Automated Compiler Generation for Extensible Data Languages

by

Emilia Elizabeth Villarreal, B.S., M.S.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

December, 1994

Automated Compiler Generation for Extensible Data Languages

Approved by
Dissertation Committee:

_____

_____

_____

_____

_____

For my family,
Chris and Nigel

Acknowledgments

First and foremost, I'd like to thank my family, Chris and Nigel, who were always there for me. Chris resonated with my angst especially well. I'd like to thank my advisor, Don Batory, who alternately provided encouragement and impetus. Finally, I'd like to thank the folks of the Computer Science Department at Cal Poly State University in San Luis Obispo, who generously provided computing and lab facilities.

Emilia Elizabeth Villarreal

The University of Texas at Austin
December 1994

Automated Compiler Generation for Extensible Data Languages

Publication No. _____

Emilia Elizabeth Villarreal, Ph.D.
The University of Texas at Austin, 1994

Supervisor: Don S. Batory

To meet the changing needs of the DBMS community, e.g., to support new database applications such as geographic or temporal databases, new data languages are frequently proposed. Most offer extensions to previously defined languages such as SQL or Quel. Few are ever implemented. The maturity of the area of data languages demands that researchers go beyond the proposal stage to have hands-on experience with their languages, if only to separate the good ideas from the bad. Tools and methodologies for building families of similar languages are definitely needed; we solve this problem by automating the generation of compilers for data languages.

Our work, Rosetta, is based on two concepts. First, underlying the domain of data languages is a common backplane of relational operations. Backplane operations are primitive building blocks for language execution and construction, where a building block has standardized semantics. The definition of a well-designed backplane is implementation-independent; that is, the backplane is defined once but can be used to model arbitrarily many data languages.

Second, there exist primitive building-blocks for language construction. From our analysis of the database data language domain, we have identified three classes of building-blocks: one class maps language syntax to backplane functions, another builds an internal representation of the backplane operator tree, and a third class manages contextual information.

For modeling data languages, we define the Rosetta specification language, a grammar-based specification language tailored to our needs with the power to define syntax, map it to the target language, and build an operator tree all in one

rule. Thus each rule is a microcosmic model of a language clause which encapsulates input parsing and code generation.

Our specification language models data languages based on the composition of primitive building blocks for semantics and the customization of the syntax for invoking the compositions. A compiler for a data language is generated by first modeling the language and then compiling the specification. The ease and efficiency with which Rosetta customizes languages derives from the reuse of the backplane operations and the high-level specification supported.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

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

In the domain of data languages, there is a seemingly endless procession of new languages that are being proposed to meet the changing needs of the DBMS community. In particular, there are numerous proposals for object-oriented data languages: GEM [Zan83], Opal [Ser86], ORION [KBB+87], etc. Until a standard is established for object-oriented data languages, multiple competing languages will persist and, indeed, proliferate.

Few are wholly new; most offer extensions to previously defined languages such as SQL or Quel. Many languages never go beyond the proposal stage; even fewer are ever implemented. But without an implementation and actual testing, they cannot be adequately evaluated and compared. The maturity of data languages hinges on such experimentation.

With few exceptions, every language is built from scratch. Despite the similarity of languages within a domain, considerable effort, time, and resources are needlessly expended in duplicating basic functionality already available in previous implementations of other languages. Unfortunately, leveraging existing ad hoc software designs is difficult, and the effort and time needed to reuse software artifacts is substantial and rarely cost effective.

Clearly, there will always be families of functionally similar but syntactically distinct languages, if only for the simple reason that no single language suits all situations. While we cannot eliminate language families, we can improve the way languages are constructed. Tools and methodologies that enable building language

families inexpensively and quickly are definitely needed. Our system, Rosetta, is designed to meet this need.

Rosetta is based on two concepts. First, underlying every family of functionally similar but syntactically distinct languages is a common backplane of operations. The backplane is an object-oriented virtual machine, whose objects are the fundamental objects of the application domain and whose operations are the fundamental operations on these objects. Languages are syntax-customized front-ends to a virtual machine, where language statements map to compositions of backplane operators. Because the objects and operations of the backplane are standardized, many different languages within the domain can be built on the same virtual machine.

Second, there exist primitive building-blocks for language construction. Every building-block has immutable semantics. From our analysis of the database data language domain, we have identified three classes of building-blocks: one class maps language syntax to backplane functions, another converts parse trees to an internal representation, and a third class manages contextual information. We view languages as compositions of building-blocks, where different languages have different compositions or, if they have identical compositions, differ in the syntax they assign to their building-blocks.

Rosetta exploits these two key observations to deliver language extensibility. Our model of data languages imposes extensible syntax on building-blocks having standardized semantics. A compiler for a target language is produced by compiling a specification, written in the Rosetta specification language, which maps syntax to the given domain backplane. The ease and efficiency with which Rosetta customizes languages derives from the reuse of libraries of standardized building-blocks.

1.1 Other Approaches

As extensible languages have been of interest for many years, many other approaches have been applied in other areas, including extensible DBMS work, extensible programming languages, compiler technology, and Artificial Intelligence (Natural Language Processing). We discuss each briefly in turn.

1.1.1 Extensible Databases

Several projects have addressed how to build extensible databases. Those of interest here include efforts to support extensible data languages. Two basic approaches are taken in designing extensible databases: the fixed data language approach, [Gut89, SK91, MD90, HFLP89], and the toolkit approach, [CDV87].

In the fixed data language approach, the extensible database includes a fixed data language but provides a methodology and (sometimes) tools to allow the user to augment it with new data types and new operations on them. Thus the basic data language cannot be changed; one cannot, for instance, abandon the built-in language and replace it with an entirely new language.

The toolkit approach is more flexible. Any language can be generated, using the tools provided by the generator for building the data language of choice. While this is an improvement, typically the support provided for specifying a language and generating its compiler is minimal.

1.1.2 Extensible Programming Languages

To realize extensible programming languages, two basic approaches are taken: preprocessing and direct compiler extension.

In the preprocessing approach, new constructs of the extended language are mapped to built-in constructs (or compositions of constructs) of the base language. The input to the preprocessor is a program in the extended language; its output is a program in the base language which can be compiled and executed just as any program written originally in the base language. There are preprocessors for many languages, e.g., C, Fortran, and PL/1, and there are preprocessors which can be used for multiple languages, e.g., M4. Embedding Quel in C, called EQuel [SWKH76], was achieved by preprocessing the EQuel program into a C program.

The preprocessing approach can be as simple or ambitious as desired. Some preprocessors are capable of only simple syntactic substitution; others are capable of code migration [CHHP91].

The other approach, compiler extension, makes the code of the compiler available to the user. Languages which can be extended in this way include Lisp and its variants, such as Scheme and Clos, and Icon, an extension of Snobol. Extensions to the language are added directly to the compiler in the form of new functions, which should adhere to a protocol supplied for adding new functions. This approach is much more powerful than the preprocessor approach, allowing the addition of significant new features, such as adding call-by-reference to a compiler which originally supported only call-by-value.

While extensible programming languages can add significant new functionality, neither approach can be used to create a compiler for a completely new language. Because we are interested in generating compilers for new as well as evolving languages, this is a major failing.

1.1.3 Compiler Technology

More related work is found in the area of compiler technology. Of particular interest to us are the syntax-directed translation method for language specification and compiler generators.

A syntax-directed translation is a context-free grammar whose rules are augmented with associated attributes [ASU86]. These attributes may be anything: numbers, strings, structures, etc. Also associated with each rule of the grammer is a set of semantic function calls which may implement any function desired, including compiler utilities.

Work in compiler generators includes both simple parser and lexical analyzer generators [LS86, Joh86, DS91] and expert systems which control other compiler generation tools [GHL+92]. A parser generator requires a specification of the language syntax, while a lexical analyzer requires a specification of the tokens of the language. Their input has no built-in capabilities for semantic specification, so they produce programs which recognize the syntax of the language but do not address mapping syntax to semantics. Input to the generators is hand-crafted anew for each compiler.

The second approach of interest is the construction of an expert system which controls off-the-shelf tools [GHL+92]. Such an expert system initiates the tools in the proper order, passing to each its associated input files and connecting the output of one to the input of another.

Our model of languages differs substantially from that of these approaches: Rosetta has built-in constructs designed for specifying semantics with a functional semantic model. Furthermore, our focus is on data languages, and none of these approaches is targeted for that field.

1.1.4 Natural Language Processing

Finally, work in natural language processing aims at facilitating the addition of natural language interfaces to existing software systems [Hen77c, HSSS78]. The approach taken is similar to that of Rosetta: an interface specification is a grammar whose rules are augmented with expressions implementing a response. The specification is stored in an internal form, and is used by the parser to map an input natural language statement to an expression of the underlying system.

One way in which Rosetta differs significantly from this approach is that Rosetta is targeted to the domain of data languages. In building Rosetta, we have analyzed our target domain to provide a backplane of functions which model it. By focusing our efforts on the data language domain, we are able to include domain-

specific support that is lacking in more general work.

1.2 Overview of Dissertation

Although we believe the concepts that underlie Rosetta to be domain-independent, we have developed and applied Rosetta only in the context of database data languages. Applying Rosetta to other domains is not within the scope of our work and may be the subject of future efforts.

In the rest of the dissertation we discuss the approach taken in Rosetta. We begin with a discussion of how we model data languages, then consider implementation details of the Rosetta prototype. Next, we continue with a discussion of the five languages we modeled to validate the Rosetta approach. The dissertation closes with a chapter on related work and another on evaluation, future work, and conclusions.

Chapter 2

The Rosetta Model

In this section we define the Rosetta model. We take a three-step approach to language modeling: we begin by identifying the objects of the language and a backplane of functions which operate on them. The set of all type-correct compositions of backplane functions is adopted as the initial model of the language. Next, we refine the model to eliminate unwanted compositions of functions. Finally, we impose syntax to dictate when particular compositions of functions should be invoked. These three steps yield a language definition consisting of a specification which defines the compositions of backplane functions, or *operator trees*, generated by the language and the syntax which invokes them. This chapter explores these steps in detail.

2.1 Domain Modeling

A *domain* is a family of related programs all having approximately the same functionality. Examples of domains include operating systems, database systems, and data languages.

A *domain model* is a library of building blocks for programs in the domain and mechanisms for composing them. These building blocks consist of a set of functions and objects which are capable of modeling elements of the domain. For example, the relational database domain includes primitive objects such as strings and integers, and complex objects such as tuples and relations. Functions on the objects of this domain include *select()*, *join()*, *project()*, etc.

A *backplane* is an object-oriented virtual machine whose objects are the fundamental objects of the domain and whose functions (or, equivalently, operations) on these objects are the fundamental functions of the domain.

Designing a backplane is a classic problem in designing reusable software.

Called *domain analysis*, the basic idea is to study a set of functionally similar systems of a domain to postulate empirically the backplane underlying the set. The goal of domain analysis is to design an *implementation-independent* backplane; that is, a backplane that can be re-used to build multiple systems. Domain analysis generally requires a considerable amount of hard work [PDA91].

Domain modeling and backplane design is a process of iterative refinement. We say that a backplane $\mathcal{B}$ *models* language $\mathcal{L}$ if each statement in $\mathcal{L}$ maps to a composition of functions in $\mathcal{B}$. After studying languages $\mathcal{L}_1, \ldots, \mathcal{L}_n$, we postulate an underlying backplane $\mathcal{B}$, then determine if $\mathcal{B}$ is sufficiently general to model an additional language $\mathcal{L}_{n+1}$ with little or no modification. If $\mathcal{B}$ models $\mathcal{L}_{n+1}$ without modification, we proceed to consider language $\mathcal{L}_{n+2}$. Otherwise, we first extend $\mathcal{B}$ to enable it to model $\mathcal{L}_{n+1}$ and then proceed to consider $\mathcal{L}_{n+2}$. Domain analysis is a never-ending process, as domain members evolve over time to meet the changing needs of their communities. Inevitably, the addition of some new feature exceeds the capabilities of the backplane. Thus, *extensibility* is inherent in the design of a backplane.

Backplane Definition. Our objective in domain modeling is the ability to map individual language statements from a multiplicity of data languages into compositions of previously identified database backplane functions.

We refer to backplane functions with a *functional* representation,

`f ( a`$_1$`:T`$_1$`, a`$_2$`:T`$_2$`, ...) : T,`

defining the function's name *"f"*, its formal parameter list *"a_1:T_1, a_2:T_2, ..."*, and its type ``T'', where "α:τ" denotes that α is of type τ.

Example. Calculators are elementary data languages. Table 2.1 shows a backplane for the domain of simple integer-arithmetic calculators. Only four data types are referenced in this backplane—Int, String, Var, and Void—where Int is an integer, Var is a variable reference, etc. This simple domain offers only four arithmetic operations +, -, *, and ÷, represented by backplane functions *add()*, *sub()*, *mul()*, and *div()*. Also included are functions which manipulate variables; turn the calculator on and off; print values; and convert strings to integers or variable references.

2.2 Language Semantics

We refer to a composition of backplane functions as an *operator tree*[1], and define the semantics of a data language by defining the set of operator trees that it can

[1]Note that the leaves of an operator tree are *constants* rather than functions.

assign	(v:Var,y:Int)	: Void	assigns to variable v the value y
clear	()	: Void	discards all active variables
list	()	: Void	lists all active variables and their values
print	(x:Int)	: Void	prints its input value
on	()	: Void	turns the calculator on
off	()	: Void	turns the calculator off and discards defined variables
add	(x:Int, y:Int)	: Int	computes the sum x + y
div	(x:Int, y:Int)	: Int	computes the division x ÷ y
mul	(x:Int, y:Int)	: Int	computes the product x * y
sub	(x:Int, y:Int)	: Int	computes the difference x - y
refvar	(v:Var)	: Int	returns the value stored in variable v
str2int	(s:String)	: Int	converts string s to an Int
varcnt	()	: Int	returns a count of the defined variables
str2var	(s:String)	: Var	interprets s as a previously defined variable
define_var	(s:String, v:Int)	: Var	defines a new variable s with initial value v

Table 2.1: Summary of Calculator Backplane Functions

express. To do so, we restrict the set of all operator trees that can be generated from the backplane functions to the set of operator trees expressible by the data language.

Catalogs. The *catalog* $\mathcal{T}$ is the set of all backplane functions that produce a result of type τ [2]. We write the elements of the set using "|" as a separator:

$$\mathcal{T} = \{ f_1() \mid f_2() \mid \ldots \}.$$

The calculator backplane of Table 2.1 is arranged in three catalogs: Void, Int, and Var.

The set of all operator trees expressible by a backplane is defined by a grammar. The productions of this grammar are the catalogs of the backplane: $\mathcal{T}$ is a production name and $f_i()$ is its ith rewrite rule (see Table 2.2). Every parameter $\alpha : \tau$ of a function is a reference to a nonterminal, whose production is defined by

[2] While our use of catalogs may *appear* to run counter to the object-oriented philosophy, it actually does not. A class in the object-oriented paradigm is defined by its objects plus the set of all functions on them. We *build* on this by imposing an *orthogonal* classification on the functions, classifying them according to the types of their results.

Void	→	assign Var Int
	|	clear
	|	list
	|	off
	|	on
	|	print Int
Int	→	add Int Int
	|	div Int Int
	|	mul Int Int
	|	sub Int Int
	|	refvar Var
	|	str2int String
	|	varcnt
Var	→	str2var String
	|	define_var String Int

Table 2.2: Interpretation of Catalogs as Grammar Rules

$\mathcal{T}$, the catalog of τ. The distinguished or start symbol of the grammar is the first catalog listed. A sentence of the grammar is an operator tree, and the language defined by the grammar is the set of all operator trees that are expressible by the backplane. For example, the expressions "5 + 7" and "v = 5/10" are represented by the operator trees shown in Figure 2.1.

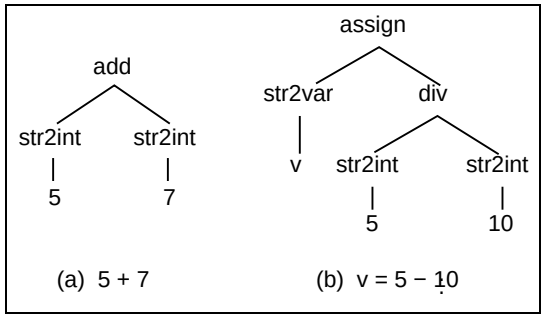

Figure 2.1: Operator Trees for Expressions

Clearly, not all compositions of functions make sense. In the case of the integer calculator, printing the square of the number of defined variables seems meaningless. Similarly, statements of data languages correspond not to arbitrary

compositions of backplane functions but to specific ones, and not every meaningful composition can be expressed by every language. The need to restrict composability to define the semantics of a language leads us to the notion of subcatalogs.

Subcatalogs. A *subcatalog* is a subset of a catalog whose membership is defined by a combination of enumeration and union (denoted by "|") of previously defined subcatalogs:

S_0 = { $f_0()$ | $f_1()$ }
S_1 = { $f_2()$ } | S_0

The subcatalog S_0 contains the functions $f_0()$ and $f_1()$, while the subcatalog S_1 contains the functions $f_0()$, $f_1()$, and $f_2()$.

Subcatalogs are further specialized by replacing each parameter $x{:}\tau$ of each function with $x{:}\mathcal{S}_\tau$, where $\mathcal{S}_\tau$ is a subcatalog of $\mathcal{T}$. This restricts the set of functions that can instantiate x to those defined in $\mathcal{S}_\tau$. By appropriately restricting catalogs to subcatalogs, the semantics of a language is declared by a grammar that defines the set of operator trees the language can express. To accommodate optional arguments to functions, we also allow catalogs to include ϵ, signifying that the empty parameter is allowable.

Example. Recall the *Int* catalog of the calculator backplane, which includes arithmetic functions and an administrative function, *varcnt()*. To preclude the use of *varcnt()* in arithmetic expressions, we define two subcatalogs, int_1 and int_2:

```
int1=   {add(x:int1,y:int1) | sub(x:int1,y:int1)
    |   mul(x:int1,y:int1) | div(x:int1,y:int1)
    |   str2int(s:string) | refvar(v:Var) }

int2=   { varcnt() }
```

and combine them in a general-purpose catalog:

```
int =   int1 | int2 .
```

The *action* subcatalog describes the top-level (user interface) actions:

```
action  =   { print(i:int) | assign(v:Var,i:int1)
            | list() | clear() | on() | off() }
```

Any integer, whether computed or a variable count, can be printed. But only the functions of the int_1 subcatalog can participate in assignment and computation.

Restricting the set of operator trees that a language can express is the most basic way a language is customized. A more visible language customization is the grafting of syntax onto the language specification, generating the same set of operator trees but associating a specific syntax with each operator tree.

2.3 Language Syntax

Customizing the syntax of a simple language consists of assigning an invoking *syntax signature* to each backplane function of every subcatalog. Syntax signatures define the composition of the backplane functions; they are used to directly translate an input statement of the language into an operator tree. Specification of these signatures is simple in some domains, harder in others.

Syntax Signatures. There are three variations on syntax signatures; all derive from the functional representation. The general form of the syntax signature is

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

where *f()* names the backplane function to be invoked, the *<invoking condition>* specifies a condition which must hold in order for the backplane function to be invoked, and the formal parameters p_1, p_2, ..., are listed in the same order as in the functional representation. Superficial changes distinguish the syntax signature from the functional representation: the parentheses are replaced with square brackets and the invoked function's type is dropped, as it can be determined from the function's definition.

All signatures succeed or fail based on some condition. For the parameterized signature and the conversion signature, the condition is the detection of specific syntax whereas for the cycle signature the condition is success of a function call.

Parameterized Signatures. The first variant, the *parameterized signature*, assigns invoking syntax to a backplane function. For this variant, the invoking condition is the recognition of particular syntax in the input.

Formal parameters may appear in any order in the invoking syntax. When referenced in the invoking syntax, they are prefixed by an underscore, '_', distinguishing them from keywords. A successful match of the invoking syntax instantiates the parameters and composes the named function into the operator tree.

As an example, a syntax signature invoking the *assign()* function in our calculator might be:

`assign[``let _v = _x; '', v:var, x:int`$_1$ `].`

The keyword `let` and the symbol `=` signal invocation of *assign()*, while x supplies the value to be assigned to the variable named in v and ; terminates the statement. Elements of the invoking syntax are separated by spaces; in the actual input, they may be separated by any amount of white space[3].

Some syntaxes may require a statement terminator to force complete parsing of the input. For example, z and $z + 8$ are both expressions. In parsing $z + 8$, the expression z may be recognized and $+ 8$ left unparsed unless a terminating symbol, such as the ';' of the assignment statement, follows the expression.

Conversion Signatures. The second variant, the *conversion signature,* manages the use of conversion functions, i.e., string-to-backplane-object mappings. Conversion functions have exactly one parameter—the literal string to be converted. Backplane objects have simple syntax signatures consisting only of the invoking syntax defined by a regular expression enclosed in single quotes. The invoked backplane function must be a conversion function which translates the character string matched by the regular expression into a backplane object.

For example, an integer defined as a string of digits which begins with a non-zero digit is matched by the regular expression [1-9][0-9]* and has the syntax signature

```
str2int[ '[1-9][0-9]*' ],
```

where *str2int()* is the string-to-integer conversion function.

Cycle Signatures. The third syntax signature, the *cycle signature*, has a boolean function as its invoking condition. The cycle signature introduces a way to add backplane functions to the operator tree based on non-syntactic considerations: a function can be injected into the operator tree based on the value of some global variable which is tested by the function of the invoking condition.

For example, suppose we wanted to add error detection to the simple calculator. We might define a function, *signal_error(s:String):Void*, to print an error message and an error detection function *variable_limit_exceeded():Boolean* which tests that the number of defined variables does not exceed the limit. Then to add the function *signal_error()* to the operator tree, we would define the cycle subcatalog:

```
signal_error[ variable_limit_exceeded(), ''VAR_LIMIT'' ]
```

But the cycle signature is much more powerful, and more will be said about it in Sections 2.5 and 2.6.

[3]White space includes spaces, tabs, and newlines; thus, actual input is free-form.

Multiple Operations. Frequently, a keyword phrase references a composition of functions that precludes simple syntax signature definitions. Two mechanisms are provided to handle this situation; both are equally expressive. Choosing between them is a matter of personal preference.

The first mechanism generalizes the instantiation of parameters in the syntax signature by allowing the insertion of calls to backplane functions (of the proper type) directly into the signature's formal parameter declaration list. The parameters of the inserted function call are instantiated from elements of the invoking syntax. For example, we can define a cube operation `cube v` for the calculator using nested calls to the *mul()* backplane function[4]:

`mul[``cube _v; '', v : int`$_1$`, mul(v,v)].`

The alternative mechanism builds on the first and uses a built-in Rosetta backplane function, *noop()*:

`noop[ <invoking syntax>, p:`τ_p `]`

The *noop()* backplane function, which has exactly one parameter, is discarded when it is seen and does not appear in the operator tree, so that the produced operator tree is the tree produced for the parameter p. We can use the *noop()* function in an alternative definition of the same calculator `cube v` operation:

`noop[``cube _v; '', mul(v : int`$_1$`,mul(v,v))].`

No Operations. There are times when keywords or symbols signal no function invocation, e.g., the use of parentheses to alter the order of evaluation. The *noop()* backplane function can handle this because it adds no additional nodes to the operator tree supplied by its parameter:

`noop[`` (_x) '', x:int`$_1$`].`

We use the *noop()* backplane function in Table 2.3 to realize parentheses in in-fix arithmetic expressions.

Language Specification. A Rosetta language specification is a file containing a set of subcatalogs whose constituent elements are syntax-and-subcatalog customized signatures. Each subcatalog maps to a bison rule, and each of its elements maps to a bison rule option. Information contained in the signatures defines the semantic actions to be associated with the bison rule options.

[4]Note that the definition $v : int_1$ need not be repeated multiple times.

Tables 2.3 and 2.4 show definitions for two integer calculators, one in-fix and the other post-fix, which, despite different syntax, generate the same set of operator trees. Generating operator trees for these calculators is quite straightforward; it is more complex for data languages.

Finally, the specification file also includes a special section for bison-style declarations of operator precedence and associativity. Operators are listed in a table with those on the same row having the same associativity and with the rows ordered by increasing precedence; e.g., the lines

```
%left + -
%left * ÷
```

define '+' and '-' to be left associative operators having the same precedence, while '*' and '÷' are also left associative operators having the same precedence but with higher precedence than '+' and '-'.

action	=	print[" _x ", x:int]
	|	assign[" let _v = _x ",v:Var,x:int$_2$]
	|	assign[" _v = _x ", v:Var, x:int$_2$]
	|	define_var[" define _s = _x ", s:string, x:int$_2$]
	|	list[" list "]
	|	on[" ON "]
	|	off[" OFF "]
	|	clear[" clear "]
int$_2$	=	add [" _x + _y ", x:int$_2$, y:int$_2$]
	|	sub [" _x - _y ", x:int$_2$, y:int$_2$]
	|	mul [" _x * _y ", x:int$_2$, y:int$_2$]
	|	div [" _x ÷ _y ", x:int$_2$, y:int$_2$]
	|	noop[" (_x) ", x:int$_2$]
	|	str2int [' [0-9]+ ']
	|	refvar[" _v ", x:Var]
int	=	varcnt[" report variables "]
	|	int$_2$
var	=	str2var['[a-z][a-z0-9]*']
%left	+ -	
%left	* ÷	

Table 2.3: In-Fix Calculator Definition

action	=	print[" _x ", x:int]
	|	assign[" _v _x = ",v:Var,x:int$_2$]
	|	define_var[" _s _x define ", s:string, x:int$_2$]
	|	list[" list "]
	|	on[" ON "]
	|	off[" OFF "]
	|	clear[" clear "]
int$_2$	=	add [" _x _y + ", x:int$_2$, y:int$_2$]
	|	sub [" _x _y - ", x:int$_2$, y:int$_2$]
	|	mul [" _x _y * ", x:int$_2$, y:int$_2$]
	|	div [" _x _y ÷ ", x:int$_2$, y:int$_2$]
	|	str2int [' [0-9]+ ']
	|	refvar[" _v ", x:Var]
int	=	varcnt[" report variables "]
	|	int$_2$
var	=	str2var['[a-z][a-z0-9]*']
%left	+ -	
%left	* ÷	

Table 2.4: Post-Fix Calculator Definition

2.4 Directive Sections

For very simple languages, such as calculators, producing an operator tree directly from the specification is quite straightforward. Because this is certainly not true for all applications, data languages in particular, we must be able to manipulate operator trees as they are being produced. (An example will be shown in Section 2.6.)

We define *directive functions* as functions (distinct from backplane functions) which manipulate operator trees, and a *directive section* as a sequence of directive function calls delimited by curly braces, '{' and '}'. Two directive sections are added to the parameterized signature—a *pre-action*, placed before the signature and used mainly to test for pre-conditions, and a *post-action*, placed after the signature and used mainly to manipulate syntax pattern elements.

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

Directive functions are evaluated sequentially at *compile* time. First the directives of the pre-action are evaluated, then the syntax pattern is matched, and finally the directives of the post-action are evaluated. Note that these are *compiler* directives which generate no additional function calls in the operator tree.

2.4.1 Local Variables

Without variables, directives' results can be obtained only by destructive side-effects on their inputs. Variables are needed for passing parsed parameters into directives and for passing results out.

The scope of a local variable is limited to *only* its associated parameterized 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, local variables defined in the post-action can be referenced in the post-action and the signature but not in the pre-action. Local variables are declared at the top of the directive section, e.g.,

```
{   α0:τ0;
    α1:τ1;
        ...
    directive0;
    directive1;
        ...
}.
```

Like parameters of backplane functions, local variables are typed with subcatalog names. The functions of the directive are responsible for enforcing type restrictions on the local variables. Local variables are assigned values only in functions or as a result of parsing; there is no direct assignment operator.

As production of operator trees is quite straightforward in the simple calculator domain, this and later examples are drawn from the domain of data languages. Tables 2.5 and 2.6 describe some data types and backplane functions from that domain which we will use extensively throughout the remainder of this section.

Attribute	:	attribute information (name, access, ...)
Boolean	:	binary truth type
Expression	:	evaluable expression of any type
List[<T>]	:	a list with any number of elements all of type T
Record	:	a composite structure with fields of differing or same types
Relation	:	relation information (name, alias, access information, ...)
Stream	:	a structure imposed on an arbitrary number of Records
Void	:	the empty type

Table 2.5: Selected Data Language Types

Example. Consider the simplest form of the Quel RETRIEVE statement,

```
RETRIEVE ( <attribute list> )
```

which simply prints the values of the listed attributes for all tuples. Attributes must be prefixed by a relation name (e.g., "R.a" instead of simply "a"). We define a signature to model this simple statement[5]:

```
retrieve[ ``RETRIEVE ( _a )'', r, a:List[Attribute], TRUE ]
    {   r:List[Relation];
```

[5] `TRUE` and `FALSE` are built-in boolean constants; `NULL` is a built-in empty value.

aggregate(a:List[Expression],s:Stream):Stream applies the aggregations of *a* to Stream *s* compute(e:List[Expression],s:Stream):Stream applies the expressions in *e* to Stream *s* retrieve(r:List[Relation], a:List[Attribute], p:Boolean):Stream gets attributes *a* satisfying *p* from relations *r*
print (s:Stream) : Void prints the records of the input Stream *s*

Table 2.6: Selected Data Language Backplane Functions

```
    add_relation_info( a, r );
}.
```

The first parameter of the *retrieve()* function, *r*, is a list of relations referenced in the query. To identify the relations referenced, we define a directive function *add_relation_info(a:List[Attribute],r:List[Relation])* which creates, from the list of attributes, a list of all relations accessed. The call to *add_relation_info()* succeeds if each attribute in *a* is successfully mapped to a valid relation reference and fails otherwise.

While local variables were sufficient for this example, sometimes information must be shared across subcatalogs. This brings us to our next topic—global variables.

2.4.2 Global Variables

Occasionally, information parsed in one syntax signature is also needed in other syntax signatures. Sometimes the entire parameter is needed; sometimes only part of it is shared. All variants of the syntax signature may reference global variables.

Rosetta shares information across subcatalogs via a mechanism for the global collection and distribution of information: the context. When an input statement is parsed, a context is supplied to make multiply-referenced information available wherever it is needed. Global variables can store information needed by multiple functions or gather information specified in multiple clauses. The scope of a context variable is global in that it can be referenced by any rule which is being satisfied while the context variable is on the stack.

Like global variables in any programming language, Rosetta global variables

should be used judiciously and sparingly: using them is easy to do but hard to justify. Usually, the parameters needed to construct a function call are bundled with the keywords indicating the function call and are not needed for the construction of other function calls; in this case, using a global variable is unnecessary. Only when information is shared across subcatalogs are global variables needed.

In the data language domain, global variables collect such information as attributes to be projected, definitions of aliases for relations, etc. For data languages, we have found that three global variables are a necessary part of the context:

rel_list – List[relation]: 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.

To increase generality, these global variables are defined as lists; however, for other domains (and for additional global variables in the data language domain), it may make sense to define global variables which are single objects and not lists.

Context definition is specified once, at the head of a Rosetta file, in a special section delimited by `BEGIN CONTEXT` and `END CONTEXT` keywords. A specification file may define more than one context. To distinguish between them, contexts are named when defined and referenced by name. Between the begin and end clauses the global variables are defined using the α:τ type notation:

```
BEGIN CONTEXT <context name>
    rel_list :  List[Relation];
    proj_list :  List[Attribute];
    xpr :  List[Expression];
END CONTEXT <context name>
```

The context is similar to the execution stack in programming languages. When a subquery requests a new context, the new context is pushed on top of the stack. A new backplane function, *new_context()*, creates and pushes the new context. The new context remains on the stack until completion of the subcatalog in which it was created, when it is popped off the context stack. Popping the context is actually an implicit pop: allocation is explicit, but the matching deallocation is not. All context elements on the stack are always visible.

2.4.3 A Simple Example

As a simple example from the data language domain which exhibits the use of the context, consider the combination of Quel `RANGE` and `RETRIEVE` statements:

```
RANGE OF T IS R

RETRIEVE (T.a)
WHERE T.a > 10
```

Once the `RANGE` statement has been processed, the alias T continues in effect for relation R until it is redefined in another `RANGE` statement. While it is in effect, it must be maintained independent of any other Quel statements so that it can be referenced as needed.

In Table 2.7 we define five subcatalogs which partially parse the Quel `RANGE` and `RETRIEVE` statements. (For brevity, we omit subcatalogs which parse standard parts of the language such as the *relation*, *identifier*, *expr_list*, and *predicate* subcatalogs, which model relation names, identifiers, expression lists, and predicates, respectively.)

```
BEGIN CONTEXT C_i
   rel_list     :   List[relation_alias]
   proj_list    :   List[Attribute]
END CONTEXT C_i
```

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

```
ret           =   { new_context(C_i); }
                  print[ " RETRIEVE _r ", r:eval ]
;
eval          =   compute[ " _e _s ", e:expr_list, s:access ]
                  { ext_attr(e,proj_list); }
;
access        =   retrieve[ " _w ", r, proj_list, w:where ]
                  { r:rel_list;
                    ext_rel(proj_list,rel_list,r);
                    ext_rel(w,rel_list,r);
                  }
;
where         =   noop[ " WHERE _p ", p:predicate ]
;
```

Table 2.7: Partial Specification of Quel RANGE and RETRIEVE Statements

The *decl* subcatalog parses the `RANGE` statement. Since the only purpose of a `RANGE` statement is to define aliases for relations, the only action is the addition of the alias to the relation list in the context, *new_alias(rel_list,[(R, T)])*[6]. Specifying the *noop()* backplane function with a NULL parameter produces an empty operator tree.

The *ret* subcatalog, recognizing the `RETRIEVE` statement, sets up the call to *print()*. Next, the *eval* subcatalog sets up a call to *compute()*, which applies the expressions listed in its expression list parameter e to its stream parameter s, computing the values to be displayed. In its post-action, the *eval* subcatalog uses the directive function *ext_attr()*[7] to create a list of all attributes referenced in the expression list. This list is used in the *retrieve()* function as the list of attributes for projection.

In the *access* subcatalog, the relation retrieval is set up by composing a call to the *retrieve()* function and by two calls in the post-action to *ext_rel()*[8]. Information for the *retrieve()* function—relation aliases and the projection list—is supplied by references to the local variable r and the global variable *proj_list*. (Since information can be added to a global variable from any subcatalog, this context element is referenced after parsing the statement is complete.)

Finally, the *where* subcatalog parses the WHERE clause. It consumes the WHERE keyword, setting up the remaining input for parsing by the *predicate* subcatalog (not shown in Table 2.7). No backplane function is called, so the *noop()* backplane function is specified.

Some directive functions have already been introduced. We now discuss more generally the functions which may be called from a directive section.

2.4.4 Directive Functions

We found in modeling data languages that for some backplane function parameters, the standard operator tree does not coincide cleanly to the desired operator tree. The directive functions provide a way to modify an operator tree even as it is being produced. Modification operations on operator trees include rearranging nodes and augmenting the operator tree with additional nodes. Any modification to the emerging operator tree can be done if a directive function is defined and added to Rosetta. Because of the likelihood that new language requirements will outstrip

[6] *new_alias()* maintains the list of aliases *rel_list*, adding new aliases, deleting re-defined aliases, and checking that all relation references are valid.

[7] *ext_attr()* returns in its second parameter a list of all attributes referenced in its first parameter.

[8] *ext_rel()* extracts the relation references from a list of expressions, making sure that they are defined either in the schema or in *rel_list*.

any set of fixed capabilities, we allow the addition of new directive functions; thus, Rosetta is itself extensible.

Note that there is a definite distinction between directive functions and backplane functions: directive functions are used solely in *syntactic* modeling of languages whereas the backplane functions are applied only in the *semantic* modeling of languages. There is no overlap between the two.

Only one requirement is imposed on directive functions: a directive function must not introduce changes which result in type violations. In particular, an operator tree (and each of its subtrees) must compute a result of the same type before and after modification.

An example of a situation when directive functions are necessary is in the computation of variance with the computational formula, $\frac{1}{n}\sum x_i^2 - \overline{x}^2$, which involves both aggregate and arithmetic operations.

When translated into a data language, the computational formula for variance is a single expression and is parsed as a unit, but it is evaluated by calling two backplane functions, *compute()* and *aggregate()*. The call to *compute()* evaluates the arithmetic operations; the call to *aggregate()* evaluates the aggregate operations. Calls to these functions can be nested; e.g., to evaluate the first term, *compute()* evaluates x_i^2, *aggregate()* sums that result, then *compute()* divides the sum by n.

An operator tree built while parsing the expression for computing variance of the attribute Age is shown in Figure 2.2a; the corresponding operator tree which actually evaluates the variance is shown in Figure 2.2b. (A more detailed explanation of this example is given in Section 2.6.)

The original operator tree of Figure 2.2a needs major surgery to transform it into the desired operator tree of Figure 2.2b. To manipulate the nodes of the original operator tree into the desired operator tree, we devised the *rewrite()* directive function.

The Rewrite Function. The *rewrite()* directive function makes node rearrangement explicit. Before explaining the operation of the *rewrite()* directive function, we present some definitions which will simplify its explanation.

In the implementation of Rosetta, every node is tagged at its creation. We make use of the tags in rearranging the tree.

Given a list t_a of tags and a tree t_r, we say that a node of t_r is *t-tagged* if its tag is an element of t_a.

We define the *cluster*(t_a,t_r): If the root of t_r is t-tagged, then it is included in the cluster; otherwise, the cluster is the empty tree. For each node in the cluster, each t-tagged child is included in the cluster. Thus, the cluster(t_a,t_r) is the *maximal*

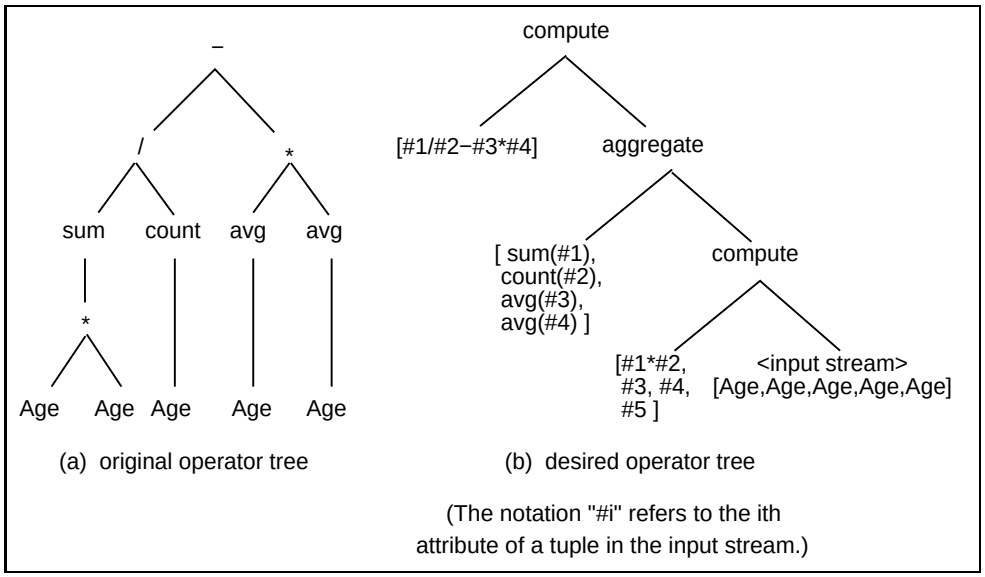

Figure 2.2: Computing Variance: Original and Desired Operator Trees

group of neighboring t-tagged nodes which includes the root. Note that the cluster is a tree.

Finally, we define the *residual subtrees* as the maximal subtrees consisting entirely of nodes which are not in the cluster(t_a,t_r). Figure 2.3 shows a tree which has been separated into its cluster and residual subtrees.

We use these definitions to define the *rewrite()* function:

```
rewrite(ta,tr,c):Boolean.
```

The first parameter, *ta*, is a list of tags (tags are strings); the second parameter, *tr*, is a list of operator trees; and the third parameter, *c*, also a list of operator trees, is an output parameter for the clusters.

One by one, and proceeding left to right through *tr*, for each operator tree tr_i in *tr*, *rewrite()* determines the *cluster(ta,*tr_i*)*, then detaches the cluster from the residual subtrees. A call to *rewrite()* leaves the residual subtrees in the list *tr* and returns the list of clusters in the output parameter *c*. As each residual subtree is detached, *rewrite()* replaces it in the cluster with a node which refers to it by its position in *tr*, providing a handle for access during evaluation to the result it computes.

In Figure 2.3, the original operator tree on the left is rewritten as a cluster and three subtrees. In the cluster, the first subtree is referenced by #1, the second by #2, and the third by #3. The references #1, #2, and #3 are used as placehold-

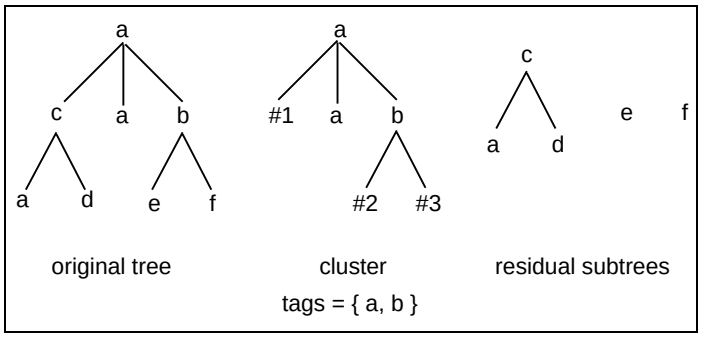

Figure 2.3: Application of *rewrite()* to an Operator Tree

ers referring to the first, second, and third attributes of the tuples of the stream generated by evaluating the subtrees.

The call to *rewrite()* succeeds if, after processing all input trees in *tr*, at least one non-empty cluster is detected, i.e., if *c* is not empty. Otherwise, *rewrite()* fails.

Combining *rewrite()* with the cycle subcatalog allows us to produce the operator tree of Figure 2.2b from the original tree of Figure 2.2a. An example combining the two will be given in Section 2.6, after a closer examination of the cycle subcatalog.

2.5 Cycle Subcatalogs

In form, the cycle subcatalog is the same as any other subcatalog:

```
<subcatalog name>  =
        < signature_0 >
    |   < signature_1 >
    |       ...
    |   < signature_n >
```

However, the cycle subcatalog consists exclusively of cycle signatures. The cycle signature, defined in Section 2.3, adds the ability to inject function calls into the operator tree. It does not rely on the detection of particular syntax in the input; instead it is enabled if a specified condition tested by a boolean function is satisfied. Input is not consumed in a cycle subcatalog; the objective is to re-process previously parsed input. The cycle subcatalog can augment the operator tree with as many additional functions as necessary.

The cycle subcatalog may have an arbitrary number of recursive signatures; that is, signatures in which the cycle subcatalog is itself named as one of the param-

eters of the invoked backplane function. As the cycle subcatalog is not referenced in the post-action or elsewhere, no variable name is needed in the parameter list.

In addition to the recursive signatures, there must be at least one non-recursive signature; this is the *exit signature* for the cycle subcatalog. The non-recursive signature names as a parameter the subcatalog through which parsing continues.

For example, consider the *proc_expr* subcatalog in Table 2.8. The first two cycle signatures are recursive, each incorporating the *proc_expr* subcatalog as the second parameter, while the third signature is the non-recursive exit signature. Parsing continues with the *f_ret* subcatalog.

Besides the cycle subcatalog, parameters of the cycle signatures' invoked functions may be local or global variables. Just as local variables referenced in the parameter list of a parameterized signature are instantiated in the syntax pattern, so local variables declared in the parameter list of a cycle signature are instantiated by the condition function. Global variables may also be instantiated elsewhere, in another subcatalog.

During parsing, the conditions of the signatures of the cycle subcatalog are tested *in the same order* as they are entered. Conditions are tested until one succeeds; whereupon the associated function is added to the operator tree. While the signature applied is recursive, this process is repeated until a non-recursive signature is applied. As the conditions of the cycle signatures are independent, they are not necessarily mutually exclusive and multiple conditions may hold simultaneously. Thus, to prevent premature exit, the exit signatures are listed last. If no condition is successful, an error is signaled and parsing fails. As for any recursive process, processing may not terminate if the conditions are not well designed.

We impose certain restrictions on cycle signatures which increase the clarity of the model without loss of generality. First, cycle signatures may have post-actions but not pre-actions. As input is not being consumed, the pre-action can be easily folded into the condition or the post-action. In addition, cycle signatures cannot be combined in a subcatalog with parameterized and conversion signatures. Such a combination would make it difficult to interpret the language specification, as it would be unclear whether processing of previously read input or consumption of additional input should take place. Instead, multiple smaller cycle subcatalogs can be interspersed with parameterized and conversion subcatalogs to achieve the same effect.

2.6 A Larger Example

To give a clearer picture of the interactions between the context, local variables, directive sections, cycle subcatalogs, and the *rewrite()* function, we now discuss a larger example—a simple form of the SQL `SELECT` statement:

```
SELECT  <attr_list>
FROM    <relation_list>
WHERE   <predicate>,
```

where items delimited by angle brackets < and > represent parameters.

Table 2.8 gives a partial specification of subcatalogs for the simplified SQL `SELECT` statement. Again, to keep the example brief, low-level subcatalogs' defini-

```
BEGIN CONTEXT C
    rel_list        :       relation_list
    proj_list       :       attribute_list
    xpr             :       expression_list
END CONTEXT C
```

```
display         =       { new_context(C); }
                        print[ " SELECT _x ", x:estream ]
;
estream         =       noop[ " _xpr ", proc_expr]
;
proc_expr       =       compute[ rewrite(xpr,"+ - * /",x), x:expression_list, proc_expr]
                |       aggregate[ rewrite(xpr,aggr-ops,x), x:expression_list, proc_expr]
                |       noop[ is_attr_list(xpr), f_ret]
;
f_ret           =       retrieve[ " FROM _r _p ", rel_list, proj_list, p:where ]
                        { r:relation_list;
                          insert_attr(proj_list,xpr);
                          insert_rel(rel_list,r);
                        }
;
where           =       noop[ " WHERE _p ", p:predicate ]
                        { a:attribute_list;
                          ext_attr(p,a);
                          insert_attr(proj_list,a);
                        }
                |       ϵ
;
```

Table 2.8: Rosetta Specification of Simplified SQL SELECT Statement

tions are not shown. (References to those subcatalogs are italicized.)

For this example, we assume the relation Student(SSN,Name,Ranking,Age) and compute the statistical variance of Age for students whose Ranking is Junior. We translate the computational formula for variance, $\frac{1}{n}\sum x_i^2 - \overline{x}^2$, into an SQL SELECT statement:

```
SELECT  (sum(Age*Age) ÷ count(Age)) - (avg(Age)*avg(Age))
FROM    Student
WHERE   Ranking=Junior
```

In processing this SQL statement with the specification of Table 2.8, the *display* subcatalog first sets up the new context for the query and consumes the `SELECT` keyword. The remaining input is parsed from the *estream* subcatalog.

The sole option of the *estream* subcatalog is exercised: it references two other subcatalogs, of which the second is a cycle subcatalog. First, the expression `(sum(Age*Age) ÷ count(Age)) - (avg(Age)*avg(Age))` is parsed via the *expression_list* subcatalog into the context element *xpr* (see Figure 2.4). Next, the second subcatalog reference, *proc_expr*, is activated.

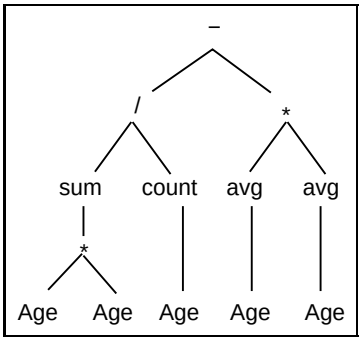

Figure 2.4: Original Operator Tree for Variance Computational Formula

The *proc_expr* subcatalog has three options of which only the first can succeed. The second option cannot succeed because it depends on the condition function *rewrite()* with aggregation operators as tags; that call to *rewrite()* fails because the root of the input operator tree is not an aggregate operator. The third option depends on the condition *is_attr_list()*, which succeeds only if the context variable *xpr* is a list of attributes; however, *xpr* is a list whose sole element is an expression.

But the first option, which is contingent on *rewrite()* being able to remove arithmetic clusters, succeeds. The call to *rewrite()* manipulates the expression list; there is one cluster and four residual subtrees (see Figure 2.5). After *rewrite()*, the context variable *xpr* contains the four residual subtrees and the local variable *x* contains the single cluster. Parsing continues with the *proc_expr* subcatalog.

This time, because the root nodes of all four clusters are aggregate operators, only the second option of *proc_expr* succeeds, finding four clusters and four residual subtrees (see Figure 2.6). Again, the clusters are elements of x; the residual subtrees are left in the context variable *xpr*; parsing continues in the *proc_expr* subcatalog.

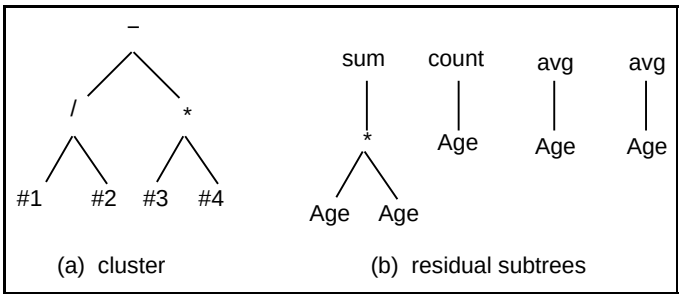

Figure 2.5: Result of First Call to *rewrite()*

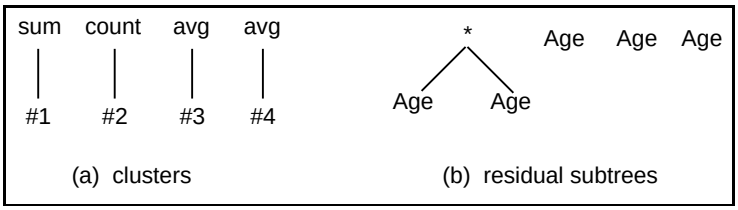

Figure 2.6: Result of Second Call to *rewrite()*

On the third iteration, only the first option of *proc_expr* succeeds. *rewrite()* finds only one cluster and five residual subtrees (see Figure 2.7). Parsing continues with the *proc_expr* subcatalog.

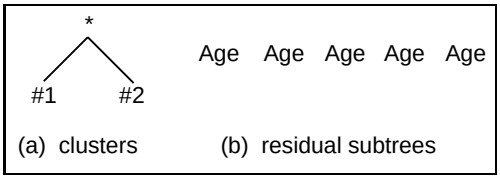

Figure 2.7: Result of Third Call to *rewrite()*

After the third iteration, *xpr* contains a list of attributes, [Age, Age, Age, Age, Age], so that neither of the conditions of the first two alternatives succeeds. As the condition of the exit signature (the third alternative) succeeds, cycling terminates successfully, and parsing continues with *f_ret*, the exit subcatalog.

The *f_ret* and *where* subcatalogs consume the remaining input:

```
FROM    Student
WHERE   Ranking=Junior
```

Figure 2.8 shows the operator tree produced by this chain of subcatalogs. Evaluation of the operator tree is bottom-up. First, *retrieve()* creates, from the tuples of the relation Student which satisfy Ranking=Junior, a stream of five-tuples, [Age,Age,Age,Age,Age] [9]. It is input to *compute()*, which produces a stream of four-tuples: [Age*Age,Age,Age,Age]. Next, *aggregate()* produces the four-tuple [sum(Age*Age),count(Age),avg(Age),avg(Age)]. Finally, *compute()* applies the division, subtraction, and multiplication to produce the final result, which is then printed.

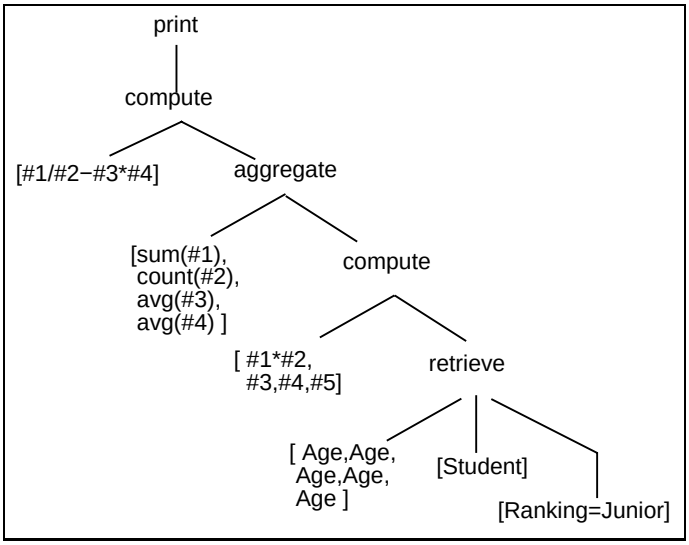

Figure 2.8: Final Operator Tree for Variance Query

The *proc_expr* subcatalog iteratively processes the input expression. Without consuming any more input, the first two options of the *proc_expr* subcatalog iteratively process the global expression list *xpr*, using *rewrite()* to extract clusters of arithmetic operations or aggregations. Finally, when the expression list consists of attributes only, neither call to *rewrite()* can remove any more clusters, cycling terminates, and parsing continues with standard subcatalogs.

During evaluation of the query, the retrieved stream will undergo either computation (arithmetic evaluation or aggregation or both) or projection. The *proc_expr* subcatalog is an artifact of the necessity of separating arithmetic operations from

[9] Clearly, a simple optimization eliminates replicating the retrieved attribute values; for clarity of exposition, we did not apply it.

aggregations[10].

2.7 Summary

Data language extensibility can be achieved by leveraging a common but extensible backplane of database operators. The semantics of data languages can be expressed in terms of the set of operator trees that can be formed; we have expressed such sets in terms of rewrite rules called subcatalogs.

Data language syntax can be customized by grafting the invoking syntax signatures onto backplane operator trees. Special functions called directive functions were introduced to add flexibility and generality to language specification. Local variables, context variables, and directive functions all interact in the directive sections to allow the manipulation of operator trees. Local variables allow information-sharing within a subcatalog; global variables allow information-sharing across subcatalogs; and directive functions can use both in manipulating nodes of the operator tree. As the generated compiler creates the operator tree in a standard form, these features are necessary to accommodate quirks in data languages which may cause the standard operator tree to fall short of the desired operator tree.

[10] Although expressions may include both aggregation and arithmetic operations, these are evaluated by different backplane functions—*aggregate()* and *compute()*.

Chapter 3

The Rosetta Prototype

In Chapter 2, we discussed the Rosetta specification language. In this chapter, we consider the implementation of the generator prototype, which maps a Rosetta language specification to a compiler for the language.

The input to Rosetta (see Figure 3.1) is a Rosetta language specification and the set of backplane function definitions. The generator type-checks the specification using the backplane function definitions and maps the specification to bison[DS91] and flex[LS86] files. These files specify a lexical analyzer and parser and, together with C files which define compiler structures and compiler utilities, comprise the generated compiler for the customized data language. The generated compiler maps a statement of the language to an operator tree which can be used by a query evaluator to evaluate the statement, possibly after optimization.

In the rest of this chapter, we explore how the prototype generates the bison and flex files from a specification. We discuss first the structure of the bison file and show how we generate its specification, then we do the same for the flex file.

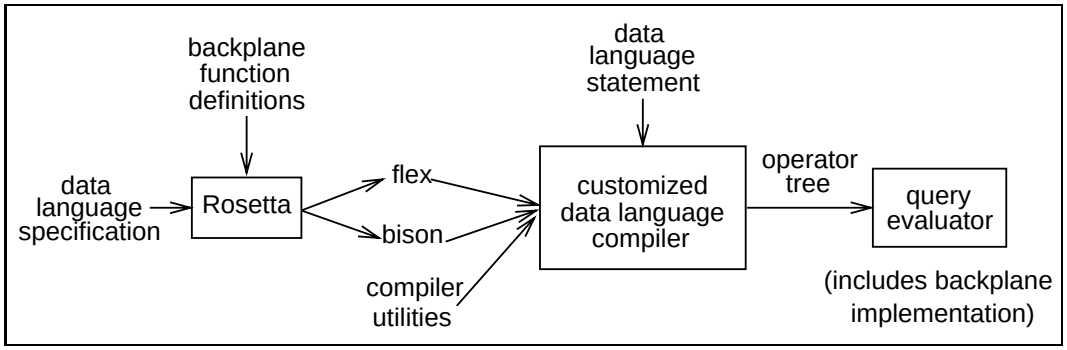

Figure 3.1: Rosetta Generator Prototype

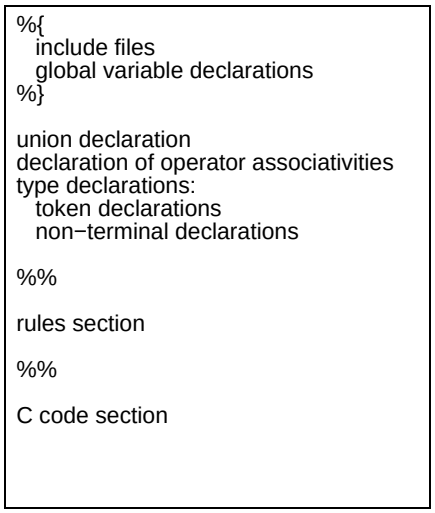

Figure 3.2: Structure of the Bison File

Next, we consider the type-checking that can be done during translation. Finally, we conclude with a summary of the prototype.

3.1 Generating the bison file

The bison file is generated after a complete pass over the specification file. A specification file is composed of three sections: a set of context definitions, possibly empty; a non-empty set of subcatalogs; and a set of associativity and precedence declarations, also possibly empty. Each of these is mapped to the bison file in its own way. We first discuss the structure of the bison file, and then discuss the role each part of the specification file plays in generating the bison file.

3.1.1 Structure of the bison file

There are three sections in a bison file: the declarations section, the rules section, and the C code section (see Figure 3.2). The sections are delimited by a line containing only the string **%%**. Information from the specification file is disseminated among the three sections but is concentrated in the second.

The declarations section consists of several distinct subparts: a section (delimited by **%{** and **%}**) to be included literally in the final C file, followed by a union declaration, declarations for operator associativities, and type declarations for terminal and non-terminal symbols.

The second section, the rules section, is the heart of the bison file, and

correspondingly most of the work goes into mapping the subcatalogs into the rules. The syntax of a bison rule is quite simple:

```
symbol  :   pattern
        |   ...
;
```

Each rule consists of a symbol on the left hand side, which defines its name, and a right hand side which defines everything else: the pattern (terminal and non-terminal symbols) which must be observed in the input and any C code which is to be executed as part of the rule. Rule options are delimited by a |, and the right hand side may include any number of options.

Finally, the third and last section of the bison file consists of C code which is copied unchanged to the bison output file. This may include additional C definitions as well as C functions.

3.1.2 Generating the bison file

The first two elements of the declarations section are generated by rote; no language-specific information is necessary. Following these, the declarations of operator precedence and associativity are copied in directly from the specification file. The last element of the declaration section declares types for the terminal and non-terminal symbols. The type declarations are language dependent; they are generated from symbol tables after the specification file has been processed. In order to make generation feasible, token types are standardized: all terminals return values of type *string* and all non-terminals return values of type *nodeptr*[1].

Most of the work in generating the bison file is in handling the context variables and subcatalogs. We discuss first how contexts are incorporated, then discuss subcatalogs.

Contexts. Contexts provide global memory during query compile-time. However, we have to generate code at compiler generation-time to manage context variable references.

A context variable is globally available whenever the context which defines it is active[2] (see Chapter 2.4.2). To manage context variables, we define a global structure and functions to store and access values of context variables. This global

[1] *string* is defined as `char *` and *nodeptr* is a pointer to a C `union` which will constitute the nodes of the operator tree.

[2] A context X is *active* whenever it has been allocated by a call to *new_CNTXT(X)* and has not yet been deallocated. Once a rule activates a context, the context is active until the rule is satisfied. Thus it is active while the sub-goals of the activating rule are being satisfied.

```
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 3.3: Mapping a Rosetta Specification to a Bison File

structure is the *context stack*. Allocating a context adds enough space at the top of the context stack to accommodate the variables of the named context.

To make context definitions available at run-time, we generate a function (in the C code section of the bison file) which both allocates and initializes a structure storing context definition information. We include in this structure the context name, the number of variables, and their names and types.

Subcatalogs. In Figure 3.3, we show how a Rosetta subcatalog maps to elements of a bison file. Each Rosetta subcatalog is mapped to a single bison rule, with its signatures mapping to rule options. Each rule generated recognizes some part of the customized language and creates a node for the operator tree. Each subcatalog name becomes a bison non-terminal name, and its type is declared in a bison `%type` statement. Subcatalogs have many parts; each is discussed in turn.

Signatures. Each signature of a subcatalog maps to an option of the associated rule, with the signature pattern mapping to the rule option pattern. Elements of the signature pattern are either keywords or parameters[3]. The keywords must appear precisely in the input while the parameters, which represent varying input, are mapped to non-terminals (bison rule names).

A keyword cannot be copied directly into the bison file as it may contain characters special to bison and flex, e.g., '*', '.', etc. For each keyword, a unique identifier is generated which represents the keyword in the pattern and is also used in the token type declaration in the bison file.

A parameter is replaced in the bison pattern by its associated subcatalog type name, which is defined in the parameter list, in a pre- or post-action's list of local variables, or in a context. As an example, the parameters of the first signature in Figure 3.3 are mapped to *display* and *display2*. In accordance with the scoping rules for Rosetta variables, we look first for a definition in the parameter list; as a parameter definition may be arbitrarily deeply nested in a function call in the parameter list, this look-up is done recursively.

A parameter may also be defined as a local variable in either a pre- or post-action. Local variables are available only in the subcatalog in which they are defined, and their definitions are maintained in a local symbol table[4].

Finally, a parameter which is not defined in either the parameter list or a pre- or post-action must be a context variable. A context variable reference is also replaced by its defining subcatalog name, but it is looked up in the stack of active contexts.

An error is reported if a parameter is not defined in the parameter list, in a pre- or post-action, or in a context.

Pre- and Post-Actions. Pre- and post-actions contribute both local variable definitions and function calls. Since local variables are available only in the subcatalog in which they are declared, each signature has an associated symbol table in which we can look up a local variable's defining subcatalog name whenever necessary. Furthermore, code is added after the pattern to initialize the value of the local variable so that its initial value can be passed to any functions that reference it. Wherever a local variable is referenced as a parameter, it is replaced with a function call which resolves the name to a pointer to be passed.

Furthermore, functions which are called in the pre- and post-actions are intended to be called before or after the pattern of the rule option is matched,

[3]Recall that parameters are distinguished from keywords by a '_' prefix, e.g., in Figure 3.3, "_x" and "_y" are parameters while UNION, INTERSECT, and EXCEPT are keywords.

[4]At run-time, memory is allocated for local variables in a global stack.

respectively. Since these functions are called at run-time, they are simply copied into the bison file instead of having nodes created for them.

Creating Operator Tree Nodes. After the post-action function calls are copied into the bison file, C code is generated to allocate and initialize an operator tree node for calling the specified backplane function. Each node of the operator tree is a union, so we allocate the node, initialize its tag, and allocate and initialize its variant fields. We convert the string naming the function to an internal function code and set the function code. From the backplane definition, we determine the number of arguments, set the argument count field, and allocate and initialize the parameter node pointers. In Figure 3.3, the function to be invoked is *union()*, and it has two arguments which are initialized to the pattern elements *display* and *display2*.

The parameters are matched by other rules and their nodes are created and initialized in those other rules, so each element of the parameter list is set to the appropriate parameter subtree using the $\$i$ notation[5]. Parameters which are matched in the pattern are initialized by the bison $\$i$ reference; other parameters are either local or context variables and are initialized by a function call to access variable storage.

Finally, we have to create and initialize nodes for the backplane functions nested in the parameter list, if any. As functions in the parameter list can be arbitrarily deeply nested, node allocation can be arbitrarily deeply nested.

Automatic Context Allocation and Deallocation. A context is explicitly allocated by a call to the backplane function *newCNTXT()* from a pre- or post-action; however, there is no corresponding deallocation function. Instead, a context is active as long as its associated rule is unsatisfied, and is automatically deactivated when the rule is satisfied. Allocating a new context adds a context element to the context stack, making variables defined in the named context available for reference in the rule and whatever rules it references.

The matching call to *dropCNTXT()* is the last function called in the action of a rule; it follows the code generated to create an operator tree node. It removes the allocated context from the active contexts stack, making context variables inaccessible, but does not reclaim memory allocated for the variables as that would potentially destroy values needed at query evaluation-time.

Conversion Subcatalogs. Conversion subcatalogs (see Chapter 2.3) are handled similarly to parameterized subcatalogs. A conversion subcatalog is also mapped to

[5] $\$i$ is bison notation which refers to the ith element matched in the pattern. There is also a special notation $$ which refers to the value of the current bison rule.

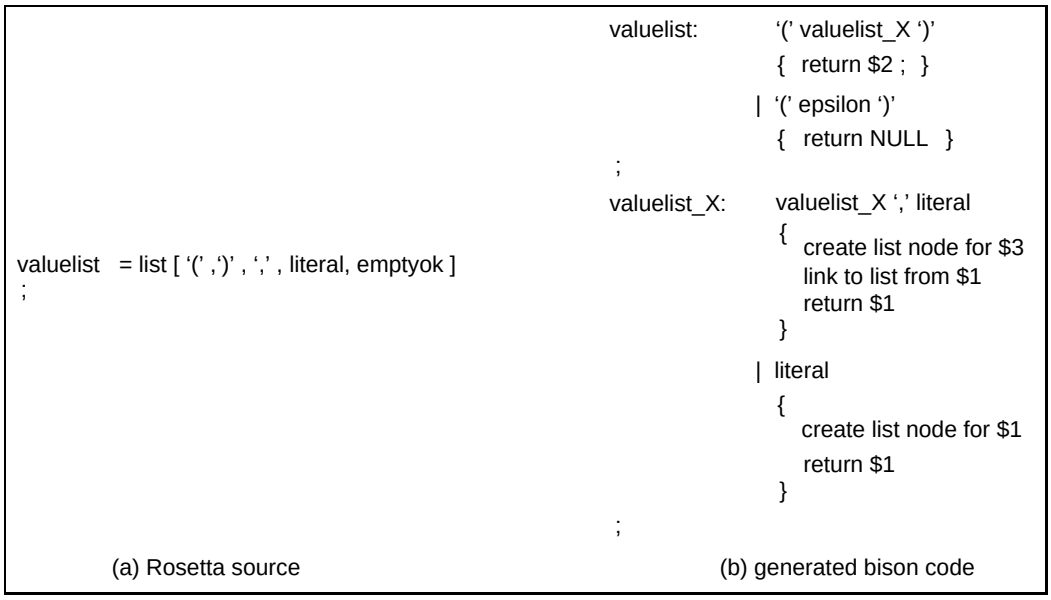

Figure 3.4: Mapping a List Subcatalog to a Bison Rule

a bison rule, with the subcatalog name acting as the non-terminal which names the bison rule. Each conversion signature is mapped to a bison rule option.

However, a conversion signature differs in that its invoking syntax is a regular expression. As for keywords, we generate a unique token identifier from it and declare its type in the bison file as <*string*>. The unique token identifier is the only element of the bison rule pattern. Following the pattern, generated C code allocates and initializes a node for the backplane conversion function specified in the signature. Node generation for conversion functions is simplified by the uniformity of conversion function definitions: all conversion functions have exactly one parameter—the string matched by the regular expression.

List Subcatalogs. List subcatalogs (see Chapter 2.3) are handled differently from either parameterized or conversion subcatalogs. In Figure 3.4, we show an example of how a list subcatalog is mapped to bison rules.

A list subcatalog maps to not one but *two* bison rules. The first rule consumes the brackets enclosing the list[6] and matches the sequence of list elements with a reference to the second rule.

The second rule has two options: the first option recursively matches a sequence of elements, a separator (if specified), and the last element of the list; the

[6]Brackets need not be specified for lists; if they are not, those positions in the generated rule are simply left empty.

second option matches a single element.

Furthermore, each rule option has some generated C code which collectively creates and initializes a list. Unlike other subcatalogs, a list subcatalog does not invoke a function. Instead, a node is allocated and initialized for each element of the list and then additional code links the nodes together tagged as list elements.

This is the standard generation pattern for a list. But list definitions have variations. The list may be a unit list[7], which has exactly one element. In this case, the first rule is the only rule, and it consumes the brackets and inside the brackets matches exactly one element.

In another variation, a list definition may allow the empty list, in which case the first rule has a second option which matches the empty string: its pattern consists of a reference to the *epsilon* subcatalog[8]. If the list *cannot* be empty, then the second option is simply not included, thereby forcing the rule to match at least one list element.

Cycle Subcatalogs. Cycle subcatalogs, explained in detail in Chapter 2.5, are mapped to the bison file completely differently than are other subcatalogs. The differences extend to subcatalogs which merely *reference* cycle subcatalogs. Figure 3.5 illustrates the complexity of rewriting a cycle subcatalog as a bison rule. This example is taken from the SQL specification.

Cycle implementation is based on controlled error generation and detection. The cycle rule matches the error condition and clears it, evaluates one of the options, then regenerates the error condition if cycling should continue.

To initiate cycling, the bison rule generated from the subcatalog referencing the cycle subcatalog raises an error condition just before the reference to the cycle subcatalog. (See the mapping of the *comp_attr* subcatalog to the `comp_attr` bison rule in Figure 3.5.) Since it generates an error just before the reference to the cycle subcatalog, the parser rejects that rule option, but tries to satisfy the same rule with another option. Therefore, that pattern must be split into two parts: one which raises the error condition and another which matches the cycle subcatalog (and any others after it). Furthermore, the exit subcatalog of the cycle subcatalog is moved to follow the reference to the cycle subcatalog.

Unlike other subcatalogs, cycle subcatalogs do not consume additional input; instead, the signatures of a cycle subcatalog test condition functions. In the generated bison rule, these condition functions become the conditions of a nested *if* state-

[7] A unit list is simply a list with exactly one element. Unit lists are useful when a backplane function parameter is specified as a list but the language being modeled is more restrictive and allows only a single item.

[8] The *epsilon* subcatalog is a built-in subcatalog which matches the empty string.

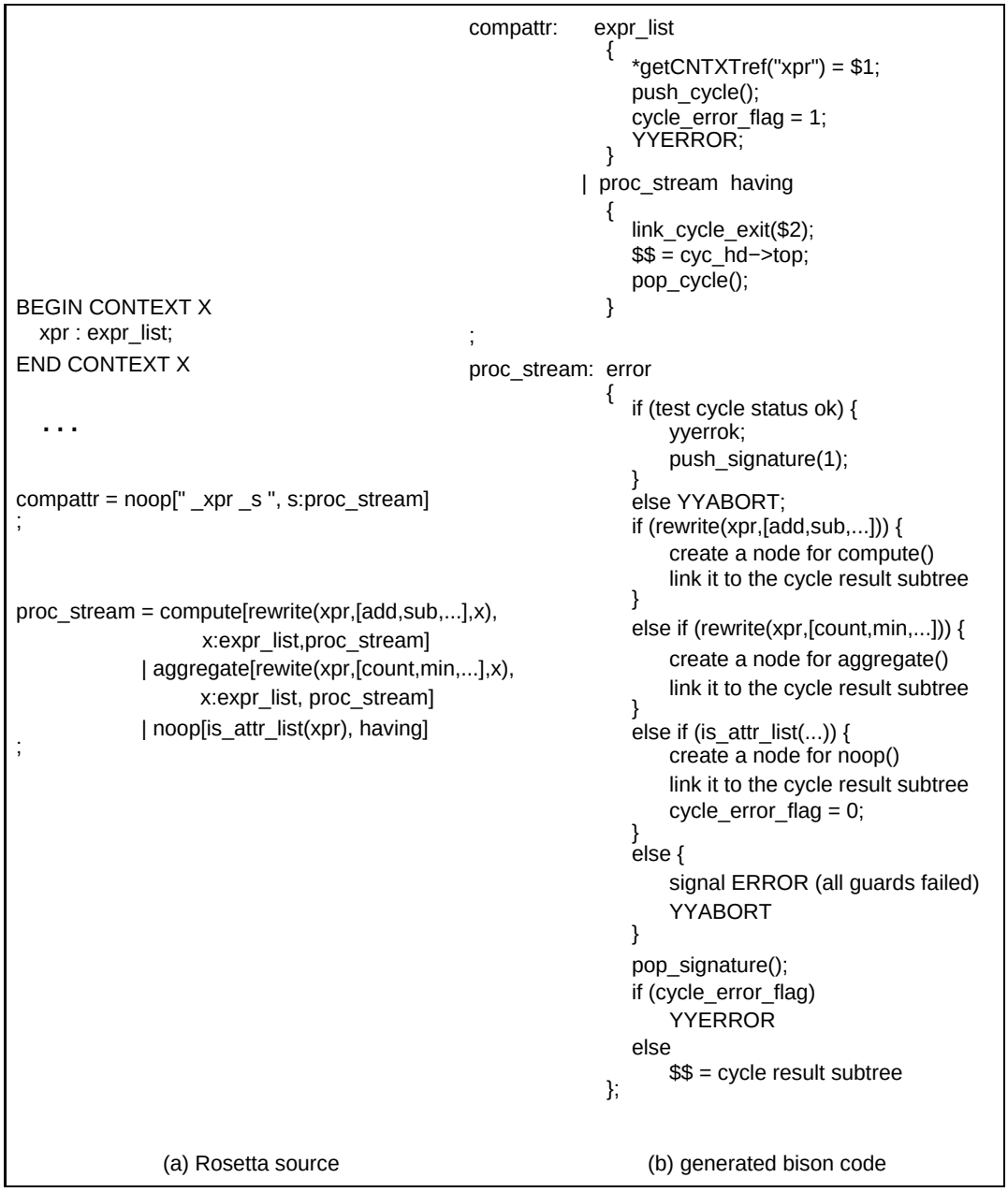

Figure 3.5: Mapping a Cycle Subcatalog to a Bison Rule

ment, and their post-actions become the conditional statements. The post-action of the first condition that is satisfied is evaluated, then either cycling terminates or condition testing begins anew with the first condition[9].

The first step in the cycle body is to test that the cycle is operating properly. If it is, the synthesized error is cleared and local variables are allocated; otherwise, parsing aborts.

The core of the cycle rule body is a nested *if* statement which selects between the alternatives of the cycle. Its conditions are the conditions of the cycle subcatalog and the conditional statements are the post-actions of the conditions of the cycle subcatalog coupled with some generated C code which creates an operator tree node and links the node to the result of the cycle. Cycling terminates when a non-recursive option is selected.

To continue cycling after the nested *if* statement, the error condition is re-generated. To terminate cycling, no new error condition is generated; instead, the result of the cycle subcatalog is assigned to the `$$` value.

The last detail is arranging for the cycle subcatalog to return a result. Cycle subcatalogs return results just as other subcatalogs do. However, the results must be accumulated in a global structure as each cycle generates a new error condition, causing the parser to discard the result of the current cycle. Instead of assigning the return value to `$$` each time, a chain of result values is maintained in a global structure and finally in the exit cycle the result is set to the accumulated structure. The global structure is initialized when we set up for cycling (in Figure 3.5, in the *comp_attr* subcatalog) and demobilized when cycling terminates.

3.2 Generating the flex file

In addition to generating the bison file, we generate a flex file which specifies the lexical analyzer for the customized compiler. Only the elements of the patterns in the signatures of the specification file play a role in generating the flex file. This section will discuss first the structure of the flex file and then how it is generated.

3.2.1 Structure of the flex file

Like the bison file, the flex file has a fixed structure to which our generated file must conform (see Figure 3.6). It too has three major sections, again separated by a line containing only a `%%`. The first section consists of C code (definitions, functions, etc.) which is copied into the flex output file. The second section consists of flex

[9]Recall that signatures of a cycle subcatalog may not have pre-actions.

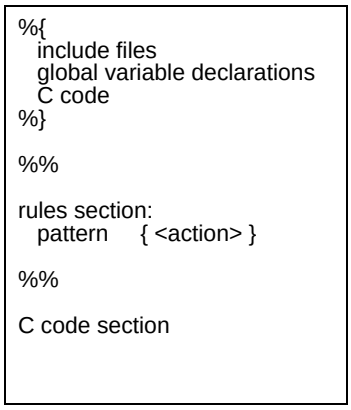

Figure 3.6: Structure of the Flex File

rules. Each rule consists of a regular expression to be matched followed by an action (written in C) to be executed when the expression is matched. The last section also consists of C code which is copied into the flex output file.

3.2.2 Generating the flex file

By comparison with generation of the bison file, generation of the flex file is quite simple (see Figure 3.7). The contents of the first and third sections are fixed so they can be generated by rote, without reference to the specification file. The rules section is somewhat more complex.

Our generated rules fall into two categories: those which match keywords and those which match regular expressions (e.g., from conversion subcatalogs). In order to avoid matching a keyword with a more general regular expression, we generate first the keyword rules and then follow them with the regular expression rules.

The keywords and the tokens generated to represent them are available from a symbol table. So we simply dump the entries of the symbol table (alphabetically ordered) and add a predefined action for each which augments the column count (useful for debugging) and returns the associated token name. For the rules matching a regular expression, we also duplicate the matched string and return the copy in a bison global variable.

Finally, two rules which match and discard white space are automatically added. One rule matches strings of spaces and tabs and augments the column count; the second matches the end-of-line character, increments the line count, and resets the column count to zero.

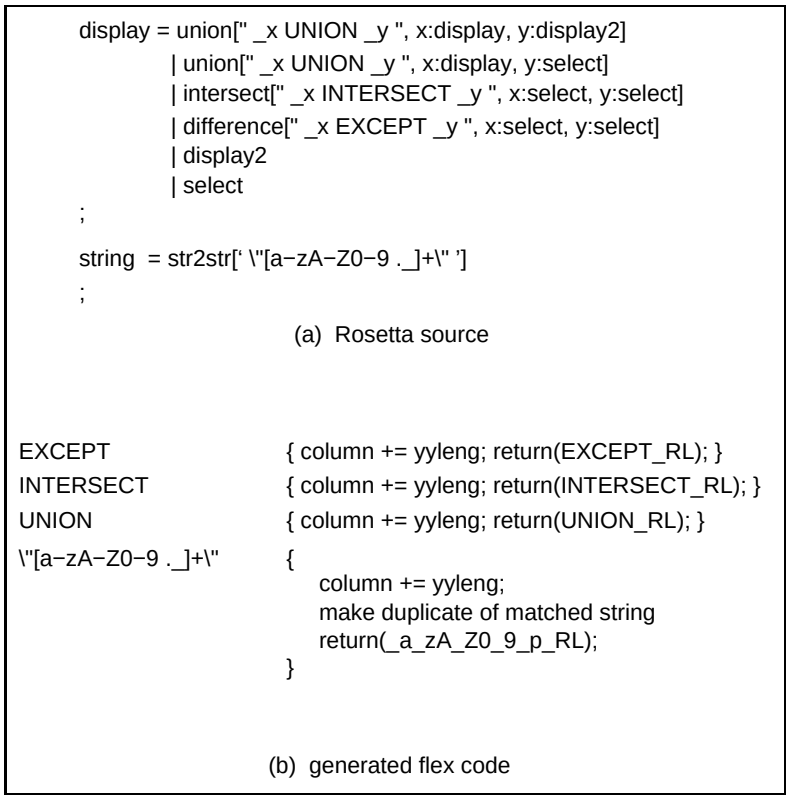

Figure 3.7: Mapping a Rosetta Specification to a Flex File

3.3 Static Type Analysis of the Specification

We type-check the specification to determine, as best we can at compiler generation-time, that the generated compiler can generate type-correct operator trees. This phase is called *static type-checking*. Generation-time type analysis is necessarily inexact as the base types of complex elements are indeterminate at compiler generation-time, e.g., an attribute of a retrieved tuple may be of any type. However, the types of all functions invoked are known: some return base types, others return streams or tuples.

Successful static type analysis means that the generated compiler constructs operator trees in which the composition of backplane functions is type-correct *if* parameters of the expected type are passed in to them. For example, an *add()* function returns either float or integer depending on its parameters, so if a float is passed in when an integer was expected, the function may return an undesirable type.

Consider an aggregation operator, such as AVG, which applies an expression

to a stream of tuples. It is incorrect for the expression to reference a string attribute; however, such an error cannot be detected until compiling an actual query. But we *can* ascertain, at generation-time, that the backplane function compositions produced by the generated compiler are potentially type correct.

Static Type Analysis. Subcatalog "types" are really just a restriction of defined base types, so subcatalog types have to be translated to defined types. The backplane function definitions are used for this.

The first step in doing the static type analysis is to analyze the subcatalogs and build the subcatalog dependency network (SDN), an array which represents the type relationships between signatures and subcatalogs. We say that a subcatalog s_i *depends* on another subcatalog s_j if some parameter of s_i is of type s_j. The SDN is a matrix, *number-of-signatures* $\times$ *number-of-subcatalogs*. The subcatalogs represent types here and the signatures represent function calls. During static type analysis, the subcatalogs are numbered sequentially, and the signatures are also numbered sequentially independently of the subcatalogs. SDN[i][j] == 1 means that the *i*th signature has a parameter whose type is the *j*th subcatalog.

After the subcatalog dependency network has been constructed, we further analyze the subcatalogs with a standard depth-first search procedure to identify the recursive cliques and the signatures that comprise them. At this point, the actual static type analysis begins.

Static type analysis proceeds recursively through the subcatalogs and signatures. A subcatalog reference can be satisfied by any of its signatures, so the type of a subcatalog can be inferred as the *union* of the types of its signatures. Thus, to determine the type of a subcatalog, we must determine the types of each of its signatures.

The type of a signature is the union of all types that can be returned by the backplane function *f()* it invokes[10]. Backplane functions may be multiply defined, with the type returned depending on the types of the parameters. For example, the *add()* backplane function returns an `integer` if both parameters are `integer` or a `float` if both parameters are `floats`, and also has other definitions. Certain combinations of parameter types will not be defined for a backplane function; we can't check for this now. All we can do is discover if it is *possible* to have a type-correct function call; if so, we include its type in the signature's type.

The type inference process is recursive (see Figure 3.8): for each subcatalog determine its type by determining the types of each of its signatures and then taking

[10] A signature can also be a simple subcatalog reference. When it is, the type of the signature is the same as that of the referenced subcatalog.

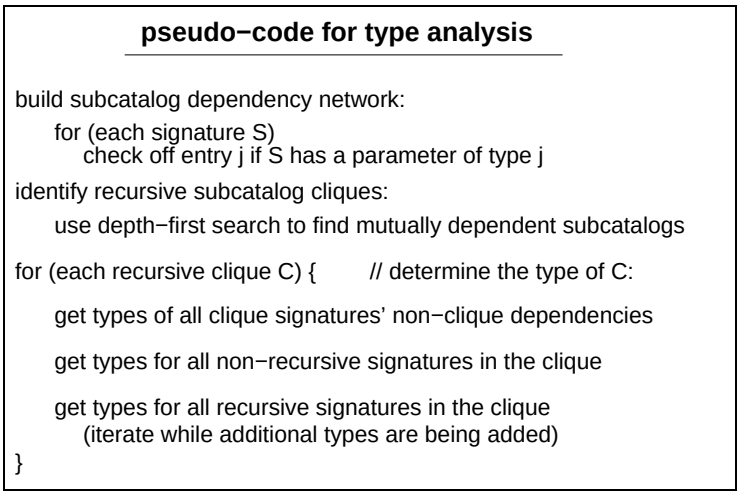

Figure 3.8: Type Analysis Algorithm

the union of those types. To determine the type of a signature, we recursively determine the types of the subcatalogs it depends on. Knowing these, we can determine from the backplane function definition which combinations of parameter types might occur and therefore which types a backplane function call might return. We take the union of the backplane function definitions which can be composed from this signature. Clearly, this process converges for non-recursive subcatalogs[11].

We treat recursive cliques similarly, but iterate over the entire clique. Type inference for the clique continues as long as we continue to infer a new type for *some* clique member in an iteration. Since types are never eliminated, the number of types inferred for clique members can only increase. Thus, as there is an upper limit on the number of types[12] that a recursive subcatalog can compute, eventually type analysis must terminate for the recursive clique.

3.4 Conclusions

In this chapter, we have discussed implementation details of how a Rosetta language specification is mapped to flex and bison files. Patterns for flex rules are generated from keywords and the regular expressions specified in conversion catalogs. Each rule returns its token in the flex variable provided for that purpose; rules matching regular expressions also return a copy of the matched string.

[11] Our base case is the conversion subcatalogs, which have known parameter type (string) and known output type.

[12] Recall that subcatalog "types" are mapped to base types, so no subcatalog can compute more types than the number of base types.

Some of the bison file is generated by rote, but most of it is generated from the specification file. Each subcatalog is mapped to one bison rule, with the exception of list subcatalogs, which are mapped to two rules. Each subcatalog name is mapped to a bison non-terminal, representing a bison rule whose options are derived from the subcatalog signatures. The patterns for the bison rule options are derived from a transformation of the invoking conditions of the signatures. Finally, C code is generated to build the operator tree which evaluates the input statement.

The prototype generator has been fully implemented and includes all Rosetta language features, including parameterized, conversion, list, and cycle subcatalogs, and local and context variables. To validate the approach, we used the Rosetta system to design models and generate compilers for five different data languages. Validation of the approach is the topic of the next chapter.

Chapter 4

Model Validation

4.1 Introduction

Previous chapters have discussed the design of the Rosetta specification language and implementation of the prototype Rosetta generator. This chapter presents experiences in modeling data languages with Rosetta. We present both anecdotal evidence of how easy it is to model languages and concrete evidence of the success we have had with code and specification reuse. Furthermore, we will show in our discussions of the modeled languages that Rosetta is able to model quite diverse languages.

Five languages were modeled: SQL [vdL89], SQL/NF [RKS89, RKB88], TSQL2 [Sno94], Quel [Dat87], and TQuel [Sno87]. Each presented its own challenges; we resolved them in a variety of ways: by modeling tricks, by defining new backplane functions, or by generalizing existing backplane functions.

We classify languages into families. A language family contains a parent language and languages derived from it. For instance, SQL/NF and TSQL2, both extensions of SQL, are members of the SQL family while TQuel, an extension of Quel, is a member of the Quel family. Our experiences with Rosetta are discussed within the context of language families because we found empirically that given a language specification, specifying a derivative of the language follows readily. Considering languages in families also demonstrates that using Rosetta can simplify language evolution.

In the following discussion, we will make use of a relation,

```
Canine(breed,size,low,high),
```

where the attributes represent a dog breed, its size category (small, medium, large, or giant), and integer values representing the minimum and maximum of the breed's

weight range.

The rest of this chapter discusses our experiences with Rosetta. Sections 4.2 and 4.3 focus on the SQL and Quel families, respectively. While discussing each language model, we will give examples of actual queries and the operator trees produced from them by the generated compilers. We close with a discussion of our results on usability and productivity in Section 4.4.

4.2 The SQL Family

The SQL family is large and diverse. Because SQL is the standard relational data language of the database community, it has had a significant impact on data language development, frequently serving as the base language to which researchers add experimental features. Although SQL itself is a traditional relational data language, the SQL family includes languages which support alternative data models, include new data types, and extend the syntax. We modeled three members of the SQL family: SQL; SQL/NF, a ¬1NF data language; and TSQL2, a temporal data language. The latter two were chosen because they both make significant but diverse extensions to SQL.

4.2.1 SQL

The first language we consider in the SQL family is SQL itself. Our model of SQL is based on the 1987 ISO standard, [vdL89]. It includes the data retrieval statement, SELECT; the set operation statements, UNION, INTERSECT, and EXCEPT; and the data modification statements, INSERT, DELETE, and UPDATE. This section considers only the high-level elements of the language model; the full SQL model, including low-level constructs such as arithmetic expressions and primitives, is specified in Appendix B.

4.2.1.1 Modeling SQL Statements

The SELECT Statement. Table 4.1a shows the syntax of the SELECT statement, one of the most complex statements in the language, and Table 4.1b outlines our model. The SQL SELECT statement includes five optional clauses: DISTINCT, WHERE, GROUP BY, HAVING, and ORDER BY. The remaining three clauses—SELECT, <*select list*>, and FROM—must appear.

In Rosetta, an optional clause is modeled by a subcatalog defined as a combination of signature enumeration and subcatalog union. For example, the *distinct* subcatalog of Table 4.1a enumerates one signature, which composes the *unique()*

```
SELECT
[ DISTINCT ]
    <select list>
FROM
    <relation list>
[ WHERE
    <predicate> ]
[ GROUP BY
    <attribute list> ]
[ HAVING
    <predicate> ]
[ ORDER BY
    <attribute list> ]
```

(a) syntax

```
BEGIN CONTEXT X
    rel_list : relation_list;
    proj_list : attr_list;
    xpr : expr_list;
END CONTEXT X

SQL         =    ...
             |  {  newCNTXT(X); }
                printstream[" _x ", x:display]
;
display     =    ...
             |  select
;
select      =  { newCNTXT(X); }
                noop[" SELECT _x ", x:order]
;
order       =  sort[" _y ORDER BY _x ", x:attr_list, y:distinct]
             |  distinct
;
distinct    =  unique[" DISTINCT _x ", x:compattr]
             |  compattr
;
compattr    =  noop[" _xpr _s ", s:proc_stream]
;
proc_stream =  compute[rewrite(xpr,[add,sub,div,mul,assign,
                   str2int,str2real],x),x:expr_list, proc_stream]
             |  aggregate[rewrite(xpr, [count,min,max,avg,sum], x),
                   x:expr_list, proc_stream]
             |  noop[is_attr_list(xpr), having]
;
having      =  having[" _y HAVING _x ", x:predicate, y:group]
             |  group
;
group       =  groupby[" _y GROUP BY _x ", x:attr_list, y:f_ret]
             |  f_ret
;
f_ret       =  retrieve[" FROM _r _w ", rel_list, xpr, w:where,NULL]
                {  r:relation_list;
                   mergerelation(r,rel_list);
                }
;
where       =  noop[" WHERE _x ", x:predicate]
             |  epsilon
;
```

(b) model

Table 4.1: SQL SELECT Statement

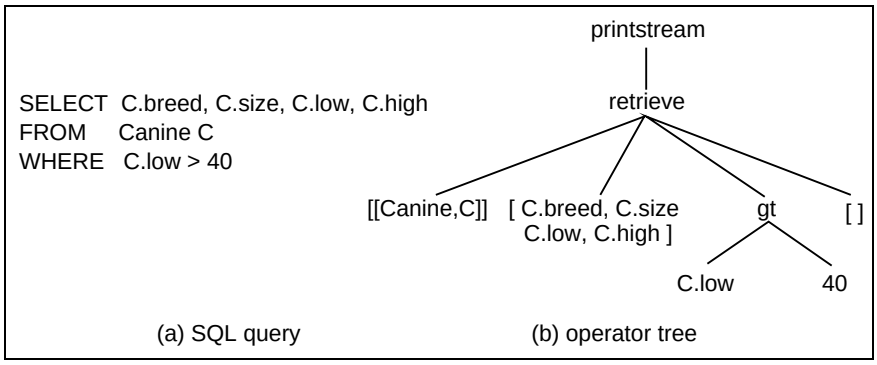

Figure 4.1: Operator Tree: SQL SELECT Statement

backplane function if the DISTINCT keyword appears, and also unions the *compattr* subcatalog, so that if the DISTINCT keyword does *not* appear, the *unique()* function is *not* composed into the operator tree and statement processing continues with the *compattr* subcatalog.

A simple SELECT statement and the operator tree which our model maps it to are shown in Figure 4.1. The root of the operator tree, the *printstream()* backplane function, is composed in the *SQL* subcatalog. Its parameter is the operator tree created beginning with the *display* subcatalog.

The SELECT keyword is not recognized until the *select* subcatalog, and it triggers no additional function composition at that point. The next clauses which may appear are the DISTINCT and ORDER BY clauses; as neither is present, parsing passes through their subcatalogs without composing additional functions.

Finally, the *compattr* and *proc_stream* subcatalogs together recognize the <*select list*>. However, since there is neither arithmetic nor aggregation, no functions are composed from these subcatalogs.

As neither of the HAVING or GROUP BY clauses are present, parsing passes through the *having* and *group* subcatalogs to the *f_ret* subcatalog, where the FROM clause is detected and the *retrieve()* backplane function is composed into the operator tree.

Last of all, the WHERE clause is recognized in the *where* subcatalog. No computation is needed to evaluate the WHERE clause; instead, its predicate is a parameter to the *retrieve()* backplane function.

Subqueries. Syntactically, subqueries are merely parenthesized SELECT statements, so we can reuse the SELECT statement specification in modeling subqueries (see Table 4.2). There is one semantic difference: the subquery's result is not printed. The differences are reflected in the *subquery* subcatalog, which consumes

```
(
    SELECT
    [ DISTINCT ]
        <select list>
        ...
)
```

(a) syntax

```
subquery    =  noop[" ( _x ) ", x:select]
;
select      =  { newCNTXT(X); }
               noop[" SELECT _x ", x:order]
;
```

(b) model

Table 4.2: SQL Subqueries

```
    <derived relation>
UNION
    <select-from-where>

    <derived relation>
UNION
    ( <derived relation> )

    <select-from-where>
INTERSECT
    <select-from-where>

    <select-from-where>
EXCEPT
    <select-from-where>
```

(a) syntax

```
SQL         =    ...
             |   { newCNTXT(X); }
                 printstream[" _x ", x:display]
;
display     =    union[" _x UNION _y ", x:display, y:display2]
             |   union[" _x UNION _y ", x:display, y:select]
             |   intersect[" _x INTERSECT _y ", x:select, y:select]
             |   difference[" _x EXCEPT _y ", x:select, y:select]
             |   display2
             |   select
;
display2    =    noop[" ( _x ) ", x:display]
;
```

(b) model

Table 4.3: SQL UNION, INTERSECT, and EXCEPT Statements

the parentheses of a parenthesized SELECT statement.

A SELECT statement which utilizes a subquery is shown in Figure 4.2, along with an operator tree which evaluates it. Parsing the subquery is almost identical to parsing a SELECT statement—the difference is that parsing the subquery begins in the *subquery* subcatalog instead of in the *SQL* subcatalog.

The UNION, INTERSECT, and EXCEPT Statements. These statements are considered together because they have similar structure and because they all perform set operations on derived relations. UNION computes the union of two derived relations; INTERSECT computes the relation consisting of tuples which appear in both of its parameters; EXCEPT computes set difference. The syntax and models of these statements are listed in Table 4.3a and Table 4.3b respectively.

Figure 4.3 shows a sample UNION statement and the operator tree it was

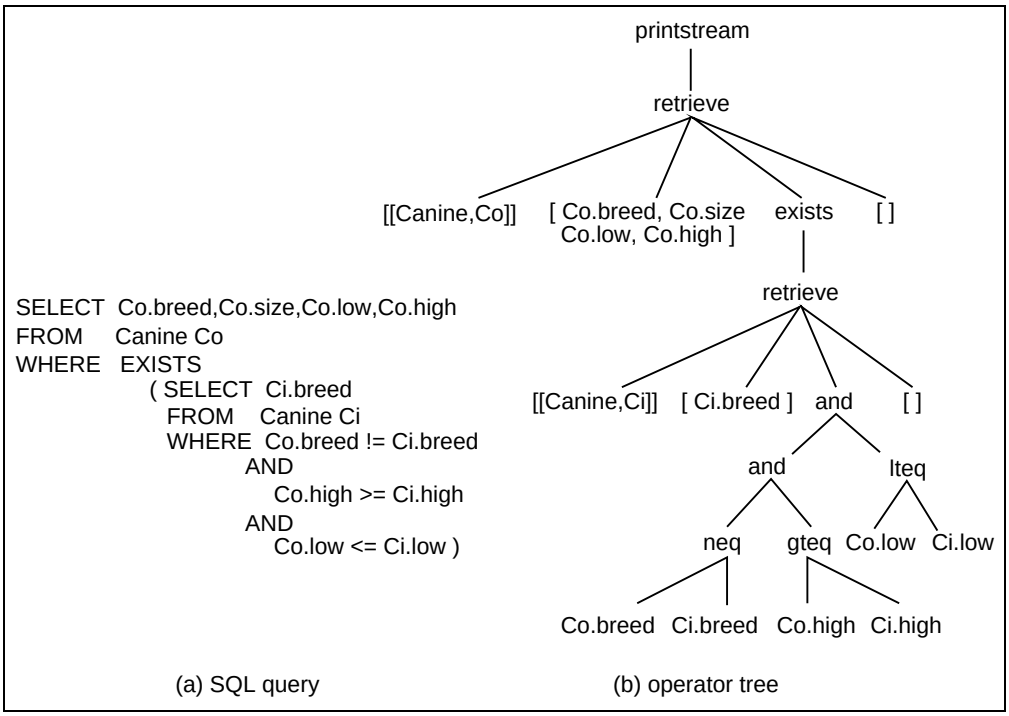

Figure 4.2: Operator Tree: SQL SELECT Statement with Subquery

mapped to. At the root of the operator tree is the *printstream()* backplane function, composed in the *SQL* subcatalog. In the *display* subcatalog, detection of the UNION keyword causes composition of the *union* backplane function. Operator trees for the two SELECT statements are generated just as for any other SELECT statement, and this holds for the parameters of the INTERSECT and EXCEPT statements as well.

In general, the operands of the *union()* backplane function can be instantiated from any of the SELECT, UNION, INTERSECT, and EXCEPT statements; therefore, we can specify these parameters with the *display* subcatalog. But the second parameter of the UNION statement *must* be parenthesized if it is not a *select*[1]. To accommodate this limitation, we defined the *display2* subcatalog to recognize a parenthesized *display*, and included two forms of the UNION signature in the *display* subcatalog, one to specify the second parameter as a *display2* and the other to specify the second parameter as a *select*.

The INSERT Statement. The SQL INSERT statement has two forms (see Table 4.4a for syntax) which differ in the source of the stream of tuples to be inserted.

[1]This is required in the 1987 ISO standard.

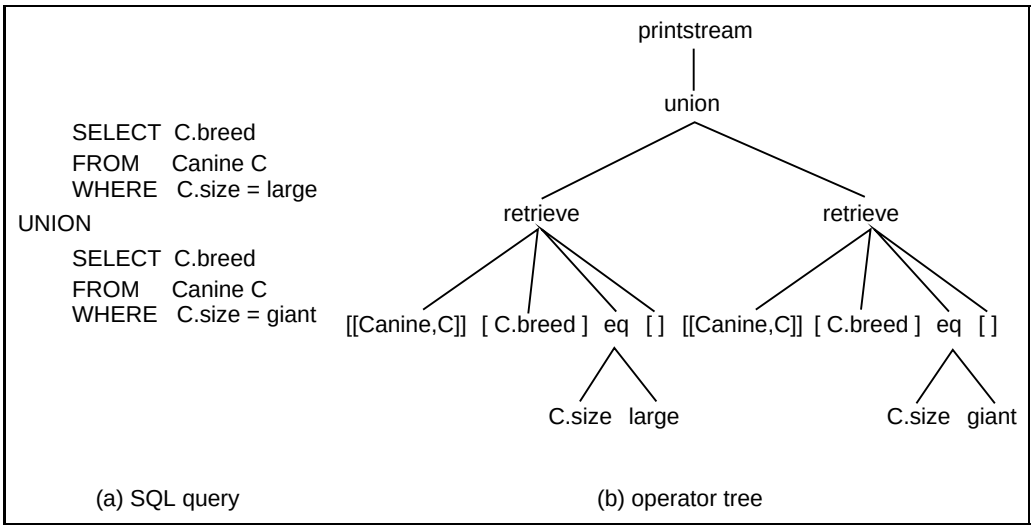

Figure 4.3: Operator Tree: SQL UNION Statement

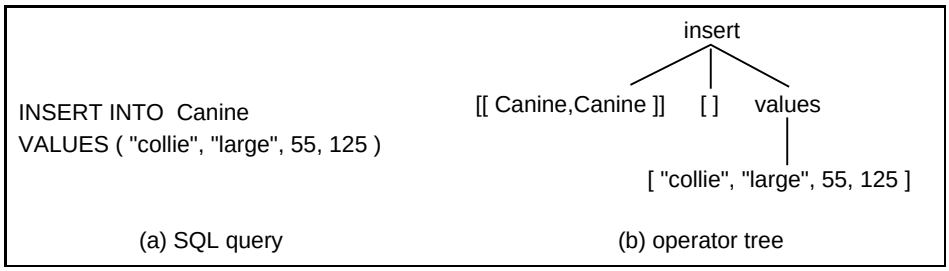

Figure 4.4: Operator Tree: SQL INSERT Statement

The first alternative is a stream of length one, consisting of a single tuple specified as a list of literal values; the second is a stream of derived tuples. Table 4.4b shows a Rosetta model for the INSERT statement.

In Figure 4.4, we show a sample INSERT statement and the operator tree to which it is mapped by the generated SQL compiler. The input stream for the *insert()* backplane function can be either literal or derived. (In the figure, it is a literal stream. The *values()* backplane function takes as input a list of values and converts it into a tuple, which is returned as a stream of length one.) Thus, the *insrtstrm* subcatalog recognizes two stream sources: the first and third signatures recognize the input stream as a literal tuple while the second and fourth signatures handle the input stream from a derived relation. The first and second, however, recognize the attribute name list (which is optional) while the other two come into play when the attribute name list is not present.

```
INSERT INTO
    [( <attribute list> )]
    <relation>
VALUES
    <literal tuple>

INSERT INTO
    [( <attribute list> )]
    <relation>
    <derived relation>
```

(a) syntax

```
BEGIN CONTEXT X
   rel_list : relation_list;
   ...
END CONTEXT X

SQL        = { newCNTXT(X); }
             insert[" INSERT INTO _x _y ",
                x:unit_rel_list, NULL, y:insrtstrm]
             { mergerelation(x,rel_list); }
           | ...
;
insrtstrm  = rename[" ( _x ) _y ", x:strng_list, y:lit_tuple]
           | rename[" ( _x ) _y ", x:strng_list, y:select]
           | lit_tuple
           | select
;
lit_tuple  = values[" VALUES _x ", x:valuelist]
;
valuelist  = list["(", ")", ",", literal, NE]
;
```

(b) model

Table 4.4: SQL INSERT Statement

The DELETE Statement. The syntax of the SQL DELETE statement (see Table 4.5a) is quite simple. Exactly one relation is specified in the FROM clause, modeled by the *d_ret* subcatalog, and the WHERE clause is optional (the model is in Table 4.5b). Tuples which satisfy the <*predicate*> of the WHERE clause (all tuples satisfy it vacuously if the WHERE clause is missing) are deleted from the named relation.

Figure 4.5 shows an SQL DELETE statement and the operator tree produced by the generated SQL compiler.

The UPDATE Statement. The SQL UPDATE statement has the most complex syntax of all the SQL modification statements (see Table 4.6a). Tuples of the named relation which satisfy the predicate of the (optional) WHERE clause are updated according to the expressions of the assignment list specified in the SET clause. Our model of the UPDATE statement is shown in Table 4.6b.

In Figure 4.6, we show a sample UPDATE statement and the operator tree to which the generated SQL compiler maps it. The *update()* backplane function is composed in the *SQL* subcatalog. The SET clause is modeled in the *u_expr* subcatalog but the assignment list is actually recognized in the *assgn_list* subcatalog. In the *u_ret* subcatalog, the *retrieve()* backplane function is composed into the operator

DELETE FROM
    <relation>
[WHERE
    <predicate>]

(a) syntax

```
BEGIN CONTEXT X
   rel_list : relation_list;
   proj_list : attr_list;
   ...
END CONTEXT X

SQL        =  ...
           |  { newCNTXT(X); }
              delete[" DELETE _x ", rel_list, x:d_ret]
;
d_ret      =  retrieve[" FROM _x _w ",
                 x:unit_rel_list,proj_list,w:where,NULL]
              { mergerelation(x,rel_list); }
;
where      =  noop[" WHERE _x ", x:predicate]
           |  epsilon
;
```

(b) model

Table 4.5: SQL DELETE Statement

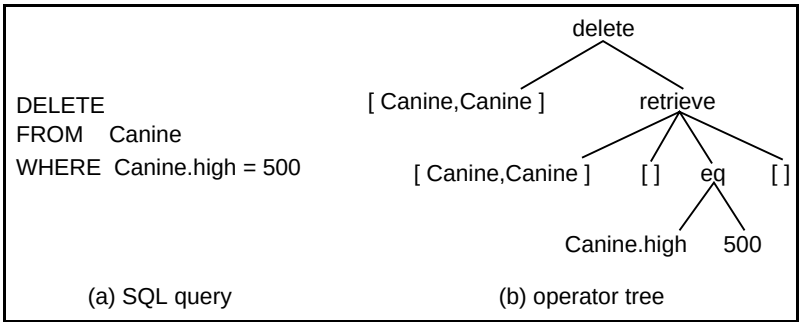

Figure 4.5: Operator Tree: SQL DELETE Statement

```
UPDATE
     <relation>
SET
     <assignment list>
[ WHERE <predicate> ]
```

```
BEGIN CONTEXT X
   rel_list : relation_list;
   proj_list : attr_list;
   xpr : expr_list;
   ...
END CONTEXT X

SQL          =  ...
             |  { newCNTXT(X); }
                update[" UPDATE _x _a _u ",
                   x:unit_rel_list,a:u_expr,u:u_ret]
                { mergerelation(x,rel_list); }
;
u_expr       =  noop[" SET _y ", y:assgn_list]
;
u_ret        =  retrieve[" _w ", rel_list, NULL, w:where, NULL]
;
where        =  noop[" WHERE _x ", x:predicate]
             |  epsilon
;
assgn_list   =  list[ , , ",", assgn, NE]
;
assgn        =  assign[" _x = _y ", x:attribute, y:expr]
                {  z:attr_list;
                   extattr(y,z);
                   mergeattr(z, proj_list);
                   extattr(x,z);
                   mergeattr(z, proj_list);
                }
;
```

(a) syntax (b) model

Table 4.6: SQL UPDATE Statement

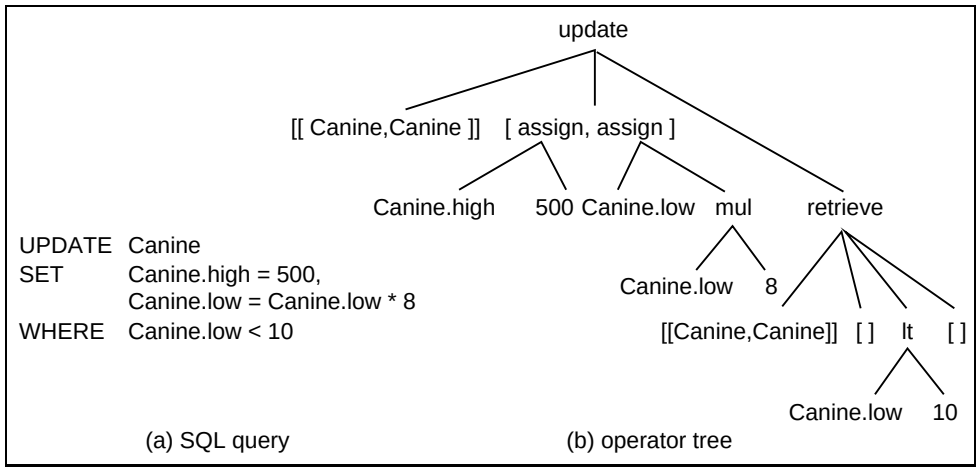

Figure 4.6: Operator Tree: SQL UPDATE Statement

tree. Finally, in the *where* subcatalog, the WHERE clause is recognized.

4.2.1.2 SQL Run-Time Experiences

Enough of the backplane functions were implemented (in Prolog) to evaluate SQL queries. We evaluated approximately forty queries, covering all data manipulation statements, including data retrieval (with aggregations and nested subqueries) and modification; all executed correctly. The queries were evaluated on relations ranging in size from ten tuples to about five hundred tuples.

4.2.1.3 Conclusions for SQL

As the SQL model was developed concurrently with the Rosetta model, we do not have the same development time numbers for this language that we will later show for the other languages. SQL was used as a test case during the development of the Rosetta specification language and the backplane. Each time the specification language or the backplane changed, the change was tried out on the evolving SQL specification. In addition, our own inexperience with this approach to specifying languages made modeling SQL slow and inefficient. For example, we did not initially fully recognize the possibilities of contexts and post-actions for information sharing. Thus, development time numbers for SQL are tainted by its relationship with development of the Rosetta model.

Nevertheless, we can say a few things about enhanced programmer productivity. Our SQL specification required fewer than 240 lines. This is sufficient to specify the statements of the data manipulation language: the data retrieval state-

ment SELECT; the set operations UNION, INTERSECT, and EXCEPT; and the data modification statements DELETE, INSERT, and UPDATE. The specification includes aggregation and subqueries but does not include all base data types.

We chose to focus on the functionality of the language rather than on adding data types *ad infinitum*; the missing data types have little impact on the functionality of our model. Varying length character strings are included, but fixed-length character strings are not. For numeric types, we modeled both real numbers and integers. However, the ISO SQL standard has three different versions of real numbers, whereas in our model precision is not specifiable. Similarly, our model has only one integer representation, whereas the SQL standard supports three. Finally, we do not support BIT strings at all. In addition to these data types, one boolean comparison operator—the LIKE string comparison operator—was not included in the specification. None of these omissions detract significantly from the expressiveness of the language; all could easily be added.

4.2.2 SQL/NF

The first extension of SQL we consider is SQL/NF, a language developed at The University of Texas at Austin by Roth, et. al. [RKS89, RKB88]. SQL/NF was designed expressly to support applications beyond traditional data processing. This language is of interest today for two reasons: because it addresses unresolved shortcomings of SQL; and because its non-traditional data model marks a significant departure from conventional relational data languages.

Overall, quite a few extensions were made to SQL. Some address shortcomings of SQL identified by Date [Dat84]. For example, Date suggested that a cleaner language design would allow a query expression to appear wherever a relation name was allowed to appear in the original language; this followed quite naturally from generalizing the data model. However, the most fundamental changes were made to support the generalized data model.

4.2.2.1 SQL/NF Data Model

SQL/NF supports the Non First Normal Form ($\neg$1NF) relational data model. This data model supersets the 1NF data model—all the power and expressiveness of the original 1NF data model remain but the $\neg$1NF data model adds nested relations: instead of restricting attribute values to scalar domains, the $\neg$1NF data model permits attributes to be relation-valued.

Consider the relations, illustrated in Figure 4.7, of the employee database popular in database literature. In the 1NF data model, two relations are needed to

Employees

name	empno	dept
Millie	462	DB
Don	283	DB

Children

empno	name	age
462	Michael	5
462	Peter	4
462	David	3
462	Patrick	2
462	Veronica	1
283	Alex	2
283	Hanna	0

(a) 1NF relations

$\Longleftrightarrow$

$\text{Employees}_{\neg 1NF}$

name	empno	dept	$\text{Children}_{\neg 1NF}$	
			name	age
Millie	462	DB	Michael	5
			Peter	4
			David	3
			Patrick	2
			Veronica	1
Don	283	DB	Alex	2
			Hanna	0

(b) ¬1NF relation

Figure 4.7: ¬1NF Data Model — Illustration

model employees and their children. But in the ¬1NF data model, the names and ages of an employee's children can be modeled as a relation attribute nested within the employee's tuple regardless of the number of children. The two 1NF relations,

```
Employee(name,empno,dept)
Children(empno,name,age),
```

can be merged into one nested relation which models the same information:

$\texttt{Employee}_{\neg 1NF}\texttt{(name,empno,dept,Children}_{\neg 1NF}\texttt{(name,age))}.$

Extending the data model to ¬1NF means that the language must be extended to support relation-valued attributes. In particular, it must include relation operations on relation-valued attributes: wherever an attribute name can appear in SQL, a relation name can appear in SQL/NF. The SQL data language was modified extensively to reflect the generalized data model. There are, of course, other issues such as storage and access, but we consider only language design issues.

4.2.2.2 Modeling SQL/NF Statements

Our model of SQL/NF includes all the data manipulation statements (for the specification, see Appendix C), including statements for data retrieval, set operations, and data modification. We discuss only high-level language elements in this section.

The SELECT Statement. Significant changes were made to the SELECT statement. The SQL/NF SELECT statement incorporates new syntax for retrieving tuples and for the outer join operation (see Table 4.7a). Furthermore, three clauses—DISTINCT, GROUP BY, and HAVING—were eliminated[2], simple changes which required only removing the subcatalogs for those clauses from the model, replacing references to them with a reference to the signature composing the next function in the composition. Other changes were not as easy (see the model in Table 4.7b).

The first of the changes we consider is a fundamental change which is propagated throughout the language. SQL/NF allows the shorthand

```
<relation>
```

to be used in place of the usual

```
SELECT * FROM <relation>
```

construct. The brief form is interpreted as retrieval of a single relation without computation or projection. Thus, the query

```
SELECT * FROM Canine WHERE Canine.low > 40
```

may be expressed more simply as

```
Canine WHERE Canine.low > 40.
```

As the SQL/NF WHERE clause is optional, the minimal SQL/NF statement consists of just a relation name. The query in Figure 4.8 utilizes this new construct to print all tuples of the Canine relation satisfying the predicate "Canine.low > 40."

In SQL, the *retrieve()* backplane function was composed in the *f_ret* subcatalog, where the FROM clause is recognized; in SQL/NF, the *retrieve()* backplane function can be composed in either the *f_ret* or *select* subcatalogs. In our sample query, the *retrieve()* backplane function is composed via the second signature of the *select* subcatalog. The extended *select* subcatalog still recognizes the SELECT-FROM-WHERE-expression (SFW-expression), just as in SQL. The last clause of our example, the `WHERE` clause, is recognized in the *where* subcatalog.

Another extension to the SQL/NF SELECT statement was the addition of the outer join operation. Outer join includes in the result all tuples from specified relations whether they satisfy the join condition or not. These tuples are padded with NULLs in the remaining attributes and added to the result.

[2]In contrast to SQL, grouping can be built in to SQL/NF relations by utilizing relation-valued attributes.

```
SELECT
    <select list>
FROM
    <relation list>
[ WHERE
    <predicate> ]
[ PRESERVE
    <relation list> ]

<relation>
[ WHERE
    <predicate> ]
```

(a) syntax

```
BEGIN CONTEXT X
    outer_list : relation_list;
    rel_list : relation_list;
    slst: select_list;
    xpr : sel_list2;
END CONTEXT X

sql_nf      = printstream[" _x ", x:display]
;
display     = select
;
select      = { newCNTXT(X); }
              noop[" SELECT _slst _p ", p:proc_stream]
            | retrieve[" _r _w ",rel_list,xpr,w:where,outer_list]
              {  r:unit_rel_list;
                 newCNTXT(X);
                 mergerelation(r,rel_list);
                 get_all_attr(rel_list,xpr);
              }
select_list = noop[" ALL ", xpr]
              { get_all_attr(rel_list,xpr); }
            | noop[" ALL BUT _a ",xpr]
              {  a:attr_list; all_but_attr(rel_list,xpr,a); }
            | noop[" _xpr ", xpr ]
;
proc_stream = compute[rewrite(xpr,[add,assign,div,mul,sub,distinct,
                 subsume],x),x:sel_list2, proc_stream]
            | aggregate[rewrite(xpr,[avg,count,max,min,sum],x),
                 x:sel_list2, proc_stream]
            | noop[is_attr_list(xpr), f_ret]
;
f_ret       = retrieve[" FROM _r _w ",rel_list,xpr,w:where,outer_list]
              {  r:relation_list; mergerelation(r,rel_list); }
;
where       = noop[" WHERE _x _p ", x:predicate]
              {  p:preserve; skip(); }
            | epsilon
;
preserve    = noop[" PRESERVE _outer_list ", outer_list]
            | epsilon
;
sel_list2   = list[ , , ",", select_item, NE ]
;
select_item = expr
            | noop[" _x AS _a ", a:attr_name]
              {  x:expr; regATTR(x,a); }
            | noop[" _r ALL ", r:dot_ref_list]
              {  mergerelation(r,rel_list); get_all_attr(r,xpr); }
;
```

(b) model

Table 4.7: SQL/NF SELECT Statement

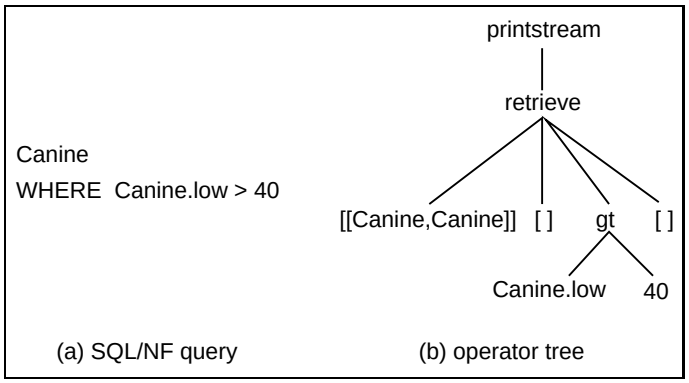

Figure 4.8: Operator Tree: SQL/NF SELECT Statement

The outer join operation was added in SQL/NF by augmenting the SELECT statement with a new clause, the PRESERVE clause, which specifies the list of relations to be outer joined. The PRESERVE clause, which follows the WHERE clause, is modeled in the *preserve* subcatalog. Like the WHERE clause, the PRESERVE clause specifies an input parameter for the *retrieve()* backplane function and composes no additional backplane functions into the operator tree.

The original *retrieve()* backplane function was not designed to implement outer join. But tuple preservation must be done *during* and not *after* relation retrieval. To handle outer join, we extended the *retrieve()* backplane function with a new parameter for the list of preserved relations. When no relations are preserved, the generalized *retrieve()* backplane function operates as originally defined. Adding outer join required modifying no backplane function definitions besides *retrieve()*[3].

Subqueries. A language design principle, the principle of *orthogonality*, states that an expression evaluating to an element of some type should be allowed wherever an element of that type is allowed. In accordance with this principle, SQL/NF generalizes relation references and allows an SFW-expression to appear wherever a relation name can occur in SQL. Our model of subqueries is shown in Table 4.8.

For example, the relation name list in the FROM clause was generalized to include SFW-expressions with an optional correlation name. The fourth signature in the *relationref* subcatalog models this.

A sample SQL/NF SELECT statement which includes a subquery is shown in Figure 4.9. Operator trees for both the SELECT statement and the subquery are generated from the *display* subcatalog.

[3]Backplane function *implementations* are, of course, a different matter. They would need to be extended to handle ¬1NF tuples.

```
(   SELECT
      <select list>
    FROM
      <relation list>
    [ WHERE
      <predicate> ]
    ...
)
|
(   <relation>
    [ WHERE
      <predicate> ]
)
|
<relation>
```

(a) syntax

```
BEGIN CONTEXT X
      rel_list : relation_list;
      xpr : sel_list2;
      ...
END CONTEXT X

nested_query  =  nqe1
              |  nqe2
;
nqe1          =  noop[" ( _d ) ", d:display]
;
nqe2          =  retrieve[" _r ", rel_list, xpr, NULL, NULL]
                 {  r:unit_rel_list;
                    newCNTXT(X);
                    mergerelation(r,rel_list);
                    get_all_attr(r,xpr);
                 }
;
relationref   =  is_relation[" _r ",r:relation, NULL]
              |  is_relation[" _n AS _a ",n:relation,a:relation]
              |  retrelation[" _n ", n:nqe1,NULL]
              |  retrelation[" _n AS _a ",n:nqe1,a:relation]
;
```

(b) model

Table 4.8: SQL/NF Subqueries

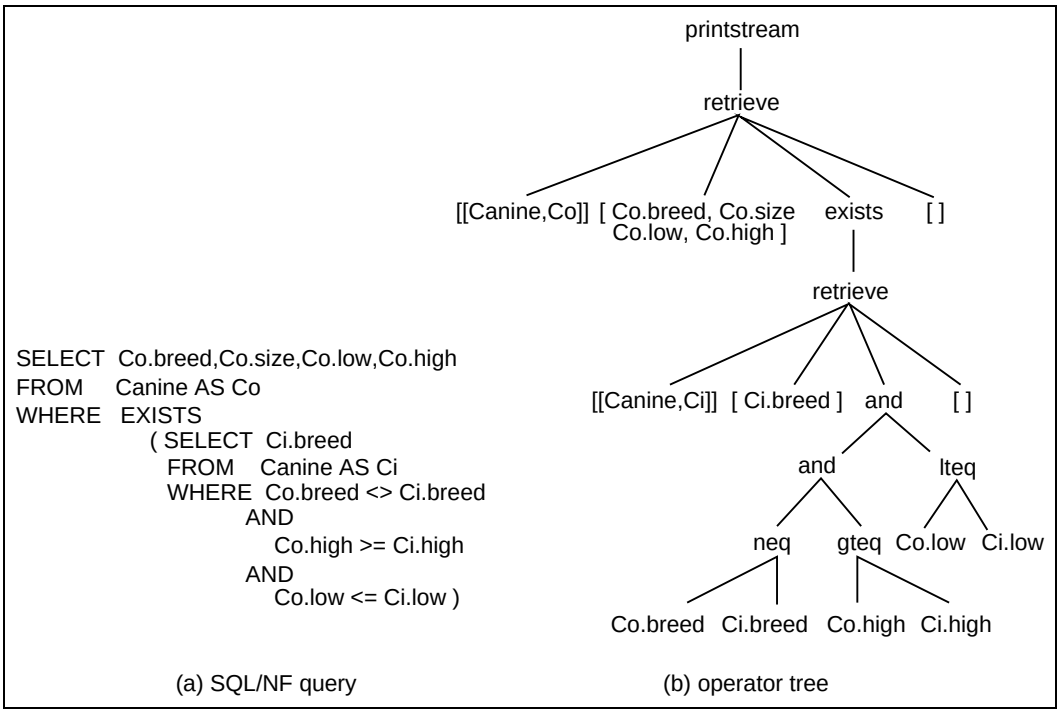

Figure 4.9: Operator Tree: SQL/NF SELECT Statement with Subquery

As another example of new subqueries, attributes in the SELECT list may be *relation-valued*; therefore relation *expressions* may appear in the SELECT list. In this way, relation-valued attributes can receive additional processing after selection, including applying a predicate to a relation-valued attribute to eliminate unwanted tuples or aggregating a relation-valued attribute.

The UNION, INTERSECT, and DIFFERENCE Statements. The set operation statements were also generalized for SQL/NF. The INTERSECT and DIFFERENCE statements may be nested to arbitrary levels, and their inputs can come from any statement computing a derived relation, including the UNION, INTERSECT, and DIFFERENCE statements. The subcatalogs modeling the SQL/NF UNION, INTERSECT, and DIFFERENCE statements are shown in Table 4.9.

By orthogonality, the new <relation> retrieve construct may appear wherever SELECT * FROM <relation> can. For instance, the UNION statement

SELECT * FROM R_0 UNION SELECT * FROM R_1

can also be expressed as

R_0 UNION R_1.

```
                    BEGIN CONTEXT X
                       outer_list : relation_list;
                       rel_list : relation_list;
                       slst: select_list;
                       xpr : sel_list2;
                    END CONTEXT X

    <derived relation>  sql_nf    =  printstream[" _x ", x:display]
UNION                               |     ...
    <derived relation>  ;
                        display   =     ...
    <derived relation>              |  union[" _x UNION _y ", x:display, y:display]
INTERSECT                           |  intersect[" _x INTERSECT _y ", x:display, y:display]
    <derived relation>              |  difference[" _x DIFFERENCE _y ", x:display, y:display]
                                    |  select
    <derived relation>  ;
DIFFERENCE              select    =  retrieve[" _r _w ",rel_list,xpr,w:where,outer_list]
    <derived relation>                 {  r:unit_rel_list;
                                          newCNTXT(X);
                                          mergerelation(r,rel_list);
                                          get_all_attr(rel_list,xpr);
                                       }
                                    |  { newCNTXT(X); }
                                          noop[" SELECT _slst _p ", p:proc_stream]
                        ;
```

(a) syntax (b) model

Table 4.9: SQL/NF UNION, INTERSECT, and DIFFERENCE Statements

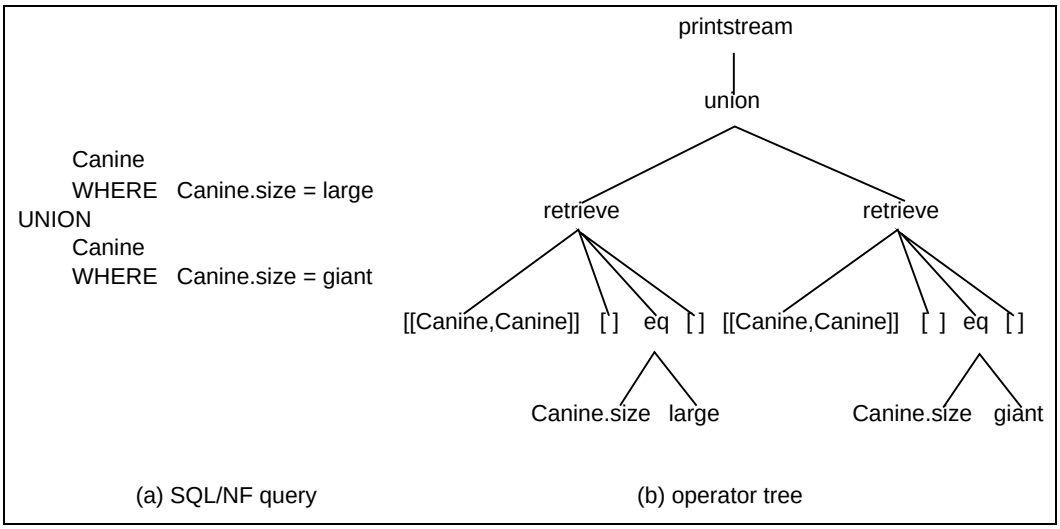

Figure 4.10: Operator Tree: SQL/NF UNION Statement

As the *select* subcatalog is an alternative in the *display* subcatalog, the signature added for the new construct to the *select* subcatalog allows the new construct to appear wherever the SFW-expression can. The change in the specification of the SELECT statement is independent of the specification of the set operation statements; therefore, the *select* and *display* subcatalogs work together to map the statement $\texttt{R}_0$ `UNION` $\texttt{R}_1$ to an operator tree (see Figure 4.10).

The MODIFY Statement. The INSERT and DELETE statements were limited to keyword changes but the MODIFY statement[4] required more complex changes (see Table 4.10).

Updating a value in the 1NF data model is simple: one merely specifies a new scalar value (or expression) to replace the former value for each attribute to be modified. But in the ¬1NF data model, attributes may be *relation-valued*, with the result that modifying an attribute value may require *relation* modification. Like other relations, relation-valued attributes may be modified in three ways: by inserting new tuples or by deleting or modifying existing tuples.

In SQL/NF, as in SQL, expressions specifying attribute updates are listed in the assignment list. However, in SQL/NF, an update expression can be not only an arithmetic expression or a literal value but may also be any of the relation modification statements INSERT, DELETE, and MODIFY. The assignment list is modeled by the *assgn_list* subcatalog.

[4]The keyword `UPDATE` was changed to `MODIFY`.

```
MODIFY
    <relation>
SET
    <assignment list>
[ WHERE
    <predicate> ]

ERASE
    <relation>
[ WHERE
    <predicate> ]

STORE
    <relation>
[ <attribute list> ]
VALUES
    <value list>

STORE
    <relation>
[ <attribute list> ]
<derived relation>
```

(a) syntax

```
BEGIN CONTEXT X
    rel_list : relation_list;
    urel_list : unit_rel_list;
    assgnmnts : assgn_list;
    proj_list : attr_list;
END CONTEXT X

update_stmt  = { newCNTXT(X); }
               update[" MODIFY _x ", urel_list, assgnmnts, x:u_ret]
             | { newCNTXT(X); }
               delete[" ERASE _x ", rel_list, x:d_ret]
             | insert["STORE _x _y",x:unit_rel_list,NULL,y:insrtstrm]
;
u_ret        = compute[" _urel_list SET _y _z ", y:assgn_list,
                  retrieve(rel_list, a, z:where,NULL)]
               {  a:attr_list;
                  mergerelation(urel_list,rel_list);
                  get_all_attr(urel_list,a);
               }
;
assgn_list   = list[, , ",", assgn, NE]
;
assgn        = assign[" _x = _y ", x:attribute, y:uvalue]
;
uvalue       = noop[" _e ", e:expr]
               {  z:attr_list;
                  extattr(e,z);
                  mergeattr(z, proj_list);
               }
             | noop[" ( _u ) ",u:update_stmt]
;
d_ret        = retrieve["_x _w",x:unit_rel_list,proj_list,w:where,NULL]
               { mergerelation(x,rel_list); }
;
insrtstrm    = rename[" ( _x ) VALUES _y ",
                  x:attr_list, values(y:tuple_seq)]
             | rename[" ( _x ) _y ", x:attr_list, y:select]
             | values[" VALUES _t ",t:tuple_seq]
             | select
;
where        = noop[" WHERE _x _p ", x:predicate]
               {  p:preserve; skip(); }
             | epsilon
;
tuple_seq    = list[ , , ,tuple_literal,NE]
;
tuple_literal = list["{","}",",",literal,NE]
;
tuple_expr   = list["<",">",",",expr,NE]
;
```

(b) model

Table 4.10: SQL/NF MODIFY, ERASE, and STORE Statements

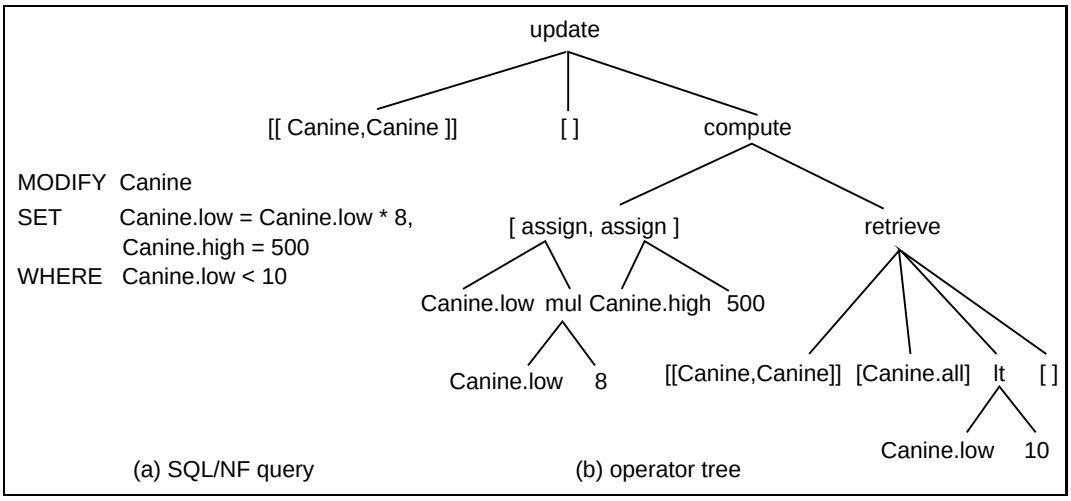

Figure 4.11: Operator Tree: SQL/NF MODIFY Statement

An example of an SQL/NF MODIFY statement is given in Figure 4.11.

Function Statements. To promote clarity of the language, certain statements were recast into a functional form so that the structure of the statement would reflect its semantics. The SQL/NF function statements are modeled in Table 4.11.

Whereas an SQL aggregation function is invoked in-line, either as an expression in the select list or as part of an arithmetic expression, an SQL/NF aggregate may be either an independent statement or a subquery.

Aggregate functions were generalized to apply to relations instead of to attributes. An aggregate function is applied to each attribute of the tuples in its input relation and produces a relation with one tuple as its output. Aggregates may appear as subqueries wherever a scalar may appear.

There are function statements besides aggregation. The DISTINCT clause which was deleted from the SQL/NF SELECT statement resurfaces as a functional statement. In addition, a new SUBSUME statement is similar to DISTINCT but allows for NULL values. Given two tuples, if all non-null attributes of the first tuple agree with their counterparts in the second tuple, the second tuple is said to *subsume* the first. The SUBSUME statement eliminates from a stream of tuples each tuple which is subsumed by some other tuple in the stream.

NEST/UNNEST: New Statements To Support ¬1NF Relations. Relations in SQL/NF may be arbitrarily deeply nested; however, occasionally a less nested or more nested version is needed. Two new statements, NEST and UNNEST

<function> (<derived relation>)

where <function> is one of
{ MAX, MIN, AVG,
SUM, COUNT,
DISTINCT, SUBSUME }

(a) syntax

```
sql_nf          =  printstream[" _x ", x:display]
                |     ...
;
display         =     ...
                |  function_stmt
;
function_stmt   =  max[" MAX ( _x ) ", NULL, x:display]
                |  min[" MIN ( _x ) ", NULL, x:display]
                |  avg[" AVG ( _x ) ", NULL, x:display]
                |  sum[" SUM ( _x ) ", NULL, x:display]
                |  count[" COUNT ( _x ) ", NULL, x:display]
                |  unique[" DISTINCT ( _x ) ", x:display]
                |  subsume[" SUBSUME ( _x ) ", x:display]
;
```

(b) model

Table 4.11: SQL/NF Function Statements

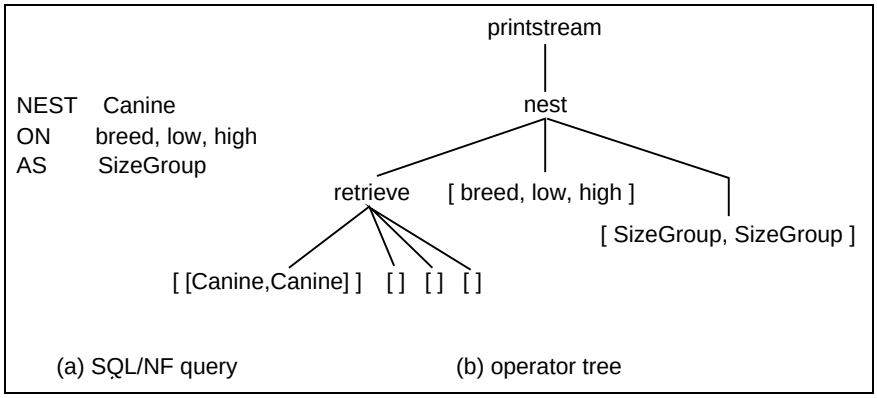

Figure 4.12: Operator Tree: SQL/NF NEST Statement

(see Table 4.12), were added to SQL/NF for dynamically adjusting the level of nesting in relations. The NEST statement nests a relation more deeply while the UNNEST statement expands nested relations.

To implement these statements, two new functions, *nest()* and *unnest()*, were added to the backplane. Figure 4.12 shows a sample SQL/NF NEST statement and the operator tree it is mapped to.

The NEST and UNNEST statements produce the 1NF and ¬1NF relations shown in Table 4.13. The SQL/NF statement

```
NEST    Canine
ON      breed,low,high
AS      SizeGroup
```

```
                                sql_nf          =  printstream[" _x ", x:display]
                                                |     ...
                                ;
                                display         =     ...
NEST <table>                                    |  operator_stmt
ON <attribute list>             ;
[ AS <nested relation name> ]   operator_stmt   =  nest[" NEST _x ON _c AS _a ",
                                                      x:nested_query, c:attr_list, a:relation]
UNNEST <table>                                  |  nest[" NEST _x ON _c ",
ON <attribute list>                                   x:nested_query, c:attr_list, NULL]
                                                |  unnest[" UNNEST _x ON _c ",
                                                      x:nested_query, c:attr_list]
                                                |     ...
                                ;
```

(a) syntax (b) model

Table 4.12: SQL/NF NEST and UNNEST Statements

produces the relation Canine$_{\neg 1NF}$, while the statement

```
UNNEST  Canine¬1NF
ON      breed,low,high
```

produces the relation Canine.

4.2.2.3 Conclusions for SQL/NF

In this section we consider the model size and development time for SQL/NF.

The SQL/NF specification file consists of approximately 270 lines. All of the data retrieval statements are modeled—SELECT, DIFFERENCE, INTERSECT, and UNION—as well as the data modification statements STORE (insert), ERASE (delete), and MODIFY (update). As for SQL, some base types are not modeled. Character strings are varying length only; fixed length strings are not supported. Integers and reals are present only with unspecified precision, and bit strings are not included at all. Again, this does not detract from the expressiveness of the language.

Rosetta maps the SQL/NF specification file into bison and flex files, comprising approximately 1320 and 120 lines, respectively. As the SQL/NF specification is approximately 270 lines, the expansion factor is approximately 1:6.

The SQL/NF specification file is approximately 30 lines longer than the SQL specification file. Although a difference of 30 lines sounds minor, simply comparing the lengths of the two specifications does not indicate the magnitude of the modifications because the changing content of subcatalogs is not reflected in the differential line count.

Canine

breed	size	low	high
australian shepherd	medium	40	60
basenji	small	15	25
beagle	small	10	25
dachshund	small	8	15
german shepherd	large	75	150
great dane	giant	150	210
keeshond	medium	30	50
labrador	large	60	150
newfoundland	giant	145	205
saint bernard	giant	175	220

(a) 1NF relation

$\text{Canine}_{\neg 1NF}$

size	$\text{SizeGroup}_{\neg 1NF}$		
	breed	low	high
small	basenji	15	25
	beagle	10	25
	dachshund	8	15
medium	australian shepherd	40	60
	keeshond	30	50
large	german shepherd	75	150
	labrador	60	150
giant	great dane	150	210
	newfoundland	145	205
	saint bernard	175	220

(b) ¬1NF relation

Figure 4.13: ¬1NF Data Model — Another Illustration

A better indication of the magnitude of the changes made to the SQL specification file is the quantity of specification reuse. The SQL specification file consists of approximately 115 signatures. Of these, 37 were reused unchanged in the SQL/NF specification. In addition, 22 signatures were reused with only minor modifications. As the SQL/NF specification contains 120 signatures, approximately half of the signatures came from the SQL specification with no more than minor changes. Thus, virtually half of the SQL/NF specification was done before we even began modeling.

Finally, we consider the time to make the modifications to the language. We needed approximately 16 days to understand the ¬1NF data model and to make the necessary extensions to the SQL model. Presumably, a modeler already experienced with the ¬1NF data model could have made the extensions more quickly.

Overall, SQL/NF differs quite dramatically from SQL. Because the data model changed from 1NF to ¬1NF, the philosophy of SQL/NF is fundamentally different from that of SQL. Generalizing relation references to allow relation expressions was the most significant extension made, as it added the relation as a primitive data type. Some syntactic changes were made to make the syntax more closely match the semantics of function calls such as the aggregate functions. Although fundamental changes were made to the data model, because the specification is so high-level the changes required consisted of adding new backplane functions or generalizing existing ones.

4.2.3 TSQL2

For our third data point, we selected TSQL2, a temporal data language [Sno94]. We developed our model for this language while the specification was still under discussion. As a result, the SELECT statement and the primitives of the language conform to the proposed specification which we obtained in December, 1993 while the modification statements, whose specifications were sketchy in the proposed specification, conform to the final specification, published in March, 1994.

TSQL2 was designed to serve the temporal database community as the standard temporal data language, providing a basis for future temporal database research. Therefore, TSQL2 augments SQL with temporal constructs derived from well-defined temporal database technology but does not address the areas of active research. For compatibility, the TSQL2 design effort does not correct deficiencies of SQL. Because this language is important to the temporal database community, it was defined by a committee drawn from the temporal database community and chaired by Richard Snodgrass, at the University of Arizona at Tucson, Arizona.

There were several reasons for choosing TSQL2 as one of our data points. First, TSQL2 is an important development in the domain of data languages because it is intended to be the *de facto* standard for temporal data languages. Furthermore, the language specification, published March, 1994, is very recent, showing that Rosetta can model recent data language developments. Finally, we wanted our last data point in the SQL family to extend the relational data model in a significantly different way than did SQL/NF, our first data point.

The rest of this section discusses briefly the extensions made to SQL; for the full model, see Appendix D. Our focus is on the data manipulation language; we do not model data definition aspects. We discuss first the temporal data model, including primitive temporal data types, and then the changes made to the SQL statements to support temporal constructs. Finally, we consider model size, development time, and productivity gains.

4.2.3.1 TSQL2 Data Model

One of the first tasks faced by the design committee was extending the relational data model to include temporal constructs. The resulting data model is the bitemporal conceptual data model (BCDM). This model of time is *linear, discrete,* and *bounded.* A model of time is *linear* if it assumes a total order on both past and future temporal elements.

A model of time is *discrete* if time is divided into units of time called chronons, and chronons are totally ordered and in 1:1 correspondence with the natural numbers.

The time line can be divided into chronons of arbitrarily fine granularity, e.g., days, hours, seconds, microseconds, etc. Time is assumed to be *bounded* both in the past and in the future, so there is a least chronon and a greatest chronon.

Temporal databases support timestamps, which can be applied to either tuples or attributes. A *valid-time* timestamp denotes the time in the modeled reality that the tuple actually becomes valid, whereas a *transaction-time* timestamp reflects the time when the information was added to the database. When both valid-time and transaction-time timestamps are supported, the data model is termed a bitemporal data model.

TSQL2 relations are either temporal relations or snapshot relations, where a *snapshot* relation is simply a traditional relation having no temporal attributes.

New Primitive Data Types of TSQL2. The chronons of the time line can be grouped in various ways into the primitive data types of TSQL2: spans, events, and intervals.

Spans. The span models the concept of temporal *distance*. The units of span constants are limited by the granularity of the chronons. Examples of spans include a day, a week, or a month.

To model spans, we define a new backplane function, *str2span()*, which converts a literal string to the corresponding span in an internal representation. TSQL2 spans are flanked by a pair of `%`, e.g., `%day%`. Our TSQL2 model includes four span literals: `%day%, %week%, %month%,` and `%year%`. A span literal may optionally specify a length multiplier, e.g., `%5 week%` or `%35 day%`.

TSQL2 includes Gregorian date to span conversion. A date can be either partial or complete, e.g., 9/95 vs. 9/1/95. The partial date, 9/95, is converted to the span `%1 month%`, while the complete date is converted to the span `%1 day%`.

Events. An event represents the moment of an occurrence, with no associated duration, and is modeled by a single chronon. Elements of type *event* are flanked by a pair of |'s. In the TSQL2 model of time, chronons are totally ordered and bounded. TSQL2 defines special literals that signal the value of negative infinite time, positive infinite time, and the current chronon: `beginning`, `forever`, and `present`. To convert these literal strings into internal event values, we extend the backplane with a conversion function, *str2event()*, which maps a string to the corresponding event in an internal representation.

Besides event literals, TSQL2 allows the coercion of dates, correlation names, and attribute references to events. Coercion is handled by the *event()* backplane

function, which takes specific inputs, such as a correlation name or a date, and converts them to the event internal format.

Intervals. An interval denotes a time span covering a specific portion of the time line and is modeled by a pair of events, representing the start and the end of the interval, enclosed in square brackets, `[` and `]`. An interval enclosed in square brackets includes its ending event; to exclude the ending event, the interval is terminated with a `)` instead.

Timestamps. In temporal databases, tuples are timestamped with either transaction-time or valid-time timestamps, or both. In TSQL2, a timestamp is referenced according to its kind: the valid-time timestamp of a tuple is referenced by the unqualified correlation name corresponding to its relation while the value of the transaction-time timestamp is accessed by a function call:

`transaction(`<*correlation name*>`).`

Referencing a timestamp yields a temporal value which may be either event or interval.

Operations on Temporal Data Types. Various operations are defined on temporal data types: constructors, deconstructors, arithmetic operations, and comparisons.

A constructor takes as input temporal elements and constructs from them another temporal element, e.g., the `interval` operator takes as input two events and returns an interval.

Conversely, the deconstructor functions extract information from their inputs. For example, given an interval, the built-in TSQL2 functions `begin` and `end` return its beginning and ending events, respectively.

The arithmetic operators were extended to operate on temporal values. For example, adding a span to an interval increases the length of the interval by the duration of the span.

Comparisons are also defined on temporal values: events can be compared to determine which occurred earlier or later; spans can be compared to determine which is longer or shorter; intervals can be compared to determine if they have a sub-interval in common or if one includes the other.

4.2.3.2 Modeling TSQL2 Statements

For simplicity, we assume the Gregorian calendar, with the finest granularity chronon being one day.

The SELECT Statement. The SELECT statement (modeled in Tables 4.13 and 4.14) produces either a snapshot or temporal relation. To indicate which, three new clauses were added: the SNAPSHOT, VALID, and VALIDINTERSECT clauses. The SNAPSHOT clause indicates that the relation is a snapshot relation while the other two clauses specify how to compute the valid time for the derived tuples.

The simplest case is the SNAPSHOT clause, as the derived relation is not a temporal relation and therefore has no valid time. The VALID clause specifies a temporal expression for computing the valid-time timestamp of the derived tuples. The VALIDINTERSECT clause is provided as a notational convenience: it calls for computing the valid-time timestamp as the temporal intersection of the valid-time timestamps of the underlying tuples and the specified temporal expression.

These three clauses are mutually exclusive; at most one may appear. If no clause is specified, TSQL2 determines the valid time of the derived tuples using as a default the intersection of the valid times of the underlying tuples[5]. However, if any participating relation is a SNAPSHOT relation, then the result relation is also a SNAPSHOT relation. Computing a valid-time timestamp is known as valid-time projection.

A sample TSQL2 SELECT statement and the operator tree to which it is mapped are shown in Figure 4.14. Parsing the statement begins with the *TSQL2* subcatalog, where the *printstream()* backplane function is composed into the operator tree. Parsing continues through the *select*, *order*, and *distinct* subcatalogs, composing no additional backplane functions from any of these, and reaches the *validtime* subcatalog.

[5]The result is `NULL` if the intersection is empty.

```
SELECT
[ DISTINCT ]
SNAPSHOT
     <select list>
FROM
     <relation list>
[ WHERE
     <predicate> ]
[ GROUP BY
     <attribute list> ]
[ HAVING
     <predicate> ]
[ ORDER BY
     <attribute list> ]

SELECT
[ DISTINCT ]
     <select list>
[
    VALID
     <temporal expression>
  | VALIDINTERSECT
     <temporal expression>
  | <epsilon>
]
FROM
     <relation list>
[ WHERE
     <predicate> ]
[ GROUP BY
     <attribute list> ]
[ HAVING
     <predicate> ]
[ ORDER BY
     <attribute list> ]
```

(a) syntax

```
BEGIN CONTEXT X
     rel_list : relation_list;
     xpr : expr_list;
     val : element;
END CONTEXT X

TSQL2      = { newCNTXT(X); }
             printstream[" _x ", x:display]
           |    ...
;
display    = display2
           |    ...
;
display2   = select
           |    ...
;
select     = { newCNTXT(X); }
             noop[" SELECT _x ", x:order]
;
order      = sort[" _y ORDER BY _x ", x:attr_list, y:distinct]
           | distinct
;
distinct   = unique[" DISTINCT _x ", x:validtime]
           | validtime
;
validtime  = valid[" SNAPSHOT _t ", NULL, t:compattr]
           | valid[" _xpr VALID _t ", val, NULL, t:compattr2]
           | valid[" _xpr VALIDINTERSECT _t ",
               overlapt(val,a), NULL,t:compattr2]
             { a:element; gen_expr(rel_list,overlapt,left,a); }
           | valid[" _t ", a, NULL, t:compattr]
             { a:element; gen_expr(rel_list,overlapt,left,a); }
;
compattr   = noop[" _xpr _s ", s:proc_stream]
;
compattr2  = noop[" _val _s ", s:proc_stream]
;
```

(b) model

Table 4.13: TSQL2 SELECT Statement

```
proc_stream= compute[rewrite(xpr,[add,sub,div,mul,assign,str2int,
                str2real,intervalc,span],x),x:expr_list,proc_stream]
           | aggregate[rewrite(xpr, [count,min,max,avg,sum,
                unique], x), x:expr_list, proc_stream]
           | noop[is_attr_list(xpr), having]
;
having     = having[" _y HAVING _x ", x:predicate, y:group]
           | group
;
group      = groupby[" _y GROUP BY _x ", x:attr_list, y:f_ret]
           | f_ret
;
f_ret      = retrieve[" FROM _r _w ", rel_list, xpr, w:where,NULL]
             { r:relation_list;
               mergerelation(r,rel_list);
               map_coalesce_depnds(r,w);
             }
;
where      = noop[" WHERE _x ", x:predicate]
           | epsilon
;
relation_list = list[ , , "," , relationref, NE]
;
relationref =   ...
           | retrelation[" _n AS _a ",n:coalescrel, a:relation]
;
coalescrel = coalesce[" _r _a ( _i ) ",retrieve(r:relation,NULL,NULL,NULL),
                a:clscattrlst,i:clscintrvl]
           | coalesce[" _r _a ",retrieve(r:relation,NULL,NULL,NULL),
                a:clscattrlst,NULL]
           | coalesce[" _r ( _i ) ",retrieve(r:relation,NULL,NULL,NULL),
                NULL,i:clscintrvl]
;
```

Table 4.14: TSQL2 SELECT Statement (cont.)

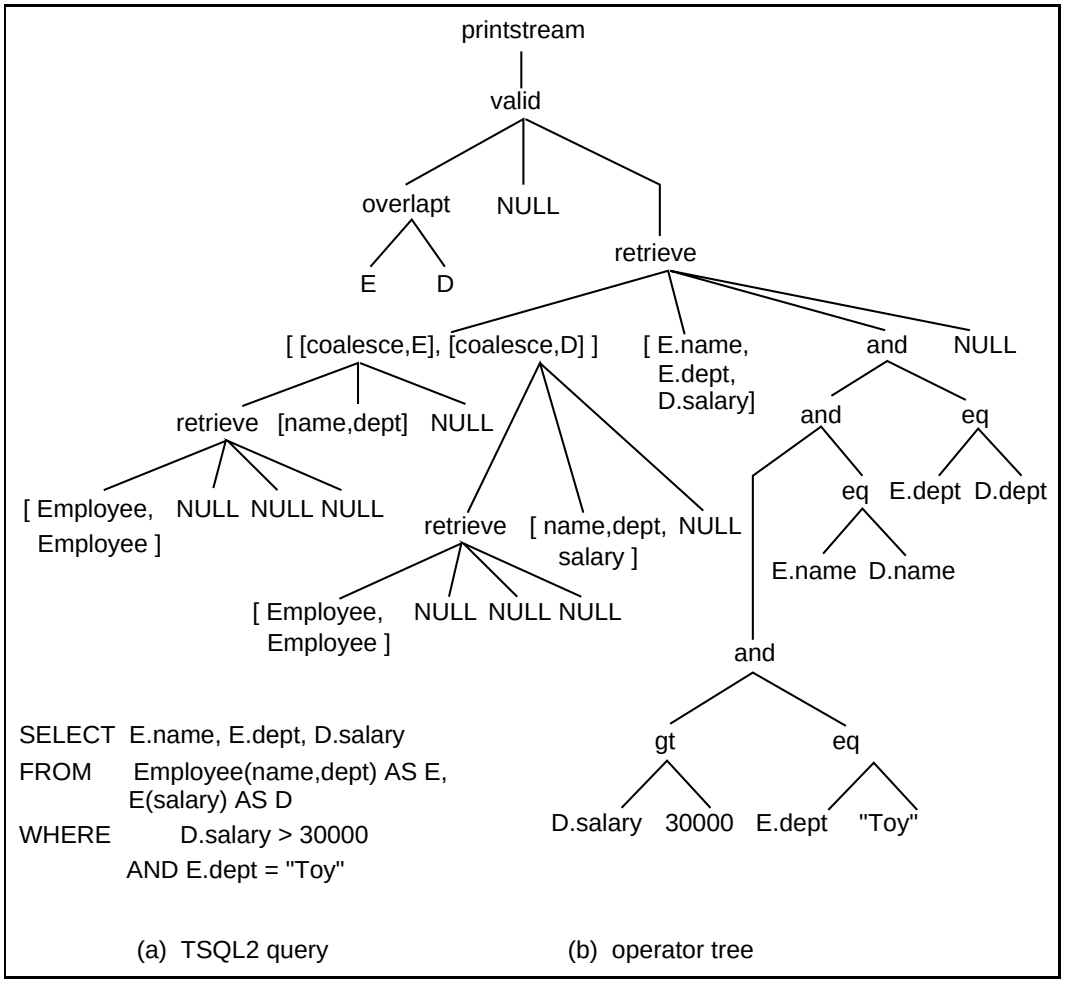

Figure 4.14: Operator Tree: TSQL2 SELECT Statement

As none of the valid-time projection clauses appear, the *valid()* backplane function must use the default temporal expression to compute the valid-time timestamp. The default valid-time timestamp of a derived tuple is computed as the temporal intersection of the valid-time timestamps of the tuples which joined to produce it. The temporal expression for computing the default valid-time timestamp is generated in the post-action of the last signature of the *validtime* subcatalog by a call to the *gen_expr()* directive function[6].

Parsing continues with the *compattr* and *proc_stream* subcatalogs, which recognize the <*select list*> and separate arithmetic computation from aggregation. The next clauses in line for recognition are the `HAVING` and `GROUP BY` clauses; as neither appears, parsing continues with the *f_ret* subcatalog.

Relations participating in the query are specified in the FROM clause. They may optionally be *coalesced*. To coalesce a relation, its name is followed in the FROM clause by a list of attributes; in the sample query, the Employee relation is coalesced on "(name,dept)." The tuples of the relation are *grouped* on the listed attributes. For each group, all tuples whose valid-time timestamps can be merged into a continuous interval are merged into a single tuple. Attributes which are not listed may not be referenced in the query.

To implement coalescing, a new function, *coalesce()*, was added to the backplane. The *coalesce()* backplane function groups its input stream on the named attributes, and then *coalesces* the stream on them: all tuples in a group whose valid-time timestamps can be merged into a single temporal element are merged into a single tuple.

Coalescing a relation is treated as a subquery appearing in the FROM clause. This is modeled by adding a signature to the *relation_ref* subcatalog to allow a coalesced relation to appear as a relation reference and then adding a new subcatalog, *coalescrel*, to model the coalesced relation. In the *coalescrel* subcatalog, the *coalesce()* and *retrieve()* backplane functions are composed to first retrieve and then coalesce the relation.

Furthermore, coalescence can be inherited. Inherited coalescence is specified in the FROM clause by referencing the correlation name of a coalesced relation as the relation name. The result is that the second coalesced relation is coalesced on the same attributes of the first, as well as the additional attributes specified in its own attribute list. In the example, the first element of the FROM clause is a temporary relation `E`, coalesced on attributes "(name,dept)," while the second element

[6]The *gen_expr()* directive function generates an expression from its inputs, a list of expressions, a binary operator, and an associativity. The generated expression applies the operator to the expressions in the list, observing the given associativity. For example, the call *gen_expr([1,2,3],"+","left")* would produce the expression ((1 + 2) + 3).

of the FROM clause, "E(salary) AS D," produces a temporary relation `D` coalesced on "(name,dept,salary)." To handle inheritance of coalescence, we added a new directive function, *map_coalesce_depnds()*, which propagates coalescing attributes.

Finally, inheriting coalesced attributes links the two relations more intimately: the predicate of the WHERE clause is augmented with conjuncts that ensure that the inherited grouped attributes are equal. In our example, the conjuncts "eq(E.name,D.name)" and "eq(E.dept,D.dept)" were generated automatically in the function *map_coalesce_depnds()* and `AND`ed with the `WHERE` predicate.

The last clause recognized, the `WHERE` clause, is parsed in the *where* subcatalog. Besides adding valid-time projection, TSQL2 adds valid-time selection, the ability to select tuples based on their timestamp values. Temporal expressions can be compared to each other and the result used to select tuples. Valid-time selection is added by augmenting the predicate of the WHERE clause with temporal comparisons. We defined subcatalogs to model temporal comparisons, then added the root subcatalog *timecompare* as an option of the *compare* subcatalog, thereby allowing temporal comparisons to participate in predicates.

The INSERT Statement. Our model of the INSERT statement is shown in Table 4.15(b); its syntax is shown in Table 4.15(a).

We show a sample *INSERT* statement in Figure 4.15. Parsing the `INSERT` statement begins with the *TSQL2* subcatalog, where the `INSERT INTO` keywords are recognized and the *tminsert()* backplane function is composed into the operator tree.

The backplane was augmented with a new backplane function, *tminsert()*, for insertion as temporal insertion differs from conventional insertion. TSQL2 specifies that modifications should not overlap existing tuples, so if the non-temporal values of a tuple are already valid in the current state of the database, the *tminsert()* backplane function must not insert them again. Instead, it must split the new tuple, potentially into two, to cover the portions of the time-line that are in the new tuple but not in the existing tuple.

Parsing the input stream continues with the *insrtstrm* subcatalog, where the *rename()* backplane function is composed. The first parameter of *rename()* is a list of strings, recognized in the *strng_list* subcatalog, which rename the attributes of the incoming stream, associating them with the attributes of the relation being augmented. The second is the input stream itself.

As for the SQL INSERT statement, input tuples may come from either a literal stream of tuples or an SFW-expression. If they are derived from the latter, their valid-time timestamps are computed with them and included in the input

INSERT INTO
    <relation>
    (*<insert list>*)
SELECT-FROM-...

INSERT INTO
    <relation>
    (*<insert list>*)
<literal tuple stream>

(a) syntax

```
BEGIN CONTEXT X
   unrel_list : unit_rel_list;
   ...
END CONTEXT X

TSQL2       =     ...
            |  { newCNTXT(X); }
               tminsert[" INSERT INTO _x _y ",
                  x:unit_rel_list, NULL, y:insrtstrm]
               { mergerelation(x,unrel_list); }
;
insrtstrm   =  rename[" ( _x ) _v ", x:strng_list, v:i_valid]
            |  rename[" ( _x ) _y ", x:strng_list, y:select]
            |  valid[" _l ", NULL, NULL, l:litrelation]
            |  select
;
i_valid     =  valid[" _l VALID _e ", e:element, NULL,
               l:litrelation]
;
litrelation =  values[" VALUES _x ", x:littuples]
;
littuples   =  list[ , , ",", valuelist, NE]
;
valuelist   =  list["(", ")", ",", literal, NE]
;
strng_list  =  list[ , , ",", String, NE]
;
```

(b) model

Table 4.15: TSQL2 INSERT Statement

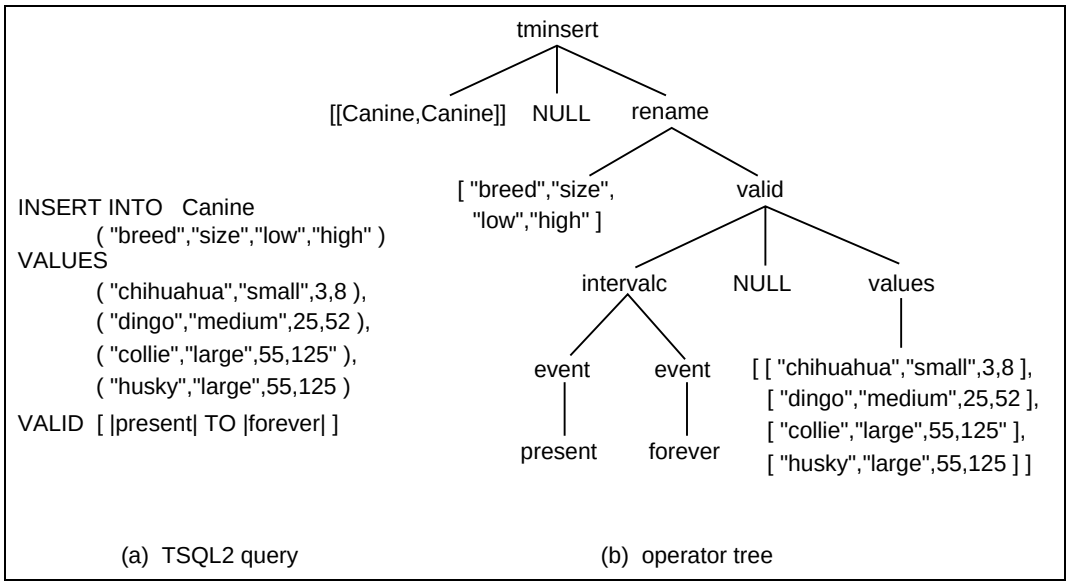

Figure 4.15: Operator Tree: TSQL2 INSERT Statement

stream. However, if the input stream of tuples is a literal relation, then the INSERT statement is augmented with a VALID clause to compute valid-time values for the tuples. In our example, the input stream to the *rename()* backplane function is recognized in the *i_valid* subcatalog, where the *valid()* backplane function is composed to add a valid-time timestamp to *its* input stream.

The actual input stream, which in our example is a literal stream of tuples specified in the VALUES clause, is recognized in the *litrelation* subcatalog. Here the *values()* backplane function is composed along with its input, a sequence of lists of literal values.

The DELETE Statement. Our model of the TSQL2 `DELETE` statement is shown in Table 4.16. In Figure 4.16, we show a sample TSQL2 DELETE statement.

Parsing the DELETE statement begins in the *TSQL2* subcatalog, where the *tmdelete()* backplane function is composed into the operator tree. Like the *tminsert()* backplane function, the *tmdelete()* backplane function must delete only the portion of an existing tuple that matches the valid time on the incoming tuple. Furthermore, it doesn't actually *delete* the tuple: by setting the value of the endpoint of the valid-time timestamp, *tmdelete()* marks the tuple invalid while leaving it in the database, thereby allowing the user to query the historical state of the database.

Parsing continues with the *d_valid* subcatalog. Augmenting the DELETE statement with temporal capabilities consists of one basic change: adding a VALID

DELETE
FROM
    <*relation*>
WHERE
    <*predicate*>
VALID
    <*temporal expression*>

(a) syntax

```
BEGIN CONTEXT X
   unrel_list : unit_rel_list;
   proj_list : attr_list;
END CONTEXT X

TSQL2  =     ...
             { newCNTXT(X); }
             tmdelete[" DELETE _x ",unrel_list,x:d_valid]
;
d_valid  =   valid[" _d VALID _v ",v:element,NULL,d:d_ret]
;
d_ret    =   retrieve[" FROM _x _w ", x:unit_rel_list,
                proj_list, w:where, NULL]
             { mergerelation(x,unrel_list); }
;
where    =   noop[" WHERE _x ", x:predicate]
         |   epsilon
;
```

(b) model

Table 4.16: TSQL2 DELETE Statement

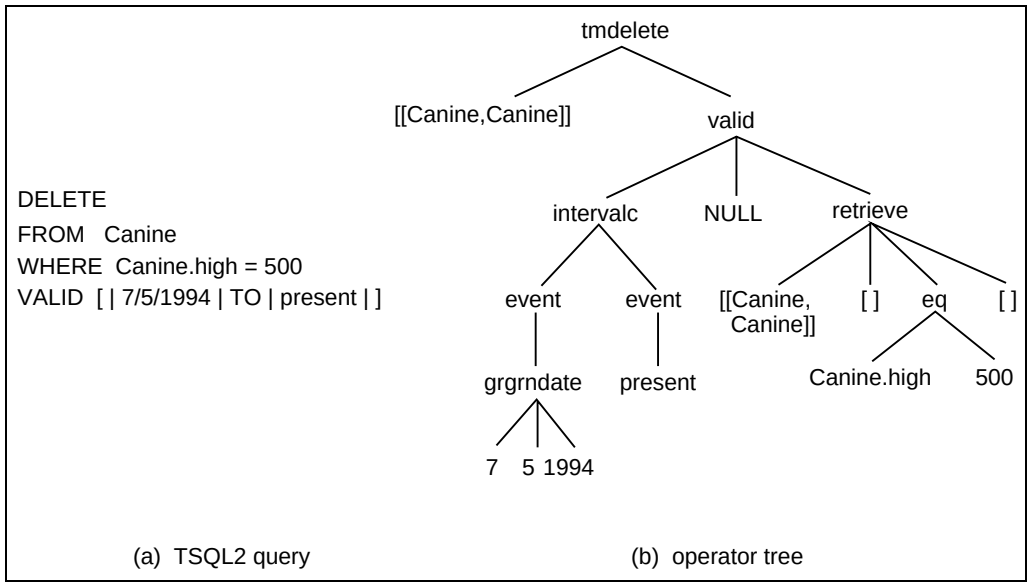

Figure 4.16: Operator Tree: TSQL2 DELETE Statement

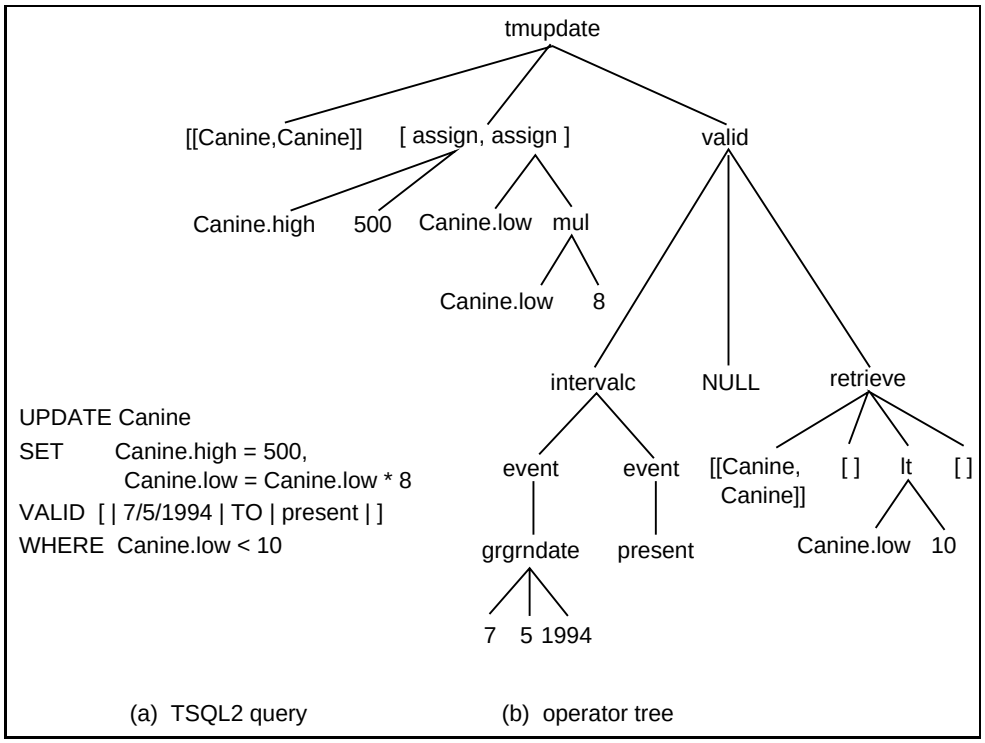

Figure 4.17: Operator Tree: TSQL2 UPDATE Statement

clause to determine the valid times which are to be deleted from the database. This valid-time timestamp is computed by the *valid()* backplane function after the tuples to be deleted have been selected. In our example, all selected tuples having a valid timestamp in the interval "7/15/1994" through the present are to be deleted. Creating the operator tree to produce the stream of tuples to be deleted continues with the *d_ret* subcatalog.

Selection of the tuples to be deleted occurs in the *retrieve()* backplane function, composed in the *d_ret* subcatalog. The WHERE clause, modeled in the *where* subcatalog, provides a predicate for the *retrieve()* backplane function.

The UPDATE Statement. The TSQL2 UPDATE statement modifies values of tuples in the database; see Table 4.17 for its syntax and model. Figure 4.17 shows a sample TSQL2 UPDATE statement and the operator tree to which it is mapped.

Parsing the UPDATE statement begins in the *TSQL2* subcatalog, where the *tmupdate()* backplane function is composed as the root of the operator tree. Like the temporal insert and delete, temporal update is more complex than non-temporal update. We augmented the backplane with a new function for temporal

UPDATE
    <*relation*>
SET
    <*assignment list*>
VALID
    <*temporal expression*>
WHERE
    <*predicate*>

(a) syntax

```
BEGIN CONTEXT X
   unrel_list : unit_rel_list;
   proj_list : attr_list;
   ...
END CONTEXT X

TSQL2       =     ...
            |  { newCNTXT(X); }
               tmupdate[" UPDATE _x _a _u ",
                  x:unit_rel_list,a:u_expr,u:u_valid]
               { mergerelation(x,unrel_list); }
;
u_expr      =  noop[" SET _y ", y:assgn_list]
;
u_valid     =  valid[" VALID _v _u ",v:element,NULL,u:u_ret]
;
u_ret       =  retrieve[" _w ",unrel_list,NULL,w:where,NULL]
;
where       =  noop[" WHERE _x ", x:predicate]
            |  epsilon
;
assgn_list  =  list[ , , ",", assgn, NE]
;
assgn       =  assign[" _x = _y ", x:attribute, y:expr]
               {  z:attr_list;
                  extattr(y,z);
                  mergeattr(z, proj_list);
                  extattr(x,z);
                  mergeattr(z, proj_list);
               }
;
```

(b) model

Table 4.17: TSQL2 UPDATE Statement

update, *tmupdate()*. The stream of tuples input to *tmupdate()* arrive with a valid-time timestamp already computed. If an existing tuple in the database has the same non-temporal values but a *different* valid-time timestamp than a tuple in the input stream, then only the portions of the existing tuple which overlap the valid-time timestamp of the new tuple are changed. An existing tuple may be split into as many as three parts.

The first parameter of the *tmupdate()* backplane function is the name of the relation to be updated; in our example, the Canine relation. The *tmupdate()* backplane function has two parameters of interest: an assignment list and an input stream of tuples selected for update. The assignment list is recognized in the *u_expr* subcatalog, where it composes no additional backplane functions. Recognizing the input stream of tuples begins in the *u_valid* subcatalog.

The UPDATE statement is augmented with a VALID clause to specify the valid-time timestamp for the new values. This clause is recognized in the *u_valid* subcatalog and the *valid()* backplane function is composed into the operator tree.

Finally, the *u_ret* subcatalog composes the *retrieve()* backplane function to select from the database the tuples to be updated. The predicate for tuple selection is specified in the WHERE clause, which is recognized in the *where* subcatalog.

4.2.3.3 Conclusions for TSQL2

In this section we consider our results for TSQL2, including the model size and development time. Our model includes all of the new temporal types and operations on them, as well as the data retrieval and modification statements. As for our other languages in the SQL family, certain primitive data types were not included: fixed length strings, BIT strings, and numeric types with specified precision. Furthermore, the LIKE predicate was not included. Finally, we assumed the Gregorian calendar, with the finest granularity chronon being one day. We do not feel that this detracts significantly from the expressiveness of the language.

A more serious omission was the run-time support for specifying calendars. TSQL2 includes a feature to support different calendric systems, and the user should be able to switch dynamically between using the different calendric systems. A date which is specified in a query is interpreted relative to the calendar in effect at the time the query is issued. Because the generated compilers do not have static run-time memory, we were unable to support this feature. Static run-time memory for generated compilers is a topic for future work on Rosetta; we will say more about this in Chapter 6.

Our model of TSQL2 required 350 lines. The specification was mapped to bison and flex files comprising a total of about 2500 lines, yielding an expansion

ratio of 1:7, quite similar to our results for SQL and SQL/NF.

Specification reuse for modeling TSQL2 was very high—87%. This is attributable to the goal of the TSQL2 designers to maintain upward compatibility with SQL.

Finally, the time needed to produce the TSQL2 model was approximately 16 days. This included the time needed to understand the specification and make the necessary additions and changes to our SQL model. Again, this was the time needed by a novice in temporal databases.

Most of the changes made to SQL were quite low-level, such as modeling the primitive temporal data types and operations on them. Other changes, such as allowing a form of subquery in the FROM clause, were higher level. Overall, TSQL2 differs significantly from both SQL and SQL/NF.

4.3 The Quel Family

The second family we consider is the Quel family. The Quel family was chosen because, like the SQL family, it is a large family with wide variation among its members. Quel is a popular language in the database community and has had a significant impact on data language development: like SQL, Quel frequently serves as the base language on which researchers graft experimental features and extensions. Members of the Quel family support non-traditional data models, include new data types, and extend the syntax. We modeled Quel and one other member of the Quel family—TQuel, a temporal data language.

4.3.1 Quel

Our interest in Quel stems from both its popularity and its historical interest: Quel was one of the earliest relational data languages [SWKH76]. Like SQL, Quel is a traditional relational data language. However, Quel is of interest in its own right, as there are significant functional and syntactic differences between SQL and Quel. Quel was developed at the University of California at Berkeley as the data language of the database management system INGRES, [HSW75].

This section focuses on only the high-level elements of the language specification; the full Quel model, including low-level constructs such as arithmetic expressions and primitives, is listed in Appendix E.

RANGE OF	<correlation list>
IS	<relation>

(a) syntax

```
Quel          =    ...
                 |  dfnCORR[" RANGE OF _v IS _r ",
                        v:relation_list, r:relation ]
;
relation_list = list[ , , "," , relation, NE]
;
```

(b) model

Table 4.18: Quel RANGE OF Statement

4.3.1.1 Modeling Quel Statements

Quel has many of the same statements (modulo syntax) that SQL has. Both languages have statements to retrieve and display tuples, to insert new tuples, and to delete or modify existing tuples. Both languages have subqueries and aggregation. But there are differences: SQL has set operation statements UNION, INTERSECT, and EXCEPT which Quel lacks; furthermore, Quel cannot group selected tuples or apply a predicate to select groups. However, Quel has more powerful aggregation constructs.

The RANGE OF Statement. The Quel RANGE OF statement performs the same function as the SQL FROM clause, associating correlations with relations, but its scope is global and therefore its consequences are more far-reaching. The syntax and model of the RANGE OF statement are shown in Table 4.18.

The RANGE OF statement defines the names in the <*correlation list*> as correlations of the named relation. Unlike the SQL FROM clause, the effect of the Quel RANGE OF statement is global. Once defined, a correlation remains active until it is redefined, and may be referenced in any number of Quel statements. Quel correlations are defined by the *dfnCORR()* backplane function[7].

The RETRIEVE Statement. The Quel RETRIEVE statement is functionally very similar to the SQL SELECT statement: both retrieve and print tuples which satisfy a predicate. Although the Quel RETRIEVE statement and the SQL SELECT statement are functionally similar, syntactically they are quite different.

The syntax of the Quel RETRIEVE statement is shown in Table 4.19a. The only clauses of the RETRIEVE statement which must appear are the RETRIEVE

[7]The *dfnCORR()* backplane function, added when modeling Quel, has parameters a list of correlation names and a relation name. It manages a global correlation list which associates each correlation name with a relation.

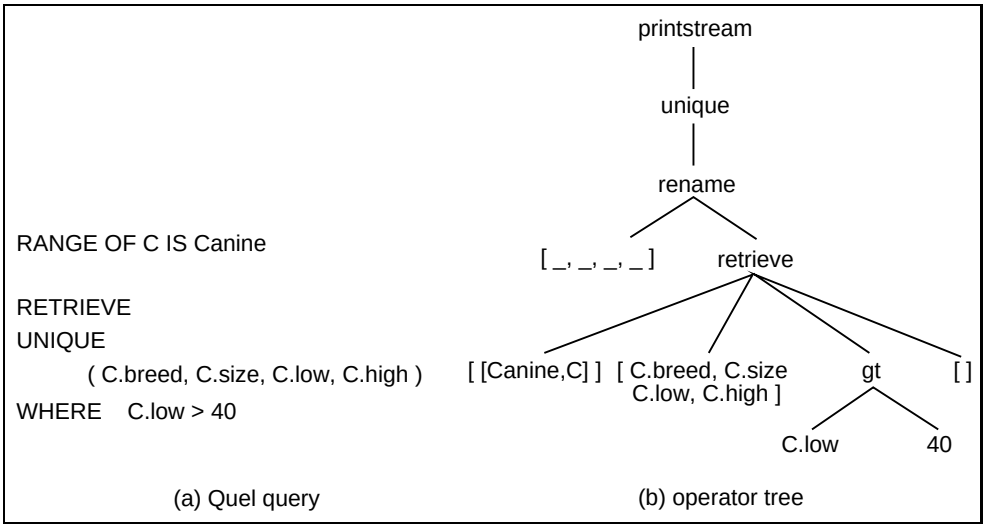

Figure 4.18: Operator Tree: Quel RETRIEVE Statement

clause and the <*list of expressions*>; all the others are optional. Our model of the Quel RETRIEVE statement is shown in Table 4.19b.

Figure 4.18 shows a Quel RETRIEVE statement and the operator tree our model maps it to. Mapping this statement to the operator tree begins with the *Quel* subcatalog, which differentiates between statements of the Quel language. The RETRIEVE keyword is detected in the *Quel* subcatalog, but no function can be composed at this point; only when keywords which differentiate the alternatives of the RETRIEVE statement have been detected can the action to be taken be determined.

In the *retstmt* subcatalog, one option is eliminated. The INTO clause, which indicates that a temporary relation should be created and populated with the retrieved tuples, is recognized in the first signature of the *retstmt* subcatalog. In the example, INTO is not present, and the second signature of the *retstmt* subcatalog is successful. No clause is recognized in the second signature, but since all other variations of the RETRIEVE statement print retrieved tuples, the *printstream()* backplane function is composed as the root of the operator tree. The input stream of tuples to be printed derives from the operator tree generated from the *display* subcatalog.

The RETRIEVE statement includes a UNIQUE clause, handled by the *display* subcatalog, which signals duplicate elimination. As the UNIQUE keyword is specified in our example, the *unique()* backplane function is composed into the operator tree. Its input stream is produced from the *rename* subcatalog.

In the *rename* subcatalog, the <*list of expressions*> which determines the

```
RETRIEVE
[ INTO
    <temporary relation> ]
    ( <list of expressions> )
[ WHERE <predicate> ]

RETRIEVE
[ UNIQUE ]
    ( <list of expressions> )
[ WHERE <predicate> ]
```

(a) syntax

```
BEGIN CONTEXT X
   rel_list : relation_list;
   xpr : expr_list;
   ...
END CONTEXT X

Quel      =     ...
          |  { newCNTXT(X); }
             noop[" RETRIEVE _x ",x:retstmt]
;
retstmt   =  create[" INTO _x _y ", x:relation, y:insertstrm ]
          |  printstream[" _x ", x:display]
;
display   =  unique[" UNIQUE _r ", r:rename]
          |  rename
;
rename    =  rename[" ( _p _c ", p:renamelst,c:process1]
;
process1  =  noop[" ) _c ",c:proc_cycl1]
;
proc_cycl1=  compute[rewrite(xpr,
                [add,sub,div,mul,uminus,str2int,str2real], x),
                x:expr_list, proc_cycl1]
          |  aggregate[rewrite(xpr,
                [count,min,max,avg,sum,filter,aggregate,groupby,
                gt,ge,lt,le,eq,str2int,str2real],x),
                x:expr_list, proc_cycl1]
          |  noop[is_attr_list(xpr), retstrm1]
;
retstrm1  =  retrieve[" _w ", rel_list, xpr, w:where, NULL]
;
where     =  noop[" WHERE _x ", x:wpredicate]
          |  epsilon
;
```

(b) model

Table 4.19: Quel RETRIEVE Statement

values of the output tuples is recognized via the *renamelst* subcatalog[8]. Next, aggregations and arithmetic computations are separated for Quel as they were for SQL; this happens in the *process1* and *proc_cycl1* subcatalogs. In our example, there is no computation or aggregation, so only the *rename()* backplane function is composed into the operator tree.

Finally, in the *retstrm1* and *where* subcatalogs, the *retrieve()* backplane function is composed into the operator tree and the WHERE clause is recognized. The <*predicate*> of the WHERE clause is passed in to the *retrieve()* backplane function to filter the retrieved tuples. The WHERE clause is optional; if it is not present, the predicate is vacuously satisfied by all tuples. Global variables are referenced to provide a relation list and an attribute projection list for the *retrieve()* backplane function.

Overall, the Quel RETRIEVE statement has almost the same functionality as the SQL SELECT statement. However, the RETRIEVE statement has no clauses to divide a stream into groups or to test predicates on groups.

Subqueries. Quel has a version of the subquery which implements existential quantification (see Table 4.20 for syntax and model). The integer construct `ANY` tests the result of a subquery, returning 1 if the result is not empty and 0 otherwise.

Figure 4.19 shows a query which uses the `ANY` construct and the operator tree our model maps it to. As our example is a RETRIEVE statement, generating the operator tree begins in the usual way, with the subcatalogs of Table 4.19 composing the *printstream()*, *rename()*, and *retrieve()* backplane functions into the operator tree. The subquery itself, which is parsed as part of the predicate, is recognized in the *wquelexpr* subcatalog.

Two backplane functions are used to model the `ANY` construct. The *exists()* backplane function tests a stream, returning the boolean values `TRUE` if non-empty and `FALSE` otherwise. To use it to model the `ANY` construct, the boolean result must be converted to an integer. We added a new backplane function *boolean2int()* which maps `TRUE` to 1 and `FALSE` to 0, and modeled the ANY construct by a composition of these two functions.

The stream argument of the *exists()* backplane function is modeled in the *subquery* subcatalog. The two remaining clauses to be modeled, BY and WHERE, are both optional. The BY clause, if present, is modeled by a *groupby()* backplane function call on the retrieved stream, while the predicate of the WHERE clause is passed as a parameter to the *retrieve()* backplane function.

[8]The *rename()* backplane function renames the attributes in a stream; its parameters are a list of names and an input stream. Unspecified name list elements result in no change to the attribute name.

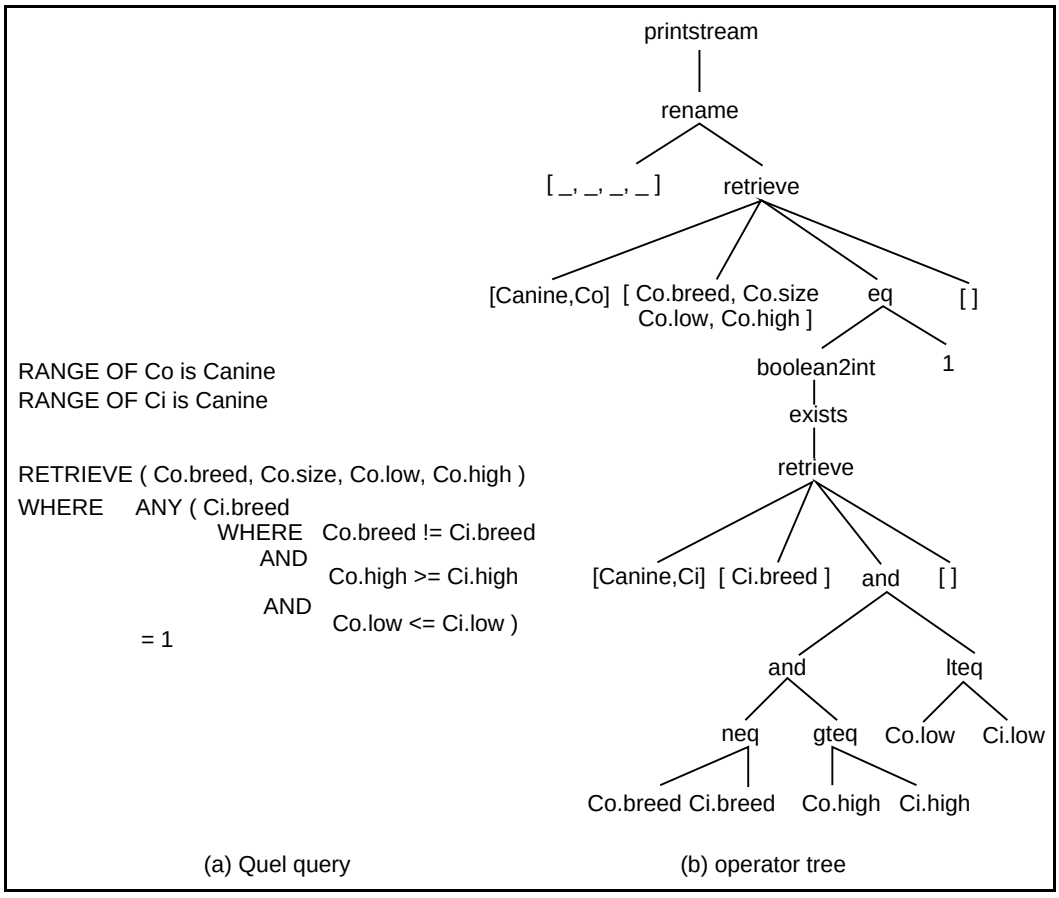

Figure 4.19: Operator Tree: Quel RETRIEVE Statement with Subquery

```
ANY
    (<expression list>)
[ BY
    <attribute list> ]
[ WHERE
    <predicate> ]
```

(a) syntax

```
BEGIN CONTEXT X
   rel_list : relation_list;
   xpr : expr_list;
   ...
END CONTEXT X

wquelexpr    =    ...
             |  { newCNTXT(X); }
                boolean2int[" ANY ( _x ) ", exists(x:subquery)]
                { enforcescope("X","rel_list"); }
;
subquery     =  groupby[" _xpr BY _g _w",g:attr_list, w:substream]
             |  retrieve[" _xpr _w ",rel_list,xpr,w:where,NULL]
;
substream    =  retrieve[" _w ",rel_list,xpr,w:where,NULL]
;
```

(b) model

Table 4.20: Quel Subquery

Aggregation. Like SQL aggregations, Quel aggregations are applied to an input stream. However, a Quel aggregation can specify additional computation on its input stream: a list of attributes on which to group the input stream; a predicate to further filter the input stream; even nested aggregations (see the syntax in Table 4.21a). These powerful aggregation expressions add a great deal of expressiveness to Quel.

Aggregates in Quel are handled differently depending on whether they occur in the select list or in the predicate. An aggregate which occurs in the select list is evaluated against the stream of tuples which satisfy the predicate of the WHERE clause, whereas an aggregate occurring in the predicate of the WHERE clause is evaluated as a subquery, so that tuples are retrieved exclusively for the aggregation. In this section, we will consider only aggregation in the select list (the model is displayed in Table 4.21b). Aggregation in the predicate is included in the Quel model in Appendix E.

Figure 4.20a shows a Quel query with aggregation in the select list. This query retrieves the size attribute for each tuple in the Canine relation and also computes the averages of the low and high values for that category. Our Quel model maps it to the operator tree of Figure 4.20b.

Recognition of the statement is initially the same as recognition of any other `RETRIEVE` statement, composing the *printstream()* and *rename()* backplane functions from the *Quel*, *retstmt*, and *display* subcatalogs (see Table 4.19).

Actually parsing the select list begins in the *renamelst* subcatalog (see Ta-

```
BEGIN CONTEXT X
    xpr : expr_list;
    ...
END CONTEXT X

rename      =  rename[" ( _p _c ", p:renamelst,c:process1]
;
renamelst   =  list[ , , ",", retcolumn, NE]
               ;
retcolumn   =  noop[" _x = _y ", x:String]
               {  y:quelexpr; linknode(xpr,y); }
            |  noop[" _x = _y ", x:String]
               {  y:String; linknode(xpr,y); }
            |  noop[" _y ", NULL]
               {  y:quelexpr; linknode(xpr,y); }
;
quelexpr    =  add[" _x + _y ", x:quelexpr, y:quelexpr]
            |  sub[" _x - _y ", x:quelexpr, y:quelexpr]
            |  mul[" _x * _y ", x:quelexpr, y:quelexpr]
            |  div[" _x / _y ", x:quelexpr, y:quelexpr]
            |  uminus[" - _x ", x:quelexpr]
            |  noop[" ( _x ) ", x:quelexpr]
            |  Numeric
            |  attribute
;
Numeric     =  LitNumeric
            |  quelaggr
;
quelaggr    =  avg[" AVG ( _e _b ) ", e:aggrexpr, b:aggrgrp]
            |  count[" COUNT ( _e _b ) ", e:aggrexpr, b:aggrgrp]
            |  max[" MAX ( _e _b ) ", e:aggrexpr, b:aggrgrp]
            |  min[" MIN ( _e _b ) ", e:aggrexpr, b:aggrgrp]
            |  sum[" SUM ( _e _b ) ", e:aggrexpr, b:aggrgrp]
            |  avg[" AVGU ( _e _b ) ", e:aggrexpr,
                  unique(b:aggrgrp)]
            |  count[" COUNTU ( _e _b ) ", e:aggrexpr,
                  unique(b:aggrgrp)]
            |  sum[" SUMU ( _e _b ) ", e:aggrexpr,
                  unique(b:aggrgrp)]
;
aggrgrp     =  groupby[ " BY _a _w ", a:attr_list, w:select]
            |  select
            |  quelaggr
;
aggrexpr    =  smexpr
            |  epsilon
;
select      =  filter[ " WHERE _x ", x:predicate, NULL]
            |  epsilon
;
```

(b) model

<aggregate operator>
    <expression>
[BY
    <attribute list>]
[WHERE
    <predicate>]

where *<aggregate operator>* is one of
    { MAX, MIN, AVG,
      COUNT, SUM,
      COUNTU, AVGU,
      SUMU }

(a) syntax

Table 4.21: Quel Aggregation

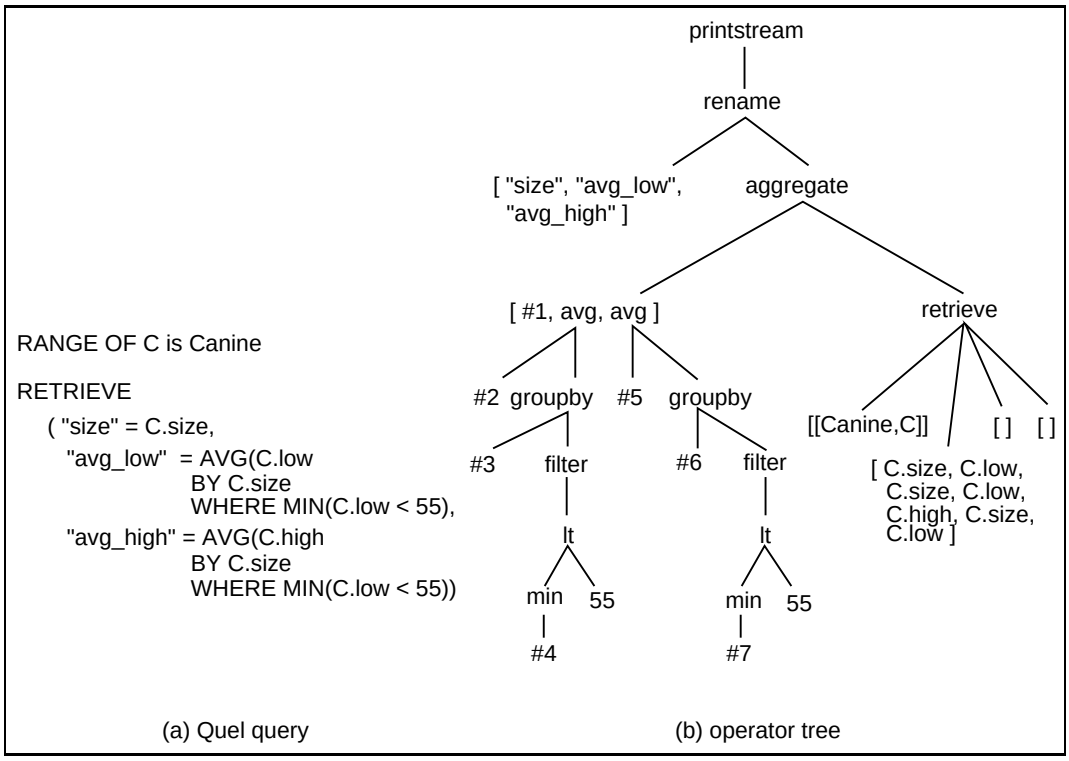

Figure 4.20: Operator Tree: Quel RETRIEVE Statement with Aggregation

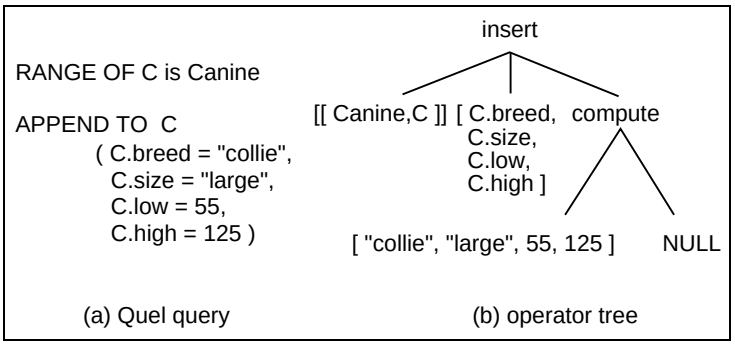

Figure 4.21: Operator Tree: Quel APPEND TO Statement

ble 4.21). The *retcolumn* subcatalog recognizes a single assignment; the *quelexpr* subcatalog recognizes the expression of the assignment, including aggregations. In the *retcolumn* subcatalog, the expressions are linked into the *xpr* context variable, from which they are later processed in the *proc_cycl1* cycle subcatalog, producing the expression list parameter of the *aggregate()* backplane function.

Finally, the subtree rooted at the *retrieve()* backplane function is produced from the *retstrm1* subcatalog (from Table 4.19). As there is no WHERE clause in this query, the predicate parameter of the *retrieve()* backplane function is empty and defaults to `TRUE`.

The APPEND TO Statement. The Quel APPEND TO statement inserts new tuples into relations (see Table 4.22a for the syntax of the APPEND TO statement). The new tuples may be either literal or derived tuples. The Rosetta model of the APPEND TO statement is shown in Table 4.22b.

In Figure 4.21, we show a sample APPEND TO statement. The APPEND TO clause is recognized in the *Quel* subcatalog, where the *insert()* backplane function is composed into the operator tree. The relation to be augmented is determined in the *unit_rel_list* subcatalog,

The *insertstrm* subcatalog models the stream of tuples to be inserted into the named relation, recognizing the expressions which compute the tuples to be inserted, while the *process2* and *proc_cycl2* subcatalogs separate arithmetic computation and aggregation, composing *compute()* and *aggregate()* backplane function calls as needed.

Finally, if there is a `WHERE` clause, it is recognized in the *retstrm2* subcatalog and the *retrieve()* backplane function is composed into the operator tree.

APPEND TO
    <*relation*>
    <*list of values*>
[WHERE <predicate>]

(a) syntax

```
BEGIN CONTEXT X
   attrlst : attr_list;
   rel_list : relation_list;
   ...
END CONTEXT X

Quel          =     ...
              |  { newCNTXT(X); }
                 insert[" APPEND TO _r _y ",r:unit_rel_list,
                    attrlst,y:insertstrm ]
;
insertstrm    =  noop[" ( _p _j ", j:process2]
                 {  p:insertlst;
                    skip();
                 }
;
process2      =  noop[" ) _c ",c:proc_cycl2]
;
proc_cycl2    =  compute[
                    rewrite(xpr,[add,sub,div,mul,uminus,
                       str2int,str2real,str2str],x),
                    x:expr_list, proc_cycl2]
              |  aggregate[
                    rewrite(xpr,[count,min,max,avg,sum,filter,
                       groupby,gt,ge,lt,le,eq,str2int,str2real,
                 str2str],x),x:expr_list, proc_cycl2]
              |  noop[is_attr_list(xpr), retstrm2]
;
retstrm2      =  retrieve[" WHERE _w ",
                    rel_list,xpr,w:wpredicate,NULL]
              |  epsilon
;
```

(b) model

Table 4.22: Quel APPEND TO Statement

DELETE <*relation*>
[WHERE <*predicate*>]

(a) syntax

```
BEGIN CONTEXT X
   rel_list : relation_list;
   ...
END CONTEXT X

Quel        = { newCNTXT(X); }
              delete[" DELETE _r _x ", r:unit_rel_list, x:delstrm]
              { mergerelation(r,rel_list); }
;
delstrm     = retrieve[" _w ",rel_list,NULL,w:where,NULL]
;
where       = noop[" WHERE _x ", x:wpredicate]
            | epsilon
;
```

(b) model

Table 4.23: Quel DELETE Statement

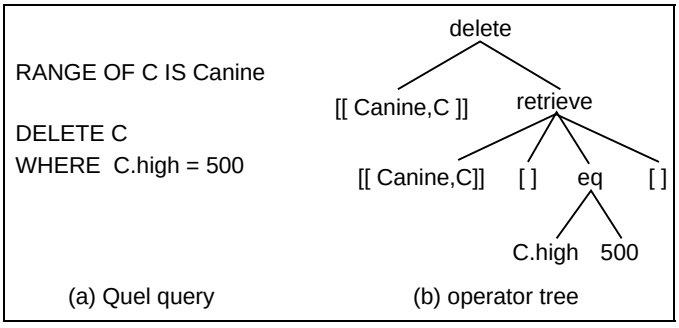

Figure 4.22: Operator Tree: Quel DELETE Statement

The DELETE Statement. The Quel DELETE statement (see syntax in Table 4.23a) deletes tuples from a named relation. An optional WHERE clause specifies a predicate which restricts the tuples to be deleted. Our model of the DELETE statement is shown in Table 4.23b.

A sample DELETE statement and its operator tree are shown in Figure 4.22. The DELETE keyword, recognized in the *Quel* subcatalog, signals composition of the *delete()* backplane function into the operator tree. Also in that signature, the name of the relation to be modified is recognized. The post-action of the *Quel* subcatalog makes the name of that relation globally available in the *rel_list* context variable. The *retrieve()* backplane function is composed in the *delstrm* subcatalog. Finally, the WHERE clause is recognized in the *where* subcatalog, where its predicate is made available to the *retrieve()* backplane function.

REPLACE <*relation*>
<*update expressions*>
[WHERE <*predicate*>]

(a) syntax

```
BEGIN CONTEXT X
   updlist : assgnlst;
   rel_list : relation_list;
   ...
END CONTEXT X

Quel        =    ...
            |  { newCNTXT(X); }
               update[" REPLACE _r _y ",
                  r:unit_rel_list,updlist,y:updatestrm]
               { mergerelation(r,rel_list); }
;
updatestrm  =  retrieve[" ( _updlist ) _w ",
               rel_list,NULL,w:where,NULL]
;
where       =  noop[" WHERE _x ", x:wpredicate]
            |  epsilon
;
```

(b) model

Table 4.24: Quel REPLACE Statement

The REPLACE Statement. The Quel REPLACE statement updates existing tuples, replacing old attribute values with new attribute values. Disregarding keywords[9], the Quel REPLACE statement is quite similar to the SQL UPDATE statement.

The syntax of the Quel REPLACE statement is shown in Table 4.24a. Only the WHERE clause is optional; the rest of the syntax must appear. Our model of the Quel REPLACE statement is shown in Table 4.24b.

A sample REPLACE statement, with the operator tree to which our model maps it, is shown in Figure 4.23. The REPLACE clause is recognized in the *Quel* subcatalog and the *update()* backplane function is composed into the operator tree. In the *updatestrm* subcatalog, the <*update list of expressions*> is recognized and the *retrieve()* backplane function is composed into the operator tree. Finally, the WHERE clause is recognized in the *where* subcatalog and the predicate made available for the call to the *retrieve()* backplane function.

4.3.1.2 Quel Run-Time Experiences

Enough of the backplane functions were implemented (in Prolog) to evaluate Quel queries. Approximately forty Quel queries were evaluated. We used relations ranging in size from ten tuples to about five hundred tuples. All statements were ex-

[9]The UPDATE keyword is replaced by REPLACE; the SET keyword is eliminated.

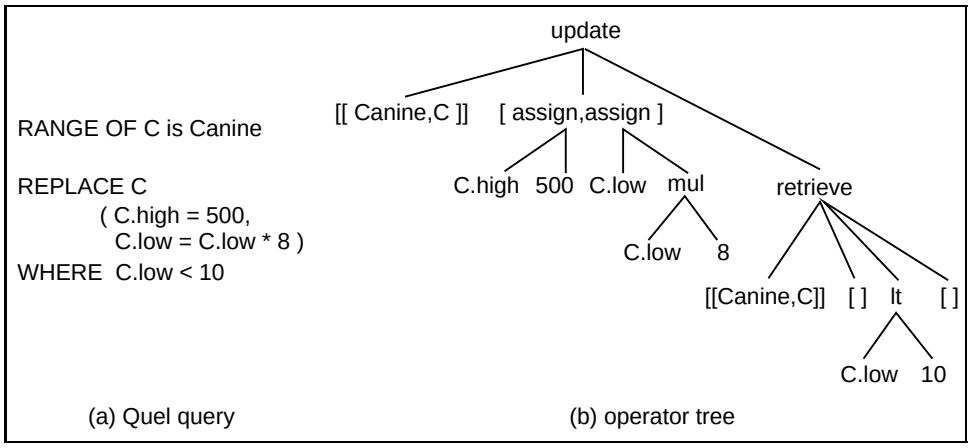

Figure 4.23: Operator Tree: Quel REPLACE Statement

ercised, including data retrieval, with nested subqueries and nested aggregations, and updates; all executed correctly. Thus, we feel that the prototype satisfactorily demonstrates a proof of concept.

4.3.1.3 Conclusions for Quel

In this section we consider the model size and development time for Quel. Our experiences in modeling Quel and other languages are used to evaluate the effectiveness of Rosetta.

Our Quel model is quite complete. All Quel data manipulation statements are included in the model: the RETRIEVE statement for data retrieval, with all its options, as well as the data modification statements APPEND TO (insert), DELETE, and REPLACE (update). The primitive types are limited to varying length character strings and integers and reals of unspecified precision.

Furthermore, our Quel model is compact. The Quel specification file consists of approximately 270 lines. From the specification, Rosetta generates a bison file and a flex file comprising approximately 100 lines and 1400 lines, respectively, yielding an expansion factor of approximately 1:6.

Overall, Quel is syntactically quite different from SQL. There is some overlap in functionality, but there are differences as well. The most significant functional difference was generalizing aggregations to allow nesting.

As an indication of the magnitude of the changes to the SQL specification file we consider specification reuse. The SQL specification file is composed of approximately 115 signatures. Approximately 25% of the SQL signatures were included either unchanged or with only minor modifications in the Quel model. Thus, even

though Quel syntax differs radically from SQL syntax, about one quarter of the Quel specification was done before we even began modeling.

Modeling Quel required generalizing one backplane function and adding four new backplane functions and five new directive functions.

Finally, the time required to develop our model of Quel was approximately four weeks, including both the time needed to understand the syntactic and semantic differences between SQL and Quel and to develop the model.

4.3.2 TQuel

Our second data point in the Quel family, and the last language we consider, is TQuel [Sno87], a temporal data language developed by Richard Snodgrass at the University of North Carolina at Chapel Hill, NC. We chose TQuel as one of our data points because it is an extension of Quel which augments the standard relational operators with temporal constructs.

Like TSQL2, TQuel is a temporal data language. Although TQuel is a derivative of Quel while TSQL2 is a derivative of SQL, TQuel bears some functional resemblance to TSQL2, perhaps because Snodgrass was instrumental in the development of both languages[10].

TQuel supersets Quel: all Quel statements are included in TQuel. Overall, quite a few extensions were made to Quel, including new temporal primitives and new functions to operate on them. No new statements were introduced; instead new clauses were added to the existing statements.

The rest of this section discusses the extensions which transform Quel into TQuel. As usual, we do not consider the data definition language; instead, we focus on the high-level elements of the data language. The full TQuel model, including such low-level constructs as arithmetic expressions and primitives, is listed in Appendix F. We consider first the temporal data model, including primitive temporal data types, then the impact of new clauses added to the Quel statements. Finally, we consider model size, development time, and productivity gains.

4.3.2.1 TQuel Data Model

With respect to temporal capabilities, databases can be classified in four categories: snapshot, rollback, historical, and temporal databases. Snapshot databases have no support for temporal capabilities; they reflect the state of the world at a particular moment in time and maintain no historical information.

[10]Snodgrass was the chair of the TSQL2 design committee.

A rollback database maintains historical information, enabling the user to issue queries with respect to a particular moment in time, but allows modifications to only the current version of the database. Each tuple in the database is given a transaction-time timestamp. Historical databases allow updates to all tuples, whether current or not. A valid-time attribute added to each tuple distinguishes between valid and invalid tuples.

Temporal databases incorporate both transaction-time and valid-time timestamps to provide the facilities of both rollback and historical databases. The transaction time timestamp is automatically computed by the DBMS while valid time is computed in clauses added to the language for that purpose.

Traditional database management systems include operations to delete and modify tuples. When tuples are deleted from the database, it as though they had never existed; when tuples are modified, the old values are written over with new values. However, in a temporal database, information is never deleted: it is marked invalid but can still participate in answering historical queries.

To support temporal queries, TQuel timestamps the tuples of derived relations and extends the query language with temporal selection. The traditional relational data model is extended with two temporal primitives: events and intervals. An event pinpoints a particular moment in time, whereas an interval covers a duration of time, starting at one event and continuing up to a second event. Interval constants are input as strings. There are also predefined constants, `beginning`, `now`, and `forever`, which are interpreted as either events or intervals depending on the context. Except for these there are no event constants.

Various operations are defined on temporal primitives. Two operations which return intervals are defined. The *overlap* operator operates on two intervals and returns the interval which is common to both. If the operands have no events in common, the result is undefined. The *extend* operator, another binary operator on interval operands, returns the maximal interval which includes both operands. Thus *overlap* can be viewed as interval intersection whereas *extend* corresponds to interval union.

Operations returning events are also defined: the *begin of* operator operates on an interval, returning its starting event, and the *end of* operator returns the interval's terminating event.

Finally, functions are provided to access valid-time and transaction-time timestamp values. All operate on a correlation name, interpreting it as referring to the current tuple. The *validat()*, *validfrom()*, and *validto()* backplane functions return valid-time timestamp values, either a valid-time event or the starting or ending event of a valid-time interval, respectively. The *transactionstart()* and *transac-*

tionstop() backplane functions return the transaction-time timestamp starting and ending events, respectively. These functions may be used only in the target list and WHERE clause.

In addition, there are comparison operations on temporal primitives which compute boolean values: *precede*, *overlap*, and *equal*. The binary operator *precede* operates on two intervals, returning `TRUE` if the first interval ends *before* the second interval begins and `FALSE` otherwise. The binary operator *overlap* also operates on two intervals, returning `TRUE` if the first interval is completely contained within the second interval and `FALSE` otherwise[11]. The third comparison operator, *equal*, operates on two events, returning `TRUE` if the two events are the same event and `FALSE` otherwise.

Finally, temporal predicates can be constructed from temporal comparisons using the boolean connectives *and*, *or*, and *not*.

4.3.2.2 Modeling TQuel Statements

TQuel includes all the statements of Quel but introduces no new ones. Our model of TQuel (see Appendix F) is restricted to the DML (even though some DDL statements were augmented). We consider the four augmented DML statements: RETRIEVE, DELETE, APPEND TO, and REPLACE. Other DML statements, such as the RANGE OF statement, and certain constructs were not changed for TQuel and are not discussed although they are included in the model.

The RETRIEVE Statement. Like the Quel RETRIEVE statement, the TQuel RETRIEVE statement (see Table 4.25 for syntax and model) retrieves from the database those tuples which satisfy a predicate. To add temporal capabilities, three new clauses were added to the TQuel RETRIEVE statement: the VALID, WHEN, and AS OF clauses. The first adds an expression to compute the valid time for the derived tuples; the second adds a temporal predicate which the retrieved tuples must satisfy; the third specifies an interval during which tuples participating in computation must be valid.

Figure 4.24 shows a sample TQuel RETRIEVE statement which retrieves pairs of breed names, where the first breed has a lower low weight value than the second breed.

Generating the operator tree commences in the *TQuel* subcatalog, where the `RETRIEVE` keyword is recognized. No action can be taken at this point as the single keyword `RETRIEVE` is insufficient to differentiate between the variations of the RETRIEVE statement.

[11] The *overlap* operator is overloaded.

```
RETRIEVE
[ UNIQUE | INTO ]
    <relation>
VALID
    <valid expression>
WHERE
    <predicate>
WHEN
    <when expression>
AS OF
    <when expression>
```

(a) syntax

```
BEGIN CONTEXT X
    rel_list : relation_list;
END CONTEXT X

TQuel     = { newCNTXT(X); }
            noop[" RETRIEVE _x ", x:retstmt]
;
retstmt   = create[" INTO _x _y ", x:relation, y:rename]
          | printstream[" _x ", x:display]
;
display   = unique[" UNIQUE _r ", coalesce(r:rename,NULL,NULL)]
          | rename
;
rename    = rename[" ( _p _c ", p:rettuple, c:process1]
;
rettuple  = list[ , , ",", retcolumn, NE]
;
process1  = noop[" ) _c ", c:proc_cycl1]
;
proc_cycl1= compute[rewrite(xpr,[add,sub,div,mul,uminus,str2int,str2rel,
              validat,validfrom,validto,xactionstart,xactionstop],x),
              x:expr_list, proc_cycl1]
          | aggregate[rewrite(xpr,[count,min,max,avg,sum,filter,
              groupby,gt,ge,lt,le,eq,str2int], x),x:expr_list, proc_cycl1]
          | noop[is_attr_list(xpr), valid1]
;
valid1    = valid[" VALID AT _e _r ",e:e_expr,NULL,r:retstrm1]
          | valid[" VALID FROM _i TO _k _r ", i:e_expr, k:e_expr,
              r:retstrm1]
          | valid[" _r ", i, NULL, r:retstrm1]
            {  i:i_expr; gen_expr(rel_list,"overlapt","left",i); }
;
retstrm1  = retrieve[" _c ", rel_list, xpr, c:retcond, NULL]
;
retcond   = and[" _w _n _a ", a:asof, and(n:when, w:where)]
;
asof      = asof[" AS OF _a ", a:e_expr, NULL]
          | asof[" AS OF _a ", a:interval, NULL]
          | asof[" AS OF _a THROUGH _b ", a:e_expr, b:e_expr]
          | asof[" _e ", "now", NULL]
            {  e:epsilon; skip(); }
;
when      = when[" WHEN _tp ", tp:tpred]
          | when[" _e ", overlapt(i,"now")]
            {  e:epsilon; i:i_expr;
               gen_expr(rel_list,"overlapt","left",i);
            }
;
where     = noop[" WHERE _x ", x:predicate]
          | epsilon
;
```

(b) model

Table 4.25: TQuel RETRIEVE Statement

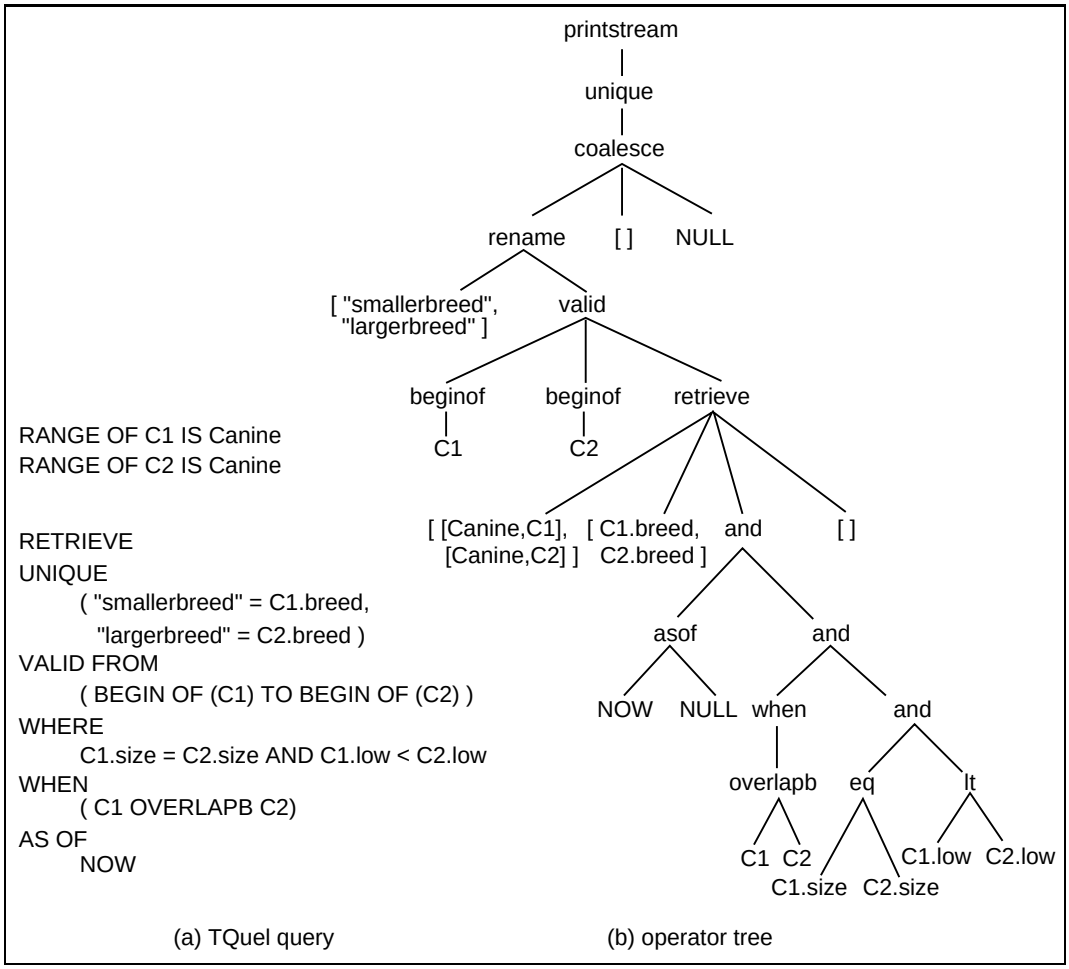

Figure 4.24: Operator Tree: TQuel RETRIEVE Statement

In the *retstmt* subcatalog, if the `INTO` keyword is detected, it is clear that a temporary relation is to be created and populated with the tuples of a derived relation, so the *create()* backplane function could be composed into the operator tree. In our example, the derived relation should be printed, and the *printstream()* backplane function is composed as the root of the operator tree.

Parsing our example continues with the *display* subcatalog, where the `UNIQUE` keyword is recognized in the first signature. Two backplane functions are composed as a result, *unique()* and *coalesce()*. The *unique()* backplane function eliminates duplicates while the *coalesce()* backplane function merges tuples having identical non-temporal attribute values *and* adjacent or overlapping valid-time timestamps.

The input stream for the *coalesce()* backplane function comes from the *rename* subcatalog, which composes the *rename()* backplane function into the operator tree. Its first parameter, the assignment list, is recognized in the *rettuple* subcatalog and processed to separate the arithmetic computation from the aggregation in the *process1* and *proc_cycl1* subcatalogs. The actual stream of tuples comes from the *valid1* subcatalog.

In the *valid1* subcatalog, the VALID clause, if present, is detected. As valid-time timestamps may be either interval or event, the VALID clause has variants which compute each: "VALID FROM <*event*> THROUGH <*event*>" and "VALID AT <*event*>," respectively. The VALID FROM clause in our example is computed by composing the *valid()* backplane function into the operator tree.

Had the optional VALID clause been omitted, the default valid time would have been computed as the interval which is the temporal intersection (overlap) of the valid times of all the underlying tuples:

$(t_0$ `overlap` t_1 `overlap` $\ldots t_k)$

where the t_i represent the relations that are involved in the query. When it is needed, the default temporal expression is generated by a call to the *gen_expr()* directive function[12] in the post-action.

The operator tree evaluating the input stream for the *valid()* backplane function comes from the *retstrm1* subcatalog, where the *retrieve()* backplane function is composed. Some parameters of the *retrieve()* backplane function are context variables, but the condition comes from the *retcond* subcatalog.

Three clauses of the RETRIEVE statement remain to be modeled: the WHERE, WHEN, and AS OF clauses. As all of these represent selection on the retrieved tuples, their conditions are *AND*'ed together in the *retcond* subcatalog and passed as one integrated condition to the *retrieve()* backplane function.

[12] see Appendix A for a description of the *gen_expr()* directive function.

The WHEN clause, modeled in the *when* subcatalog, specifies selection based on a temporal condition. The TQuel RETRIEVE statement eliminates all tuples which do not satisfy the predicates of the WHERE clause *and* the WHEN clause. The *when()* backplane function was added to test temporal predicates. The default WHEN predicate is

`(`t_0 `overlap` t_1 `overlap ...`t_k`) overlap now`

where the t_i represent the relations that are involved in the query. The default predicate stipulates that the underlying tuples must share an interval of validity, and that that interval must include the present, i.e., that all underlying tuples are current. Note that this is consistent with the VALID default, which computes the valid time for the derived tuple as the interval when *all* the underlying tuples are valid. Once again, the *gen_expr()* directive function is used to generate the default temporal expression.

Finally, the AS OF clause, modeled in the *asof* subcatalog, specifies a moment or interval in time to which the database is rolled back; no tuples added later are considered in evaluating the query. With the optional keyword THROUGH,

`AS OF <event expression> [ THROUGH <event expression> ]`

the rollback is further refined; the two events specify an *interval* $<e_0, e_1>$ which brackets the tuples participating in the query, eliminating from consideration information not known in that interval. We added a boolean *asof()* backplane function which tests that a tuple is valid in the proper interval. The AS OF default, `AS OF now`, requires that all the underlying tuples be current in the database.

The APPEND TO Statement. The TQuel APPEND TO statement (see Table 4.26 for the Rosetta model) inserts new tuples into a relation. Tuples may be either literal or derived.

Figure 4.25 shows an example of the TQuel APPEND TO statement. Parsing the APPEND TO statement begins in the *TQuel* subcatalog, where the `APPEND TO` clause is recognized and the *tminsert()* backplane function is composed as the root of the operator tree. This function has temporal functionality not defined for the standard *insert()* backplane function: the *insert()* backplane function simply adds to the database the new tuples from its input stream but the *tminsert()* backplane function must check for existence of any *portion* of the tuple to be inserted and add only the new interval of the new tuple to the database.

The input stream is derived from the *insertstrm* subcatalog, where the *rename()* backplane function is composed but no input is consumed. Parsing passes on to the *valid2* subcatalog.

```
APPEND TO
    <relation>
    <list of values>
[VALID
    <valid expression>]
[WHERE
    <predicate>]
[WHEN
    <when expression>]
```

(a) syntax

```
BEGIN CONTEXT X
    urel_list : unit_rel_list;
    rel_list : relation_list;
    all_attr : attr_list;
    proj_list : attr_list;
    ...
END CONTEXT X

TQuel     =     ...
          |  { newCNTXT(X); }
             tminsert[" APPEND TO _urel_list _y ",
                urel_list, NULL, y:insertstrm]
             {  a:attr_list;
                get_all_attr(urel_list, all_attr);
             }
;
insertstrm =  rename[" ( _j ", proj_list, j:valid2]
;
valid2    =  valid[" VALID AT _e _r ", e:e_expr, NULL, r:process2]
          |  valid[" VALID FROM _i TO _k _r ",
                i:e_expr, k:e_expr, r:process2]
          |  valid[" _r ", "now", "forever", r:process2]
;
process2  =  noop[" _rtt ) _c ", c:proc_cycl2]
;
proc_cycl2 = compute[rewrite(xpr,[add,sub,div,mul,uminus,str2int,
                str2real,str2str],x),x:expr_list,proc_cycl2]
          |  aggregate[rewrite(xpr,[count,min,max,avg,sum,filter,
                groupby,gt,ge,lt,le,eq,str2int],x),x:expr_list,proc_cycl2]
          |  noop[is_attr_list(xpr), retstrm2]
;
retstrm2  =  retrieve[" _w ", rel_list, all_attr, w:modcond, NULL]
          |  epsilon
;
modcond   =  and[" _w _n ", asof("now",NULL),
             and(n:modwhen,w:where)]
;
where     =  noop[" WHERE _x ", x:predicate]
          |  epsilon
;
modwhen   =  when[" WHEN _tp ", tp:tpred]
          |  when[" _e ", overlapt(i,"now")]
             {  e:epsilon; i:i_expr;
                mergerelation(urel_list,rel_list);
                gen_expr(rel_list,"overlapt","left",i);
             }
;
```

(b) model

Table 4.26: TQuel APPEND TO Statement

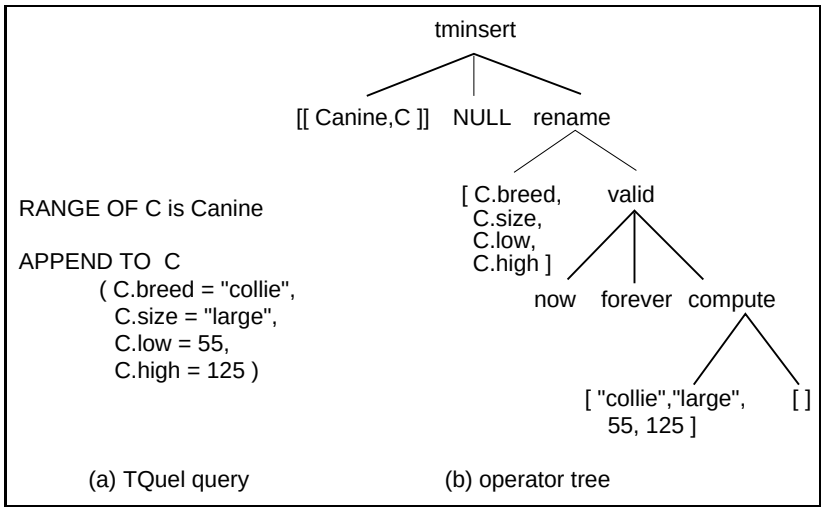

Figure 4.25: Operator Tree: TQuel APPEND TO Statement

Like the RETRIEVE statement, the APPEND TO statement is augmented with a VALID clause which computes the valid time for the derived tuples. The *valid()* backplane function is composed even though no `VALID` clause is specified in the example. The timestamp assigned is the default valid-time timestamp, `now` to `forever`. The input stream for the *valid()* backplane function comes from the *process2* and *proc_cycl2* subcatalogs, where the *compute()* backplane function is composed.

Finally, parsing terminates in the *retstrm2* subcatalog. As no attributes are referenced, the *retrieve()* backplane function is not composed.

The DELETE Statement. The TQuel DELETE statement (the syntax and Rosetta model are in Table 4.27) deletes tuples from relations. However, deletion in a temporal database has different semantics than deletion in a conventional database. In a conventional database, deleted tuples cease to exist; in a temporal database, deleted tuples are marked invalid so that although the information they represent is no longer valid, it is still available for answering historical queries.

In Figure 4.26, we show an example of a TQuel DELETE statement. Mapping the DELETE statement to an operator tree begins in the *TQuel* subcatalog, where the `DELETE` keyword is recognized and the *tmdelete()* backplane function is composed as the root of the operator tree.

The backplane function *tmdelete()* implements tuple deletion in temporal databases, with temporal functionality not available in the standard *delete()* backplane function. First, the *tmdelete()* backplane function sets the transaction stop

```
DELETE
    <relation>
[VALID
    <valid expression> ]
[WHERE
    <predicate> ]
[WHEN
    <when expression >]
```

```
BEGIN CONTEXT X
    all_attr : attr_list;
    rel_list : relation_list;
    urel_list : unit_rel_list;
END CONTEXT X

TQuel    = { newCNTXT(X); }
           tmdelete[" DELETE _urel_list _x ", urel_list, x:valid3]
           {  a:attr_list;
              get_all_attr(urel_list, all_attr);
           }
         |    ...
;
valid3   = valid[" VALID AT _e _d ", e:e_expr, NULL, d:delstrm]
         | valid[" VALID FROM _i TO _k _d ",
              i:e_expr,k:e_expr,d:delstrm]
         | valid[" _r ", "now", endof(urel_list), r:delstrm]
;
delstrm  = retrieve[" _y ", rel_list, all_attr, y:modcond, NULL]
;
modcond  = and[" _w _n ", asof("now",NULL),
           and(n:modwhen,w:where)]
;
where    = noop[" WHERE _x ", x:predicate]
         | epsilon
;
modwhen  = when[" WHEN _tp ", tp:tpred]
         | when[" _e ", overlapt(i,"now")]
           {  e:epsilon;
              i:i_expr;
              mergerelation(urel_list,rel_list);
              gen_expr(rel_list, "overlapt", "left", i);
           }
;
tpred    = precede[" _a PRECEDE _b ", a:evnt_intrvl, b:evnt_intrvl]
         | overlapb[" _a OVERLAPB _b ",a:evnt_intrvl,b:evnt_intrvl]
         | sametime[" _a EQUAL _b ", a:evnt_intrvl, b:evnt_intrvl]
         | and[" _x AND _y ", x:tpred, y:tpred]
         | or[" _x OR _y ", x:tpred, y:tpred]
         | not[" NOT _x ", x:tpred]
         | noop[" ( _x ) ", x:tpred]
;
```

(a) syntax (b) model

Table 4.27: TQuel DELETE Statement

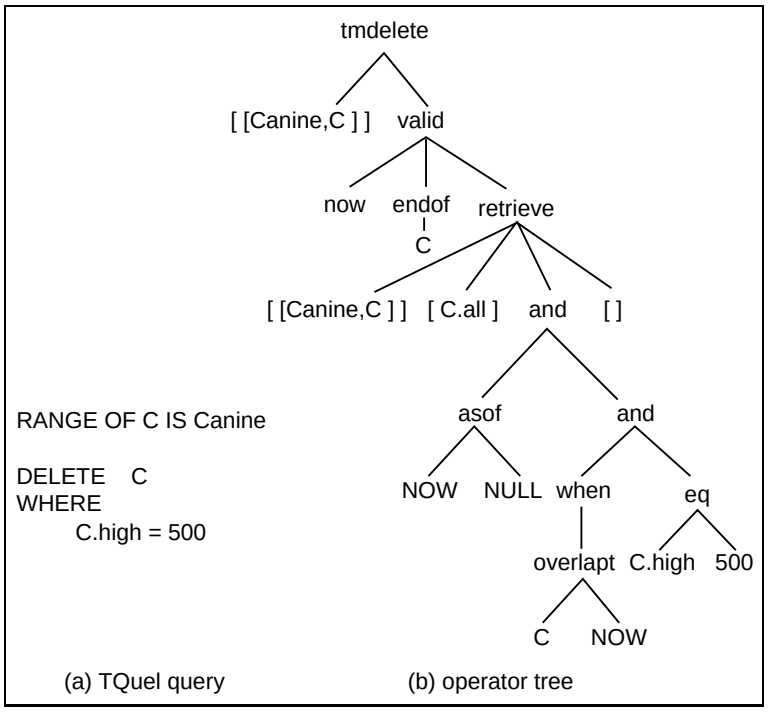

Figure 4.26: Operator Tree: TQuel DELETE Statement

time on these tuples to the current transaction time, which marks them invalid[13]. Then a new tuple with identical values, but with a *known* valid interval and transaction interval `now` to ∞, is added. Finally, if the valid time of the selected tuple overlaps the interval to be deleted, some of it must be "added back" so a compensating tuple (or possibly two) is created and inserted into the database.

Generation of the operator tree continues with the *valid3* subcatalog. In the `DELETE` statement, the `VALID` clause specifies an event or interval during which the tuple values are to be deleted. As the `VALID` clause is missing, the default valid-time timestamp is used and parsing continues with the *delstrm* subcatalog.

The *delstrm* and *modcond* subcatalogs combine to compose the *retrieve()* backplane function into the operator tree. The WHEN clause specifies a temporal predicate which must be satisfied by the tuples to be deleted; if it is not specified, the default temporal expression is used. To select only current tuples for deletion, the AS OF clause is not specified and defaults to "AS OF now." Thus, the tuples targeted for deletion are those which satisfy the WHEN and WHERE predicates

[13] Tuples in a temporal database are never directly modified or deleted. Setting the transaction stop time effectively deletes a tuple as that marks it as containing information which is no longer valid. Only a tuple with a transaction stop time value of ∞ is current.

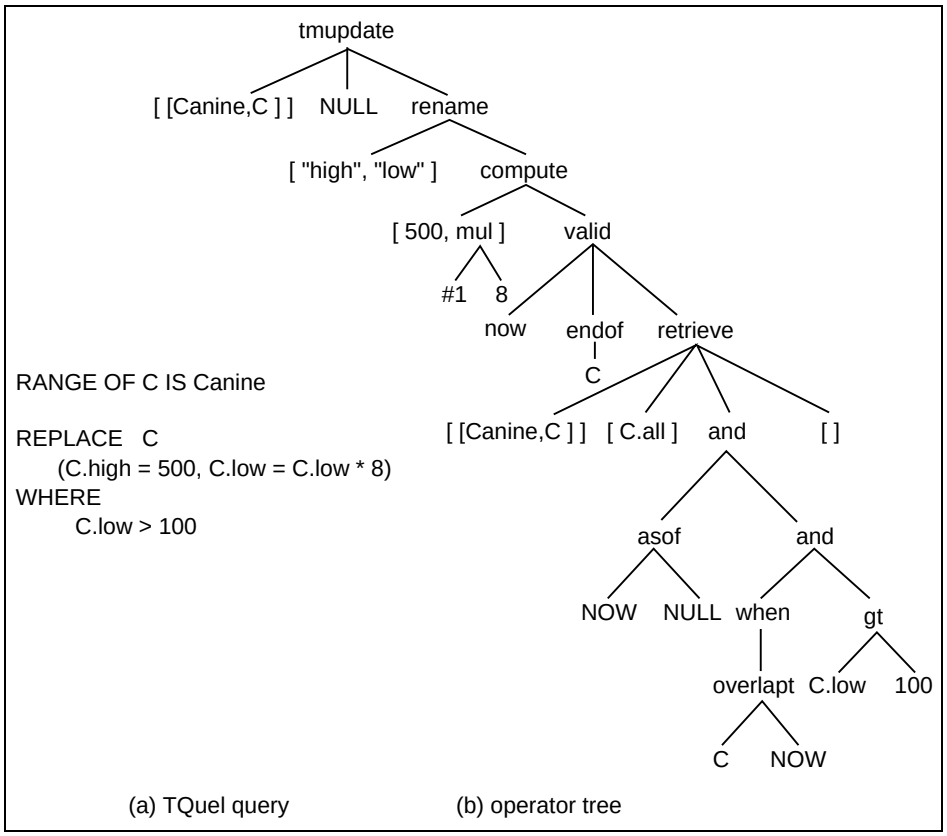

Figure 4.27: Operator Tree: TQuel REPLACE Statement

and are current during the interval specified by the VALID clause.

The REPLACE Statement. The TQuel REPLACE statement (see Table 4.28 for its syntax and model) updates tuple values. Like the DELETE and APPEND TO statements, extending the Quel REPLACE statement to support the temporal model adds complexity to the statement. Both the WHEN and VALID clauses were added to the REPLACE statement; just as for the other tuple modification statements, the AS OF clause defaults to `AS OF now`.

A sample TQuel REPLACE statement is shown in Figure 4.27. Generating an operator tree for it begins in the *TQuel* subcatalog, where the REPLACE keyword is recognized and triggers the composition of the *tmupdate()* backplane function as the root of the operator tree.

The *tmupdate()* backplane function handles modification of temporal relations. As for other modifications, the computed valid-time of the derived tuple might span only part of the interval of an actual tuple in the database. In this case,

```
REPLACE
    <relation>
    <update expressions>
[ VALID
    <valid expression> ]
[ WHERE
    <predicate> ]
[ WHEN
    <when expression> ]
```

(a) syntax

```
BEGIN CONTEXT X
    urel_list : unit_rel_list;
    all_attr : attr_list;
    rel_list : relation_list;
    xpr : sel_list2;
END CONTEXT X
TQuel      = { newCNTXT(X); }
             tmupdate[" REPLACE _urel_list _y ",urel_list,
                 NULL,y:updatestrm]
             {  a:attr_list;
                get_all_attr(urel_list, all_attr);
             }
;
updatestrm= rename[" ( _a _p ", a:assgnlst, p:process3]
;
process3   = noop[" ) _p ", p:proc_cycl3]
;
proc_cycl3 = compute[rewrite(xpr,[add,sub,div,mul,uminus,
                 str2int,str2str,str2real], x),x:expr_list,proc_cycl3]
           | aggregate[rewrite(xpr,[count,min,max,avg,sum,filter,
                 groupby,gt,ge,lt,le,eq], x),x:expr_list,proc_cycl3]
           | noop[is_attr_list(xpr), valid4]
;
valid4     = valid[" VALID AT _e _r ", e:e_expr, NULL, r:retstrm2]
           | valid[" VALID FROM _i TO _k _r ", i:e_expr,
                 k:e_expr,r:retstrm2]
           | valid[" _r ", "now", endof(urel_list), r:retstrm2]
;
retstrm2   = retrieve[" _w ", rel_list, all_attr, w:modcond, NULL]
           | epsilon
;
modcond    = and[" _w _n ",asof("now",NULL),and(w:modwhen,n:where)]
;
where      = noop[" WHERE _x ", x:predicate]
           | epsilon
;
modwhen    = when[" WHEN _tp ", tp:tpred]
           | when[" _e ", overlapt(i,"now")]
             {  e:epsilon; i:i_expr;
                mergerelation(urel_list,rel_list);
                gen_expr(rel_list, "overlapt", "left", i);
             }
;
tpred      = precede[" _a PRECEDE _b ", a:evnt_intrvl, b:evnt_intrvl]
           | overlapb[" _a OVERLAPB _b ",a:evnt_intrvl,b:evnt_intrvl]
           | sametime[" _a EQUAL _b ", a:evnt_intrvl, b:evnt_intrvl]
           | and[" _x AND _y ", x:tpred, y:tpred]
           | or[" _x OR _y ", x:tpred, y:tpred]
           | not[" NOT _x ", x:tpred]
           | noop[" ( _x ) ", x:tpred]
;
```

(b) model

Table 4.28: TQuel REPLACE Statement

only during the spanned part of the original tuple should the new values replace the old, and the original tuple must be split.

The stream of derived tuples with their new values comes from the *updatestrm* subcatalog, where the assignment list is recognized and the *rename()* backplane function is composed into the operator tree. The list of assignments are processed in the *process3* and *proc_cycl3* subcatalogs, which separate the arithmetic computations from aggregations. In the process, the *compute()* backplane function is composed into the operator tree.

Processing the statement continues with the *valid4* subcatalog, where the VALID clause is recognized. As usual, the VALID clause computes a valid time (interval or event) for the derived tuples. In the case of the REPLACE statement, the valid time computed is the interval (or event) during which the new values should replace the old. In our example, there is no VALID clause, so the default temporal expression is used.

The input stream for the *valid()* backplane function comes from the *retstrm2* subcatalog, where the *retrieve()* backplane function is composed into the operator tree. The predicate for the *retrieve()* backplane function is generated by the combination of the *modcond*, *when*, and *where* subcatalogs. The `WHERE` and `WHEN` clauses specify predicates which must be satisfied by the tuples to be updated. The `WHERE` and `WHEN` predicates and the default `AS OF` predicate are *and*'ed together in the *modcond* subcatalog and supplied to the *retrieve()* backplane function as the predicate.

4.3.2.3 Conclusions for TQuel

Finally, we consider the model size and development time for TQuel. Our model of TQuel DML statements is based on the language definition in [Sno87]. All of the TQuel DML statements are included in the model: the RETRIEVE statement for data retrieval, with all its options, as well as the data modification statements APPEND TO (insert), DELETE, and REPLACE (update). The primitive types are limited to varying length character strings; integers and reals of unspecified precision; and of course temporal primitives.

There are some shortcomings in our TQuel model. As Rosetta does not have a module for checking semantic constraints, we cannot discriminate between occurrences of an overloaded operator. In particular, to handle ambiguities in the language definition from an overloaded operator (OVERLAP returns either boolean or interval depending not on the types of its parameters but on the context), we had to slightly modify the operator (OVERLAPB for boolean vs. OVERLAPI for interval) so that its function could be distinguished from context.

Our specification of TQuel is compact. The TQuel specification file consists of approximately 250 lines. From the specification, Rosetta generates a bison file and a flex file comprising approximately 125 lines and 1300 lines, respectively, yielding an expansion factor approximately 1:6.

As an indication of the magnitude of the changes to the Quel specification file we consider specification reuse. The Quel specification file is composed of approximately 115 signatures. Approximately 44% of the Quel signatures were included either unchanged or with only minor modifications in the TQuel model. Thus, even though TQuel is a superset of Quel, less than half of the Quel specification was reusable.

Modeling TQuel required generalizing six existing backplane functions and adding eighteen new backplane functions. This number reflects the numerous additional temporal functions added to TQuel.

Finally, the time required to develop our model of TQuel was approximately one and one half weeks, including both the time needed to understand the syntactic and semantic differences between Quel and TQuel and to develop the model.

4.4 Comparison of Language Models

Numbers for the language models are summarized in Table 4.29. The first row of the table shows the range of number of lines in our language models, while the second row shows the sizes of the generated flex and bison files. The ratio of the two, the expansion ratio, illustrates the difference between using Rosetta and writing the flex and bison files by hand. All the expansion ratios are in the neighborhood of 1:6. Typically, using Rosetta saves writing about 1200 lines of flex and bison code. Considering that Rosetta comprises about 6000 lines of C, this is a nice, but not particularly impressive, gain.

However, there are considerations besides the flex and bison file expansion ratios: development time; reusability of language specifications; reusability of backplane and directive functions and compiler utilities; clarity; and maintainability.

Our development times for languages range 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.

Specification reuse was measured in terms of signature reuse. We counted only those signatures which were reused unchanged or with only minor modifications[14]. Percentages refer to the fraction of the signatures of the parent language which were reused.

[14]Minor modifications include variable and subcatalog name changes and keyword changes.

	SQL	SQL/NF	TSQL2	Quel	TQuel
Specification Size	240	270	350	275	240
flex/bison input files	1400	1440	2530	1500	1425
Expansion Ratio	1:7	1:6	1:7	1:6	1:6
Development Time	NA	3 wk.	3 wk.	4 wk.	1.5 wk.
Specification Reuse	NA	60%	87%	27%	44%
New Functions	NA	7	16	7	18
Generalized Functions	NA	7	7	1	6

Table 4.29: Summary of Modeled Languages

Specification reuse ranged from a high of 87% for TSQL2 to a low of 27% for Quel. We achieved a very high reusability rate for TSQL2 because it was designed to be upward compatible with SQL, its parent language. The low specification reusability rate for Quel can be explained by noting that our starting point for the Quel model was the SQL model. The two languages are quite different, so achieving even a 27% reuse rate is gratifying. Most of the signature reuse in modeling Quel comes from the primitives shared by the two languages.

Another consideration is that of backplane reusability. Our goal is to design the backplane functions to be as general and flexible as possible, so that additional languages can be modeled with few or no changes to the backplane.

The graph in Figure 4.28 shows the cumulative lines of code written during the specification of the five languages we modeled[15]. 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 lines of code. 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 point of writing Rosetta is that one can write it once and then derive the benefit of using it to generate multiple compilers, writing only a simple specification

[15]Overhead consists of lines of code for building Rosetta.

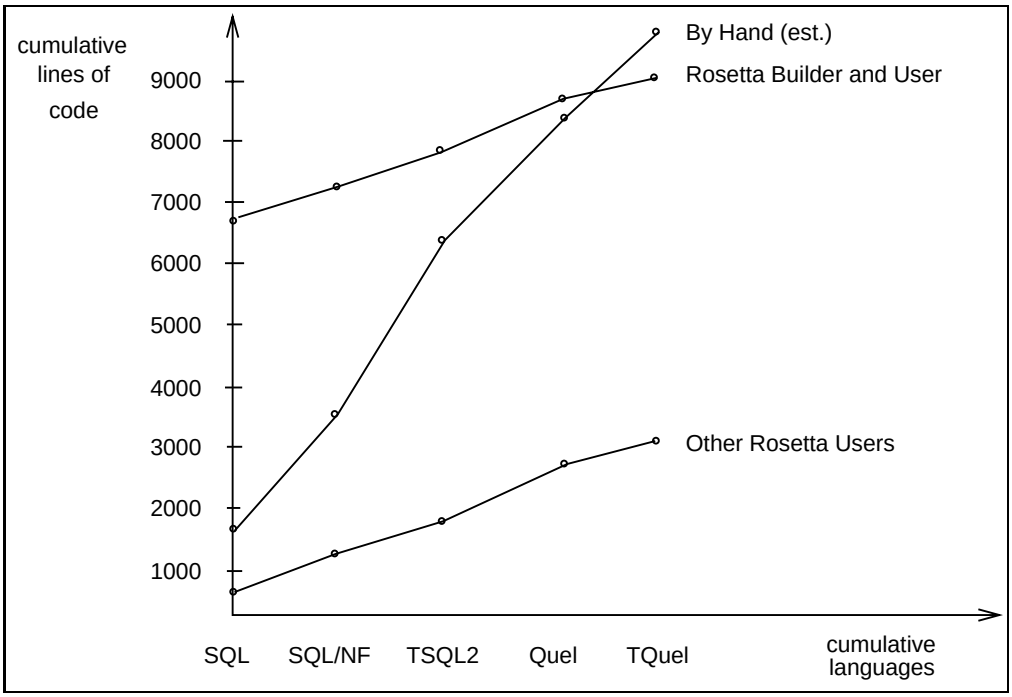

Figure 4.28: Compiler Generation: Cumulative Lines of Code

for each language.

Looking across the graph, it can be seen that the development overhead for Rosetta becomes more and more cost-effective (in terms of lines of model specification) with each additional language generated, until the fifth language, when it becomes competitive.

Finally, the last row of the graph shows the cumulative lines of code that a user who was *given* Rosetta would have had to write, i.e., the cumulative count of lines in the specifications.

In addition, there is the advantage of code reuse. Every backplane function and directive function that is used in the SQL specification is available for reuse in the specification of any other language. Furthermore, there is a core body of infrastructure compiler code which is included in every compiler generated. These files, written in C, include the main function, the backplane function table, symbol tables, etc., and total approximately 4400 lines. Because we are concerned only with declarative languages at this time, we can operate with only a minimal symbol table.

Not only is the C code reusable; the specification itself is reusable. For example, specifications of languages which extend SQL might well include parts of

the SQL specification. New subcatalogs can be added directly to the SQL model, or subcatalogs can be borrowed from the SQL model and incorporated into models for other languages.

4.5 Recap

This chapter has shown the diversity of languages and language families that can be modeled using Rosetta. We have modeled traditional data languages as well as languages that significantly extend these languages. We have also modeled a very recent language (TSQL2), showing that Rosetta is general enough to capture contemporary ideas in data languages. Other approaches, discussed in the next chapter, have not done this with the degree of automation and software reuse that we have achieved.

Chapter 5

Related Work

As Rosetta is a compiler generator for data languages, there is plenty of related work in the areas of database and programming languages, and even some related AI/Natural Language Processing (NLP) work. There are several ongoing extensible DBMS projects which have implemented or are implementing extensible data languages; none of these are generators. Compiler generators are a live area of programming languages research, but no projects have the emphasis that we do: declarative languages and code reuse. Finally, some NLP work used a similar approach in the generation of NL interfaces.

5.1 Extensible DBMS Projects

There are two basic approaches to extensibility in data languages. The more popular (but more restrictive) choice is to build a particular data language into the DBMS and then provide a well-defined interface through which extensions can be made. A second approach is to supply a toolkit which can be used to implement a broad class of data languages. Rosetta is an example of the latter approach.

At a minimum, a data language should be extensible with new data types and new operations on data items, e.g., the addition of a polygon data type and the area operation on it. A more powerful extensible data language allows the addition of new operations on relations, e.g., transitive closure or new aggregation functions (such as standard deviation). Rosetta is capable of generating both kinds of extensions.

5.1.1 Fixed Data Language Approach

Most extensible DBMSs have a fixed language to which extensions can be added. The usual approach to adding extensibility to data languages is to make the parser and optimizer table-driven. Of the extensible DBMSs of which we are aware, all of those which have a fixed data language use this approach.

If extensions can be added dynamically, the data language will also include a protocol for adding extensions. New data types are implemented by first defining the data type (usually by a type definition in the implementation language) and then creating procedures to manipulate the object, e.g., operations to store, load, print, etc. The type name and the names of its supporting procedures are entered in a table by the type manager, where they are available for use by the parser and optimizer. Addition of descriptive and executable operators is similar: first they are implemented, then they are registered with an operator manager module which adds definitional information to a system table.

Gral [Gut89] has its own query language based on an extended relational algebra—a many-sorted algebra—which is basically relational algebra extended with types. Gral differentiates between declarative conceptual operations and *concrete* operations. Extensions may be made to both. All extensions supported, including the addition of new data types and operations, the addition of information for optimization of new operations, and the addition of new relation representations or index structures to support the new type, are made using basically the same mechanism of dynamic registration with the DBMS.

Postgres [SK91] also has a fixed data language, called PostQuel, with dynamic definition of functions and data types, including new base types. Postgres closely follows the general table-driven approach described earlier (and, indeed, popularized it).

In Probe [MD90], the general approach as outlined above is used. To add a new class, the DBI produces an *adapter*, a function which implements the class in terms of lower-level DBMS primitives. Since the data model is object-oriented while the underlying DBMS is relational, when a new class is added the adapter must reconcile the two views.

Starburst [HFLP89], was designed with the goal of total extensibility. Its query language, Hydrogen, is a generalized and extensible variant of SQL. New functions can be added to it, including new aggregate and relational functions. Hydrogen queries are rewritten into an internal form, and the customizer is able to write code to translate language extensions into this internal form and to supply heuristics for optimizing extensions.

5.1.2 The Toolkit Approach

In the toolkit approach, a collection of tools for the generation of a DBMS is supplied. We are aware of only two DBMS generators—Exodus [CDV87] and Genesis [BBG+90, Bat88, BLW88].

The Exodus toolkit includes components which are a mix of generic solutions, generators, libraries, and tools. There is no generator for query languages; instead, a programming language E is supplied for use in the implementation of query languages. E is an extension of C++ augmented with generic classes, iterators, and support for persistent object types. The E compiler is *part of* the generated DBMS. Queries are first translated to E programs, then compiled and executed.

The query language can also be dynamically augmented with extensions written in E using the usual protocol of registration with the DBMS. Exodus supplies a Dependency Manager for managing dynamic extensions.

Genesis, the other DBMS generator, uses the same building-blocks approach to extensibility that Rosetta does. There are facilities for generating other DBMS modules, but no generator for data languages is supplied. However, the interface for the data language is standardized, and hand-written language modules can be plugged in to replace the supplied language modules.

5.1.3 Summary

While all of these projects offer extensibility in the data language, none of them offers the flexibility of a compiler generator. All but Exodus are limited to a single data language. Exodus does have the flexibility of choice of data language, but doesn't offer much besides E for building it.

One attractive feature of these DBMSs is dynamic extension. Rosetta does not offer dynamic extension; languages are extended by generating a new compiler from extended specifications.

5.2 Other Tool Generators

The Genoa generator of code analyzer tools [Dev92, DRW94] takes a similar approach to Rosetta. Genoa is a language independent generator for code analyzer tools. It uses an independent lexical analyzer and parser (produced, for example, by lex and yacc) to produce a parse tree for a program of the language. After the parse tree is converted into an internal form, it is traversed by the generated analyzers during analysis of the program.

Genoa has two major components. The first is a component called Genii which takes as input a syntax specification for the target language and a description of the nodes of the syntax tree. This node description includes code to access various fields of the node. Genii produces two results: a translator which maps the parse trees into the Genoa internal form, and an *Abstract Syntax Dictionary*[1]. The second major component is the generator Genoa, which produces an analyzer from a specification of the analyzer.

An analyzer is generated to answer a specific query. The specification of an analyzer contains commands (in a language peculiar to Genoa) to traverse a parse tree (in Genoa internal form) to collect data to answer its query. Given a transformation of a program to the Genoa internal parse tree form, the analyzer carries out the traversal commands in its specification and produces an answer.

Similar approaches are taken by Genoa and Rosetta. The Genii component produces from a description of the target language and the input parse tree nodes a specification of the internal form of the parse tree which corresponds to a Rosetta language specification. Genii produces node descriptions which combine node tag values and code fragments; the Rosetta signatures combine syntax and semantics.

From an input specification, Genoa produces a tool generator which corresponds to a Rosetta-generated compiler. The input to the tool generator is a tool specification which addresses a specific question; this corresponds to a query in the data language. Finally, the produced tool corresponds to an operator tree.

Despite these similarities, the two are fundamentally different: Genoa is not syntax-oriented but value-oriented and thus cannot produce a data language compiler. More importantly, Genoa lacks extensibility in the operations that can be applied in its analyzer specifications.

5.3 Extensible Programming Languages

Extensibility in programming languages can be realized in several ways. Often, extensible programming languages utilize a preprocessor which translates a program written with extensions into a program utilizing only the built-in constructs of the language. Another possibility is augmenting the compiler with new functions implementing extensions to the language.

[1] The Abstract Syntax Dictionary is used by Genoa to validate an analyzer specification, e.g., to determine that accessed fields of the parse tree actually exist.

5.3.1 Preprocessing

Before compilation, a preprocessor translates the extensions into language constructs recognized by the compiler. The input to the preprocessor is a program written in the *extended* form of the language; its output is a program in the standard form of the language. The output of the preprocessor is then translated and executed just as any other program written originally in the base language. Many languages have preprocessors, e.g., C, Fortran, PL/1, and there are preprocessors which can be used for multiple languages, e.g., M4. Embedding the data language Quel in C, called EQuel [SWKH76], was achieved by preprocessing the EQuel program, converting it into a C program.

One quite general preprocessor is TXL [CHHP91], a project at Queens University at Kingston. TXL can be used to extend any imperative language for which a compiler and a complete syntactic description of the base language are available. A bipartite description is supplied for each extension, consisting of a context-free specification of the extension and rules specifying semantic transformations which map it to the base language. TXL decouples extension syntax from semantic transformation; the two are independently specified and independently applied.

TXL parses the extension syntax and applies semantic transformations in separate passes. First, it parses an input program in the extended language and builds a parse tree, then applies semantic transformations to the parse tree, and finally de-parses the parse tree, generating code in the base language. Like other preprocessors, TXL maps extensions into the base language; however, it also allows extensions to reference library functions which augment functionality of the base language.

5.3.2 Compiler Extension

Another way to make languages extensible is to allow extensions to be added directly to the compiler. This flexibility makes it possible to add significantly new capabilities, for example, adding call-by-reference to a compiler which originally supported only call-by-value. Making such additions also allows the creation of new notation which does not rely on constructs available in the base language, for instance, an infix notation for complex numbers can be added.

Lisp and its variants, e.g., Scheme, CLOS, as well as others, are extensible by virtue of the self-reference capability of Lisp which allows the redefinition of built-in interpreter functions. New functions which can be used just like the built-in ones can be defined.

Scheme provides for adding new syntax with an extend-syntax function which

maps syntactic extensions into core syntactic forms. CLOS (Common Lisp Object System) also allows the user to modify some built-in methods for metaclasses and to define new metaclasses.

This approach to extensibility is also used in Icon [GG86], a descendent of SNOBOL. Functions implementing the extension are written and registered (by editing the appropriate compiler tables), then the entire compiler is re-compiled. Functions implementing extensions are required to follow specified conventions, e.g., they must use standard macros to signal success or failure, and they must use the memory allocation and deallocation functions of the compiler. Because functions are added directly to the compiler, Icon can accommodate many kinds of extensions: new types, new or improved operations on types, generalization of the way it handles parameter passing for function calls, etc.

5.3.3 Summary

From our perspective, a major weakness of extensible languages is that a working compiler must be available before extensions can be made. The point of Rosetta is to *automatically generate* a compiler for a broad class of data languages. Some data languages are extensions to previously defined languages, but others are wholly new.

5.4 Compiler Technology

Not surprisingly, there is a lot of related work from the compiler field, ranging from methods for language specification, such as syntax directed translation, to compiler generators, including early bare-bones generators such as lex and yacc, and later, more mature compiler generators such as Eli.

5.4.1 Syntax Directed Definition.

A syntax-directed translation [ASU86] is a context-free grammar whose symbols are extended with associated attributes. Attributes may be just about anything—strings, numbers, etc. A rule of the grammar has an associated set of *semantic rules* which assign to attributes values computed by function calls. The functions can do virtually anything: generate code, manipulate the symbol table, generate error messages, and so forth.

A Rosetta specification is essentially a syntax-directed translation which is augmented with global variables. Furthermore, Rosetta rules have implicit attributes which constitute the nodes of the automatically constructed parse tree.

5.4.2 Compiler Generators.

Early work in compiler generation immediately brings to mind lex [LS86], which generates a lexical analyzer, and yacc [Joh86], which generates a parser.

A lex input file consists of regular expressions to be matched and associated actions (written in C) to be executed when a matching string is found. The regular expressions match tokens of the language and the actions can specify manipulations to be performed on them. From its input, lex generates a file, `lex.yy.c`, containing C source code for a lexical analyzer.

A yacc input file consists of a context-free grammar with optional semantic actions associated with each alternative of the rules. From its input yacc generates a file (called `y.tab.c`), consisting of C source code for an LALR(1) parser. If there are ambiguities in the grammar, yacc uses its own built-in precedence rules to resolve them, but it also reports the conflicts. The parser requires a separate lexical analyzer (perhaps produced using lex) and a separate error handling routine to be supplied.

Lex and yacc generate programs but supply no actions except for those put in by the user. From a specification, Rosetta generates complete input for *both* lex and yacc, including both patterns and semantic actions and eliminating the tedious enumeration of tokens. One reason for using Rosetta instead of lex and yacc is that a Rosetta specification is more readable than lex and yacc input. But Rosetta is more than a preprocessor for lex and yacc and its framework is more expressive. A data language specification expressed in Rosetta's functional semantic model captures both syntax and semantics. Furthermore, Rosetta automates code reuse and encapsulates reusable components.

Another compiler generator, the Eli compiler-compiler [GHL+92], a project at the University of Colorado, Boulder, consists of an expert system controlling off-the-shelf tools including lex, yacc, LIDO[2], and other tools.

Specifying a compiler to Eli consists of supplying specification files which contain information used for lexical analysis and parsing, semantic analysis, and code generation. Six different files are necessary to convey this information to Eli: one for type analysis, one for lexical analysis and parsing, one describing an attribute grammar used in translation, one for a name analysis module, one for a definition table module, and one for specifying the target language for code generation. Each of these specifications is written using its own special syntax with no uniformity of interface among them.

Our model of languages is substantially different from that of Eli. Rosetta provides a functional language model and maps language constructs to backplane functions, simplifying the code generation/assembly phase. Furthermore, we have

[2]LIDO is an attribute grammar tool.

standardized attributes for nodes, eliminating the use of an attribute grammar for specification.

5.5 AI/NLP

LIFER [Hen77c] is a general package of tools which facilitates the rapid addition of natural language interfaces to existing software systems. It has two major components—a set of interactive functions for language specification and a parser. A language specification is a context free grammar whose rules are augmented with expressions implementing a response. The language specification functions map the specification to an Augmented Transition Network (ATN) which the parser uses to parse input and build expressions in the language of the underlying system. LIFER also provides for other extensions: synonyms, paraphrases, and elliptical inputs.

LIFER was used to build a natural language interface for LADDER [HSSS78], a front-end to a distributed database. LADDER is comprised of three components: INLAND, IDA, and FAM[3]. INLAND translates a natural language query into a set of Lisp-syntax-like constraints; IDA converts the constraints into a query or queries in the underlying language; and FAM requests file accesses at the appropriate site.

INLAND, the module produced with LIFER, cannot handle joins. Nor does it translate input natural language queries to the data language of the underlying DBMS. In [HSSS78], Hendrix, et.al. state that joins are handled by handing off a predicate to the IDA module. While LADDER does have join capability, the IDA splits the predicate into a sequence of *single file queries*, and *it* composes the records obtained to produce the result. Thus, LADDER *does not actually pass a join query* to the underlying DBMS.

Q&A, a commercial product from Symantec, is a commercial version of LIFER technology. Like LIFER, Q&A allows the user to query individual relations via natural language and does not allow the formation of queries involving joins.

The similarity of LIFER to Rosetta is the triggering of an operation when an input text pattern is recognized. A point of divergence is an important part of our research: the definition of a standardized (but open) set of operations for a DBMS backend. A fundamental part of our approach in Rosetta is the use of standardized semantic actions with user-definable syntax.

[3]These are acronyms for Informal Natural Language Access to Navy Data, Intelligent Data Access, and File Access Manager, respectively.

5.6 Summary

In short, while there is plenty of related work, none does everything we want to do. Specifically, we automate reuse in several ways.

First, a Rosetta language specification can be reused when extensions are added. To generate a language variant we can start from a definition of the language, extend it, and then generate a compiler for the extended language.

Alternatively, components of a language can be reused in the design of another language. For example, most data languages have the same base types and operations on them; these can be copied from one language to another.

Furthermore, both the backplane functions and compiler components are reused in the construction of new languages. While extensible languages encourage reuse of the base compiler, there is no structure in place for reuse across languages.

Finally, Rosetta will generate most data languages, not merely extensions to one language, and it is not necessary to have a working compiler in order to build a variant.

Chapter 6

Conclusions

We have developed an approach to model a broad class of data languages and to generate a compiler for a data language from its specification. Components of our approach include a backplane of functions derived from a domain analysis of data languages and a specification language which models both data language syntax and semantics. Versions of a data language can be readily spawned from earlier versions, enabling experimentation with evolving language features.

A prototype has been implemented and demonstrated. Five diverse data languages were modeled and compilers were generated for them. Sufficient backplane functions were implemented to enable evaluation of queries in two of these languages.

6.1 Limitations

However, there are limitations both in the model and in the implementation. Limitations in the model diminish the expressiveness of Rosetta while implementation limitations reduce the effectiveness of the generated compilers.

6.1.1 Model Limitations

Various model limitations surfaced during the course of modeling five data languages. Some were generalized immediately; others were not recognized as genuine limitations until later. The primary deficiencies are documented in the following sections.

6.1.1.1 Run-time Global Memory

One of the more versatile features of the Rosetta specification language is the ability to define and use compile-time global memory, i.e., context variables, to share

information across subcatalogs.

However, context variables are available only at compile-time, and their scope is limited to only a single query. This is insufficient to meet all needs for global memory.

One example of the need for persistent global memory is the definition of correlation variables in Quel and its derivatives. The `RANGE OF` statement defines a correlation $\mathcal{C}$ which is known to all subsequent statements until it is redefined in another `RANGE OF` statement. This was modeled using a suite of directive functions to define and look up correlations. The opportunity to generalize the model was lost by the assumption that global correlations represent an isolated need for persistent compiler global memory.

We encountered this need a second time. The TSQL2 data language specification allows users to specify a default calendar to use in interpreting dates. Like correlations, the default calendar is specified in a TSQL2 statement and persists through subsequent statements until it is redefined.

Having needed persistent compiler global memory modeling each of two language families, we feel confident that it will be needed again in modeling future languages. Rosetta should supply persistent global memory in its generated compilers.

6.1.1.2 Random Clause Recognition Order

Another weakness in the model is that the specified clause recognition order cannot be truly random. This limitation stems from bison limitations, which requires us to specify a linear progression on parsing the input string.

Consider the syntax of an abstract language statement, consisting of a sequence of three clauses "C_2 C_0 C_1" where clause C_i signals the composition of the function f_i into the operator tree. Bison requires that the clauses be processed in left-to-right order: first C_2, then C_0, and finally C_1. Thus, to get the composition order $f_0(f_1(f_2))$, we have to define the monolithic signature $\mathcal{S}_0$:

$\mathcal{S}_0$ = f$_0$[`` C$_2$ C$_0$ C$_1$ '', f$_1$(f$_2$())]

But a cleaner definition would define independent signatures:

$\mathcal{S}_0$ = f$_0$[`` _v$_2$ C$_0$ _v$_1$ '', v$_2$||v$_1$:$\mathcal{S}_1$]
$\mathcal{S}_1$ = f$_1$[`` _v$_2$ C$_1$ '', v$_2$:$\mathcal{S}_2$]
$\mathcal{S}_2$ = f$_2$[`` C$_2$ '']

where the symbol "||" means string concatenation.

There are ways to work around this limitation: one way is to specify a monolithic signature as above. Another is to store parameter values in context variables until their backplane function(s) can be composed. While this is not a serious limitation, it does detract from the clarity of the language specifications.

6.1.1.3 Semantic Constraints Checking

Languages often impose semantic constraints on statements in addition to syntactic ones. Such a constraint is exemplified in the `SORT` statement, with the requirement that all attributes to be sorted on must be included in the input stream. Another example comes from the SQL `SELECT` statement, which requires that if the `GROUP BY` clause is present, then only aggregations or attributes which are grouped on may appear in the select list. It is not possible to specify constraints such as these in the current model of Rosetta.

Adding semantic constraints checking to Rosetta would entail the definition of a formalism to specify semantic requirements for backplane functions and the implementation of a generator which would convert a specification in the formalism into a function for testing semantic constraints.

This same module would also handle the analysis of overloaded operators to determine which capacity they are being used in. For example, the result of the TQuel `OVERLAP` operator depends on the context. Given two intervals, the `OVERLAP` operator returns *either* the interval which is common to both (a temporal intersection), *or* a boolean, `TRUE` if its inputs share a non-NULL common interval and `FALSE` otherwise. Note that the types of the inputs are identical in both cases so that even if a call to `OVERLAP` is type-correct, the type of its result must still be determined from context. Which is the type of the result is determined by the context in which the `OVERLAP` operator appears.

6.1.2 Implementation Limitations

In addition to model limitations, there are implementation limitations. Although all of the Rosetta modeling constructs were satisfactorily implemented, the implementation encompasses more than just the generator, and it provides somewhat inadequate infrastructure for the generated compilers. There are several areas in which the generated compilers should have additional included code.

6.1.2.1 Type Checking

The generated compilers lack included infrastructure for type-checking operator trees. When an operator tree is created, each function's parameters should be

compared against the function definition to determine that the parameters are of the proper type. Type-checking is a lower-level operation than semantic constraints checking; it can be done automatically without any specification whereas checking semantic constraints requires definition of a formalism to specify constraints and a generator to map the formalism to an implementation.

Static type-checking serves only to determine that the generated compiler *may* produce type-correct operator trees, and not that it will produce *only* type-correct operator trees. While static type checking assures that constants are type-correct, it cannot make the same claim for attributes. For example, we can specify using subcatalogs that a particular operation applies only to integer constants, but we cannot specify that the operation applies only to integer attributes. More generally, a composite structure, such as a tuple, may contain an element whose type is incompatible with operations applied higher in the operator tree. This means, of course, that a function in an operator tree may receive actual parameters of improper types undetected.

6.1.2.2 Schema Lookup

Another limitation is that schema lookup was not fully implemented. The generated compilers should include a module which determines if a referenced relation exists in the database, gets its schema, and also generates access information for it. This same module should also test for the existence of attributes in the referenced relations. The ramifications of unimplemented schema lookup are that non-existent relations or attributes may be referenced in an operator tree.

6.1.2.3 Error Detection

Another shortcoming of the generated compilers is the lack of infrastructure for trapping run-time errors. Some of these implementation limitations may potentially crash the generated compiler. Parsing errors are announced gracefully, but generated compilers have no infrastructure to avoid a catastrophic failure during evaluation of an operator tree. For example, failure in a backplane function can cause the compiler to fail. If a backplane function signals a "Floating point exception," the error is not detected and the compiler fails.

An error trapping component should be included with every generated compiler. It should also be possible to specify different errors and to associate with each error the action to be taken if it is detected.

6.1.2.4 Backplane Implementation

Furthermore, it could be quite rewarding to reimplement the interpreter in C. The backplane functions were implemented in Quintus Prolog and interfaced using the Quintus IPC library with the generated compilers, which were written in C. Unquestionably, we took both a performance and portability hit by using Quintus Prolog instead of C for backplane function implementation, both in the slower running time of the functions and in the extra overhead of communication.

However, this was offset by the advantage offered by using Quintus Prolog to implement the prototype backplane functions: since Prolog is a very high-level language with built-in primitive database capabilities, we were able to quickly implement these functions and get the prototype running queries so as to have hands-on experience with our system.

6.2 Extensions

There are several potentially rewarding directions for extending the model which would increase its expressiveness.

As it stands, the expressiveness of the model was sufficient to allow us to model diverse data languages. However, extensions to the model would permit us to model more complex languages, as well as adding new functionality not available in the current generated compilers.

6.2.1 Cycle Subcatalogs

In our experience with Rosetta, limitations on cycle subcatalogs have not been a problem. Nevertheless, we surmise that the expressiveness of the model could be increased by generalizing cycle subcatalogs to allow multiple exit conditions.

The current definition of cycle subcatalogs allows only one non-recursive exit subcatalog. For example, in the cycle subcatalogs separating computation from aggregation (e.g., the *proc_stream* subcatalogs in Tables 4.1, 4.7, and 4.13), the only exit condition is the reduction of the expression list to a list of attributes and constants. A more general definition of cycle subcatalogs would permit multiple exit conditions, so that different actions could be taken depending on the exit condition selected.

6.2.2 Backplane Redefinition and Refinement

It is hard to overestimate the importance of thorough domain analysis in developing the backplane and of scrupulous backplane maintenance. However, what can be

done with the backplane depends on its definition. Our definition did not limit our development of language models, but it does impose limitations on other activities.

In particular, our backplane function definitions are too coarse for query optimization. For example, the *retrieve()* backplane function encapsulates relation access, application of predicates to select tuples and join relations, and projection. Such a monolithic encapsulation of functionality defies optimization.

Another reason to redefine and refine the backplane is to add the ability to model more expressive languages than those we have considered. Currently, the backplane has no constructs which can model either loops or alternation (*if* statements). However, more sophisticated data languages may include these constructs, and a generalized backplane would increase the expressiveness of Rosetta.

Currently, our data language backplane comprises about 95 functions. We implemented about two-thirds of the backplane functions, sufficient to evaluate SQL and Quel queries, both data retrieval and data modification. The rest of the backplane should be implemented so that queries could be evaluated in the other modeled languages as well as languages modeled in the future.

6.2.3 Generalize Compiler Memory

Our generated compilers currently have global compile-time memory (contexts). While it sufficed to model many language features, it was not sufficiently powerful to model all; the need for persistent compiler memory was discussed in Section 6.1.1.1.

But generalizing global compile-time memory would also be useful. Foremost is to make assignment of global variables "instantaneous." Currently, assignment to a global variable is made in the post-action. This means that its value is not immediately available to subcatalogs which are active before the post-action is evaluated. While this is not difficult to work around, it is aesthetically displeasing. Thus, the generated compilers should have a full complement of memory: persistent memory as well as transitory compile-time memory (contexts).

6.2.4 Syntax Representation

Extending Rosetta's capacity for specifying primitives would allow us to distinguish during parsing between syntactically identical but semantically distinct language elements. Allowing such primitives to sometimes (but not always) appear interchangeably in the language leads to ambiguities in the language specification. In order to eliminate these ambiguities, the lexical analyzer should be able to distinguish between them. Since the lexical analyzer is generated, it follows that the language specification must include a description of how they are to be distinguished.

For each primitive element to be distinguished, we would supply a boolean function to differentiate it. Consider an identifier, which may be interpreted as either a relation or an attribute. To distinguish identifiers, functions such as *is_a_relation()* and *is_an_attribute()* would be supplied, returning `TRUE` if the identifier is a relation or an attribute, respectively.

But augmenting the syntax representation in this way is more general than schema lookup. Such a distinction could be made for any class of primitives having subclasses some of which subsume others. For example, suppose that an integer constant is automatically stored as a long integer if its value exceeds a certain threshold. Then one might distinguish long integer constants from small integer constants by testing the value of an integer constant in the lexical analyzer using supplied distinguishing functions.

Finally, automating the detection of identical conversion patterns would enable merging the patterns in the flex file. Generalizing the actions generated for the lexical analyzer to select the token type returned based on the response of the distinguishing functions would enable making this distinction during token recognition.

Another extension to the specification of primitives would add the specification of output functions for primitives. The conversion subcatalogs handle conversion of input strings to primitive types but there is no inverse definition of functions which convert primitives to strings for output. Currently, all primitives are printed out as strings. No additional characters, e.g., units such as '$' or *mg.* can be printed with the value. Furthermore, more complex types need more sophisticated formatting. This has not been a serious limitation as the languages modeled do not have complex data types. Nevertheless, the model should include the specification of output functions for primitives.

6.3 Conclusions

In order to support new applications of databases, new features for data languages are frequently proposed. Typically, these new features are added to existing languages. To evaluate new features thoroughly, researchers need hands-on experience with their proposed languages. However, because it is a time- and resource-intensive task to build a compiler from scratch, few of these proposed languages are ever implemented. This was the motivation for Rosetta.

Our goal was to develop a systematic approach which would speed and simplify the generation of compilers for data languages. We began with definitional work—a domain analysis to develop a backplane of functions which model the do-

main of data languages. A composition of backplane functions specifies the semantics of a data language statement. A data language can express a subset of the set of all compositions of backplane functions. Finally, we defined the Rosetta specification language to explicitly specify which compositions can be expressed by the language and to associate compositions with statements.

Following development of these definitions, the next step was to validate our model. We wanted to discover the limitations on the expressiveness of our approach: what features, if any, can not be expressed by Rosetta. We hoped that the specifications would prove to be compact, because compactness goes hand in hand with readability and maintainability. Another hoped-for characteristic was ease of use of the system. Finally, we needed to evaluate the correctness of the operator trees produced by the generated compilers.

To answer these questions, we needed empirical evidence which could only be obtained by implementing Rosetta and using it. The Rosetta prototype was built, including all specification language features. Models were designed and compilers generated for five different data languages. About two thirds of the backplane functions were implemented; this was sufficient to evaluate SQL and Quel queries in all statements of both languages.

We found that Rosetta simplifies specifying data languages and generating compilers for them by providing high-level, easy-to-use, powerful constructs. The statements of a data language are logically divided into clause-sized specification elements. The Rosetta specification reflects this logical division as each clause specification includes both the syntax and semantics of the clause, clearly delimited. The high-level specification eliminates concern with minutiae, yielding a further advantage: clarity, readability, and (we claim) maintainability.

Furthermore, as discussed in Section 4.5, Rosetta speeds the development of compilers by encouraging reuse of backplane functions, directive functions, and compiler utilities. The savings in time and effort which can be realized by specification and code reuse makes this a potentially viable development approach.

Overall, we feel that Rosetta was successful. The first phase of our work defined a backplane and specification language; the validation phase utilized them to demonstrate that compilers for five diverse data languages could be built using this approach. Thus, the prototype satisfactorily demonstrates proof of concept.

Appendix A

A Database Backplane

This appendix lists the definitions of the backplane and directive functions used in our data language models. Many backplane functions are multiply defined, e.g., the comparison function *equal()* is defined as *equal(x:Int,y:Int):Boolean*, *equal(x:Int, y:Real):Boolean*, etc. Definitions of multiply defined functions are listed together with a single function description.

Currently, thirteen types are known to Rosetta. The first ten are primitive: Attribute, Boolean, Date, Event, Int, Interval, Real, Span, String, and Void. There is one composite type: Relation. The remaining two are built-in generic types, List and Stream. The List[τ] type is a convenience for modeling languages which avoids the tedious repetition of list structure definition, while the Stream type imposes structure on a group of records. The Stream type is a stream of tuples. Stream is a generic type, and the attributes of its constituent tuples are not known until compile-time.

Each function is listed in a fixed format:

function_name (parameter_list) : type
parameter$_0$ description of parameter$_0$
...
parameter$_n$ description of parameter$_n$
function description

where elements of the parameter_list are defined as "$\alpha : \tau$", indicating parameter α is of type τ.

A.1 Backplane Function Definitions

abs (x:Int) : Int
abs (x:Real) : Real
abs (x:Span) : Span
x : numeric or temporal expression
abs() computes and returns the absolute value of its operand.

add (x:Int, y:Int) : Int
add (x:Int, y:Real) : Real
add (x:Real, y:Int) : Real
add (x:Real, y:Real) : Real
add (x:Event, y:Span) : Event
add (x:Span, y:Event) : Event
add (x:Span, y:Span) : Span
x : first addend
y : second addend
add() computes and returns the sum of x and y.

aggregate (a:List, s:Stream) : Stream
a : list of expressions
s : stream of tuples
The first parameter is a list of expressions to evaluate; the expressions are *either* aggregations or simple references to stream attributes. *aggregate()* evaluates each aggregation operation in *a* (e.g. count, max, min, ...) over the entire stream *s*. Other elements of *a* are references to stream attributes and are simply passed on. Note that *s* may be grouped; if so, *aggregate()* evaluates the list of expressions over each group independently.

and (x:Boolean, y:Boolean) : Boolean
x : Boolean
y : Boolean
and() computes the logical AND of its inputs and returns the result.

asof (x:Event, y:Void) : Boolean asof (x:Event, y:Event) : Boolean asof (x:Interval, y:Void) : Boolean
x : event or interval expression
y : event or Void
asof() is applied by *retrieve()* to the current tuple to determine that its transaction-time timestamp is after the specified event or within the specified interval. If the first parameter is an Event and the second parameter is Void, the second parameter

defaults to ``now'', so that the interval becomes $x$ through ``now''.

assign (x:Attribute, y:Int) : Void
assign (x:Attribute, y:Real) : Void
assign (x:Attribute, y:String) : Void
assign (x:Attribute, y:Boolean) : Void

x : attribute name
y : new attribute value

assign() is an operator applied to a stream by a function (such as *compute()*. The attribute named in x is assigned the value of y.

attribute (x:List, y:String) : Attribute

x : unit list of relation
y : attribute name

attribute() verifies that the name y is actually an attribute of relation x.

avg (x:Int, y:Stream) : Int
avg (x:Real, y:Stream) : Real

x : numeric expression
y : stream of tuples

avg() computes the average value of the expression x over the stream of tuples y.

beginof (x:Interval) : Event

x : temporal interval

beginof() returns the start event of the interval x.

boolean2int (x:Boolean) : Int

x : Boolean

boolean2int() converts a boolean value to an integer. `TRUE` is converted to `1`; `FALSE` is converted to `0`.

coalesce (x:Stream, y:List, z:String) : Stream

x : stream of tuples
y : list of attributes
z : string

The *coalesce()* function *coalesces* the stream x on the attributes named in y. The input stream x is grouped on the attributes named in y. In each group, the tuples whose valid-time timestamps can be merged into a single temporal element are merged into one tuple with the composite valid-time timestamp. The parameter z directs how the time-stamps should be merged, as *Interval*s or *Element*s.

compute (e:List, s:Stream) : Stream

e : list of expressions

s : stream of tuples

The first parameter, *e*, is a list of expressions to be evaluated over the second parameter, *s*, a stream of tuples. *compute()* does not handle aggregations; all elements of *e* are either arithmetic expressions or stream attribute references. A new tuple is produced for each of the tuples in the input stream; these new tuples are returned as a Stream.

contains (x:Stream, y:Stream) : Boolean

x : stream of tuples

y : stream of tuples

contains() compares stream *x* with stream *y*; if every tuple of *x* occurs in *y*, *contains()* returns `TRUE`; otherwise, *contains()* returns `FALSE`.

count (x:Int, y:Stream) : Int
count (x:Real, y:Stream) : Real

x : numeric expression

y : stream of tuples

count() counts the number of tuples in the stream of tuples *y*.

create (n:Relation, t:Stream) : Void

n : relation identifier

t : stream of tuples

create() creates a new relation and populates it with the tuples listed in *t*. The new relation is given the name specified in n; the names and types of its attributes are as specified in the header of the stream *t*.

delete (r:List, s:Stream) : Void

r : a unit list of relation

s : stream of tuples

delete() deletes the tuples of *s* from the relation *r*.

difference (x:Stream, y:Stream) : Stream

x : stream of tuples

y : stream of tuples

difference() compares stream *x* with stream *y*; it returns a stream comprised of the tuples of *x* that *do not* occur in *y*.

div (x:Int, y:Int) : Real
div (x:Int, y:Real) : Real
div (x:Real, y:Int) : Real
div (x:Real, y:Real) : Real
div (x:Span, y:Span) : Real

div (x:Span, y:Int) : Span
div (x:Span, y:Real) : Span

x : dividend
y : divisor

div() computes and returns the quotient of x and y.

earliest (x:Event, y:Event) : Event

x : event expression
y : event expression

event() determines which event, x or y, occurred earlier and returns it.

endof (x:Interval) : Event

x : interval

endof() returns the terminating event of the interval x.

eq (x:Int, y:Int) : Boolean
eq (x:Int, y:Real) : Boolean
eq (x:Real, y:Int) : Boolean
eq (x:Real, y:Real) : Boolean
eq (x:String, y:String) : Boolean
eq (x:Attribute, y:Stream) : Stream
eq (x:Int, y:Stream) : Stream
eq (x:Real, y:Stream) : Stream
eq (x:String, y:Stream) : Stream
eq (x:τ, y:Stream) : Stream

x : first item to be compared
y : second item to be compared

eq() tests the equality of x and y, returning `TRUE` if they are equal and `FALSE` if not. In the last definition of *eq()*, the first parameter, x, and the elements of the second parameter, y, must be comparable by some definition of *eq()*. If x is a tuple, then each of its attributes must be comparable by some definition of *eq()* to the corresponding attribute in the tuples of y. The returned Stream is a stream of Boolean, with one value for each input tuple of y.

event (x:Date) : Event

x : a date

event() converts a date into an Event.

exists (s:Stream) : Boolean

s : stream of tuples

exists() tests existential quantification, returning `TRUE` if s is non-empty and `FALSE`

otherwise.

extend (x:Interval, y:Interval) : Interval
extend (x:Event, y:Interval) : Interval
extend (x:Interval, y:Event) : Interval
extend (x:Event, y:Event) : Interval

x : event or interval expression
y : event or interval expression

extend() creates an interval which begins with the earlier of the starting points of its interval parameters x and y and ends with the later of the terminating points of its parameters. If either parameter is an event, it is first coerced into an interval with itself as starting and ending events. If there is no overlap between the two parameters, the returned interval is empty.

filter (x:Boolean, y:Stream) : Stream

x : boolean expression (predicate)
y : stream of tuples

filter() evaluates the predicate x over the tuples of y, returning only those which satisfy x.

forall (s:Stream) : Boolean

s : stream of tuples

forall() tests universal quantification. Each tuple in stream s is a tuple with one Boolean attribute. If the attribute values are all `TRUE`, *forall()* returns `TRUE`; otherwise, *forall()* returns `FALSE`.

forany (s:Stream) : Boolean

s : stream of tuples

forany() tests universal quantification. Each tuple in stream s is a tuple with one Boolean attribute. If any of the attributes is `TRUE`, *forany()* returns `TRUE`; otherwise, *forany()* returns `FALSE`.

grgrndate (x:Int, y:Int, z:Int) : Date

x : integer (month)
y : integer (day)
z : integer (year)

grgrndate() converts its arguments into a Gregorian date.

groupby (a:List, s:Stream) : Stream

a : list of attributes
s : stream of tuples

groupby() takes as input a stream s and separates the tuples into groups having iden-

tical attribute values over the attributes listed in *a*. *groupby()* returns the grouped tuples in a stream augmented with markers delimiting the groups. Note that the attributes listed in *a must* be included in the attributes of *s*.

gt (x:Int, y:Int) : Boolean
gt (x:Int, y:Real) : Boolean
gt (x:Real, y:Int) : Boolean
gt (x:Real, y:Real) : Boolean
gt (x:String, y:String) : Boolean
gt (x:Attribute, y:Stream) : Stream
gt (x:Int, y:Stream) : Stream
gt (x:Real, y:Stream) : Stream
gt (x:String, y:Stream) : Stream
gt (x:τ, y:Stream) : Stream

x : first item to be compared
y : second item to be compared

gt() compares x to y, returning `TRUE` if x greater than y, `FALSE` if not. In the last definition of *gt()*, the first parameter, x, and the elements of the second parameter, y, must be comparable by some definition of *gt()*. If x is a tuple, then each of its attributes must be comparable by some definition of *gt()* to the corresponding attribute in the tuples of y. The returned Stream is a stream of Boolean, with one value for each input.

gteq (x:Int, y:Int) : Boolean
gteq (x:Int, y:Real) : Boolean
gteq (x:Real, y:Int) : Boolean
gteq (x:Real, y:Real) : Boolean
gteq (x:String, y:String) : Boolean
gteq (x:Attribute, y:Stream) : Stream
gteq (x:Int, y:Stream) : Stream
gteq (x:Real, y:Stream) : Stream
gteq (x:String, y:Stream) : Stream
gteq (x:τ, y:Stream) : Stream

x : first item to be compared
y : second item to be compared

gteq() compares x to y, returning `TRUE` if x is greater than or equal to y and `FALSE` if not. In the last definition of *gteq()*, the first parameter, x, and the elements of the second parameter, y, must be comparable by some definition of *gteq()*. If x is a tuple, then each of its attributes must be comparable by some definition of *gteq()* to the corresponding attribute in the tuples of y. The returned Stream is a stream

of Boolean, with one value for each input tuple of y.

having (p:Boolean, s:Stream) : Stream

p : boolean expression
s : stream of tuples

having() evaluates the boolean expression p over each group in the stream s. If p is satisfied, the unmodified stream is returned; otherwise, the empty stream is returned. An input stream with no group markers is treated as one group. Often p includes aggregations; when it does, aggregations are computed separately for each group and not over the stream as a whole. Furthermore, a restriction on an *attribute* must be satisfied by *each* tuple in the group, e.g. if p requires "R.A > 5", then each tuple of the group must have R.A > 5.

insert (r:Relation, l:List, s:Stream) : Void

r : relation identifier
l : list of attribute
s : stream of tuples

insert() adds the tuples of stream s to the named relation r. If necessary, *insert()* rearranges the attributes of the s according to the list of attributes in l.

intersect (x:Stream, y:Stream) : Stream

x : stream of tuples
y : stream of tuples

intersect() compares stream x with stream y; it returns a stream comprised of the tuples that occur in both x and y.

intervalc (x:Event, y:Event) : Interval
invervalc (x:Date, y:Void) : Interval

x : event or date
y : event or `NULL`

intervalc() computes the closed interval having starting and ending events x and y, respectively. If x is a date and y is NULL, then the date is converted to a closed interval having as its starting and ending events the first and last chronons of the date, respectively.

intervalo (x:Event, y:Event) : Interval
invervalo (x:Date, y:Void) : Interval

x : event or date
y : event or `NULL`

intervalo() computes the open interval having starting and ending events x and the last chronon before y, respectively. If x is a date and y is NULL, then the date is converted to a closed interval having as its starting and ending events the first and

last but one chronons of the date, respectively.

iselement (x:Tuple, y:Stream) : Boolean
iselement (x:Int, y:Stream) : Boolean
iselement (x:Real, y:Stream) : Boolean

x : tuple
y : stream of tuples

iselement() determines if tuple x occurs in the stream of tuples y. If the first parameter x is not a tuple, then the tuples of the stream parameter must have only one attribute.

isunique (x:Stream) : Boolean

x : stream of tuples

isunique() returns `TRUE` if stream x contains no duplicates and `FALSE` otherwise.

latest (x:Event, y:Event) : Event

x : event
y : event

latest() returns the later of the events, x or y.

lt (x:Int, y:Int) : Boolean
lt (x:Int, y:Real) : Boolean
lt (x:Real, y:Int) : Boolean
lt (x:Real, y:Real) : Boolean
lt (x:String, y:String) : Boolean
lt (x:τ, y:Stream) : Stream

x : first item to be compared
y : second item to be compared

lt() compares x to y, returning `TRUE` if x is less than y and `FALSE` if not. In the last definition of *lt()*, the first parameter, x, and the elements of the second parameter, y, must be comparable by some definition of *lt()*. If x is a tuple, then each of its attributes must be comparable by some definition of *lt()* to the corresponding attribute in the tuples of y. The returned Stream is a stream of Boolean, with one value for each input tuple of y.

lteq (x:Int, y:Int) : Boolean
lteq (x:Int, y:Real) : Boolean
lteq (x:Real, y:Int) : Boolean
lteq (x:Real, y:Real) : Boolean
lteq (x:String, y:String) : Boolean
lteq (x:τ, y:Stream) : Stream

x : first item to be compared
y : second item to be compared

lteq() compares x to y, returning `TRUE` if x is less than or equal to y and `FALSE` if not. In the last definition of *lteq()*, the first parameter, x, and the elements of the second parameter, y, must be comparable by some definition of *lteq()*. If x is a tuple, then each of its attributes must be comparable by some definition of *lteq()* to the corresponding attribute in the tuples of y. The returned Stream is a stream of Boolean, with one value for each input tuple of y.

max (x:Int, y:Stream) : Int
max (x:Real, y:Stream) : Real
max (x:String, y:Stream) : String

x : numeric expression
y : stream of tuples

max() determines the maximum value of the expression x over the stream of tuples y.

min (x:Int, y:Stream) : Int
min (x:Real, y:Stream) : Real
min (x:String, y:Stream) : String

x : numeric expression
y : stream of tuples

min() determines the minimum value of the expression x over the stream of tuples y.

mul (x:Int, y:Int) : Int
mul (x:Int, y:Real) : Real
mul (x:Real, y:Int) : Real
mul (x:Real, y:Real) : Real

x : multiplicand
y : multiplicand

mul() computes and returns the product of x and y.

neq (x:Int, y:Int) : Boolean
neq (x:Int, y:Real) : Boolean
neq (x:Real, y:Int) : Boolean
neq (x:Real, y:Real) : Boolean
neq (x:String, y:String) : Boolean
neq (x:Attribute, y:Stream) : Stream
neq (x:Int, y:Stream) : Stream
neq (x:Real, y:Stream) : Stream

neq (x:String, y:Stream) : Stream
neq (x:τ, y:Stream) : Stream

x : first item to be compared
y : second item to be compared

neq() tests the inequality of x and y, returning `FALSE` if they are equal and `TRUE` if not. In the last definition of *neq()*, the first parameter, x, and the elements of the second parameter, y, must be comparable by some definition of *neq()*. If x is a tuple, then each of its attributes must be comparable by some definition of *neq()* to the corresponding attribute in the tuples of y. The returned Stream is a stream of Boolean, with one value for each input tuple of y.

nest (x:Stream, y:List, z:String) : Stream

x : stream of tuples
y : list of attributes
z : string

nest() converts the stream x into a more deeply nested stream, nesting its input stream on the attributes named in y. If z is not NULL, the nested relation, which is comprised of the nested attributes, is named z.

noop (x:Attribute) : Attribute
noop (x:Boolean) : Boolean
noop (x:Int) : Int
noop (x:Real) : Real
noop (x:String) : String
noop (x:List) : List
noop (x:Stream) : Stream

x : parameter

noop() returns the value of its parameter.

not (x:Boolean) : Boolean

x : Boolean

not() returns the logical negation of its input.

or (x:Boolean, y:Boolean) : Boolean

x : Boolean
y : Boolean

or() returns the logical OR of its inputs.

overlapb (x:Interval, y:Interval) : Boolean

x : interval
y : interval

overlapb() returns `TRUE` if its input intervals x and y have some non-empty subinterval in common and `FALSE` otherwise.

overlapt (x:Interval, y:Interval) : Interval

x : interval
y : interval

overlapt() returns `NULL` if its input intervals x and y have no non-empty subinterval in common and returns the subinterval otherwise.

precede (x:Interval, y:Interval) : Boolean

x : interval
y : interval

precede() returns `TRUE` if the starting event of x occurs before the terminating event of y; it returns `FALSE` otherwise.

printstream (s:Stream) : Void

s : stream of tuples

printstream() takes as input a stream of tuples and displays them.

rename (a:List, s:Stream) : Stream

a : list of attribute identifiers
s : stream of tuples

The tuple attribute names of the stream s are renamed with the names specified in a and the (otherwise unmodified) stream is returned. *rename()* modifies the stream's header.

retrelation (x:Stream, y:String) : Stream

x : stream of tuples
y : string

retrelation () associates with the input stream of tuples x the correlation name y, returning the stream.

retrieve (r:List, a:List, p:Boolean, l:List) : Stream

r : list of relations
a : list of attributes
p : boolean expression
l : list of attribute identifiers for outer join

retrieve() fetches a stream of tuples from the relations specified in the relation list r. The tuples which do not satisfy the Boolean expression p are discarded. *retrieve()* projects the accepted tuples on the attributes listed in a and returns them. If a is empty, all tuples are returned. In addition, all timestamps are returned. Note that as multiple relations can be specified in r, *retrieve()* implements join and cross

product.

sametime (x:Interval, y:Interval) : Boolean

x : interval
y : interval

sametime() returns `TRUE` if the two intervals are identical and `FALSE` otherwise.

sort (a:List, s:Stream) : Stream

a : list of attributes
s : stream of tuples

sort() sorts the input stream s over the specified attributes and returns the sorted stream.

span (x:Int, y:Span) : Span
span (x:Date, y:Void) : Span
span (x:Relation, y:Void) : Span
span (x:Interval, y:Void) : Span

x : integer or date or relation or interval expression
y : span or Void

span() converts its inputs into a span. Its inputs may be an integer and a span, in which case the length of the returned span is x times as long as the span y. If x is a date, the returned span is its length. For example, the length of a day is "24 hours." If x is a relation (or correlation), the returned span is the length of the valid-time timestamp of the current tuple. Finally, if x is an interval, the returned span is the length of that interval.

str2Boolean (x:String) : Boolean

x : string

str2Boolean() converts a string to a boolean constant.

str2attr (x:String) : Attribute

x : string

str2attr() converts a string to an attribute reference.

str2event (x:String) : Event

x : string

str2event() converts a string to an event constant.

str2int (x:String) : Int

x : string

str2int() converts a string to an integer constant.

str2mon (x:String) : Int

x : string

str2mon() converts a string to an integer representation of a month.

str2real (x:String) : Real

x : string

str2real() converts a string to a Real constant.

str2rel (x:String) : Relation

x : string

str2rel() converts a string to a relation reference.

str2span (x:String) : Span

x : string

str2span() converts a string to a span constant.

str2str (x:String) : String

x : string

str2str() converts a string to a string (in internal representation).

str2wild (x:String) : String

x : string

str2wild() converts a string constant to an internal representation of a "wild card" value.

sub (x:Int, y:Int) : Int
sub (x:Int, y:Real) : Real
sub (x:Real, y:Int) : Real
sub (x:Real, y:Real) : Real
sub (x:Event, y:Span) : Event
sub (x:Event, y:Event) : Span
sub (x:Span, y:Span) : Span
sub (x:Span, y:Int) : Span
sub (x:Span, y:Real) : Span
sub (x:Int, y:Span) : Span
sub (x:Real, y:Span) : Span

x : minuend

y : subtrahend

sub() computes and returns the difference of x and y.

subsume (x:Stream) : Stream

x : stream of tuples

subsume() removes subsumed tuples from the stream of tuples x and returns the

stream. (Given two tuples, if all non-null attributes of the first tuple agree with their counterparts in the second tuple, the second tuple is said to *subsume* the first.)

sum (x:Int, y:Stream) : Int
sum (x:Real, y:Stream) : Real

x : numeric expression
y : stream of tuples

sum() computes the sum of the expression x over the stream of tuples y.

tmdelete (x:List, y:Stream) : Void

x : unit list of relation
y : stream of tuples

tmdelete() implements temporal deletion, deleting the tuples of stream y from the relation x. Deleted temporal tuples are not expunged from the database; they are merely marked invalid. Furthermore, the semantics of temporal deletion are somewhat trickier than the semantics of non-temporal deletion. *tmdelete()* must not invalidate portions of existing tuples which are not covered by some tuple specified in the input stream.

tminsert (x:Relation, y:List, z:Stream) : Void

x : unit list of relation
y : list of attributes
z : stream of tuples

tminsert() adds the tuples of stream s to the named relation r. If necessary, *insert()* rearranges the attributes of the s according to the list of attributes in z. However, *tminsert()* implements temporal insertion, inserting the tuples of stream z into the relation x. The semantics of temporal insertion are somewhat trickier than the semantics of non-temporal insertion. *tminsert()* must not duplicate portions of existing tuples which are covered by some tuple specified in the input stream.

tmupdate (x:Relation, y:List, z:Stream) : Void

x : unit list of relation
y : list of update expressions
z : stream of tuples

tmupdate() implements temporal update, updating the relation x with the tuples of z. If the list of update expressions y is non-empty, they are applied to the tuples of stream z before the relation is updated. Temporal update shares elements with both temporal deletion and temporal insertion, as once again the timestamps of the tuples of the incoming stream z may not completely cover the timestamps of the existing tuples. Only the temporal portions included in the timestamps of the incoming stream of tuples z should be updated.

uminus (x:Int) : Int
uminus (x:Real) : Real
uminus (x:Span) : Span

x : numeric value

uminus() computes unary minus.

union (x:Stream, y:Stream) : Stream

x : stream of tuples
y : stream of tuples

union() computes the union of the two input streams and returns it. Duplicates are not removed. *union()* does not print its result; it must be composed with *printstream()* to print the union. The two input streams must agree on type, that is, they must have the same number of attributes with the same types.

unique (s:Stream) : Stream

s : stream of tuples

unique() removes duplicates from a stream of tuples s and returns the (duplicate-free) stream. The input stream of tuples need not be sorted; however, if it is not, *unique()* will sort it (invoking *sort(s)*) before duplicate elimination.

unnest (x:Stream, y:List) : Stream

x : stream of tuples
y : list of attributes

unnest() removes one level of nesting from a nested ($\neg$1NF) relation. The relation is unnested on the named nested relation attributes y.

update (x:Relation, y:List, z:Stream) : Void

x : unit list of relation
y : list of expressions
z : stream of tuples

update() rewrites each tuple from stream z into relation x. If the list of expressions y is not empty, the expressions are applied to the tuples of z before updating the relation x.

uspan (x:Int, y:Int, w:String, z:String) : Span
uspan (x:Int, y:Void, w:String, z:String) : Span

x : integer representing year or month
y : integer representing month or day
w : string representing span unit
z : string representing span unit

uspan() converts an SQL2-style span literal (an `INTERVAL` to a span in our internal representation. The integers x and y represent numbers of years, months, or days

while the strings w and z specify the units of x and y. The units must be specified in decreasing value, e.g., w as `YEAR` and z as `MONTH` or *DAY*. *uspan()* converts x and y to spans, and then subtracts the span of y from the span of x.

valid (x:Interval, y:Void, z:Stream) : Stream
valid (x:Event, y:Event, z:Stream) : Stream

x : temporal expression, event or interval
y : temporal expression, event or interval
z : stream of tuples

From the temporal expressions x and y, *valid()* computes a valid-time timestamp for the tuples of stream z. The timestamp is either an interval or an event. Incoming timestamps are projected out and replaced by the new timestamp. If both x and y are NULL, no valid-time timestamp is computed but the temporal attributes are still projected out, producing a non-temporal relation.

validat (x:Relation) : Event

x : relation name or correlation

validat() returns the event valid-time timestamp of the current tuple of the relation x.

validfrom (x:Relation) : Event

x : relation name or correlation

validfrom() returns the starting event of the interval valid-time timestamp of the current tuple of the relation x.

validto (x:Relation) : Event

x : relation name or correlation

validto() returns the terminating event of the interval valid-time timestamp of the current tuple of the relation x.

values (v:List) : Stream

v : list of values

values() takes as input a list of values and constructs from them a single tuple which it returns as a stream of length 1.

when (x:Void) : Boolean

xactionstart (x:Relation) : Interval

x : relation name or correlation

xactionstart() returns the starting event of the transaction-time timestamp of the current tuple of relation x.

xactionstop (x:Relation) : Interval

x : relation name or correlation

xactionstop() returns the terminating event of the transaction-time timestamp of the current tuple of relation *x*.

A.2 Directive Function Definitions

all_but_attr (x:List, y:List, z:List) : Void

x : list of relation
y : list of attribute
z : list of attribute

all_but_attr() lists in its output parameter *z* all attributes of the relations in *x except* for those listed in *y*.

dfnCORR (x:List, y:Relation) : Void

x : list of identifiers
y : relation name

dfnCORR() defines the identifiers listed in *x* to be correlations of the relation named in *y*.

dropCNTXT () *dropCNTXT()* removes the topmost context from the context stack.

enforcescope (x:String, y:List):Void

x : string
y : relation name list

enforcescope() renames, internally to an operator tree, relation correlations in order to enforce the scoping of relation references in subqueries.

extattr (x:Int, y:List) : Void
extattr (x:Real, y:List) : Void
extattr (x:Boolean, y:List) : Void

x : expression
y : list of attribute

extattr() extracts the attributes referenced in the input expression and returns them in the list *y*.

extrels (x:Int, y:List) : Void
extrels (x:Real, y:List) : Void
extrels (x:Boolean, y:List) : Void

x : expression
y : list of relations

extrels() extracts the relations referenced in the input expression and returns them in the list *y*.

gen_expr (x:List, y:String, z:String) : Interval
gen_expr (x:List, y:String, z:String) : Span
gen_expr (x:List, y:String, z:String) : Int
gen_expr (x:List, y:String, z:String) : Boolean
gen_expr (x:List, y:String, z:String) : Real
gen_expr (x:List, y:String, z:String) : String

x : list of operands
y : operator
z : string (associativity)

The *gen_expr()* directive function generates an expression from its inputs, which are a list of operands (expressions), a binary operator, and an associativity. The generated expression applies the operator to the expressions in the list, observing the given associativity. For example, the call *gen_expr([1,2,3],"+","left")* would produce the expression $((1 + 2) + 3)$.

get_all_attr (x:List, y:List) : Void

x : list of relation
y : list of attribute

get_all_attr() lists in its output parameter *y* all attributes of the relations in *x*.

is_attr_list (x:List) : Boolean

x : list of expressions

is_attr_list() determines if all elements in its input list *x* are attributes. The value `TRUE` is returned if they are; otherwise `FALSE` is returned.

is_literal_list (x:List) : Boolean

x : list of expressions

is_literal_list() determines if all elements in its input list *x* are literals. The value `TRUE` is returned if they are; otherwise `FALSE` is returned.

is_relation (x:String, y:String) : Relation

x : string
y : string

is_relation() determines if its first parameter *x* is a relation; access information for it is returned if it is, and NULL otherwise. The second parameter, if present, defines a correlation for the relation. Otherwise, the default correlation, the relation name *x*, is used.

linkcopy (x:List, y:<*node*>) : Void

x : list of nodes
y : an operator tree node
linkcopy() duplicates the node y and links it to the list x.

linknode (x:List, y:List) : Void
x : list of nodes
y : an operator tree node
linknode() links the node y to the list x.

map_coalesce_depnds (x:List, y:Boolean) : Void
x : list of relations
y : predicate
map_coalesce_depnds() maps coalesce dependencies, e.g., it pushes through attribute lists when we detect a reference to an earlier definition. Also, *map_coalesce_depnds()* creates and adds conjuncts to force equivalence of inherited coalesce attributes. The created conjuncts are *and*'ed with the predicate y.

mergeattr (x:List, y:List) : Void
x : list of attributes
y : list of attributes
mergeattr() adds the attributes in the input attribute list, x, to those in the output list, y. Duplicates are not added.

mergerelation (x:List, y:List) : Void
x : list of relations
y : list of relations
mergerelation() adds relations in the list x to the relation list y. Duplicates are not added.

newCNTXT (x:String) : Void
x : string
newCNTXT() allocates space on the context stack for the new context named in x.

regATTR (x:Attribute, y:String) : Void
x : attribute reference node
y : string
regATTR() registers the correlation name y for the attribute x. (As SQL/NF has nested relations, it allows correlations for attributes.)

revise_dummy (x:List, y:List) : Void
x : list of attributes
y : list of operator trees
revise_dummy() revises the attribute references in the aggregate `BY` clause for Quel

and TQuel to accommodate the semantics of Quel and TQuel aggregates in the `WHERE` clause, where the `BY` attributes are bound to values of the outer query but other attribute references are not.

rewrite (x:List, y:List, z:List) : Boolean

x : list of operator trees
y : list of tags
z : list of operator trees

rewrite() divides each operator tree in its input list x into a cluster and subtrees. The subtrees remain in the input node, x, while the clusters are placed in an output list, z. For a more complete explanation, see Section 2.4.4.

skip () : Void *skip()* is a no-op.

Appendix B

SQL: Specification

```
BEGIN CONTEXT X
    rel_list : relation_list;
    proj_list : attr_list;
    xpr : expr_list;
END CONTEXT X

SQL         =  { newCNTXT(X); }
               delete[" DELETE _x ", rel_list, x:d_ret]
            |  { newCNTXT(X); }
               insert[" INSERT INTO _x _y ",x:unit_rel_list,NULL,y:insrtstrm]
               { mergerelation(x,rel_list); }
            |  { newCNTXT(X); }
               printstream[" _x ", x:display]
            |  { newCNTXT(X); }
               update[" UPDATE _x _a _u ",x:unit_rel_list,a:u_expr,u:u_ret]
               { mergerelation(x,rel_list); }
;
d_ret       =  retrieve["FROM _x _w",x:unit_rel_list,proj_list,w:where,NULL]
               { mergerelation(x,rel_list); }
;
f_ret       =  retrieve[" FROM _r _w ", rel_list, xpr, w:where,NULL]
               {  r:relation_list;
                  mergerelation(r,rel_list);
               }
;
```

```
group       =  groupby[" _y GROUP BY _x ", x:attr_list, y:f_ret]
            |  f_ret
;
having      =  having[" _y HAVING _x ", x:predicate, y:group]
            |  group
;
insrtstrm   =  rename[" ( _x ) _y ", x:strng_list, y:lit_tuple]
            |  rename[" ( _x ) _y ", x:strng_list, y:select]
            |  lit_tuple
            |  select
;
strng_list  =  list[ , , ",", String, NE]
;
lit_tuple   =  values[" VALUES _x ", x:valuelist]
;
select      =  { newCNTXT(X); }
               noop[" SELECT _x ", x:order]
;
order       =  sort[" _y ORDER BY _x ", x:attr_list, y:distinct]
            |  distinct
;
distinct    =  unique[" DISTINCT _x ", x:compattr]
            |  compattr
;
compattr    =  noop[" _xpr _s ", s:proc_stream]
;
proc_stream =  compute[rewrite(xpr,[add,sub,div,mul,assign,str2int,str2real],x),
                  x:expr_list, proc_stream]
            |  aggregate[rewrite(xpr, [count,min,max,avg,sum], x),
                  x:expr_list, proc_stream]
            |  noop[is_attr_list(xpr), having]
;
display     =  union[" _x UNION _y ", x:display, y:display2]
            |  union[" _x UNION _y ", x:display, y:select]
            |  intersect[" _x INTERSECT _y ", x:select, y:select]
            |  difference[" _x EXCEPT _y ", x:select, y:select]
            |  display2
            |  select
```

```
;
display2      =  noop[" ( _x ) ", x:display]
;
u_expr        =  noop[" SET _y ", y:assgn_list]
;
u_ret         =  retrieve[" _w ", rel_list, NULL, w:where, NULL]
;
where         =  noop[" WHERE _x ", x:predicate]
               |  epsilon
;
assgn_list    =  list[ , , ",", assgn, NE]
;
assgn         =  assign[" _x = _y ", x:attribute, y:expr]
                  {  z:attr_list;
                     extattr(y,z);
                     mergeattr(z, proj_list);
                     extattr(x,z);
                     mergeattr(z, proj_list);
                  }
;
attribute     =  attr_name
               |  attribute[" _r._a ", r:unit_rel_list, a:attr_name]
;
attr_name     =  str2attr[' [a-z][a-zA-Z0-9_]* ']
;
attr_list     =  list[ , , ",", attribute, NE]
;
expr_list     =  list[ , , ",", expr, NE]
;
literal       =  String
               |  Integer
               |  Real
;
String        =  str2str[' \"[a-zA-Z0-9 ._]+\" ']
;
relation      =  str2rel[' [A-Z][a-zA-Z0-9_]* ']
;
relationref   =  is_relation[" _r ", r:relation, NULL]
```

```
                | is_relation[" _r _a ", r:relation, a:relation]
;
relation_list   = list[ , , "," , relationref, NE]
;
unit_rel_list   = list[ , , "," , relation, unit]
;
Boolean         = str2Boolean['true']
                | str2Boolean['false']
;
valuelist       = list["(", ")", ",", literal, NE]
;
compare         = lteq[" _x <= _y ", x:expr, y:expr]
                | gteq[" _x >= _y ", x:expr, y:expr]
                | neq[" _x != _y ", x:expr, y:expr]
                | lt[" _x < _y ", x:expr, y:expr]
                | gt[" _x > _y ", x:expr, y:expr]
                | eq[" _x = _y ", x:expr, y:expr]
                | and[" _v BETWEEN _l AND _h ",gteq(v:expr,l:expr),
                    lteq(v,h:expr)]
                | or[" _v NOTxBETWEEN _l AND _h ",lt(v:expr,l:expr),
                    gt(v,h:expr)]
;
predicate       = not[" NOT _x ", x:predicate]
                | and[" _x AND _y ", x:predicate, y:predicate]
                | or[" _x OR _y ", x:predicate, y:predicate]
                | noop[" ( _x ) ", x:predicate]
                | compare
                | sqe
                | Boolean
;
sqe             = iselement[" _x IN _y ", x:smexpr, y:subquery]
                | not[" _x NOT IN _y ", iselement( x:smexpr, y:subquery)]
                | exists[" EXISTS _y ", y:subquery]
                | forany[" _x =ANY _y ", eq(x:expr, y:subquery)]
                | forany[" _x <>ANY _y ", neq(x:expr, y:subquery)]
                | forany[" _x <ANY _y ", lt(x:expr, y:subquery)]
                | forany[" _x >ANY _y ", gt(x:expr, y:subquery)]
                | forany[" _x <=ANY _y ", lteq(x:expr, y:subquery)]
```

```
                | forany[" _x >=ANY _y ", gteq(x:expr, y:subquery)]
                | forany[" _x =SOME _y ", eq(x:expr, y:subquery)]
                | forany[" _x <>SOME _y ", neq(x:expr, y:subquery)]
                | forany[" _x <OME _y ", lt(x:expr, y:subquery)]
                | forany[" _x >OME _y ", gt(x:expr, y:subquery)]
                | forany[" _x <=SOME _y ", lteq(x:expr, y:subquery)]
                | forany[" _x >=SOME _y ", gteq(x:expr, y:subquery)]
                | forall[" _x =ALL _y ", eq(x:expr, y:subquery)]
                | forall[" _x <>ALL _y ", neq(x:expr, y:subquery)]
                | forall[" _x <ALL _y ", lt(x:expr, y:subquery)]
                | forall[" _x >ALL _y ", gt(x:expr, y:subquery)]
                | forall[" _x <=ALL _y ", lteq(x:expr, y:subquery)]
                | forall[" _x >=ALL _y ", gteq(x:expr, y:subquery)]
;
sqlaggr         = count[" count ( * ) ", NULL, NULL]
                | count[" count ( _x ) ", x:smexpr, NULL]
                | max[" max ( _x ) ", x:smexpr, NULL]
                | min[" min ( _x ) ", x:smexpr, NULL]
                | avg[" avg ( _x ) ", x:smexpr, NULL]
                | sum[" sum ( _x ) ", x:smexpr, NULL]
;
expr            = add[" _x + _y ", x:expr, y:expr]
                | sub[" _x - _y ", x:expr, y:expr]
                | mul[" _x * _y ", x:expr, y:expr]
                | div[" _x / _y ", x:expr, y:expr]
                | uminus[" - _x ", x:expr]
                | noop[" ( _x ) ", x:expr]
                | attribute
                | Numeric
                | String
;
smexpr          = expr
                  { is_arith_expr(); }
;
subquery        = noop[" ( _x ) ", x:select]
;
Numeric         = Integer
                | Real
```

```
                | sqlaggr
;
Integer         = str2int['-[1-9]+[0-9]*']
                | str2int['[1-9]+[0-9]*']
;
Real            = str2real[' [0-9]*.[0-9]+ ']
                | str2real[' [0-9]+.[0-9]* ']
;
%left '+' '-'
%left '*' '/'
%left OR
%left AND
%right NOT
```

Appendix C

SQL/NF: Specification

```
BEGIN CONTEXT X
    rel_list : relation_list;
    urel_list : unit_rel_list;
    outer_list : relation_list;
    assgnmnts : assgn_list;
    proj_list : attr_list;
    xpr : sel_list2;
    slst: select_list;
END CONTEXT X

sql_nf        =  printstream[" _x ", x:display]
              |  update_stmt
;
update_stmt   =  { newCNTXT(X); }
                 delete[" ERASE _x ", rel_list, x:d_ret]
              |  { newCNTXT(X); }
                 update[" MODIFY _x ", urel_list, assgnmnts, x:u_ret]
              |  insert[" STORE _x _y ", x:unit_rel_list, NULL, y:insrtstrm]
;
display       =  function_stmt
              |  union[" _x UNION _y ", x:display, y:display]
              |  intersect[" _x INTERSECT _y ", x:display, y:display]
              |  difference[" _x DIFFERENCE _y ", x:display, y:display]
              |  operator_stmt
              |  select
```

```
;
select          =  retrieve[" _r _w ",rel_list,xpr,w:where,outer_list]
                   {  r:unit_rel_list;
                      newCNTXT(X);
                      mergerelation(r,rel_list);
                      get_all_attr(rel_list,xpr);
                   }
                |  { newCNTXT(X); }
                   noop[" SELECT _slst _p ", p:proc_stream]
;
select_list     =  noop[" ALL ", xpr]
                   { get_all_attr(rel_list,xpr); }
                |  noop[" ALL BUT _a ",xpr]
                   {  a:attr_list;
                      all_but_attr(rel_list,xpr,a);
                   }
                |  noop[" _xpr ", xpr ]
;
sel_list2       =  list[ , , ",", select_item, NE ]
;
select_item     =  expr
                |  noop[" _x AS _a ", a:attr_name]
                   {  x:expr;
                      regATTR(x,a);
                   }
                |  noop[" _r ALL ", r:dot_ref_list]
                   {  mergerelation(r,rel_list);
                      get_all_attr(r,xpr);
                   }
;
proc_stream     =  compute[rewrite(xpr,[add,assign,div,mul,sub,distinct,subsume],x),
                      x:sel_list2, proc_stream]
                |  aggregate[rewrite(xpr,[avg,count,max,min,sum],x),
                      x:sel_list2, proc_stream]
                |  noop[is_attr_list(xpr), f_ret]
;
f_ret           =  retrieve[" FROM _r _w ",rel_list,xpr,w:where,outer_list]
                   {  r:relation_list;
```

```
                    mergerelation(r,rel_list);
                }
;
preserve        =   noop[" PRESERVE _outer_list ", outer_list]
                |   epsilon
;
where           =   noop[" WHERE _x _p ", x:predicate]
                    {   p:preserve;
                        skip();
                    }
                |   epsilon
;
function_stmt   =   max[" MAX ( _x ) ", x:display, NULL]
                |   min[" MIN ( _x ) ", x:display, NULL]
                |   avg[" AVG ( _x ) ", x:display, NULL]
                |   sum[" SUM ( _x ) ", x:display, NULL]
                |   count[" COUNT ( _x ) ", x:display, NULL]
                |   unique[" DISTINCT ( _x ) ", x:display]
                |   subsume[" SUBSUME ( _x ) ", x:display]
;
operator_stmt   =   nest[" NEST _x ON _c AS _a ", x:nested_query, c:attr_list,
                        a:relation]
                |   nest[" NEST _x ON _c ", x:nested_query, c:attr_list, NULL]
                |   unnest[" UNNEST _x ON _c ", x:nested_query, c:attr_list]
                |   { newCNTXT(X); }
                    sort[" ORDER _x BY _s ", x:nested_query, s:attr_list]
;
nested_query    =   nqe1
                |   nqe2
;
nqe1            =   noop[" ( _d ) ", d:display]
;
nqe2            =   retrieve[" _r ", rel_list, xpr, NULL, NULL]
                    {   r:unit_rel_list;
                        newCNTXT(X);
                        mergerelation(r,rel_list);
                        get_all_attr(r,xpr);
                    }
```

```
;
unit_rel_list   =  list[, , "," , relation, unit]
;
relation        =  str2rel[' [A-Z][a-zA-Z0-9_]* ' ]
;
relation_list   =  list[, , "," , relationref, NE]
;
relationref     =  is_relation[" _r ",r:relation, NULL]
                |  is_relation[" _n AS _a ",n:relation,a:relation]
                |  retrelation[" _n ", n:nqe1,NULL]
                |  retrelation[" _n AS _a ",n:nqe1,a:relation]
;
attr_list       =  list[, , ",", attribute, NE]
;
attribute       =  attr_name
                |  attribute[" _r _a ", r:dot_ref_list, a:attr_name]
;
attr_name       =  str2attr[' [a-z][a-zA-Z0-9_]* ' ]
;
dot_ref_list    =  list[, ".", ".", relation, NE]
;
compare         =  lteq[" _x <= _y ", x:expr, y:expr]
                |  gteq[" _x >= _y ", x:expr, y:expr]
                |  neq[" _x <> _y ", x:expr, y:expr]
                |  gt[" _x > _y ", x:expr, y:expr]
                |  lt[" _x < _y ", x:expr, y:expr]
                |  eq[" _x = _y ", x:expr, y:expr]
                |  iselement[" _x ELEMENT OF _y ",x:expr,y:expr]
                |  not[" _x NOT ELEMENT OF _y ",iselement(x:expr,y:expr)]
                |  contains[" _x CONTAINS _y ",x:expr,y:expr]
                |  not[" _x NOT CONTAINS _y ",contains(x:expr,y:expr)]
                |  contains[" _x SUBSET OF _y ",y:expr,x:expr]
                |  not[" _x NOT SUBSET OF _y ",contains(y:expr,x:expr)]
                |  and[" _v BETWEEN _l AND _h ",gteq(v:expr,l:expr),
                      lteq(v,h:expr)]
                |  or[" _v NOT BETWEEN _l AND _h ",lt(v:expr,l:expr),
                      gt(v,h:expr)]
                |  eq[" _x IS NULL ", x:attribute,NULL]
```

```
                | not[" _x IS NOT NULL ", eq(x:attribute,NULL)]
                | exists [" EXISTS _y ", y:nested_query]
                | iselement[" _x IN _y ", x:tuple_expr, y:nested_query]
;
predicate       = not[" NOT _x ", x:predicate]
                | and[" _x AND _y ", x:predicate, y:predicate]
                | or[" _x OR _y ", x:predicate, y:predicate]
                | noop[" ( _x ) ", x:predicate]
                | compare
;
expr            = add[" _x + _y ", x:expr, y:expr]
                | sub[" _x - _y ", x:expr, y:expr]
                | mul[" _x * _y ", x:expr, y:expr]
                | div[" _x / _y ", x:expr, y:expr]
                | uminus[" - _x ", x:expr]
                | noop[" ( _x ) ", x:expr2]
                | value
;
expr2           = add[" _x + _y ", x:expr, y:expr]
                | sub[" _x - _y ", x:expr, y:expr]
                | mul[" _x * _y ", x:expr, y:expr]
                | div[" _x / _y ", x:expr, y:expr]
                | uminus[" - _x ", x:expr]
;
literal         = Boolean
                | Numeric
                | String
                | tuple_literal
                | dont_care
;
value           = attribute
                | literal
                | nested_query
;
tuple_seq       = list[ , , ,tuple_literal,NE]
;
tuple_literal   = list["{","}",",",literal,NE]
;
```

```
tuple_expr    =  list[“<”,“>”,“,”,expr,NE]
;
dont_care     =  str2wild[‘ ? ’]
;
Boolean       =  str2Boolean[‘true’]
              |  str2Boolean[‘false’]
;
Numeric       =  Integer
              |  Real
;
Integer       =  str2int[‘-[1-9]+[0-9]*’]
              |  str2int[‘[1-9]+[0-9]*’]
;
Real          =  str2real[‘ [0-9]*.[0-9]+ ’]
              |  str2real[‘ [0-9]+.[0-9]* ’]
;
String        =  str2str[‘ \“[a-zA-Z0-9 ._]-\” ’]
;
d_ret         =  retrieve[“ _x _w ”,x:unit_rel_list,proj_list,w:where,NULL]
                 { mergerelation(x,rel_list); }
;
insrtstrm     =  rename[“ ( _x ) VALUES _y ”,x:attr_list,values(y:tuple_seq)]
              |  rename[“ ( _x ) _y ”, x:attr_list, y:select]
              |  values[“ VALUES _t ”,t:tuple_seq]
              |  select
;
u_ret         =  compute[“ _urel_list SET _y _z ”, y:assgn_list,
                    retrieve(rel_list, a, z:where,NULL)]
                 {  a:attr_list;
                    mergerelation(urel_list,rel_list);
                    get_all_attr(urel_list,a);
                 }
;
assgn_list    =  list[, , “,”, assgn, NE]
;
assgn         =  assign[“ _x = _y ”, x:attribute, y:uvalue]
;
uvalue        =  noop[“ _e ”, e:expr]
```

```
                    {   z:attr_list;
                        extattr(e,z);
                        mergeattr(z, proj_list);
                    }
                |   noop[" ( _u ) ",u:update_stmt]
;
%left '+' '-'
%left '*' '/'
%left OR
%left AND
%right NOT
%left UNION
%left DIFFERENCE
%left INTERSECT
```

Appendix D

TSQL2: Specification

```
BEGIN CONTEXT X
    unrel_list : unit_rel_list;
    rel_list : relation_list;
    proj_list : attr_list;
    xpr : expr_list;
    val : element;
END CONTEXT X

TSQL2       =  { newCNTXT(X); }
               tmdelete[" DELETE _x ", unrel_list, x:d_valid]
            |  { newCNTXT(X); }
               tminsert[" INSERT INTO _x _y ",x:unit_rel_list,NULL,y:insrtstrm]
               { mergerelation(x,unrel_list); }
            |  { newCNTXT(X); }
               printstream[" _x ", x:display]
            |  { newCNTXT(X); }
               tmupdate[" UPDATE _x _a _u ",x:unit_rel_list,a:u_expr,u:u_valid]
               { mergerelation(x,unrel_list); }
;
d_ret       =  retrieve["FROM _x _w",x:unit_rel_list,proj_list,w:where,NULL]
               { mergerelation(x,unrel_list); }
;
d_valid     =  valid[" _d VALID _v ", v:element, NULL, d:d_ret]
;
f_ret       =  retrieve[" FROM _r _w ", rel_list, xpr, w:where,NULL]
```

```
                    {  r:relation_list;
                       mergerelation(r,rel_list);
                       map_coalesce_depnds(r,w);
                    }
;
group          =    groupby[" _y GROUP BY _x ", x:attr_list, y:f_ret]
               |    f_ret
;
having         =    having[" _y HAVING _x ", x:predicate, y:group]
               |    group
;
insrtstrm      =    rename[" ( _x ) _y ", x:strng_list, y:i_valid]
               |    rename[" ( _x ) _y ", x:strng_list, y:select]
               |    valid[" _l ", NULL, NULL, l:litrelation]
               |    select
;
i_valid        =    valid[" _l VALID _e ", e:element, NULL, l:litrelation]
;
strng_list     =    list[ , , ",", String, NE]
;
litrelation    =    values[" VALUES _x ", x:littuples]
;
select         =    { newCNTXT(X); }
                    noop[" SELECT _x ", x:order]
;
order          =    sort[" _y ORDER BY _x ", x:attr_list, y:distinct]
               |    distinct
;
distinct       =    unique[" DISTINCT _x ", x:validtime]
               |    validtime
;
validtime      =    valid[" _t ", a, NULL, t:compattr]
                    {  a:element;
                       gen_expr(rel_list,"overlapt","left",a);
                    }
               |    valid[" SNAPSHOT _t ", NULL, NULL, t:compattr]
               |    valid[" _xpr VALIDINTERSECT _t ",overlapt(val,a),NULL,
                       t:compattr2]
```

```
                    {  a:element;
                       gen_expr(rel_list,"overlapt","left",a);
                       }
                  |  valid[" _xpr VALID _t ", val, NULL, t:compattr2]
;
compattr2     =  noop[" _val _s ", s:proc_stream]
;
compattr      =  noop[" _xpr _s ", s:proc_stream]
;
proc_stream   =  compute[rewrite(xpr,[add,sub,div,mul,assign,str2int,
                       str2real,intervalc,span],x),x:expr_list, proc_stream]
                  |  aggregate[rewrite(xpr, [count,min,max,avg,sum,unique], x),
                       x:expr_list, proc_stream]
                  |  noop[is_attr_list(xpr), having]
;
display       =  union[" _x UNION _y ", x:display, y:display2]
                  |  intersect[" _x INTERSECT _y ", x:display2, y:display2]
                  |  difference[" _x EXCEPT _y ", x:display, y:display2]
                  |  display2
;
display2      =  noop[" ( _x ) ", x:display]
                  |  select
;
u_expr        =  noop[" SET _y ", y:assgn_list]
;
u_valid       =  valid[" VALID _v _u ", v:element, NULL, u:u_ret]
;
u_ret         =  retrieve[" _w ", unrel_list, NULL, w:where, NULL]
;
where         =  noop[" WHERE _x ", x:predicate]
                  |  epsilon
;
assgn_list    =  list[ , , ",", assgn, NE]
;
assgn         =  assign[" _x = _y ", x:attribute, y:expr]
                    {  z:attr_list;
                       extattr(y,z);
                       mergeattr(z, proj_list);
```

```
                    extattr(x,z);
                    mergeattr(z, proj_list);
                }
;
attribute     =  attr_name
              |  attribute[" _r._a ", r:unit_rel_list, a:attr_name]
;
attr_name     =  str2attr[' [a-z][a-zA-Z0-9_]* ']
;
attr_list     =  list[ , , ",", attribute, NE]
;
expr_list     =  list[ , , ",", expr, NE]
;
literal       =  String
              |  Integer
              |  Real
;
String        =  str2str[' \"[a-zA-Z0-9 ._]+\" ']
;
relation      =  str2rel[' [A-Z][a-zA-Z0-9_]* ']
;
relationref   =  is_relation[" _r ", r:relation, NULL]
              |  is_relation[" _r _a ", r:relation, a:relation]
              |  is_relation[" _r AS _a ", r:relation, a:relation]
              |  retrelation[" _n AS _a ",n:coalescrel, a:relation]
              |  retrelation[" _n AS _a ",n:subquery, a:relation]
;
coalescrel    =  coalesce[" _r _a ( _i ) ", retrieve(r:relation,NULL,NULL,NULL),
                    a:clscattrlst,i:clscintrvl]
              |  coalesce[" _r _a ", retrieve(r:relation,NULL,NULL,NULL),
                    a:clscattrlst, NULL]
              |  coalesce[" _r ( _i ) ", retrieve(r:relation,NULL,NULL,NULL),
                    NULL, i:clscintrvl]
;
clscattrlst   =  list["(", ")", ",", attribute, NE]
;
clscintrvl    =  str2str[' Interval ']
              |  str2str[' Element ']
```

```
;
relation_list   =  list[ , , “,” , relationref, NE]
;
unit_rel_list   =  list[ , , “,” , relation, unit]
;
Boolean         =  str2Boolean[‘true’]
                |  str2Boolean[‘false’]
;
littuples       =  list[ , , “,”, valuelist, NE]
;
valuelist       =  list[“(”, “)”, “,”, literal, NE]
;
compare         =  lteq[“ _x <= _y ”, x:expr, y:expr]
                |  gteq[“ _x >= _y ”, x:expr, y:expr]
                |  neq[“ _x != _y ”, x:expr, y:expr]
                |  gt[“ _x > _y ”, x:expr, y:expr]
                |  lt[“ _x < _y ”, x:expr, y:expr]
                |  eq[“ _x = _y ”, x:expr, y:expr]
                |  timecompare
;
predicate       =  not[“ NOT _x ”, x:predicate]
                |  and[“ _x AND _y ”, x:predicate, y:predicate]
                |  or[“ _x OR _y ”, x:predicate, y:predicate]
                |  noop[“ ( _x ) ”, x:predicate]
                |  compare
                |  sqe
                |  Boolean
;
sqe             =  iselement[“ _x IN _y ”, x:smexpr, y:subquery]
                |  not[“ _x NOT IN _y ”, iselement( x:smexpr, y:subquery)]
                |  exists[“ EXISTS _y ”, y:subquery]
                |  isunique[“ UNIQUE _y ”, y:subquery]
                |  forany[“ _x =ANY _y ”, eq(x:expr, y:subquery)]
                |  forany[“ _x <>ANY _y ”, neq(x:expr, y:subquery)]
                |  forany[“ _x <ANY _y ”, lt(x:expr, y:subquery)]
                |  forany[“ _x >ANY _y ”, gt(x:expr, y:subquery)]
                |  forany[“ _x <=ANY _y ”, lteq(x:expr, y:subquery)]
                |  forany[“ _x >=ANY _y ”, gteq(x:expr, y:subquery)]
```

```
                | forany[" _x =SOME _y ", eq(x:expr, y:subquery)]
                | forany[" _x <>SOME _y ", neq(x:expr, y:subquery)]
                | forany[" _x <SOME _y ", lt(x:expr, y:subquery)]
                | forany[" _x >SOME _y ", gt(x:expr, y:subquery)]
                | forany[" _x <=SOME _y ", lteq(x:expr, y:subquery)]
                | forany[" _x >=SOME _y ", gteq(x:expr, y:subquery)]
                | forall[" _x =ALL _y ", eq(x:expr, y:subquery)]
                | forall[" _x <>ALL _y ", neq(x:expr, y:subquery)]
                | forall[" _x <ALL _y ", lt(x:expr, y:subquery)]
                | forall[" _x >ALL _y ", gt(x:expr, y:subquery)]
                | forall[" _x <=ALL _y ", lteq(x:expr, y:subquery)]
                | forall[" _x >=ALL _y ", gteq(x:expr, y:subquery)]
;
tsql2aggr       = count[" count _s ( * ) ", NULL, s:aggrstrm]
                | count[" count _s ( _x ) ", x:smexpr, s:aggrstrm]
                | max[" max _s ( _x ) ", x:smexpr, s:aggrstrm]
                | min[" min _s ( _x ) ", x:smexpr, s:aggrstrm]
                | avg[" avg _s ( _x ) ", x:smexpr, s:aggrstrm]
                | sum[" sum _s ( _x ) ", x:smexpr, s:aggrstrm]
                | count[" count _s ( _e ) ", e:eventexpr, s:aggrstrm]
                | count[" count _s ( _i ) ", i:intrvlexpr, s:aggrstrm]
                | count[" count _s ( _sx ) ", sx:spanexpr, s:aggrstrm]
                | max[" max _s ( _e ) ", e:eventexpr, s:aggrstrm]
                | min[" min _s ( _e ) ", e:eventexpr, s:aggrstrm]
                | avg[" avg _s ( _e ) ", e:eventexpr, s:aggrstrm]
                | max[" max _s ( _sx ) ", sx:spanexpr, s:aggrstrm]
                | min[" min _s ( _sx ) ", sx:spanexpr, s:aggrstrm]
                | avg[" avg _s ( _sx ) ", sx:spanexpr, s:aggrstrm]
                | sum[" sum _s ( _sx ) ", sx:spanexpr, s:aggrstrm]
;
aggrstrm        = unique[" DISTINCT ", NULL]
                | noop[" ALL ", NULL]
                | epsilon
;
expr            = add[" _x + _y ", x:expr, y:expr]
                | sub[" _x - _y ", x:expr, y:expr]
                | mul[" _x * _y ", x:expr, y:expr]
                | div[" _x / _y ", x:expr, y:expr]
```

```
                | div[" _s0 / _s1 ", s0:spanexpr, s1:spanexpr]
                | uminus[" - _x ", x:expr]
                | noop[" ( _x ) ", x:expr]
                | attribute
                | Numeric
                | String
;
smexpr          = expr
                  { is_arith_expr(); }
;
subquery        = noop[" ( _x ) ", x:select]
;
Numeric         = Integer
                | Real
                | tsql2aggr
;
Integer         = sgndint
                | unsgndint
;
sgndint         = str2int['-[1-9]+[0-9]*']
;
unsgndint       = str2int['[1-9]+[0-9]*']
;
Real            = str2real[' [0-9]*.[0-9]+ ']
                | str2real[' [0-9]+.[0-9]* ']
;
timecompare     = lt[" _s0 < _s1 ", s0:spanexpr, s1:spanexpr]
                | lteq[" _s0 <= _s1 ", s0:spanexpr, s1:spanexpr]
                | gt[" _s0 > _s1 ", s0:spanexpr, s1:spanexpr]
                | gteq[" _s0 >= _s1 ", s0:spanexpr, s1:spanexpr]
                | eq[" _s0 = _s1 ", s0:spanexpr, s1:spanexpr]
                | sametime[" _t0 = _t1 ", t0:element, t1:element]
                | sametime[" _a0 MEETS _a1 ", endof(a0:element),
                    beginof(a1:element)]
                | precede[" _a0 PRECEDES _a1 ", a0:element, a1:element]
                | overlapb[" _a0 OVERLAPS _a1 ", a0:element, a1:element]
                | and[" _a0 CONTAINS _a1 ",
                    lteq(beginof(a0:element),beginof(a1:element)),
```

```
                    gteq(endof(a0:element),endof(a1:element))]
;
element      =  eventexpr
             |  intrvlexpr
             |  overlapt[" INTERSECT ( _e0 , _e1 ) ", e0:element, e1:element]
;
eventexpr    =  sub[" _e - _s ", e:eventexpr, s:spanexpr]
             |  add[" _e + _s ", e:eventexpr, s:spanexpr]
             |  add[" _s + _e ", s:spanexpr, e:eventexpr]
             |  earliest[" FIRST ( _e0 , _e1 ) ", e0:eventexpr, e1:eventexpr]
             |  latest[" LAST ( _e0 , _e1 ) ", e0:eventexpr, e1:eventexpr]
             |  beginof[" BEGIN ( _e ) ", e:element]
             |  endof[" END ( _e ) ", e:element]
             |  Event
;
Event        =  event[" | _d | ", d:fulldate]
             |  event[" | _e | ", e:eventconv]
             |  event[" | _r | ", r:relation]
             |  event[" | _a | ", a:attribute]
             |  sql2date
             |  sql2tmstmp
;
eventconv    =  str2event[' beginning ']
             |  str2event[' forever ']
             |  str2event[' present ']
             |  str2event[' current_date ']
             |  str2event[' current_timestamp ']
;
intrvlexpr   =  extend[" _i - _s ", i:intrvlexpr, s:spanexpr]
             |  extend[" _s + _i ", s:spanexpr, i:intrvlexpr]
             |  extend[" _i + _s ", i:intrvlexpr, s:spanexpr]
             |  earliest[" FIRST ( _i0 , _i1 ) ",i0:intrvlexpr,i1:intrvlexpr]
             |  latest[" LAST ( _i0 , _i1 ) ", i0:intrvlexpr, i1:intrvlexpr]
             |  Interval
;
Interval     =  intervalc[" [ _p ] ", p:partdate, NULL]
             |  intervalo[" [ _p ) ", p:partdate, NULL]
             |  intervalc[" [ _e0 , _e1 ] ", e0:eventexpr, e1:eventexpr]
```

```
            |  intervalo[" [ _e0 , _e1 ) ", e0:eventexpr, e1:eventexpr]
            |  intervalc[" [ _e0 TO _e1 ] ", e0:eventexpr, e1:eventexpr]
            |  intervalo[" [ _e0 TO _e1 ) ", e0:eventexpr, e1:eventexpr]
            |  intervalc[" [ _r ] ", r:relation, NULL]
            |  intervalc[" [ _a ] ", a:attribute, NULL]
;
spanexpr    =  add[" _s0 + _s1 ", s0:spanexpr, s1:spanexpr]
            |  div[" _spx / _sqx ", spx:spanexpr, sqx:expr]
            |  sub[" _s0 - _s1 ", s0:spanexpr, s1:spanexpr]
            |  sub[" _e0 - _e1 ", e0:eventexpr, e1:eventexpr]
            |  sub[" _spx - _sqx ", spx:spanexpr, sqx:expr]
            |  sub[" _sqx - _spx ", sqx:expr, spx:spanexpr]
            |  mul[" _spx * _sqx ", spx:spanexpr, sqx:expr]
            |  mul[" _sqx * _spx ", sqx:expr, spx:spanexpr]
            |  span[" SPAN ( _i ) ", i:intrvlexpr,NULL]
            |  abs[" ABSOLUTE ( _s ) ", s:spanexpr]
            |  noop[" ( _s ) ", s:spanexpr]
            |  uminus[" - _s ", s:spanexpr]
            |  Span
;
Span        =  noop[" % _s % ", s:spanconv]
            |  span[" % _i _s % ", i:Integer, s:spanconv]
            |  span[" % _d % ", d:grgrndate, NULL]
            |  span[" % _r % ", r:relation, NULL]
            |  span[" % _a % ", a:attribute, NULL]
            |  sql2span
;
spanconv    =  str2span[' day ']
            |  str2span[' week ']
            |  str2span[' month ']
            |  str2span[' year ']
;
grgrndate   =  fulldate
            |  partdate
;
fulldate    =  grgrndate[" _m / _d / _y ",m:unsgndint,d:unsgndint,y:unsgndint]
            |  grgrndate[" _m _d , _y ", str2mon(m:String), d:unsgndint,
                  y:unsgndint]
```

```
;
partdate      =  grgrndate[" _m , _y ", str2mon(m:String), NULL, y:unsgndint]
;
sql2span      =  uspan[" INTERVAL ' _y - _m ' _u0 TO _u1 ",
                     y:unsgndint, m:unsgndint, u0:spanunit, u1:spanunit]
               |  uspan[" INTERVAL ' _y - _m ' _u ", y:unsgndint,
                     m:unsgndint, u:spanunit, u:spanunit]
               |  uspan[" INTERVAL ' _mdy ' _u ", mdy:unsgndint, NULL,
                     u:spanunit, u:spanunit]
;
spanunit      =  str2str[' YEAR ']
               |  str2str[' MONTH ']
               |  str2str[' DAY ']
;
sql2date      =  event[" DATE ' _yy - _mm - _dd ' ",
                     grgrndate(mm:unsgndint, yy:unsgndint, dd:unsgndint)]
;
sql2tmstmp    =  event[" TIMESTAMP ' _yy - _mm - _dd ' ",
                     grgrndate(mm:unsgndint, yy:unsgndint, dd:unsgndint)]
;
%left UNION EXCEPT
%left INTERSECT
%left '+' '-'
%left '*' '/'
%left OR
%left AND
%right NOT
```

Appendix E

Quel: Specification

```
BEGIN CONTEXT X
    agcol : whereaggrcol;
    attrlst : attr_list;
    bylist : attr_list;
    proj_list: attr_list;
    rel_list : relation_list;
    updlist : assgnlst;
    xpr : expr_list;
END CONTEXT X

Quel        =  { newCNTXT(X); }
               delete[" DELETE _r _x ", r:unit_rel_list, x:delstrm]
               { mergerelation(r,rel_list); }
            |  { newCNTXT(X); }
               insert[" APPEND TO _r _y ",r:unit_rel_list,attrlst,y:insertstrm]
            |  { newCNTXT(X); }
               update["REPLACE _r _y",r:unit_rel_list,updlist,y:updatestrm]
               { mergerelation(r,rel_list); }
            |  dfnCORR[" RANGE OF _v IS _r ", v:relation_list, r:relation]
            |  { newCNTXT(X); }
               noop[" RETRIEVE _x ",x:retstmt]
;
retstmt     =  create[" INTO _x _y ", x:relation, y:insertstrm]
            |  printstream[" _x ", x:display]
;
```

```
display     =  unique[" UNIQUE _r ", r:rename]
            |  rename
;
rename      =  rename[" ( _p _c ", p:renamelst,c:process1]
;
process1    =  noop[" ) _c ",c:proc_cycl1]
;
proc_cycl1  =  compute[rewrite(xpr,[add,sub,div,mul,uminus,str2int,str2real],x),
                  x:expr_list, proc_cycl1]
            |  aggregate[rewrite(xpr,[count,min,max,avg,sum,filter,aggregate,
                  groupby,gt,ge,lt,le,eq,str2int,str2real],x), x:expr_list, proc_cycl1]
            |  noop[is_attr_list(xpr), retstrm1]
;
renamelst   =  list[ , , ",", retcolumn, NE]
;
retcolumn   =  noop[" _x = _y ", x:String]
               {  y:quelexpr;
                  linknode(xpr,y);
               }
            |  noop[" _x = _y ", x:String]
               {  y:String;
                  linknode(xpr,y);
               }
            |  noop[" _y ", NULL]
               {  y:quelexpr;
                  linknode(xpr,y);
               }
;
retstrm1    =  retrieve[" _w ", rel_list, xpr, w:where, NULL]
;
where       =  noop[" WHERE _x ", x:wpredicate]
            |  epsilon
;
delstrm     =  retrieve[" _w ",rel_list,NULL,w:where,NULL]
;
insertstrm  =  noop[" ( _p _j ", j:process2]
               {  p:insertlst;
                  skip();
```

```
                }
;
insertlst       =  list[ , , ",", insertcol, NE]
;
insertcol       =  noop[" _x = _y ", x:iattr]
                   {  y:String;
                      z:attr_list;
                      linknode(xpr,y);
                      extattr(x,z);
                      mergeattr(z, attrlst);
                   }
                |  noop[" _x = _y ", x:iattr]
                   {  y:quelexpr;
                      z:attr_list;
                      linknode(xpr,y);
                      extattr(y,z);
                      mergeattr(z, proj_list);
                      extattr(x,z);
                      mergeattr(z, attrlst);
                   }
;
process2        =  noop[" ) _c ",c:proc_cycl2]
;
proc_cycl2      =  compute[rewrite(xpr,[add,sub,div,mul,uminus,str2int,str2real,
                      str2str],x),x:expr_list, proc_cycl2]
                |  aggregate[rewrite(xpr,[count,min,max,avg,sum,filter,groupby,
                      gt,ge,lt,le,eq,str2int,str2real,str2str],x), x:expr_list, proc_cycl2]
                |  noop[is_attr_list(xpr), retstrm2]
;
retstrm2        =  retrieve[" WHERE _w ", rel_list, xpr, w:wpredicate, NULL]
                |  epsilon
;
compare         =  lteq[" _x <= _y ", x:quelval, y:quelval]
                |  gteq[" _x >= _y ", x:quelval, y:quelval]
                |  neq[" _x != _y ", x:quelval, y:quelval]
                |  gt[" _x > _y ", x:quelval, y:quelval]
                |  lt[" _x < _y ", x:quelval, y:quelval]
                |  eq[" _x = _y ", x:quelval, y:quelval]
```

```
;
quelval      =  quelexpr
             |  String
;
predicate    =  not[" NOT _x ", x:predicate]
             |  and[" _x AND _y ", x:predicate, y:predicate]
             |  or[" _x OR _y ", x:predicate, y:predicate]
             |  noop[" ( _x ) ", x:predicate]
             |  compare
             |  Boolean
;
expr_list    =  list[ , , ",", quelexpr, NE]
;
smexpr       =  add[" _x + _y ", x:smexpr, y:smexpr]
             |  sub[" _x - _y ", x:smexpr, y:smexpr]
             |  mul[" _x * _y ", x:smexpr, y:smexpr]
             |  div[" _x / _y ", x:smexpr, y:smexpr]
             |  uminus[" - _x ", x:smexpr]
             |  noop[" ( _x ) ", x:smexpr]
             |  LitNumeric
             |  attribute
;
quelexpr     =  add[" _x + _y ", x:quelexpr, y:quelexpr]
             |  sub[" _x - _y ", x:quelexpr, y:quelexpr]
             |  mul[" _x * _y ", x:quelexpr, y:quelexpr]
             |  div[" _x / _y ", x:quelexpr, y:quelexpr]
             |  uminus[" - _x ", x:quelexpr]
             |  noop[" ( _x ) ", x:quelexpr]
             |  Numeric
             |  attribute
;
subquery     =  groupby[" _xpr BY _g _w",g:attr_list, w:substream]
             |  retrieve[" _xpr _w ",rel_list,xpr,w:where,NULL]
;
substream    =  retrieve[" _w ",rel_list,xpr,w:where,NULL]
;
quelaggr     =  avg[" AVG ( _e _b ) ", e:aggrexpr, b:aggrgrp]
             |  count[" COUNT ( _e _b ) ", e:aggrexpr, b:aggrgrp]
```

```
              |  max[" MAX ( _e _b ) ", e:aggrexpr, b:aggrgrp]
              |  min[" MIN ( _e _b ) ", e:aggrexpr, b:aggrgrp]
              |  sum[" SUM ( _e _b ) ", e:aggrexpr, b:aggrgrp]
              |  avg[" AVGU ( _e _b ) ", e:aggrexpr, unique(b:aggrgrp)]
              |  count[" COUNTU ( _e _b ) ", e:aggrexpr, unique(b:aggrgrp)]
              |  sum[" SUMU ( _e _b ) ", e:aggrexpr, unique(b:aggrgrp)]
;
aggrgrp       =  groupby[ " BY _a _w ", a:attr_list, w:select]
              |  select
              |  quelaggr
;
aggrexpr      =  smexpr
              |  epsilon
;
select        =  filter[ " WHERE _x ", x:predicate, NULL]
              |  epsilon
;
attr_list     =  list[ , , ",", attribute, NE]
;
iattr         =  attr_name
              |  attribute[" _r._a ", r:unit_rel_list, a:attr_name]
;
attribute     =  attribute[" _r._a ", r:unit_rel_list, a:attr_name]
                 { mergerelation(r,rel_list); }
;
updatestrm    =  retrieve["( _updlist ) _w",rel_list,NULL,w:where,NULL]
;
assgnlst      =  list[ , , ",", assgn, NE]
;
assgn         =  assign[" _x = _y ", x:attribute, y:quelval]
                 {  z:attr_list;
                    extattr(y,z);
                    mergeattr(z, proj_list);
                    linknode(xpr,y);
                 }
;
String        =  str2str[' \"[a-zA-Z0-9 _;.,-]+\" ']
;
```

```
attr_name     =  str2attr[‘ [a-z][a-zA-Z0-9_]* ’]
;
relation_list =  list[ , , “,” , relation, NE]
;
unit_rel_list =  list[ , , “,” , relation, unit]
;
relation      =  str2rel[‘ [A-Z][a-zA-Z0-9_]* ’]
;
Boolean       =  str2Boolean[ ‘true’ ]
               |  str2Boolean[ ‘false’ ]
;
LitNumeric    =  Int
               |  Real
;
Numeric       =  LitNumeric
               |  quelaggr
;
Int           =  str2int[ ‘[0-9]+’ ]
               |  str2int[ ‘-[1-9]+[0-9]*’ ]
;
Real          =  str2real[‘ [0-9]*.[0-9]+ ’]
               |  str2real[‘ [0-9]+.[0-9]* ’]
;
wpredicate    =  not[“ NOT _x ”, x:wpredicate]
               |  and[“ _x AND _y ”, x:wpredicate, y:wpredicate]
               |  or[“ _x OR _y ”, x:wpredicate, y:wpredicate]
               |  noop[“ ( _x ) ”, x:wpredicate]
               |  wcompare
               |  Boolean
;
wcompare      =  lteq[“ _x <= _y ”, x:wquelval, y:wquelval]
               |  gteq[“ _x >= _y ”, x:wquelval, y:wquelval]
               |  neq[“ _x != _y ”, x:wquelval, y:wquelval]
               |  gt[“ _x > _y ”, x:wquelval, y:wquelval]
               |  lt[“ _x < _y ”, x:wquelval, y:wquelval]
               |  eq[“ _x = _y ”, x:wquelval, y:wquelval]
;
wquelval      =  wquelexpr
```

```
                | String
;
wquelexpr       = add[" _x + _y ", x:wquelexpr, y:wquelexpr]
                | sub[" _x - _y ", x:wquelexpr, y:wquelexpr]
                | mul[" _x * _y ", x:wquelexpr, y:wquelexpr]
                | div[" _x / _y ", x:wquelexpr, y:wquelexpr]
                | uminus[" - _x ", x:wquelexpr]
                | { newCNTXT(X); }
                  boolean2int[" ANY ( _x ) ", exists(x:subquery)]
                  { enforcescope("X","rel_list"); }
                | noop[" ( _x ) ", x:wquelexpr]
                | wNumeric
                | attribute
;
wNumeric        = LitNumeric
                | { newCNTXT(X); }
                  aggregate[" _w _s ",w:whereaggrlst,s:subqret]
                  { revise_dummy(bylist,s); }
;
whereaggrlst    = list[ , , "," , whereaggr, unit]
;
whereaggr       = avg[" AVG ", agcol, NULL]
                | count[" COUNT ", agcol, NULL]
                | max[" MAX ", agcol, NULL]
                | min[" MIN ", agcol, NULL]
                | sum[" SUM ", agcol, NULL]
                | avg[" AVGU ", agcol, unique(NULL)]
                | count[" COUNTU ", agcol, unique(NULL)]
                | sum[" SUMU ", agcol, unique(NULL)]
;
subqret         = retrieve["( _agcol BY _bylist _w )",rel_list,aggr_lst,w:where,NULL]
                  {  aggr_lst:attr_list;
                     extattr(agcol,aggr_lst);
                  }
                | whereaggr
                | retrieve[" ( _agcol ) ",rel_list,aggr_lst,NULL,NULL]
                  {  aggr_lst:attr_list;
                     extattr(agcol,aggr_lst);
```

```
                    extrels(agcol,rel_list);
                }
;
whereaggrcol  =  attribute[" _r._a ", r:unit_rel_list, a:attr_name]
              |  epsilon
;
%right NOT
%left AND
%left OR
%left '*' '/'
%left '+' '-'
```

Appendix F

TQuel: Specification

```
BEGIN CONTEXT X
    rtt : rettuple;
    urel_list : unit_rel_list;
    rel_list : relation_list;
    all_attr : attr_list;
    proj_list : attr_list;
    xpr : expr_list;
END CONTEXT X

TQuel       =  { newCNTXT(X); }
               tmdelete[" DELETE _urel_list _x ", urel_list, x:valid3]
               {  a:attr_list;
                  get_all_attr(urel_list,all_attr);
               }
            |  { newCNTXT(X); }
               tminsert[" APPEND TO _urel_list _y ",urel_list, NULL,
                  y:insertstrm]
               {  a:attr_list;
                  get_all_attr(urel_list,all_attr);
               }
            |  { newCNTXT(X); }
               tmupdate[" REPLACE _urel_list _y ",urel_list, NULL,
                  y:updatestrm]
               {  a:attr_list;
                  get_all_attr(urel_list,all_attr);
```

```
                }
            |   { newCNTXT(X); }
                noop[" RETRIEVE _x ",x:retstmt]
            |   dfnCORR[" RANGE OF _v IS _r ", v:relation_list, r:relation]
;
retstmt     =   create[" INTO _x _y ", x:relation, y:rename]
            |   printstream[" _x ", x:display]
;
display     =   unique[" UNIQUE _r ", coalesce(r:rename,NULL,NULL)]
            |   sort[" _r SORT BY _a ", a:attr_list, r:rename]
            |   rename
;
rename      =   rename[" ( _p _c ", p:rettuple,c:process1]
;
process1    =   noop[" ) _c ",c:proc_cycl1]
;
proc_cycl1  =   compute[rewrite(xpr,[add,sub,div,mul,uminus,str2int,str2rel,
                    validat,validfrom,validto,xactionstart,xactionstop],x),
                    x:expr_list, proc_cycl1]
            |   aggregate[rewrite(xpr,[count,min,max,avg,sum,filter,groupby,
                    gt,ge,lt,le,eq,str2int],x),x:expr_list,proc_cycl1]
            |   noop[is_attr_list(xpr), valid1]
;
rettuple    =   list[ , , ",", retcolumn, NE]
;
retcolumn   =   noop[" _x = _y ", x:String]
                {   y:tquelval;
                    linknode(xpr,y);
                    linkcopy(proj_list,x);
                }
            |   noop[" _x = _y ", x:iattr]
                {   y:tquelval;
                    linknode(xpr,y);
                    linkcopy(proj_list,x);
                }
            |   noop[" _a ", NULL]
                {   a:attribute;
                    linknode(xpr,a);
```

```
                    linkcopy(proj_list,a);
                }
;
iattr           =   attr_name
                |   attribute[" _r._a ", r:unit_rel_list, a:attr_name]
;
retstrm1        =   retrieve[" _c ", rel_list, xpr, c:retcond,NULL]
;
retcond         =   and[" _w _n _a ", a:asof, and(n:when, w:where)]
;
delstrm         =   retrieve[" _y ",rel_list,all_attr,y:modcond,NULL]
;
insertstrm      =   rename[" ( _j ", proj_list, j:valid2]
;
process2        =   noop[" _rtt ) _c ",c:proc_cycl2]
;
proc_cycl2      =   compute[rewrite(xpr,[add,sub,div,mul,uminus,str2int,str2real,
                        str2str],x),x:expr_list, proc_cycl2]
                |   aggregate[rewrite(xpr,[count,min,max,avg,sum,filter,groupby,
                        gt,ge,lt,le,eq,str2int],x), x:expr_list, proc_cycl2]
                |   noop[is_attr_list(xpr), retstrm2]
;
retstrm2        =   retrieve[" _w ", rel_list, all_attr, w:modcond, NULL]
                |   epsilon
;
modcond         =   and[" _w _n ", asof("now",NULL), and(n:modwhen, w:where)]
;
where           =   noop[" WHERE _x ", x:predicate]
                |   epsilon
;
predicate       =   not[" NOT _x ", x:predicate]
                |   and[" _x AND _y ", x:predicate, y:predicate]
                |   or[" _x OR _y ", x:predicate, y:predicate]
                |   noop[" ( _x ) ", x:predicate]
                |   compare
                |   Boolean
;
compare         =   lteq[" _x <= _y ", x:tquelval, y:tquelval]
```

```
                | gteq[" _x >= _y ", x:tquelval, y:tquelval]
                | neq[" _x != _y ", x:tquelval, y:tquelval]
                | gt[" _x > _y ", x:tquelval, y:tquelval]
                | lt[" _x < _y ", x:tquelval, y:tquelval]
                | eq[" _x = _y ", x:tquelval, y:tquelval]
;
tquelval        = tquelexpr
                | temporalfn
                | String
;
tquelexpr       = add[" _x + _y ", x:tquelexpr, y:tquelexpr]
                | sub[" _x - _y ", x:tquelexpr, y:tquelexpr]
                | mul[" _x * _y ", x:tquelexpr, y:tquelexpr]
                | div[" _x / _y ", x:tquelexpr, y:tquelexpr]
                | uminus[" - _x ", x:tquelexpr]
                | noop[" ( _x ) ", x:tquelexpr]
                | exists[" ANY ( _x ) ", x:tquelexpr]
                | Numeric
                | attribute
;
temporalfn      = xactionstart[" transactionstart ( _r ) ", r:relation]
                | xactionstop[" transactionstop ( _r ) ", r:relation]
                | validat[" validat ( _r ) ", r:relation]
                | validfrom[" validfrom ( _r ) ", r:relation]
                | validto[" validto ( _r ) ", r:relation]
;
tquelaggr       = count[" COUNT ( _e _b ) ", e:tquelexpr, b:aggrgrp]
                | avg[" AVG ( _e _b ) ", e:tquelexpr, b:aggrgrp]
                | sum[" SUM ( _e _b ) ", e:tquelexpr, b:aggrgrp]
                | max[" MAX ( _e _b ) ", e:tquelexpr, b:aggrgrp]
                | min[" MIN ( _e _b ) ", e:tquelexpr, b:aggrgrp]
                | count[" COUNTU ( _e _b ) ", e:tquelexpr, unique(b:aggrgrp)]
                | avg[" AVGU ( _e _b ) ", e:tquelexpr, unique(b:aggrgrp)]
                | sum[" SUMU ( _e _b ) ", e:tquelexpr, unique(b:aggrgrp)]
;
aggrgrp         = groupby[" BY _a _w ", a:attr_list, w:select]
                | select
;
```

```
select          =  filter[" WHERE _x ", x:predicate, NULL]
                |  epsilon
;
attr_list       =  list[ , , ",", attribute, NE]
;
attribute       =  attr_name
                |  attribute[" _r._a ", r:unit_rel_list, a:attr_name]
                   { mergerelation(r,rel_list); }
;
attr_name       =  str2attr[' [a-z][a-zA-Z0-9_]* ']
;
relation_list   =  list[ , , "," , relation, NE]
;
unit_rel_list   =  list[ , , "," , relation, unit]
;
relation        =  str2rel[' [A-Z][a-zA-Z0-9_]* ']
;
String          =  str2str[' \"[a-zA-Z0-9 _;:,-]+\" ']
;
Boolean         =  str2Boolean['true']
                |  str2Boolean['false']
;
Numeric         =  LitNumeric
                |  tquelaggr
;
LitNumeric      =  Int
                |  Real
;
Int             =  str2int['[1-9]+[0-9]*']
;
Real            =  str2real[' [0-9]*.[0-9]+ ']
                |  str2real[' [0-9]+.[0-9]* ']
;
expr_list       =  list[ , , ",", tquelexpr, NE]
;
updatestrm      =  rename[" ( _a _p ", a:assgnlst,p:process3]
;
process3        =  noop[" ) _p ",p:proc_cycl3]
```

```
;
proc_cycl3      =  compute[rewrite(xpr,[add,sub,div,mul,uminus,str2int,str2str,
                      str2real],x),x:expr_list, proc_cycl3]
                |  aggregate[rewrite(xpr,[count,min,max,avg,sum,filter,groupby,
                      gt,ge,lt,le,eq],x), x:expr_list, proc_cycl3]
                |  noop[is_attr_list(xpr), valid4]
;
assgnlst        =  list[ , , ",", assgn, NE]
;
assgn           =  noop[" _x = _y ", x:String]
                   {  y:tquelval;
                      linknode(xpr,y);
                      linkcopy(proj_list,x);
                   }
;
when            =  when[" WHEN _tp ",tp:tpred]
                |  when[" _e ", overlapt(i,"now")]
                   {  e:epsilon;
                      i:i_expr;
                      gen_expr(rel_list,"overlapt","left",i);
                   }
;
modwhen         =  when[" WHEN _tp ",tp:tpred]
                |  when[" _e ", overlapt(i,"now")]
                   {  e:epsilon;
                      i:i_expr;
                      mergerelation(urel_list,rel_list);
                      gen_expr(rel_list,"overlapt","left",i);
                   }
;
valid1          =  valid[" VALID AT _e _r ",e:e_expr,NULL,r:retstrm1]
                |  valid[" VALID FROM _i TO _k _r ",i:e_expr,k:e_expr,r:retstrm1]
                |  valid[" _r ", i, NULL, r:retstrm1]
                   {  i:i_expr;
                      gen_expr(rel_list,"overlapt","left",i);
                   }
;
valid2          =  valid[" VALID AT _e _r ",e:e_expr,NULL,r:process2]
```

```
                | valid[" VALID FROM _i TO _k _r ",i:e_expr,k:e_expr,r:retstrm2]
                | valid[" _r ", "now", "forever", r:process2]
;
valid3          = valid[" VALID AT _e _r ",e:e_expr,NULL,r:delstrm]
                | valid[" VALID FROM _i TO _k _r ",i:e_expr,k:e_expr,r:delstrm]
                | valid[" _r ", "now", endof(urel_list), r:delstrm]
;
valid4          = valid[" VALID AT _e _r ",e:e_expr,NULL,r:retstrm2]
                | valid[" VALID FROM _i TO _k _r ",i:e_expr,k:e_expr,r:retstrm2]
                | valid[" _r ", "now", endof(urel_list), r:retstrm2]
;
asof            = asof[" AS OF _a ",a:e_expr,NULL]
                | asof[" AS OF _a ",a:interval,NULL]
                | asof[" AS OF _a THROUGH _b ",a:e_expr,b:e_expr]
                | asof[" _e ", "now", NULL]
                  {  e:epsilon; skip(); }
;
tpred           = precede[" _a PRECEDE _b ", a:evnt_intrvl,b:evnt_intrvl]
                | overlapb[" _a OVERLAPB _b ",a:evnt_intrvl, b:evnt_intrvl]
                | sametime[" _a EQUAL _b ", a:evnt_intrvl,b:evnt_intrvl]
                | and[" _x AND _y ", x:tpred, y:tpred]
                | or[" _x OR _y ", x:tpred, y:tpred]
                | not[" NOT _x ", x:tpred]
                | noop[" ( _x ) ", x:tpred]
;
i_expr          = overlapt[" _a OVERLAPI _b ", a:evnt_intrvl,b:evnt_intrvl ]
                | extend[" _a EXTEND _b ", a:evnt_intrvl,b:evnt_intrvl ]
                | noop[" ( _i ) ", i:i_expr]
                | interval
;
e_expr          = beginof[" BEGIN OF _e ", e:evnt_intrvl ]
                | endof[" END OF _e ", e:evnt_intrvl ]
                | noop[" ( _e ) ", e:e_expr]
;
evnt_intrvl     = i_expr
                | e_expr
;
interval        = tconst1
```

```
                | relation
                | String
;
tconst1         = str2str[' NOW ']
                | str2str[' FOREVER ']
                | str2str[' BEGINNING ']
;
%right NOT BEGIN END OF
%left OVERLAPB OVERLAPI EXTEND
%left PRECEDE EQUAL
%left AND OR
%left '*' '/'
%left '+' '-'
```

Bibliography

[Ara88] G. Arango. *Domain Engineering for Software Reuse*. PhD thesis, University of California Irvine, 1988.

[Ara89] G. Arango. Domain analysis - from art form to engineering discipline. *Software Engineering Notes*, pages 152–159, May 1989.

[ASU86] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley Publishing Company, Inc., 1986.

[Bat87a] D. Batory. Extensible cost models and query optimization in Genesis. *Database Engineering*, pages 206–212, November 1987.

[Bat87b] D. Batory. Principles of database management system extensibility. *Database Engineering*, pages 100–106, June 1987.

[Bat88] D. Batory. Building blocks of database management systems. Technical Report TR-87-23, The University of Texas at Austin, February 1988.

[BBG+90] D. Batory, J. Barnett, J. Garza, K. Smith, K. Tsukuda, B. Twichell, and T. Wise. Genesis: An extensible database management system. In S. Zdonik and D. Maier, editors, *Readings in Object Oriented Database Systems*, chapter 7.4. Morgan Kaufman, 1990.

[BG89] L. Becker and R. Güting. Rule-based optimization and query processing in an extensible geometric database system. Technical Report CS Technical Report 312, Universität Dortmund, August 1989.

[BLW88] D. Batory, T. Leung, and T. Wise. Implementation concepts for an extensible data model and data language. *ACM Transactions on Database Systems*, 13(3):231–262, September 1988.

[Car87] M. Carey, editor. *IEEE Database Engineering Special Issue on Extensible Database Systems*. IEEE Computer Society, June 1987.

[CDG+90] M. Carey, D. DeWitt, G. Graefe, D. Haight, J. Richardson, D. Schuh, E. Shekita, and S. Vandenberg. The EXODUS extensible DBMS project: An overview. In S. Zdonik and D. Maier, editors, *Readings in Object Oriented Database Systems*, chapter 7.3. Morgan Kaufman, 1990.

[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.

[Dat84] C. J. Date. A critique of the SQL database language. *SIGMOD Record*, pages 8–54, September 1984.

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

[Dev92] P. Devanbu. Genoa—a customizable, language- and front-end independent code analyzer. In *Proceedings of the Fourteenth International Conference on Software Engineering*, pages 307–319, 1992.

[DRW94] P. Devanbu, D. Rosenblum, and A. Wolf. Automated construction of testing and analysis tools. In *Proceedings of the Sixteenth International Conference on Software Engineering*, 1994.

[DS86] U. Dayal and J. Smith. PROBE: A knowledge-oriented database management system. In M. Brodie and J. Mylopoulos, editors, *On Knowledge Base Management Systems*, chapter 19. Springer-Verlag, 1986.

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

[Gar90] D. Garlan. The role of formal reusable frameworks. In *Proceedings of the ACM SIGSOFT International Workshop on Formal Methods in Software Development*, pages 42–44, May 1990.

[GCK+89] G. Gardarin, J-P. Cheiney, G. Kiernan, D. Pastre, and H. Stora. Managing complex objects in an extensible relational DBMS. In *Proceedings of the Fifteenth International Conference on Very Large Data Bases*, pages 55–65, 1989.

[GD87] G. Graefe and D. DeWitt. The EXODUS optimizer generator. In *ACM SIGMOD*, pages 160–172, 1987.

[GG86] R. Griswold and M. Griswold. *The Implementation of the Icon Programming Language*. Princeton University Press, 1986.

[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.

[Gog89] J. Goguen. Principles of parameterized programming. In T. Biggerstaff and A. Perlis, editors, *Software Reusability, vol. 1*, chapter 7. Addison Wesley Publishing Company, Inc., 1989. ACM Press Frontier Series; NY, NY.

[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.

[HCF+88] L.M. Haas, W.F. Cody, J.C. Freytag, G. Lapis, B.G. Lindsay, G.M. Lohman, K. Ono, and H. Pirahesh. An extensible processor for an extended relational query language. Technical Report RJ 6182 (60892), IBM Almaden Research Center, April 1988.

[Hen77a] G. Hendrix. Human engineering for applied natural language processing. In *International Joint Conference on Artificial Intelligence*, pages 183–191, 1977.

[Hen77b] G. Hendrix. LIFER: A natural language interface facility. *SIGART Newsletter*, pages 25–26, February 1977.

[Hen77c] 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.

[HSSS78] G. Hendrix, E. Sacerdoti, D. Sagalowicz, and J. Slocum. Developing a natural language interface to complex data. *ACM Transactions on Database Systems*, pages 105–147, June 1978.

[HSW75] G. Held, M. Stonebraker, and E. Wong. Ingres—a relational database system. In *National Computer Conference*, pages 409–416, 1975.

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

[Jen94] C. Jensen. A consensus glossary of temporal database concepts. *SIGMOD Record*, pages 52–64, March 1994.

[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.

[KBB+87] W. Kim, N. Ballou, J. Banerjee, H. Chou, J. Garza, and D. Woelk. Features of the ORION object-oriented database system. Technical Report ACA-ST-308-87, Microelectronics and Computer Technology Corporation, September 1987.

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

[KS86] H. Korth and A. Silberschatz. *Database System Concepts*. McGraw-Hill Book Company, 1986.

[LLPS91] G. Lohman, B. Lindsay, H. Pirahesh, and K. Schiefer. Extensions to Starburst: Objects, types, functions, and rules. *Communications of the ACM*, pages 94–109, October 1991.

[LMP87] B. Lindsay, J. McPherson, and H. Pirahesh. A data management extension architecture. In *ACM SIGMOD*, pages 220–226, May 1987.

[LR89] J. Lingat and C. Rolland. PROQUEL: a PROgramming QUEry Language. In *Proceedings of the Second International Workshop on Database Programming Languages*, pages 281–295, 1989.

[LS86] M. Lesk and E. Schmidt. Lex—a lexical analyzer generator. In *UNIX Programmer's Manual: Supplementary Documents 1*. University of California, Berkeley, 1986.

[Man88] N. Mano. Modeling of data-processing software for generating and reusing their programs. In *Proceedings of the Tenth International Conference on Software Engineering*, pages 231–240, April 1988.

[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.

[Nei84] J. Neighbors. The Draco approach to constructing software from reusable components. *IEEE Transactions on Software Engineering*, pages 564–574, September 1984.

[PDA91] R. Prieto-Diaz and G. Arango. *Domain Analysis and Software Systems Modeling*. IEEE Computer Society Press, 1991.

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

[RKS85] M. Roth, H. Korth, and A. Silberschatz. Null values in ¬1NF relational databases. Technical Report TR-85-32, The University of Texas at Austin, December 1985.

[RKS88] M. Roth, H. Korth, and A. Silberschatz. Extended algebra and calculus for nested relational databases. *ACM Transactions on Database Systems*, 13(4):389–417, December 1988.

[RKS89] M. A. Roth, H. F. Korth, and A. Silberschatz. Null values in nested relational databases. *Acta Informatica*, 26(7):615–642, September 1989.

[RS87] L. Rowe and M. Stonebraker. The Postgres data model. In *Proceedings of the Thirteenth International Conference on Very Large Data Bases*, pages 83–96, 1987.

[Ser86] Servio Logic Development Corporation, Beaverton, Oregon. *Programming in OPAL*, 1986.

[SHP88] M. Stonebraker, E. Hanson, and S. Potamianos. The Postgres rule manager. *IEEE Transactions on Software Engineering*, 14(7):897–907, July 1988.

[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.

[SR86] M. Stonebraker and L. Rowe. The design of Postgres. In *ACM SIGMOD*, pages 340–355, 1986.

[SS87] A. Segev and A. Shoshani. Logical modeling of temporal data. In *ACM SIGMOD*, pages 454–466, May 1987.

[Sto86a] M. Stonebraker. Inclusion of new types in relational database systems. In *Proceedings Second International Conference on Database Engineering*, February 1986.

[Sto86b] M. Stonebraker. Object management in Postgres using procedures. In *Proceedings of the 1986 International Workshop on Object-Oriented Database Systems*, September 1986.

[Sto87] M. Stonebraker. The design of the Postgres storage system. In *Proceedings of the Thirteenth International Conference on Very Large Data Bases*, pages 289–300, 1987.

[SWKH76] M. Stonebraker, E. Wong, P. Kreps, and G. Held. The design and implementation of Ingres. *ACM Transactions on Database Systems*, 1(3):189–222, September 1976.

[Ull82] J. Ullman. *Principles of Database Systems*, chapter 8. Computer Science Press, Inc., second edition, 1982. chapter 6 on Relational Query Languages.

[Ull88] J. Ullman. *Principles of Database and Knowledge-Base Systems*, pages 271–288. Computer Science Press, Inc., 1988. subchapters 2.7 on Object-Oriented Data Model and 5.6-5.7 on OPAL.

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

[Zan83] C. Zaniolo. The database language GEM. In *ACM SIGMOD*, pages 207–218, 1983.

Vita

Emilia Elizabeth Villarreal was born in Aransas Pass, Texas, the daughter of Esperanza Villarreal and Francisco Villarreal. After graduation from McAllen High School, she enrolled in the Massachusetts Institute of Technology, in Boston, Massachusetts, where she received the Bachelor of Science degree in February, 1980. During the following years, she was employed at Technology + Economics in Cambridge, Massachusetts and the Boston Consulting Group in Boston, Massachusetts. In May, 1987 she received the Master of Science degree from The University of Texas at Austin. Subsequently, she was employed at the Microelectronics and Computer Technology Corporation, in Austin, Texas, and at International Business Machines in Austin, Texas. She completed her doctoral work in August, 1994.

Permanent Address: 10120 Huer Huero Rd.
Creston, CA 93432

This dissertation was typeset with LaTeX 2_ε[1] by the author.

[1] LaTeX 2_ε is an extension of LaTeX. LaTeX is a collection of macros for TeX. TeX is a trademark of the American Mathematical Society. The macros used in formatting this dissertation were written by Dinesh Das, Department of Computer Sciences, The University of Texas at Austin.