PQL Language Guide and Reference

David A. Holland

August 23, 2009
Chapter 1

Introduction

1.1 PQL

PQL (Path Query Language, or “pickle”) is a query language for semistructured data organized as a directed (possibly cyclic) graph.

Semistructured data is distinguished from structured data as found in a typical RDBMS by having no particular formal schema; it is distinguished from unstructured data such as plain text files by nonetheless still being organized in terms of values and relationships. In PQL, data is organized into objects with named fields, akin to C structures. Fields can contain values or point to other objects; the set of all objects forms a directed graph. This is a very general model suitable for encoding many kinds of data structured in many different ways. It is also a very flexible model allowing the integration of heterogeneous data from many sources.

The PQL query model is based on following paths through the object graph to find and retrieve data. The typical query returns a set of values. This distinguishes it from some other graph-related query languages that are based on graph transformation, where a typical query returns a new graph. (Examples include GOOD and StruQL.) PQL is also distinct from tools meant to store and retrieve many individual graphs each considered as a database value, by pattern matching or graph isomorphism or whatever. (Examples include GraphGrep and GraphQL.) While PQL can be used for both of these additional purposes, it is not (for the present time at least) especially well suited to them and some other tool may be a better choice. PQL is also declarative, in the same sense SQL is, which distinguishes it from graph query languages based on procedural structural recursion. (Examples include UnQL and CouchDB.)

PQL is derived from Lorel, a similar language introduced in 1997 as part of the Lore database project at Stanford. The PQL engine was originally intended as a reimplementation of Lorel; thus, there are relatively few visible differences between Lorel and PQL. In the course of defining PQL’s semantics carefully it became clear that some of the language-level magic in Lorel was unsupportable and had to be abandoned. Mostly, however, PQL extends Lorel. A full discussion may be found in Chapter ??.

PQL is also a descendent of SQL, via the object query language OQL that served as a basis for Lorel. Thus the general structure and appearance of queries is familiar. PQL is, however, more like a conventional programming language
in feel; in particular, it does not share SQL’s tendency towards extraneous noise keywords and pointless synonyms, and it is tighter about identifiers and other basic syntactic elements.

Because PQL works on data for which there is no formal schema, it contains many features to allow queries to still work effectively on data whose structure is not uniform and perhaps not fully known. It also contains features for inspecting the structure of the data itself, and for using the structure of the data to guide or restrict queries. These considerations are likely less familiar and are discussed at some length in this manual, both from a user perspective and from the perspective of the detailed language semantics.

1.2 Purpose of This Document

This manual, the PQL Language Guide and Reference, is divided into two primary sections. The first section, the Guide to Using PQL, contains an introduction to the PQL query language and its uses. The second section, the PQL Language Reference, provides a comprehensive discussion of language features organized by topic.

The intent is that novices may learn how to use PQL by reading the Guide, whereas experienced users may get answers to specific questions quickly via the Reference.

The Guide does not in general explain every last detail of the language, preferring instead to explain the broad outline and common usage. Cross-references within the Guide provide further reading in the Reference if a full treatment of the gory details is desired.

It is assumed that the reader is reasonably familiar with basic concepts of databases and database access, such as transactions and data types. The reader should also be comfortable with general concepts of programming and should have at least a nodding acquaintance with C.

Some familiarity with SQL may be helpful but is not required. Specific knowledge of OQL or Lorel is not assumed.

This document is not meant to explain how to use any particular program, tool, or database engine that may support PQL queries.

1.3 Typographical Conventions

Material appearing on a computer screen (e.g. that one might type in or that the system might print out) is set in typewriter font. Important technical terms are set in italics when first introduced or where fully defined.
Chapter 2

Using PQL

This chapter explains how to use PQL and how to write simple (and not so simple) queries.

2.1 Graph-Oriented Data

The purpose of PQL is to allow making queries on graph-structured data. Graph-structured data is not well handled by relational databases or SQL; while one can represent a graph in a relational database as an adjacency list, this is not a particularly natural representation to work with, and even simple queries like “is A reachable from B?” must be translated by hand into extremely opaque SQL incantations.

Formally, PQL is a query language for semistructured data, which means that there is no schema for its database. Data is attached to objects, like in a traditional object database, but the objects are untyped: any particular object may contain an arbitrary collection of named fields, and each field may contain a value of arbitrary type. These values can be atoms (ordinary values) or references to other objects.

This is a good model to use for graph-structured data; the interconnections among objects form a directed graph, and any directed graph can be stored as a collection of objects. Meanwhile, the data attached to the graph nodes can be stored in other fields of these objects and addressed by searching the graph.

This model is a superset of the relational model; individual objects are single database rows, object fields name columns, and groups of similar objects can be thought of as forming tables. See Section ?? for further details.

The lack of any formal schema is a conscious design decision; it makes writing queries harder, but also allows queries to be more powerful, and allows PQL to be used with heterogeneous data or to integrate data from multiple sources. The chief drawback is that queries that do not match the database structure will return no results rather than failing with a type error. However, one can also write queries that are polymorphic over different database structures (which might all exist at once in the data) — this would be impossible, or at least very difficult, with a strong schema and type system.

This property does make PQL less desirable for highly structured graphs such as those found in compiler backends. For this reason it is intended that some form of schema support (perhaps for partial or approximate schemas) will be developed in the future.
2.2 Purpose of PQL

PQL is meant for extracting material from graph-structured data, the same way SQL is meant for extracting material from tabular data. The database or databases behind it are organized as a graph of interconnected objects, and querying is based on following paths through that graph to find values.

This is important, because traditional relational databases are not well suited to handling graph-structured data, and writing queries that retrieve data based on graph structure is at best difficult. Similarly, the XML databases that have become popular in recent years handle only trees, not graphs, and while a number of query languages exist for handling XML document trees, none of them generalize well to graphs. The current alternative to these is to use RDF and SPARQL. Unfortunately, SPARQL is quite limited; PQL is strictly more powerful.

This does not mean that PQL will walk the dog or wash the windows.

PQL is not meant to be a query language for a database of graphs, where graphs are the data and are to be retrieved as objects by subgraph matching or other graph algorithms. PQL can be used for this purpose — probably — but it would be awkward and cumbersome to do so, and the internals of the current implementation are not intended for such workloads.

PQL is also not meant to be a query language for extracting subgraphs from a large graph, or computing graph homomorphisms from the database graph to some simpler but analogous representation. Currently, PQL cannot easily be used for this purpose at all, because its select statement can only return trees, not graphs. This problem is expected to be rectified in the future, but even then, PQL will still probably not be very suitable.

Other query languages exist that will be better for these purposes; some of these are mentioned in Section 2.2.1. However, because PQL's data model can be considered a superset of the relational model, as mentioned in the previous section, PQL is potentially suitable for use with data sets where only some data is graph-structured and the rest is tabular. PQL may also be an attractive choice for tabular data where the schema is not fixed.

2.2.1 Example Applications

PQL was developed in conjunction with the PASS data provenance project [1] and was first intended for making provenance and ancestry queries. Ancestry (of data or of humans) tends to form a graph rather than a tree and a graph-based query language is required [2]. This application provides one source of example data and queries.

PQL's ancestor Lorel was developed less for graphs than for heterogeneous tabular or semi-hierarchical data. The motivating examples in the Lorel paper [3] are based on a hypothetical restaurant database. This provides another source of example queries.

For a further set of examples we will hypothesize that PQL has been connected up to the Internet Movie Database [4]. In this context PQL can be used to explore the interconnections among IMDB entities and work with concepts (such as Bacon numbers [5]) that are hard to handle in SQL.

Most of the examples used in this document are restaurant-related, partly as a result of over-immersion in the Lorel examples and partly because they tend to be short and easy to follow. Others, typically those illustrating a spe-
2.2. PURPOSE OF PQL

Figure 2.1: Fragment of a possible restaurant database. Objects are red boxes and plain values are blue diamonds.

Specific point of the language, are ancestry-related or IMDB-based.

2.2.2 Example Database

Figure ?? shows a small example database organized according to PQL’s data model. The red boxes are objects; the arrows represent fields of those objects. The blue diamonds are plain values. The global variable Guide points to the leftmost object.

This figure also serves to illustrate other points about PQL’s data model. First, objects are not themselves named. While we think of the leftmost object as the “Guide” object, and the next three objects to the right as restaurants, they are only really identified that way by their position in the graph.

Second, graph edges and object fields are the same thing. The .restaurant fields of the Guide object are the graph edges that lead to the restaurant objects. (Though the plain values are not formally objects, access to them is functionally equivalent and we consider them part of the graph.)

Third, the graph need not be acyclic. Note the circular .nearby relationship of the top two restaurants. The query engine must have logic to avoid trundling around such cycles forever. This is discussed in Section ??.

Fourth, the database is anchored by global variables, such as Guide. These serve as roots for garbage collection: unreachable portions of the graph are discarded. They also serve as starting points for queries. In fact, all queries must start from a global variable. This seems restrictive at first but is actually a feature: the storage backend can easily provide extra dynamically computed or otherwise magic objects to point to interesting things, and provide them to PQL as globals.

Finally, while the edges in the graph are di-
rected, they are bidirectional. They can be followed forwards from the restaurants to the addresses, or backwards from the addresses back to the restaurants. (It is up to the backend storage code, or the data, to decide which direction is forwards.)

### 2.2.3 Example Query

The following query retrieves the names of Japanese restaurants.

```sql
select N
from Guide.restaurant as R,
    R.name as N,
    R.type as T
where T = "Japanese"
```

It starts at the global variable `Guide`, follows the fields/edges named `.restaurant` (note that there are several of these), and calls the objects it finds `R`. Then it follows the `.name` and `.type` edges from those objects to find all corresponding values for `N` and `T`. This produces a collection of pairs with values for `N` and `T`. (Because `R` is not further used, it can be dropped.)

Each such pair is a candidate result, a possible assignment of values to the variables used in the query. The collection of candidates is then iterated; each one is used one at a time to substitute values for `N` and `T`. These values are used to evaluate the where-clause; if that yields `true`, they are used to evaluate the select-clause and the results added to the set of values returned.

If this query is applied to the example database drawn in Figure ??, it will return the name “Sushi on Bloor”. The other names would not be returned because their corresponding types do not match.

### 2.3 Design Principles

The design of Lorel, and thus also of PQL, was largely motivated by a number of fundamental considerations.

First is that querying over semistructured data requires a certain degree of robustness: the data may not have a uniform structure, and a query must be able to inspect portions of the data that have one kind of structure without bombing on other portions with a different structure. Ideally, in fact, a query ought to be able to inspect multiple structures in one query, with some parts of the query logic responding to some regions of the data and other parts to other regions. As a simple example, the apparently nonsensical expression `X > 3 or X = "abc"` is perfectly legal (if `X` is a value that came out of the database) and must be true if `X` is e.g. 5.

Similarly, exact type matching is not required; comparisons that can be made to make sense should be interpreted so they make sense, and those that cannot, such as the subexpression `X = "abc"` under the substitution of 5 for `X`, should simply be false; they are not errors. This means for example that the integer value 3 is equal to the string value "3". Type conversions are discussed in section ?? and the complete set of possible conversions can be found in section ??.

The principle expressed in the Lorel paper is that no syntactically valid query should fail at runtime merely because it fails to match the data that happens to be in the database.

That said, a further principle in PQL is that any query we can tell in advance will never work properly should be rejected at query compilation time. Thus, even though `X > 3 or X = "abc"` is valid, the expression `3 = "abc"` is rejected by the type checker.
2.4. TUPLES, SETS, AND PATHS

The second major principle is that if the same text is repeated, it should mean the same thing each time, unless an explicit notation is made to the contrary. For example, in the query

```sql
select R.name
from Guide.restaurant as R
where R.address.state = "MA"
  or R.address.state = "NH"
```

the PQL engine should not do anything that leads to using more than one value of `R.address.state` at the same time, even for restaurants with multiple addresses. The precise need for this rule is difficult to explain without further background. It is, however, fundamental to the language structure, and is discussed in varying degrees of detail in sections ?? and ??.

2.4 Tuples, Sets, and Paths

The database consists of objects and atoms (plain values); however, other types of values can arise during query processing. These values are always transient; they do not conform to the objects-and-fields data model and cannot be stored in the database. (They can, however, both be converted to objects for storage.)

There are several kinds of non-database values. A **tuple** is an ordered and fixed-size group of values. While tuples scarcely appear in PQL at the language level, they are fundamental to the internals of the engine implementation.

A **set** (really a multiset or “bag”) is an arbitrary collection of values. A **sequence** is a set that has been sorted or otherwise has a fixed ordering.

Paths are represented as sequences of a primitive type called **pathelement**, which identifies a single database edge.

There is an additional internal type called **distinguisher** that arises internally as part of the implementation of certain language constructs; these values should never escape from the engine.

2.4.1 Sets

Sets (and sequences) arise in queries where multiple values appear at once. They can be constructed explicitly, and the result of a `select` expression is a set or sequence. However, they appear most frequently when a query follows an edge in the database. Because the data model allows multiple fields of the same name, looking up the field of an object might yield more than one value. (Or it might yield no values.) And because we have no schema, we do not know a priori if exactly one value is to be expected. Thus, the result of every field lookup/edge traversal is a set. When a set or sequence needs to be stored into the database, it is converted to an object.

A set is converted to an object with however many fields of some (all the same) name; thus, reading that field name back from that object restores the set. A sequence is converted to an object with numbered fields (starting at 0) so as to preserve the ordering.

Because of these properties (and because sequences are a special kind of set) we say that “everything is a set”; much of the mechanism in PQL exists to deal with this property. This is discussed further in Section ??.

2.4.2 Paths

Path values hold specific physical routes through the database graph. They can be created while traversing the database, and then used to inspect the route taken, repeat an analogous route from
a different starting point, or other things. These mechanisms are discussed below in Section ??.

Note that path values are not the same as path expressions. Path expressions are object field references as seen in the examples already given. (They will be defined properly in a moment.) They specify groups of physical routes through the database; when evaluated, they return the values found at the end of the routes, which because they come from the database are never path values.

Path values are generated by additional annotations on path expressions (see Section ??) and represent individual physical routes.

That is, path expressions do not (directly) yield path values. This is confusing at first, but an unavoidable consequence of supporting path values at all without making the language syntax horrendous.

(Note that most graph- and tree-oriented query languages do not support path values at all and certainly do not make them first-class language-level entities.)

### 2.5 Simple Expressions

Expressions in PQL have a similar flavor to SQL. However, in an effort to make the syntax saner, some of the lexical constructs of C have been adopted – string constants, for example.

#### 2.5.1 Constants

String constants are written with are written with double quotes (" ") and support C-style escape sequences for encoding control characters. Integer constants can likewise be written in hex or octal in the C manner by giving a prefix of 0 or 0x, and floating point constants take the C form.

There is a null value, but it has different semantics from SQL’s null value. SQL’s null value represents an unknown value; PQL’s nil represents no value, and consequently behaves somewhat differently. This is discussed in Section ??, ??, and belabored to death in Sections ??, ??, and ??.

#### 2.5.2 Operators

PQL has unary prefix operators, binary infix operators, and function calls with arguments in parentheses. The operators have essentially the SQL spellings: the logical and set operators are words rather than symbols (and, union, etc.) and equality is tested with = or <>.

PQL also provides like, glob and grep for string matching. (These perform SQL-like, shell-like, and regular expression matches respectively.) PQL also defines a soundex operators for approximate spelling matching; this is currently unimplemented. String concatenation is performed with ++.

For the complete list of operators see Section ??.

#### 2.5.3 Logical Quantifiers

There are no loops in PQL, because PQL is declarative rather than procedural. To iterate explicitly over sets, the logical quantifiers for all and exists are used.

The expression for all X in S: X > 3 is true if every element in S is greater than 3; the quantifier variable X is bound to each value of S in turn to evaluate the predicate after the colon. The expression exists X in S: X >
2.6. WRITING BASIC QUERIES

3 is true if there is at least one element in \( S \) greater than 3.

There is also a shorthand form of these expressions: \( 3 < \text{all} \) (or any) \( S \). This is translated internally into the long form.

The long form can be used with any boolean expression; the short form can be used with any (single) comparison operator. Note that the reverse order of the short syntax, all \( S > 3 \), is not supported.

To find restaurants whose addresses are all in Cambridge, one might write:

```plaintext
select N from Guide.restaurant as R,
       R.name as N,
       R.address.city as C
group by R
having
   for all C1 in C: C1 = "Cambridge"
```

The `group by` operation is used to collect \( C \) into a set (much like in SQL) and is discussed in Section ??.

2.6 Writing Basic Queries

There are two fundamental elements to queries in PQL. The first is the path expression; the second is the selection.

2.6.1 Basic Path Expressions

Path expressions are the heart of the PQL syntax. Think of path expressions as a generalization of the structure member reference found in languages like C and Java. That is, a simple path of the form \( A.b \) means “retrieve the fields \( b \) of the object \( A \).” (The analogy is not perfect because fields can be multiply defined in PQL.)

A longer path expression follows a longer chain: \( A.b.c.d \) starts from the object \( A \) and dereferences three fields, passing through two (sets of) intermediate objects before retrieving a (set of) final value(s).

The string \( .b \) in \( A.b \) is called a path step; it matches an object field. Thus, the term path step can be loosely used to refer to the field (or, equivalently, graph edge) it matches.

2.6.2 Selection

The `select` syntax is essentially the same as in SQL, except that PQL does not support all the bells and whistles. The general form is:

```plaintext
select outputs
from sources
where condition
group by variables
having condition
order by expressions
```

In SQL, broadly, the sources are tables and the outputs are column names. In PQL, the sources are path expressions and the outputs can be anything computed from the sources.

For syntactic convenience PQL allows paths to be placed outside the from-clause. These paths are moved to the from-clause and implicitly iterated.

Like in SQL, the model is that the sources are evaluated first, taking a cartesian product of the results. The results are then filtered by the condition expression. If necessary, these results are grouped and then filtered again by the second condition. The surviving results are then used to evaluate the output expressions.

This is perhaps best thought of, however, in the OQL fashion of iteration: the sources name sets, and each of the sets is in turn iterated, so
CHAPTER 2. USING PQL

the from-clause is a list of nested loops. The syntax from Guide.restaurant as R can be thought of as implying a for-loop where R is the iteration variable. (In some query languages, such as XQuery for XML, this iteration is explicit.)

In the earlier example

```plaintext
select N
from Guide.restaurant as R,
    R.name as N,
    R.address.city as C
group by R
having
  for all C1 in C: C1 = "Cambridge"
```

the restaurants, names, and cities are all iterated. The group by clause combines results for the same restaurant, collecting the corresponding name and city values together into sets. (This is discussed further in Section ??.)

The iteration behavior of PQL is comparable to the behavior in SQL, even though we do not usually think of it as iteration in that context.

Note, however, that unlike in SQL, it is essential to use the as R syntax to bind a variable name to each path in the from-clause. One cannot use the edge name as a variable like one can use column names in SQL. (While conceivably the parser could interpret a plain from Guide.restaurant as from Guide.restaurant as restaurant, it does not.)

2.6.3 Detailed Example

Here is a simple query. It searches the restaurant database for cheap eats.

```plaintext
select N
from Guide.restaurant as R,
    R.name as N
where R.price = "cheap"
```

The evaluation (conceptually) proceeds as follows.

1. We look up the global variable Guide. This gives us the Guide object that points to all the restaurants.

2. We follow the restaurant edge from the Guide object. This gives us one restaurant object for every restaurant in the database.

3. From each restaurant object, we follow the name edge. This gives us, for each restaurant, some number (possibly zero, possibly several, but usually one) name. These names are probably all strings, but maybe if the database contains some weirdness some of them might be objects.

4. Now, because the path R.price is to be implicitly iterated, we follow the price edge from each restaurant object. Now we iterate these. This gives us a group of prices for each restaurant and each name of that restaurant.

5. For each restaurant, each name, and each price, we repeat the remaining steps.

6. If the price value is the string "cheap", the comparison is true. If it is not (including if it is a dollar value represented as a number, or whatever else) the comparison is construed as false.

7. If the comparison and thus the whole where-clause is not true, give up on this restaurant, name, and price and continue with the next one, from step 4.
2.8. EVERYTHING IS A SET

8. Now we take the name, which was iterated by N, and add it to the set of results.

9. Finally, after we have generated all the results, we return the result set as the value of the select expression.

Note that implicitly iterating the price can result in getting extra results in the output: if a restaurant has two prices and they are both “cheap”, both will match and the name will be added to the results twice. This may not be what was intended. On the other hand, if one writes the query in this form it probably means one expects (or knows) there to be exactly one price, so in the common case this makes no difference. If it matters one can use group-by to inspect all prices in the aggregate (discussed in the next section) or filter the results by using select distinct.

2.7 Group By

Sometimes one wants to compute or inspect aggregates over multiple results. The group by feature allows this.

In the earlier example

```pql
select N
from Guide.restaurant as R,
    R.name as N,
    R.address.city as C
group by R
having
    for all C1 in C: C1 = "Cambridge"
```

we group by R; this collects all the results whose R value is the same into groups. Each group has a single value for R; the values of other variables are collected into sets. Thus C in the having-clause is a set and can be quantified over; N in the select-clause is also a set.

Note that this group-by is more powerful than SQL; since in SQL columns cannot be multivalued, all that can be applied to columns not listed in the group-by clause is a small set of specially licensed aggregator functions, like count. PQL supports these aggregator functions but also allows arbitrary inspection of the grouped values, such as found in this example.

2.8 Everything is a Set

this section withdrawn

2.9 Issuing Queries

Right now, the only PQL statements that can be issued are queries that return values. In the future, a language for update languages is planned; this will result in further statement types being defined.

Furthermore, right now there are no PQL statements for controlling transactions; these will be added in the near future.

This section will be expanded once these features are available.

2.10 Query Results

A query returns a value. In general, queries will be selections or at least path expressions, and will thus return sets. The value returned to the PQL client or printed for the user will thus generally be a set.

However, because sets are not allowed to exist in the database, if the result of a selection is
assigned to a database variable or object field, it will be converted to a database object instead: a struct with one field per member of the set.

If those fields are tuples (as will be the case if the selection generates multiple values) each tuple will in turn be converted to a database object as well, with one field per column.

The names of these fields can be specified in the same way that one specifies result column names in SQL: by appending as name to the expression in the select-clause. An additional (QQL-derived?) syntax for this is also supported: name:value.

If edge names are not provided for some or all values, the PQL engine chooses a name based on the path edge used to fetch the value, if one exists. When no suitable name can be extracted, the string default is used.

Consider this example query:

```sql
select R.name, R.price
from Guide.restaurant as R
```

This query returns a set of pairs. If this is stored into the database, it will be converted to an object (for the set) pointing to many objects (one for each pair) each holding a name and price.

The pair objects will have name and price fields because those edge names fetched the values. The set object will have fields named restaurant because the subobjects all came from R, which was fetched via an edge called restaurant. If the values in the select clause were not all derived from a single source, the name default would be used instead.

The precise rules for choosing default names are given in Sections ?? and ??.

The new function, which constructs fresh database objects, allows the same edge-naming syntax for its arguments: new(height:3, width:5) creates an object with two fields, height and width, with the values 3 and 5 respectively.

### 2.11 More on Path Expressions

#### 2.11.1 Matching

For better handling of semistructured data, Lorel and PQL generalize path expressions so they can match patterns of edges rather than only specific sequences. There are two matching mechanisms for path steps, which are independent: matching of edge names and matching of edge sequences.

To match path edge names, use the metacharacters % and _. These behave the same as in the like matching operator, described in Section ??.

Thus for example the path

```sql
Guide.restaurant.address.zip%
```

would match both zip and zipcode fields in the restaurant address.

This is the only matching behavior supported for edge names. However, path variables (introduced below) can be used for more elaborate matching if needed. This is discussed in section ??.

To match path edge sequences, PQL provides full regular expression support. The regular expression operators are the same as in Unix text regular expressions: concatenation names a sequence, | names alternatives, ? marks an optional element, and * and + name repetition zero or more, or one or more (respectively) times, and parentheses () are used for grouping. The precedence of these operators is the same as in egrep. Thus for example R.nearby* returns all the restaurants reachable from R by zero or more “nearby” steps, and R(.nearby|.nextdoor)+ follows
one or more steps that can be either “nearby” or “nextdoor”.

The important difference between path regular expressions in PQL and normal regular expressions in Unix is that the elements of the regular expression are whole path edges rather than characters. Thus, A.a* matches A.a.a.a but not A.abc, while A.a% matches A.abc but not A.a.a.a.

Special shorthand syntax is provided for the very common construct “zero or more arbitrary edges”. (This is effectively the only matching expression available in most path-based query languages.) It can be written as .# instead of the longer form (.%)*.

2.11.2 Going Backwards

The backwards direction of an edge is named by appending the reserved word -of. The spelling -OF is also accepted. Thus, given Guide.restaurant as R, the path R.address as A fetches an address into A and A.address-of yields R again.

The -of particle is not properly part of the edge name and is therefore not matched by the % metacharacter. This is intentional so that repeated edge sequences do not, by default at least, vacillate back and forth across the same edge indefinitely. The edge expression %-of can be used to follow all backwards edges; thus the path expression (%| %-of) follows all edges, both backwards and forwards.

2.11.3 Non-Constant Edges

Edge names can also be computed. Edge names can be strings or integers; any expression yielding a string or integer value (or path value; see below) can be used to name an edge to follow.

The expression is wrapped in [[ ]] to group it apart from the rest of the path. For example, in our provenance application we have argv data from processes. The argv objects have edges with integer names. To check if a program was run with some particular option that spans multiple words in argv, one might write an expression of the form argv.[[i]] == "-m" and argv.[[i+1]] == "12".

This syntax can also be used anywhere else a path edge name is required, including the field-naming constructs of select and new. For example, one might create an index to allow finding restaurants by their cuisine type as follows:

```
select R as [[T]]
from Guide.restaurant{R}.type{T}
```

2.11.4 Variable Binding

While a path expression directly names only the value or values found at the right-hand end, sometimes values found in the middle of the path are interesting as well. In PQL these can be retrieved for inspection or other handling. The syntax {V}, placed between two edges in a path expression, binds the variable V to the object found at that point along the path. For example, Guide.restaurant{R}.address{A} binds R to a restaurant and A to its address.

Note that if placed in the from-clause, Guide.restaurant{R} is exactly equivalent to Guide.restaurant as R. However, the variable-binding syntax can be used outside of the from-clause as well, such as in this query:

```
select R.name
from Guide.restaurant as R
where R.address{A} grep "Elm"
or A.street grep "Elm".
```
This query deals with the case where some addresses are strings and others are structures.
This kind of variable binding is sometimes referred to as an “object variable binding” to distinguish it from a “path variable binding”.

### 2.11.5 Path Variable Binding

Path expressions can also bind *path variables*, variables whose values represent a segment of the physical path the expression matched. The syntax for this is @var. In the path A(.b.c)@CP, the variable CP is bound to the physical path that matched the sequence .b.c.

There are a number of uses for this functionality. One is to inspect the structure of the database:

```sql
select distinct tostring(P)
from Guide.restaurant(.#.zip%)@P
```

will return the various ways zipcodes are entered in the database.

Another use is to distinguish different paths:

```sql
select R1.name, R2.name
from Guide.restaurant as R1,
     Guide.restaurant as R2,
     R1(.nearby)@P as E
     R1(.nearby)@Q as F
where E = R2 and F = R2
and P <> Q
```

This query returns all pairs of restaurants for which there is more than one way to get from one to the other by “nearby” relationships.

A further use of path variables is to follow a sequence of edges that has just been discovered. This allows “backrefs” as found in Perl regular expressions. This is done with the `[[ ]]` syntax: A(.a|b|.c)*@P.[[P]]. (This needs an example.)

Similarly, one may use path variables to name results, by writing `select ... as [[X]]` or `new(... as [[X]])`. (This also needs an example.)

Finally, one can use path variables for more elaborate edge name matching than allowed by the `%` syntax. This is done by binding the edge to a path variable name, then performing string matching in the where-clause.

```sql
select distinct Z
from Guide.restaurant.#.%@P as Z
where tostring(P) grep "[Zz]ipcode"
    or tostring(P) = "zip"
    or tostring(P) = "ZIP"
```

The internal representation of path values includes both the edge names traversed and the objects at either end. These can be extracted for inspection, via functions that haven’t been defined yet. The sequence of edges can also be printed using the `tostring` function. Path variables can also be concatenated using the same `++` operator used for string concatenation.

The operator precedence of a path variable binding is between repetition and alternatives; that is, A.b|.c@P and A.b.c@P both bind P to .c, never .b. Parentheses change this: A(.b|.c)@P binds either .b or .c, and A(.b.c)@P binds the sequence .b.c.

Note that because the select-clause comes logically last, variables (of either kind) bound in the select-clause are *not in scope* in the from-clause or where-clause.

### 2.11.6 Variables Bound Under Repetition

Combining the variable binding syntax with the regular expression syntax yields several interesting cases.
2.12 Query Canonicalization

Several of the previous sections allude to rewriting steps that are taken in order to transform queries into a more precise form. These processes are now described in more detail.

2.12.1 Syntactic Sugar

The following transformations are basically trivial:

Zero-or-more arbitrary edges. The transformation from \( .\# \) to \((.%)*\) is sufficiently trivial that it is also made in reverse when dumping out the internal representation.

Short-form quantifiers. The form \( X < \text{all}\ S \) is translated to \( \text{for all} S_0 \text{ in } S: X < S_0 \). The same transformation is used for \( \text{any} \) and \( \exists \).

2.12.2 Path Unification

If no further steps were taken, simple queries would return bizarre and nonsensical results. Consider the query

\[
\text{select R.name from Guide.restaurant as R,}
\text{R.(nearby(NBR))}^+ \text{ as TGT where TGT.name = "BBQ Heaven"}
\text{and for all N in NBR.name: N <> "Moe's Place"}
\]

which finds a nearby-path to someplace we want without going by the local disreputable bar.

The proper behavior in the optional case is even less clear: an optional element could be held to be a special case of either repetition or alternatives. Currently it behaves like an empty alternate.
This is clearly meant to return restaurants that are on Elm St. in Somerville. However, if no further processing were done, a restaurant with a branch on Elm St. in Waltham and another branch on Union Ave. in Somerville would be returned.

To see why, consider the form of the query after moving the implicitly iterated paths to the from-clause:

```sql
select N
from Guide.restaurant as R,
R.address.street as S,
R.address.city as C,
R.name as N
where S grep "Elm"
and C = "Somerville"
```

Since there are two addresses, we get two streets and two cities, and when we iterate each in turn, we get four combinations, two that match up and two that are crosslinked.

This behavior is clearly undesirable. By the second of our design principles, when we write `R.address` twice in the where clause, we expect this to mean the same address both times.

To make the behavior match this expectation, PQL performs path unification: repeated paths, common path prefixes, and other duplicate path elements are combined by introducing new variables. The query above becomes

```sql
select N
from Guide.restaurant as R,
R.address(A).street as S,
A.city as C,
R.name as N
where S grep "Elm"
and C = "Somerville"
```

Now only one address is considered at a time, and the query behaves as intended.

Paths are unified within each clause of a selection, and paths in the where-clause and select-clause are unified with paths in the from-clause. However, paths appearing only in the where-clause and select-clause but not in the from-clause are not unified with one another.

Paths and path segments are also only unified if they don't bind variables in the same places. `Guide.restaurant(R).name` and `Guide.restaurant.name(N)` will be unified to `Guide.restaurant(R).name(N)`, but `Guide.restaurant(R1).name` and `Guide.restaurant(R2).name` will not. Thus, to prevent path unification, insert one or more dummy variables at the desired points.

### 2.12.3 nil

Most of this section withdrawn

The truth tables for boolean logic in the presence of `nil` are given in Section ?? . They are not the same as the truth tables for SQL’s `null`. The best way to think about this is that SQL’s `null` represents an unknown value, whereas PQL’s `nil` represents no value.

### 2.12.4 Other Type Conversions

to be written

### 2.13 Query Debugging

to be written
Chapter 3

PQL vs. Lorel

3.1 Caveats

Lorel is rather lightly documented; the Lore distribution includes some simple examples and a suggestion that one read the published Lorel paper. This paper is in turn an overview, not a specification, and it is furthermore not entirely self-consistent; for example, the grammar given in its appendix does not accept all the sample queries provided in its text.

Meanwhile, the Lore implementation, which is available only as a binary distribution, is no longer runnable. It requires an obsolete version of libstdc++.so that is not available on current Linux installs. On a very old Linux machine (Red Hat 7.3) that does have that library, it does not run properly; it dumps core on almost every query. This is presumably due to Linux backwards compatibility issues. There is a Solaris binary distribution as well, which is more likely to work, but there has been no opportunity to try it so far.

Therefore, by “Lorel” this chapter means “My/our best understanding from reading and re-reading the Lorel paper”. Pragmatically, and given the reality of published papers about real projects, this is likely to diverge substantially from the implementation, particularly upon fine semantic distinctions and other minutiae. For the time being at least, however, it is the best interpretation of “Lorel” available.

One of the long-term goals of the PQL project is to provide a clear and unambiguous language specification.

3.2 Syntax

For the most part PQL is supposed to be syntactically compatible with Lorel.

The following things have been intentionally changed:

- Lorel allows \( A < \text{some } S \) as a pointless synonym for \( A < \text{any } S \). This has been removed from PQL.
- Lorel apparently allows and ignores the OQL indirection operator \( * \) in at least some contexts, even though in Lorel’s world it makes no sense. This has been removed from PQL.
- In Lorel, the \text{tostring} function on paths is called \text{pathof}. PQL currently supports \text{pathof} but it is likely to be removed in a future release.
• The `unquote` operator in Lorel has been replaced in PQL by the syntax `[[ ]]`. PQL currently still supports `unquote` but it is likely to be removed in a future release.

• Lorel’s `new_oem` is just `new` in PQL.

• PQL uses the `like` operator for matching edge names, so both `%` and `_` are metacharacters. Lorel uses `%` with the same meaning in this context; it is not currently clear if Lorel also treats `_` as a metacharacter or only allows `%`.

Furthermore, the following extensions have been made:

• String constants in PQL support the standard C escape sequences for representing non-printing characters. This may not be true of Lorel.

• Lorel, or at least the grammar in the Lorel paper, requires many extraneous parentheses in expressions. PQL’s parser accepts but does not require most of these. Parentheses are required only when mixing paths with repetition operators and arithmetic. See Section ??.

• Path expressions in PQL can contain field names that are integer constants, as well as identifiers, without using the `unquote` operator or `[[ ]]` syntax.

• The `glob` operator only exists in PQL and is not supported by Lorel. Lorel only provides `like` and `grep` for text pattern-matching.

### 3.3 Visible Semantics

The following visible but not fundamental extensions have been made to the language semantics.

• Lorel apparently does not allow boolean values to appear in the database or be query results. PQL lifts this restriction.

• In Lorel, path edges may only be identifiers. In PQL, path edges may be strings or integers. While strings that are not identifiers cannot be written directly in path expressions, they can be handled using the `[[ ]]` syntax.

• In Lorel, the `unquote` operator is apparently restricted to path variables only. In PQL, the equivalent `[[ ]]` syntax can also be used with string or integer values.

• In Lorel, the repetition operators (`*`, `+`, and `?`) can only be applied to a single path step. In PQL, arbitrary regular expressions over paths are allowed.

• In Lorel, variables (whether path or object variables) bound inside a repetition operator either have unspecified behavior or are not allowed at all. (It is not entirely clear which.) In PQL, these variables become sets of sequences with clearly defined semantics. Such variables are important for writing “pruning” into searches. See Sections ?? and ??.

• Lorel prohibits path expressions appearing in the select clause from binding variables. In PQL there is no such restriction. (However, because the select clause comes logically last in the select statement, variables
bound in the select clause are not in scope in the corresponding from clause or where clause.

- In Lorel, the results of a standalone select (whose results are not assigned anywhere) are implicitly assigned to the special variable answer. PQL does not do this, and the variable answer is not special. To use answer, assign it explicitly, or rewrite the query using let-binding.

### 3.4 Picayune Semantics

There are also some differences that most users will never notice but affect the behavior of the language in corner cases.

**Data model.** PQL’s data model is slightly different from Lorel’s.

In Lorel, structure members are always pointers to database objects, and database objects can hold arbitrary values, both structures and atoms.

In PQL, structure members hold arbitrary values, both values of atoms and references to structures, and database objects are always structures.

This has some practical ramifications in the backend, which are not pertinent here. It also has one user-visible consequence: in Lorel it is possible to set up a family of objects with members that all share the same object containing an integer, such that changing that integer changes the member value in the whole family.

This is not possible in PQL; to get this shared update behavior in PQL a level of indirection through an additional structure is required. However, it seems that in general such a construction would be confusing to work with and undesirable, so disallowing it is arguably a feature.

### 3.5 nil

The oemnil value introduced in Lorel differs from the nil in PQL. The purpose of oemnil as described in the Lorel paper is that all comparisons involving it should yield false. This is the right general idea, but incorrect in detail.

In PQL, comparisons to nil yield nil, not false, and boolean logic involving nil proceeds according to the truth tables of the three-valued logic discussed in ?? If the entire where-clause evaluates to nil, it is considered false. This produces the desired effect but works properly on a much larger family of expressions.

### 3.6 Group By

In Lorel, paths may be placed outside the from-clause and are then not iterated. They thus formally name sets; they can then be handled as sets, or, alternatively, language-level magic is applied to allow handling them as single values.

In PQL, paths placed outside the from-clause are implicitly moved to the from-clause and iterated anyway. The language-level magic need to do what Lorel does turns out to be quite problematic, and arguably not even desirable from a user perspective. Instead, one can use group by (which Lorel does not support) to collect related values into sets for aggregate inspection.

In the Lorel query

```plaintext
select N from Guide.restaurant as R, R.name as N,
where
```

In PQL:

```plaintext
select N from Guide.restaurant as R
where
```
for all C in R.address.city: C = "Cambridge"

the set of cities retrieved is not iterated by the
selection, and can thus be inspected with for
all. This query is not legal in PQL: the path
R.address.city is moved to the from-clause
and iterated, and the scratch variable left be-
hind in its place does not name a set, so the type
checker rejects the query.

The (nearly) equivalent query in PQL is writ-
ten as follows:

select N
from Guide.restaurant as R,
    R.name as N,
    R.address.city as C
group by R, N
having
    for all C1 in C: C1 = "Cambridge"

Here the group-by collects the various values of
C for each distinct R and N into a set, which can
then be inspected.

This PQL query is subtly different:

select N
from Guide.restaurant as R,
    R.name as N,
    R.address.city as C
group by R
having
    for all C1 in C: C1 = "Cambridge"

Here we group by R only, so N becomes a set. If
some restaurant has two names, this query will
return one result with both names; the query
above will return two results with one name
each. This is (nearly) equivalent to the following
Lorel query:

select R.name
from Guide.restaurant as R

One subtle difference remains, which is that
in PQL the various sets are expanded and then
grouped back together, whereas in Lorel they are
not. This can lead to different results if values
are repeated. Consider a case where a resto-
rant has two names in the database and these
two names are the same. The group by R, N
clause will fold these together and generate one
group with more cities (and thus one result); the
Lorel query will treat them as separate and gen-
erate two results.

As of this writing there is no way to avoid this
behavior. Internally the PQL engine has objects
called distinguishers that it uses in certain con-
texts to prevent this kind of conflation. Making
these available to the query writer would take
care of the issue, but at the cost of making an
ugly mess. Syntax like group by R, all N
would solve the problem neatly, but creates im-
plementation difficulties. It should be possible
to bridge the gap, but it is not yet immediately
clear how.

3.7 Translating Lorel Queries

Make the following syntactic transformations:

- Remove any uses of * in *X or *X.y.z.
- A < some S becomes A < any S.
- new_oem becomes new.
- pathof becomes tostring.
- unquote(X) becomes [[X]].
3.7. TRANSLATING LOREL QUERIES

It may be necessary to add parentheses around paths used in arithmetic: \( X.\text{height} \times X.\text{width} \) regretfully needs to become \((X.\text{height}) \times (X.\text{width})\). (However, many other (extraneous) parentheses can be removed.)

Any explicit uses of `oemnil` must be converted to uses of PQL's `nil`; this transformation must be done cautiously because the semantics are different.

Any explicit uses of `answer` will need to be adjusted because PQL does not implicitly assign `answer`.

Queries using Lorel’s update language will need to be rewritten in terms of PQL’s update language, which has yet to be defined and is likely to be somewhat different.

Uses of paths outside the from-clause need to be inspected and converted to equivalent queries using group-by.

If a path appearing in the where-clause or select-clause of a Lorel query is used as if it were a scalar value, the equivalent PQL query requires first group-by to turn it into a set, then usage of the `exists` quantifier to get a scalar value again. It is important to make sure any particular set is quantified only once (otherwise extraneous results appear) and additional explicit tests for empty sets and/or `nil` may be needed (to avoid having `exists` become vacuously false) and some possible queries may have multiple (or no) correct interpretations. The Lorel paper [?] discusses some simple instances of the problems that can arise, but unfortunately part of the exposition is wrong, and furthermore the problem is much deeper than the paper acknowledges or seems to realize. The best approach is to figure out what the intent of the query is supposed to be and write a PQL query that satisfies that intent, rather than trying to derive an exactly equivalent query.
Chapter 4

PQL Language Reference

4.1 Lexical Structure

In general the basic lexical structure of PQL is more like C than like SQL; for example, identifiers are case sensitive and cannot be quoted as they can be in SQL, and strings use C escape sequences.

Reserved words are also case sensitive. A complete list of reserved words is given in Appendix ??.

4.1.1 Whitespace

Spaces, tabs, and newline characters are treated as whitespace. Whitespace separates tokens but is not semantically significant.

4.1.2 Identifiers

Identifiers in PQL must begin with a letter, and may contain letters or digits. The underscore character _ is considered a letter.

4.1.3 Operators.

In general, operators are sequences of punctuation characters. However, many operators are also reserved words. See section ??.

4.1.4 Edge Names

As written in the language, the names of object fields, graph edges, and single path steps (which are all equivalent things) may be identifiers or integer constants. Internally, edge names may be arbitrary strings, not just identifiers; however, such names must be handled in the language using the [[ ]] syntax discussed in section ??.

4.1.5 Constants

Boolean. The reserved identifiers true and false are boolean constants.

Integers. Integer constants are sequences of digits. A constant beginning with 0 is interpreted as octal. A constant beginning with 0x or 0X is interpreted as hex.

Floating point. Floating point constants consist of a mantissa (one or more digits surrounding exactly one decimal point) followed by an optional exponent, in the standard form e+123. The decimal point need not be preceded (or followed) by any digits.
CHAPTER 4. PQL LANGUAGE REFERENCE

Strings. String constants are written with double quotes. C-style escape sequences are supported. For brain compatibility with SQL, single-quoted SQL-like strings with no escapes are also recognized. (This latter feature is provisional and may be removed.)

4.2 Syntax

The general overall syntax is similar to SQL. However, there are fewer “noise” keywords, and in general an effort has been made to tighten up the syntax and make it more like a programming language and less like COBOL.

4.2.1 Paths

The general syntax of a path expression is as follows.

Every path has a root - some variable, either a global variable or one previously bound in the same query. The root is followed by an edge expression, which can be created from the following elements, in syntactic precedence order:

- Sequencing: two or more edge expressions in a row are matched in sequence.

- Alternates: two or more edge expressions separated by the | character are treated as alternatives. Any or all may be matched, but each different match generates a distinct concrete path.

- Repetition: the ?, *, and + characters, used as suffix operators on an edge expression, cause it to be matched zero or one, zero or more, or one or more times respectively.

- Bind Path: the @ character followed by an identifier, used as a suffix operator on an edge expression, causes the concrete path matched by that expression to be extracted. This value is bound to a variable named by the identifier.

- Bind Object: an identifier enclosed in braces ({}), placed after an edge expression causes the object found by following the path up to that point to be extracted. This value is bound to a variable named by the identifier.

- Label: the primitive element of an edge expression is the name (or label) for a single graph edge, prefixed by a dot for separation. This can be an identifier, an integer constant, or an expression surrounded by the characters [[ ]], and may be optionally followed by the particle -of to denote following an edge backwards.

- Subexpression: subexpressions may be enclosed in parentheses.

Note that because the dot goes with the edge name, each alternative is written with its own dot: .a| .b rather than .a|b. Similarly, parentheses must include the leading dot: ( .a) rather than .(a).

Also note that while an object binding can be inserted both inside and outside parentheses like this: (.b{X}){Y} the two variables resulting name the same object and are completely equivalent. Attempting to use both positions at once will result in an error.

However, variables bound similarly both inside and outside repetition are not equivalent: the “inside” variable names every object reached at that point, including the ones that resulted in looping back, and the “outside” variable names only the last of these found before continuing onward.
4.2. SYNTAX

4.2.2 Expressions

Operators

The available operators arranged among other constructs by order of precedence:

- select.
- The formal form of quantifier expression.
- The logical operators and and or.
- The set operator in, which can mean element-of or subset-of based on the types.
- The set operators union, intersect, and except (set difference).
- The comparison operators = (equal), <> (not equal), <, >, <=, >=, like, glob, grep, and soundex, along with the short form of quantifier expression.
- The binary arithmetic operators + and -, or alternatively a path expression. (See below.)
- The arithmetic operators *, /, and mod.
- The prefix operators not and -. 
- Function calls and new.

The like operator is the same as in SQL. It is similar to Unix shell “globs”, except with different metacharacters: % means *, _ means ?, and there is no way to express character classes or sets.

The glob operator implements Unix shell “globs”, that is, the familiar mydoc*.txt notation.

The grep operator implements regular expression matching. In PQL this uses “extended” (egrep-style) regular expressions.

The soundex operator provides Soundex approximate-by-spoken-English matching.

Path expressions have the same precedence as arithmetic and cannot be mixed without using parentheses. This is because the path expression syntax uses + and * as suffix operators, whereas arithmetic expressions use them as infix operators. It turns out that using the same token as a suffix operator and an infix operator requires an LR(2) parser; commonly available parser generators are only LR(1). Accepting this restriction seemed a better choice than adopting unfamiliar alternative spellings for any of these cases.

Quantifiers

There are two syntaxes for the quantifier logic: the formal form, and a shorter form for use with comparison operators.

The formal form has the syntax exists T in S: predicate(T) where T is a fresh variable that varies over the elements of the set S. The predicate must be a boolean expression. The quantifier may be exists or for all.

The shorter form has the syntax A < all S where A is some existing variable and S is a set. Any comparison operator can be used. The quantifier can be all or any.

Internally the short form is translated into the formal form in the obvious way.

Functions

Currently the available functions are:

Set:

- element(S) - given that S is a set with one element, return that element.
• set(E) - construct a set containing the one element E.

• exists(S) - true if the set S is nonempty.

Aggregation:
• count(S) - number of elements in set S.
• min(S) - minimum value in set S of numbers.
• max(S) - maximum value in set S of numbers.
• sum(S) - total value in set S of numbers.
• avg(S) - average value in set S of numbers.

Numeric:
• abs(N) - absolute value of number N.

Time:
• ctime() - return Unix-formatted time and date string.

New
The new function constructs a new database object. Each argument becomes the value of a field of the new object.

Additional magic syntax is allowed on these arguments to allow naming the field. This uses the keyword as: new(3 as x) constructs an object with a single field x that contains the value 3. An additional (OQL-derived?) syntax for this is also supported: name:value. In either form, the name can be taken from an expression instead by wrapping the expression in [ [ ] ]. If the name is not given, the default name is the name of the graph edge that was used to retrieve the value. Thus after `Guide.restaurant as R and R.name as N` the default edge name in new(N) would be name. If there is no such edge or it is not uniquely defined, the default name reverts to default.

Select
A select is an expression that returns a set of values. (These may be values of almost any type. Currently path values may not be returned from a selection; however, this restriction is likely to be lifted.)

The select-clause contains a comma-separated list of expressions, each optionally tagged with a field/edge name in the same way as the arguments of new.

The from-clause contains a comma-separated list of path expressions. It may be omitted if no paths are needed.

The where-clause is a boolean expression. If omitted, it is treated as true.

The evaluation of a selection occurs starting from the from-clause; the select-clause is logically last. The scopes of variables bound by path expressions in the select-clause reflect this.

If necessary, the returned sets (and tuples, if appropriate) are converted to the object representation of those types, a single object with one edge leading to each member. This is done if the selection results are assigned to a database variable or object field, because language-level sets and tuples may not appear in the database. It may also be done for query results if the query client arrangements are not prepared to handle sets as sets.

There are potentially two layers of edge
4.3. TYPE SYSTEM

names in the results: the edges in the object representation of the ultimate result set, if it is generated, and, if more than one field was selected, the edges leading from the returned objects to the individual fields. These names are chosen the same way as in `new`, with one additional rule: if there are multiple values selected, the objects generated to hold those values, which come from no database edge and would thus always be default under the basic rules, can instead be named as follows: if all values selected are subfields of some single object, the default name for that object is used.

Note that if multiple fields are selected, there is no way to explicitly name the enclosing objects. For explicit control of this name, an explicit `new` must be written; then the field names can be specified on its arguments and the name for the new objects themselves can be given at the top level of the select-clause.

4.2.3 Statements

Select

A selection standing alone (not assigned to anything) is treated as a query whose results are to be sent back to the client.

A simple expression standing alone is implicitly wrapped in a selection and treated the same way.

Other Statements

No other statements are defined in PQL yet. An update language is planned for PQL; this will result in other statements being defined.

4.3 Type System

Though PQL has no support for formal schemas and does not impose restrictions on what types of values may appear in what places in the database, it does have a type system.

PQL is partially statically typed: tuples, sets, sequences, and paths arise only during query processing, so the type system can be strict about them.

However, because there is no schema, a value from the database can legitimately have any of the other types, and which such type is not known until the value is found during query execution. Thus, these types cannot be checked in advance during query compilation/planning, and are allowed to mix. (The typechecker will nonetheless reject expressions whose types are inherently mismatched, such as `false + 3`.)

Note that mismatched types that appear during query execution do not result in errors. Because the database is not strictly typed, the logic in any particular query may not match the types of the data it finds. When this happens, the PQL engine does the best it can to convert types to match; if that fails, it creates a null result and continues. This makes it possible to query over data that does not have a single uniform organization, by writing logic that tries all the various possibilities. The ones that match generate results; the ones that do not get dropped without causing the query to fail. This reflects the first of the design principles stated in Section ??.

4.3.1 Types

Sets. Sets, which are really multisets or “bags”, may contain elements of any type, including other sets. Set values are generated during query evaluation, but may not be stored directly
Sequences. Sequences are sets that have been sorted. Sequence values are generated during query evaluation, but may not be stored directly in the database.

Pathelements. The `pathelement` type holds an instance of a single step in a specific path through the database graph. A pathelement holds the name of the matched edge plus the identity of the object on each end. Pathelement values can be generated as the database is searched, but may not be stored directly in the database.

Paths. There is no `path` type as such; full paths are sequences of pathelement. Path values can be generated as the database is searched, but may not be stored directly in the database.

Structs. The `struct` type is an “object”: a collection of named fields and values. Values of type `struct` are references to objects. Thus, a field that holds a `struct` is a reference to another object and thus an edge in the graph. PQL formally uses the term `struct` for this type, rather than `object`, to avoid confusion.

Structure fields, that is, values appearing in the database, may contain values that are structs or atoms. They may not contain paths or sets; paths and sets appear only during query processing.

Atoms. The boolean, string, and all number types, as well as any other types defined by the storage backend, are collectively referred to as `atom`.

Booleans. Booleans range over the two constants `true` and `false`.

Strings. Strings are as one would expect. Currently there is no support for wide-character strings or multiple character encodings.

Numbers. The integer and floating types (and bignums, if extant) are collectively referred to as `number`.

Integers. The `integer` type is by default a signed 32-bit integer, specifically the `int` type of the underlying system. Explicitly-sized integer types are not currently available but are expected to be added in the future.

Floats. The `float` type is an IEEE double-precision floating point number.

Bignums. Bignums are not currently supported, but would be nice to add in a future version.

Dates and Times. Dates and times are not currently supported, but would be nice to add in a future version.

4.3.2 Explicit Conversions
to be written

4.3.3 Implicit Conversions
to be written
4.4 Semantics

The basic evaluation model for selections is that the elements in the from-clause are each evaluated and the cartesian product of all of them is taken. This produces a set of candidate results. Each such result constitutes an assignment of values to the variables bound by the from-clause. The where-clause is evaluated for each candidate result using these values. If this yields true, the select-clause is evaluated using the same values. This generates a result value. Then ultimately all the result values are returned as a set.

The physical evaluation may have only a loose resemblance to this model; the query is transformed heavily by the planning and optimization stages in the query engine, and aggressive steps are taken to short-cut evaluation and avoid generating doomed candidate results.

4.4.1 Variable binding

to be written

... the below is no longer very valid ...

When one binds multiple values in the same path expression, the implicit linkage between them is retained. For example, in the expression `Guide.restaurant{R}.address{A}`, each address A remains associated with its restaurant R. No substitution of values, or iteration of the sets R and A, even with an explicit quantifier expression, will create cases where an address is paired with a restaurant it does not belong to.

For example, given the above, in the expression

```
exists A0 in A: A0 grep R.name, only
the name of the restaurant corresponding to A0
will be tried.
```

To defeat the linkage, bind the objects independently, such as like this:

```
select R.name
where exists A0 in
    Guide.restaurant{R}.address{A}:
    A0 grep Guide.restaurant{RR}.name
```

Note that the extraneous RR is needed to indicate that the two usages of Guide.restaurant are themselves meant to be distinct. Otherwise they would be folded together, as described in section ??.

This behavior is somewhat odd, but logically reasonable, and necessary for the implicit quantifier insertion to work properly.

4.4.2 Loop Zapping

to be written

4.4.3 nil

Like in SQL, missing values are represented with a magic null value. This value is called nil and has a type such that it can be mixed with any other value, except for a set.

When nil is used in logical expressions, the truth tables are as follows:

For the and operator:

```
nil false true
nil nil nil nil
false nil false false
true nil false true
```

For the or operator:

...
The `not` operator applied to `nil` yields `nil`, as do comparisons of `nil`, including to itself; use the `nilp` function to test for a value being `nil`.

**NOTE:** This is not the same behavior as the SQL null value! PQL’s `nil` value represents *no* value, whereas SQL’s null behaves more like an unknown or unspecified value. This particular choice of behavior is required for correct handling of absent or mismatched database values. Also see Section ??.

4.4.4 **Quantifier Insertion**

*to be written* (but see Section ??)

4.4.5 **Path Unification**

*to be written* (but see Section ??)

4.4.6 **Relational Subset**

*to be written*
Chapter 5

Formal Treatment

5.1 Syntax
For the time being please consult the grammar in the source.

5.2 Abstract Syntax
Under development.

5.3 Semantics (Canonicalization)
Under development.

5.4 Semantics (Evaluation)
Under development.
## Appendix A

### List of Reserved Words

<table>
<thead>
<tr>
<th>Word</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs</td>
<td>numeric function (absolute value)</td>
</tr>
<tr>
<td>all</td>
<td>short-form universal quantifier</td>
</tr>
<tr>
<td>and</td>
<td>boolean operator</td>
</tr>
<tr>
<td>any</td>
<td>short-form existential quantifier</td>
</tr>
<tr>
<td>as</td>
<td>result naming particle</td>
</tr>
<tr>
<td>avg</td>
<td>aggregation operator</td>
</tr>
<tr>
<td>count</td>
<td>aggregation operator</td>
</tr>
<tr>
<td>ctime</td>
<td>time function</td>
</tr>
<tr>
<td>distinct</td>
<td>selection keyword</td>
</tr>
<tr>
<td>element</td>
<td>set function</td>
</tr>
<tr>
<td>except</td>
<td>set operator (set difference)</td>
</tr>
<tr>
<td>exists</td>
<td>existential quantifier</td>
</tr>
<tr>
<td>false</td>
<td>boolean constant</td>
</tr>
<tr>
<td>for</td>
<td>with all, universal quantifier</td>
</tr>
<tr>
<td>from</td>
<td>selection keyword</td>
</tr>
<tr>
<td>glob</td>
<td>string operator</td>
</tr>
<tr>
<td>grep</td>
<td>string operator</td>
</tr>
<tr>
<td>in</td>
<td>quantification keyword</td>
</tr>
<tr>
<td>intersect</td>
<td>set operator</td>
</tr>
<tr>
<td>like</td>
<td>string operator</td>
</tr>
<tr>
<td>max</td>
<td>aggregation operator</td>
</tr>
<tr>
<td>min</td>
<td>aggregation operator</td>
</tr>
<tr>
<td>mod</td>
<td>numeric function</td>
</tr>
<tr>
<td>new</td>
<td>object constructor</td>
</tr>
<tr>
<td>nil</td>
<td>special constant</td>
</tr>
<tr>
<td>not</td>
<td>boolean unary operator</td>
</tr>
<tr>
<td>of</td>
<td>label reversing particle</td>
</tr>
<tr>
<td>or</td>
<td>boolean operator</td>
</tr>
<tr>
<td>pathof</td>
<td>path function (deprecated)</td>
</tr>
<tr>
<td>select</td>
<td>selection keyword</td>
</tr>
<tr>
<td>set</td>
<td>set function</td>
</tr>
<tr>
<td>soundex</td>
<td>string operator</td>
</tr>
<tr>
<td>sum</td>
<td>aggregation operator</td>
</tr>
<tr>
<td>tostring</td>
<td>general function</td>
</tr>
<tr>
<td>true</td>
<td>boolean constant</td>
</tr>
<tr>
<td>union</td>
<td>set function</td>
</tr>
<tr>
<td>unquote</td>
<td>edge naming operator (deprecated)</td>
</tr>
<tr>
<td>where</td>
<td>selection keyword</td>
</tr>
</tbody>
</table>
Appendix B

Known Implementation Defects

This appendix is not meant to be a bug list (there's one of those in the source) but more of a high-level list of unimplemented or improperly implemented features.

withdrawn - see the README in the source for the current list
APPENDIX B. KNOWN IMPLEMENTATION DEFECTS
Appendix C

some old text

C.0.1 To Schema or Not

The absence of a formal schema is a two-edged sword: on one hand, it allows working with arbitrary real-world data, or data merged from multiple sources, which may not be uniformly organized or contain enough structure to allow loading it into a traditional database. But on the other hand, it offers the opportunity to make a giant mess if data is inserted into the database without some level of discipline and organization.