b[c]]). Rosetta, in contrast, enables type equations to be expressed as declarative statements of data languages (e.g., SQL SELECT-FROM-WHERE).

Extensible Database Systems. Extensible DBMSs offer some support for data language extensibility. The usual approach is to make data language parsers table-driven, thus limited (but useful) sets of features (e.g., user-written ADTs) can be added easily (see Gral[Gut89], Postgres[SK91], Probe[MD90], and Starburst[HFLP89]).

DBMS toolkits, such as Exodus[CDV87] and Genesis[BBG+88], are software design and development environments for building customized DBMSs. Genesis is a GenVoca generator that can assemble customized DBMSs from components. Among Genesis components are those that implement different data languages. However, these components (data language compilers) must be hand-written; like Exodus, there is no specific tool support for building customized data languages.

Extensible Programming Languages. Language extensibility can be achieved using preprocessors. TXL[CHHP91] can extend any imperative language for which a compiler and a complete syntactic description of the base language are available. TXL maps extensions into the base language; however, it also allows extensions to reference library operators which augment the functionality of the base language.

Eli[GHL+92] is an example of a compiler generator. It is an expert system that controls off-the-shelf tools like lex and yacc. Specifying a compiler to Eli consists of supplying specification files which contain information used for lexical analysis and parsing, semantic analysis, and code generation. Multiple specification files are necessary, where each is written using its own special syntax with no uniformity among them.

IP is an extensible language/compiler that is being developed at Microsoft Research [Sim95]. IP's interface is a language-neutral structure editor, which takes programmers input and directly represents source code as semantic trees. To view a program (e.g., during editing) an unparser converts semantic trees to text. Additional semantic nodes can be added to IP, so language extensibility is straightforward. An intent of IP is that once programs are represented in a semantic tree format, analyses and program transformations (e.g., lift operations) can be easily accomplished. Rosetta is related to IP in that they share the idea that a backplane of operations can support many languages.

Natural Language Processing. LIFER[Hen77] is a general package of tools which facilitates the rapid addition of natural language interfaces to existing software systems. The similarity of LIFER to Rosetta is the triggering of an operator when an input text pattern is recognized. A point of divergence is an important part of our work: the definition of a standardized (but open) set of operators for a DBMS backend and our emphasis on building blocks and their reuse.

7 Conclusions and Future Work

Achieving significant increases in software productivity requires automating well-understood (and potentially difficult) programming tasks. Generators are designed to achieve these goals by exploiting the similarities of software design and development in families of closely-related applications. Generators are based on domain models that identify the basic building blocks of these applications, and that formalize the similarities and differences among family members as the presence or absence of particular blocks. In this way, generators substantially reduce the effort needed in both constructing new software systems/applications and evolving existing applications.

In this paper, we have outlined a generative approach that demonstrates how families of relational data language compilers can be developed economically. We first identified the basic units of data language computation as backplane operators. The actions of specific data language statements are expressed as compositions of these operators (e.g., operator trees). Next, we grafted syntax onto individual operators to indicate where these operators should appear in an operator tree that implements a language statement. The result is a compact specification of a data language. Our prototype, called Rosetta, translates such specifications into compilers. We noted that our approach to compiler construction is different than typical compiler-construction tools: that is, we define the semantics of a target language first and then its syntax, rather than the reverse.

We validated Rosetta by modeling five diverse data languages and generating compilers for them. Enough backplane operations were implemented to enable evaluation of SQL and Quel queries. We found that the overhead for constructing Rosetta was amortized after using it to construct compilers for these five target languages; the generation of additional compilers would have been accomplished at rather low, incremental costs. Further research is needed to determine effective and compact ways of specifying and handling error recovery.

Although Rosetta is a valuable tool for prototyping data language compilers, we believe that the true value of Rosetta lies in its potential for generating compilers for other domain-specific (e.g., OO) programming languages. One of the main lessons learned from the generator and architecture communities is that domain-specific languages can substantially simplify the specification of domain-specific applications. Thus, the need for extensible and evolvable languages will become progressively more important, driving the need for tools that economically produce compilers for these languages and that allow them to evolve easily. We believe that a Rosetta-like approach may be the key to a practical solution to these problems. The generalization and application of Rosetta to other language domains is the subject of further research.

References

[BBG+88] D. Batory, J. Barnett, J. Garza, K. Smith, K. Tsukuda, B. Twichell, and T. Wise. Genesis: An extensible database management system. *IEEE Transactions on Software Engineering*, pages 1711–1730, 1988.

[BG96] D. Batory and B.J. Geraci. Validating component compositions in software system generators. In *International Conference on Software Reuse*, April 1996.

[BO92] D. Batory and S. O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Transactions on Software Engineering and Methodology*, 1(4):355–398, October 1992.

[CDV87] M. Carey, D. DeWitt, and S. Vandenberg. A data model and query language for EXODUS. Technical Report CS Technical Report 734, University of Wisconsin, December 1987.

[CHHP91] J. Cordy, C. Halpern-Hamu, and E. Promislow. TXL: A rapid prototyping system for programming language dialects. *Computer Languages*, pages 97–107, January 1991.

[Dat87] C. J. Date. *A Guide to Ingres*, chapter 4. Addison Wesley Publishing Company, Inc., 1987.

[DS91] C. Donnelly and R. Stallman. *BISON The YACC-Compatible Parser Generator*, December 1991. on-line documentation for Bison Version 1.16.

[GHL+92] R. Gray, V. Heuring, S. Levi, A. Sloane, and W. Waite. Eli: A complete, flexible compiler construction system. *Communications of the ACM*, pages 121–131, February 1992.

[Gut89] R. Guting. Gral: An extensible relational database system for geometric applications. In *Proceedings of the Fifteenth International Conference on Very Large Data Bases*, pages 33–44, 1989.

[Hen77] G. Hendrix. The LIFER manual—a guide to building practical natural language interfaces. Technical Report Technical Note 138, SRI International, Menlo Park, CA, 1977.

[HFLP89] L. Haas, J. Freytag, G. Lohman, and H. Pirahesh. Extensible query processing in Starburst. In *ACM SIGMOD*, pages 377–388, May 1989. also, IBM Almaden Tech Report RJ 6610 (63921) 12/21/88.

[ISO72] ISO. *ISO Recommendation R1538, Programming Language ALGOL*, first edition, March 1972.

[Joh86] S. Johnson. Yacc: Yet another compiler compiler. In *UNIX Programmer's Manual: Supplementary Documents 1*. University of California, Berkeley, 1986.

[JW78] K. Jensen and N. Wirth. *Pascal: User Manual and Report*. Springer-Verlag, second edition, 1978.

[KR78] B. Kernighan and D. Ritchie. *The C Programming Language*. Prentice-Hall Inc., 1978.

[MD90] F. Manola and U. Dayal. PDM: An object-oriented data model. In S. Zdonik and D. Maier, editors, *Readings in Object Oriented Database Systems*, chapter 3.4. Morgan Kaufman, 1990.

[RKB88] M. Roth, H. Korth, and D. Batory. SQL/NF: A query language for ¬1NF relational databases. *Information Systems*, 12(1):99–114, 1988.

[Sal95] J. Salasin. Evolutionary design of large complex software. ARPA BAA 95-40, Advanced Research Projects Agency, 1995.

[Sim95] C. Simonyi. The death of computer languages, the birth of intentional programming. In *28th Annual International Seminar on the Teaching of Computing Science at University Level The Future of Software*. University of Newcastle upon Tyne, September 1995.

[SK91] M. Stonebraker and G. Kemnitz. The Postgres next-generation database management system. *Communications of the ACM*, pages 78–93, October 1991.

[Sno87] R. Snodgrass. The temporal query language TQuel. *ACM Transactions on Database Systems*, 12(2):247–298, June 1987.

[Sno94] R. Snodgrass. TSQL2 language specification. *SIGMOD Record*, pages 65–86, March 1994.

[vdL89] R. van der Lans. *The SQL Standard*. Prentice Hall International, 1989.

[Vil94] E. E. Villarreal. *Automated Compiler Generation for Extensible Data Languages*. PhD thesis, The University of Texas at Austin, 1994.