Advances in Programming Languages

APL10: Bridging Query and Programming Languages

Ian Stark

School of Informatics
The University of Edinburgh

Friday 29 October 2010
Semester 1 Week 6

Topic: Domain-Specific vs. General-Purpose Languages

This is the second of three lectures on integrating domain-specific languages with general-purpose programming languages. In particular, SQL for database queries.

- Using SQL from Java
- Bridging Query and Programming Languages
- Heterogeneous Metaprogramming

Topic: Domain-Specific vs. General-Purpose Languages

This is the second of three lectures on integrating domain-specific languages with general-purpose programming languages. In particular, SQL for database queries.

- Using SQL from Java
- Bridging Query and Programming Languages
- Heterogeneous Metaprogramming

Outline

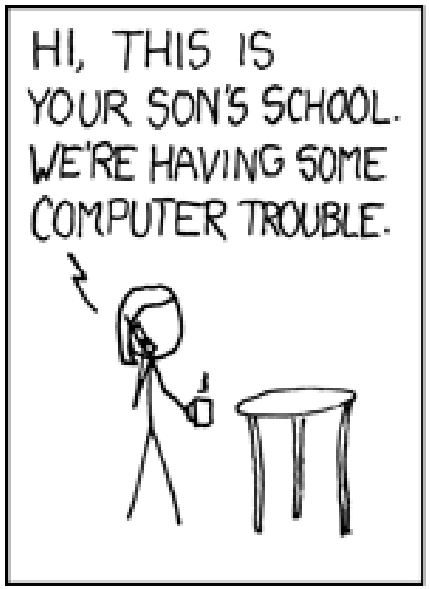
Review

SQL is a *domain-specific language* for programming queries over relational databases. Queries may be complex, with declarative and imperative components, and are often constructed by other programs rather than by hand.

Programs generating SQL code use frameworks like JDBC or ADO.NET; and these do construct queries using unstructured string manipulation. Using *prepared strings* begins to add back some structure.

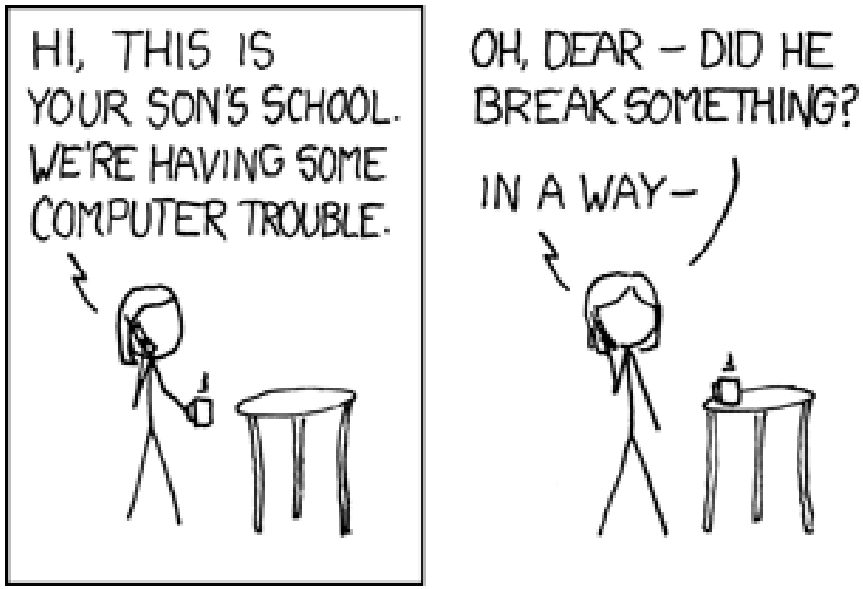
SQL queries are programs in a structured high-level language, but we treat them as unstructured text.

XKCD on SQL Injection



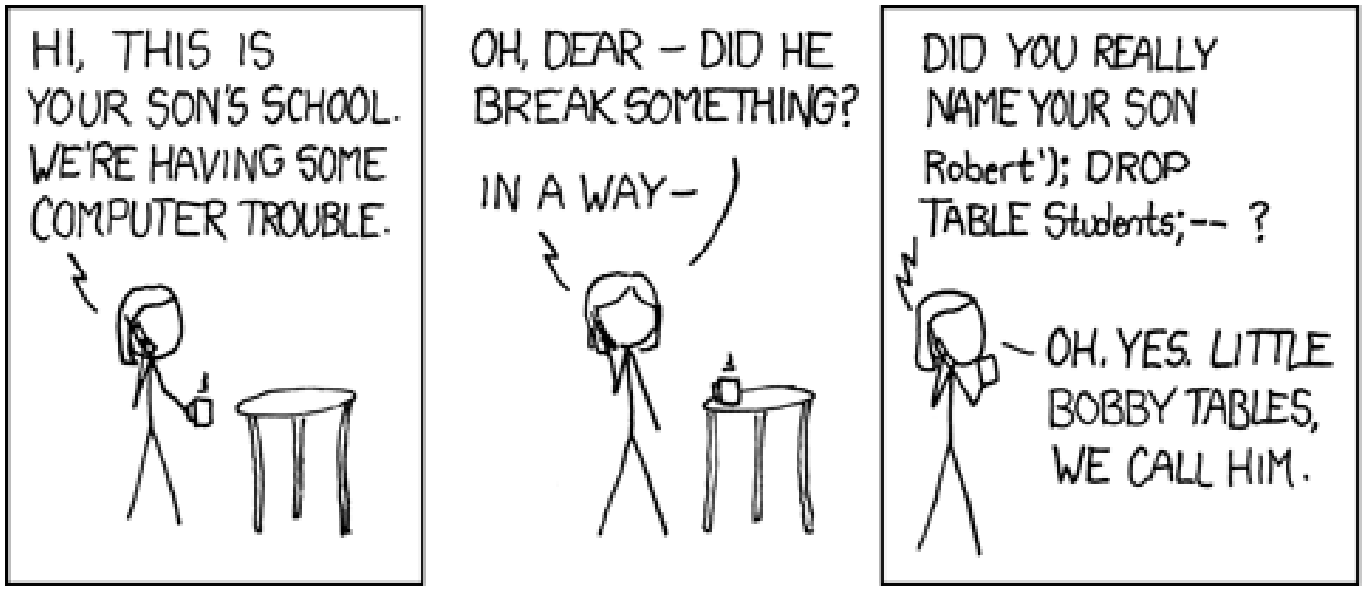
http://xkcd.com/327

XKCD on SQL Injection



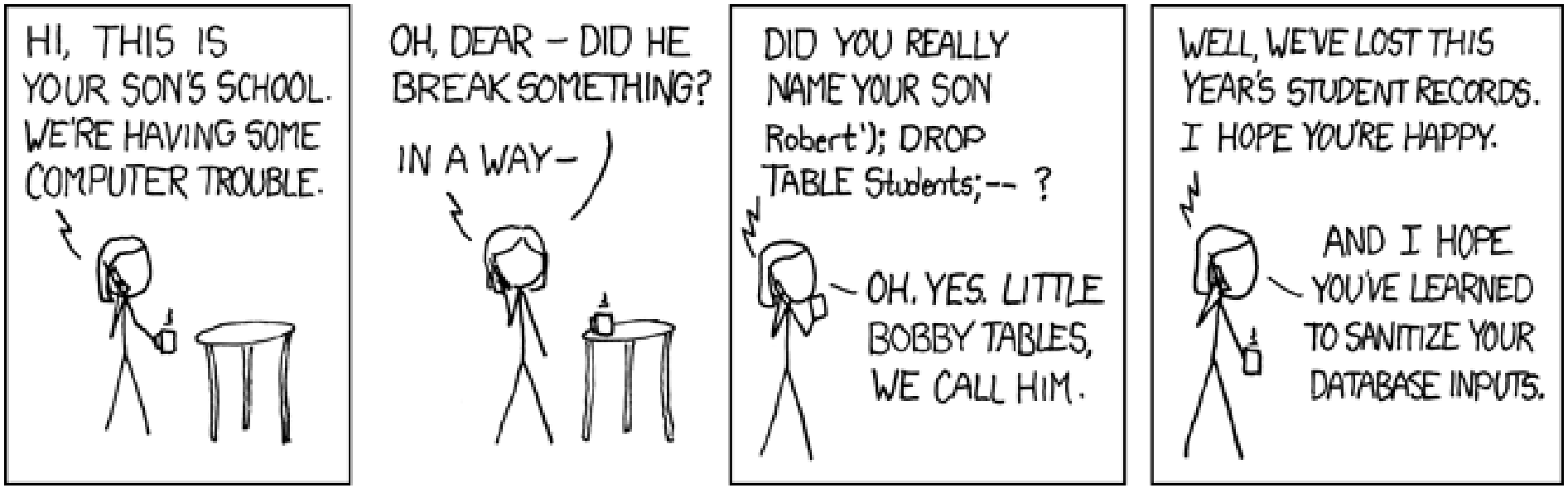
http://xkcd.com/327

XKCD on SQL Injection



http://xkcd.com/327

XKCD on SQL Injection



http://xkcd.com/327

Outline

The Microsoft .NET Framework

Microsoft's *.NET* is a large framework for developing, deploying, and running applications. It now forms a substantial part of the Windows platform, and most additions to Windows arrive as part of .NET.

From the skewed perspective of this course, we can conveniently divide .NET features into two domains:

- Application management infrastructure
- Interesting programming language provision

.NET Application Management

The .NET framework supplies extensive support for building and managing large applications.

- Building:
  - General-purpose base classes: collections, datatypes, text manipulation, networking, crypto, file access, graphics, ...
  - High-level Windows specials: Forms, Presentation, Communication, Active Directory, Workflow, Cardspace, ...
- Managing:
  - Library control and access
  - Application packaging and deployment
  - Name spaces and versioning

.NET *assemblies* provide rich metadata and other facilities for managing deployment and execution.

.NET Programming Language Support

.NET is comparatively language-neutral, providing a shared platform for multiple programming languages.

The *Common Language Infrastructure* is intended to allow high-level interworking between languages.

- A *Common Language Runtime* (*CLR*) provides memory management, garbage collection, code security and other runtime services.
- The *Common Intermediate Language* (*CIL*, or Microsoft's *MSIL*) is a bytecode that serves as the binary format for .NET components.
- The *Common Type System* (*CTS*) means that applications and libraries written in different languages can sensibly communicate high-level data structures.

MSIL is comparable to the Java virtual machine bytecode, but with a few refinements built in (generics, unboxed datatypes) and better support for different language paradigms.

.NET Programming Languages

Several programming languages are available for .NET, all compiling to MSIL, and all sharing access to the .NET libraries and to each other.

Visual Studio includes C#, Visual Basic, C++, and F#; with add-on tools available for (at least) Python, Ruby and Standard ML.

Wikipedia lists another 50 or so .NET languages (right down to LOLcode.net)

For legacy code, and facilities not directly available in the CLR, .NET provides explicit handling of "managed" and "native" code assemblies.

Overall, .NET is similar to Java/JavaEE except for: multiple-language support; symbiotic with Microsoft Windows.

Outline

Database Query from Java

```
Connection c = DriverManager.getConnection(url, user, password);

Statement s = c.createStatement();

ResultSet rs = s.executeQuery("SELECT name, id, score FROM Users");

while (rs.next()) // Loop through each row returned by the query
{
  String n  = rs.getString("name");
  int i     = rs.getInt("id");
  float s   = rs.getFloat("score");
  System.out.println(n+i+s);
}
```

Database Query from C#

```
SqlConnection con = new SqlConnection(dataSourceString);

con.Open();

string query = "SELECT name, id, score FROM Users";

SqlCommand command = new SqlCommand(query, con);

SqlDataReader rdr = command.ExecuteReader();

while (rdr.Read())
  { Console.WriteLine("{0} {1} {2}", rdr[0], rdr[1], rdr[2]); }
rdr.Close();
```

Could Do Better

These existing arrangements for database access have good and bad points:

- ✓ Industrial strength: alternative back-end drivers, scalable, supported, familiar.
- ✓ Straightforward: strings are easily to read and edit. (For humans, at least.)
- ✗ Fragile: concatenating and manipulating strings easily goes wrong.
- ✗ Insecure: sanitizing user input becomes essential but also difficult.
- ✗ Unchecked: the strong static checking of Java/C# is abandoned within the query string.
- ✗ Semantically lossy: the high-level abstraction and structure of SQL as a domain-specific declarative programming language is all gone.

Aside: Hiding Everything Can Work Sometimes

One approach is to wrap up all database access in a library. For example, the Java Persistence API, known in its *Hibernate* implementation, uses database backing to provide persistent object storage.

Good:

- Excellent language integration, user can work purely in host language.
- Using a *data access object* or *active record* can provide an OO view on relational databases.
- Can import features like persistence, transaction support from one language into another.

Not so good:

- Anything not already in the library, or not fitting the OO model, requires going back to coding in SQL (or HQL, or similar).
- In particular, this applies to the very thing an RDBMS does best: efficient execution of complex queries across large datasets.

Parameterized Queries

Prepared statements

```
...
String prequery =
  "SELECT id, name FROM Users WHERE ? < score AND score < ?";

PreparedStatement stmt = con.prepareStatement(prequery);

stmt.setFloat(1,low);    // Fill in the two
stmt.setFloat(2,high);   // missing values

rs = stmt.executeQuery(query); // Now run the completed query
...
```

This is less fragile, and offers opportunities for sanitization: but to go further reinvents features that host programming languages already have.

Could Still Do Better

These existing arrangements for database access have good and bad points:

- ✓ Industrial strength: alternative back-end drivers, scalable, supported, familiar.
- ✓ Straightforward: strings are easily to read and edit. (For humans, at least.)
- ? Fragile: concatenating and manipulating strings easily goes wrong.
- ? Insecure: sanitizing user input becomes essential but also difficult.
- ✗ Unchecked: the strong static checking of Java/C# is abandoned within the query string.
- ✗ Semantically lossy: the high-level abstraction and structure of SQL as a domain-specific declarative programming language is all gone.

Limits to Parameterized Queries

We might like, but cannot express in Java...

```
Boolean valid(int score), high(int score);        // Two tests

String prequery =
  "SELECT id, name FROM Users WHERE ?(score) AND ?(score)";

PreparedStatement stmt = con.prepareStatement(prequery);

stmt.setTest(1,valid);  // Fill in the two        !!! Not Java
stmt.setTest(2,high);   // missing tests          !!! Not Java

rs = stmt.executeQuery(query); // Now run the completed query
```

We can't begin to do this in Java: even if we could pass around first-class functions, they wouldn't fit into SQL. Yet many functions could be mapped to SQL.

Outline

LINQ

LINQ, *Language Integrated Query*, aims to improve the alignment between programming languages and query languages.

```
float findUsersInRange(SqlConnection con, float low, float high) {

  Table<Person> users = con.GetTable<Person>()

  var query = from u in users
              where low < u.Score && u.Score < high
              select new { u.Id, u.Name };

  foreach(var item in query)
    { Console.WriteLine("{0}: {1}", item.Id, item.Name); }
}
```

LINQ

There is more here than just extra SQL-like keywords. The Table<Person> has typed records, field selection u.Score can be checked at compile time, and each item has a correct static type.

```
float findUsersInRange(SqlConnection con, float low, float high) {

  Table<Person> users = con.GetTable<Person>()

  var query = from u in users
              where low < u.Score && u.Score < high
              select new { u.Id, u.Name };

  foreach(var item in query)
    { Console.WriteLine("{0}: {1}", item.Id, item.Name); }
}
```

LINQ

Recall how, even with prepared statements, the compiler makes no use of the types and structures available in SQL expressions.

```
...
String prequery =
  "SELECT id, name FROM Users WHERE ? < score AND score < ?";

PreparedStatement stmt = con.prepareStatement(prequery);

stmt.setFloat(1,low);    // Fill in the two
stmt.setFloat(2,high);   // missing values

rs = stmt.executeQuery(query); // Now run the completed query
...
```

LINQ

There is more here than just extra SQL-like keywords. The Table<Person> has typed records, field selection u.Score can be checked at compile time, and each item has a correct static type.

```
float findUsersInRange(SqlConnection con, float low, float high) {

  Table<Person> users = con.GetTable<Person>()

  var query = from u in users
              where low < u.Score && u.Score < high
              select new { u.Id, u.Name };

  foreach(var item in query)
    { Console.WriteLine("{0}: {1}", item.Id, item.Name); }
}
```

LINQ

Note also that while **var** query = **from** ... builds a query, here of type IEnumerable<...>, it need not necessarily execute it; this can be deferred until the data itself is required by the **foreach**(...) statement.

```
float findUsersInRange(SqlConnection con, float low, float high) {

  Table<Person> users = con.GetTable<Person>()

  var query = from u in users
              where low < u.Score && u.Score < high
              select new { u.Id, u.Name };

  foreach(var item in query)
    { Console.WriteLine("{0}: {1}", item.Id, item.Name); }
}
```

LINQ

The special SQL-like syntax is sugar that expands into a sequence of method invocations, each of which returns an IEnumerable<...> object.

```
float findUsersInRange(SqlConnection con, float low, float high) {

  Table<Person> users = con.GetTable<Person>()

  var query = from u in users
              where low < u.Score && u.Score < high
              select new { u.Id, u.Name };

  foreach(var item in query)
    { Console.WriteLine("{0}: {1}", item.Id, item.Name); }
}
```

LINQ

The special SQL-like syntax is sugar that expands into a sequence of method invocations, each of which returns an IEnumerable<...> object.

```
float findUsersInRange(SqlConnection con, float low, float high) {

  Table<Person> users = con.GetTable<Person>()

  var query =  users.Where(u => (low < u.Score && u.Score < high))
                    .Select(u => new { u.Id, u.Name }) ;

  foreach(var item in query)
    { Console.WriteLine("{0}: {1}", item.Id, item.Name); }
}
```

LINQ

In this case, the Where and Select methods take lambda-expressions and act much like filter and map do on lists in a functional language.

```
float findUsersInRange(SqlConnection con, float low, float high) {

  Table<Person> users = con.GetTable<Person>()

  var query =  users.Where(u => (low < u.Score && u.Score < high))
                    .Select(u => new { u.Id, u.Name }) ;

  foreach(var item in query)
    { Console.WriteLine("{0}: {1}", item.Id, item.Name); }
}
```

LINQ

Although the SQL-like syntax is natural for requesting records from a database, in fact the expansion to regular methods means that it can be used for any kind of IEnumerable<...> objects.

```
float findUsersInRange(SqlConnection con, float low, float high) {

  Table<Person> users = con.GetTable<Person>()

  var query =  users.Where(u => (low < u.Score && u.Score < high))
                    .Select(u => new { u.Id, u.Name }) ;

  foreach(var item in query)
    { Console.WriteLine("{0}: {1}", item.Id, item.Name); }
}
```

LINQ

This expansion into standard method calls also opens up query handling to compiler optimization: we are no longer just piecing together an SQL string, but building a structured query.

```
float findUsersInRange(SqlConnection con, float low, float high) {

  Table<Person> users = con.GetTable<Person>()

  var query =  users.Where(u => (low < u.Score && u.Score < high))
                    .Select(u => new { u.Id, u.Name }) ;

  foreach(var item in query)
    { Console.WriteLine("{0}: {1}", item.Id, item.Name); }
}
```

Outline

Language Support for LINQ

Beyond these small examples, LINQ is a general technique for managing data queries in .NET programming languages: currently supported for $\{\text{Object}, \text{SQL}, \text{XML}\}$ queries in $\{\text{C\# 3}, \text{Visual Basic 9}\}$.

LINQ maps the structure of queries into the host programming language, which allows rich possibilities for manipulation and optimization. However, to do this requires several language extensions, including:

- Lambda expressions
- Free-standing method declarations
- Structural datatypes
- Anonymous record types
- Type inference

These are new to C#, but based on well-established concepts from other existing languages.

Lambda expressions

Java *inner classes* and C# *delegates* allow for local declaration of methods:

```
int max = start + offset; // Some value calculated at run time

Func accept = delegate(int id){ return id < max; }

... userlist . filter (accept) ...
```

A *lambda* expression elides the declaration so that anonymous functions become first-class values:

```
... userlist . filter (id => (id<max)) ...
```

Extension methods

Object-oriented programming allows related classes to implement methods in different ways. With *extension methods*, a third party can add further methods to an existing class.

```
// Extension to String class from standard libraries
public static String Bracket(this String source, String pre, String post)
  { return pre+source+post; }

// This can be used as if it was part of the class from the start
String s = "Hello, World";
s.Bracket( "[", "]" );      // Invokes method Bracket(s, "[", "]" )
```

This is used for Where, Select and other LINQ methods.

Structural datatypes

Using *data-centric* programming in LINQ means that many classes serve only to hold structured values, without object-style state or behaviour.

To support this a new *object initialization* constructor creates a structured data value with an *anonymous type*:

```
object v = new { title = "OED", volumes = 20, mass = 65.68 };
```

For precise static typing in these cases, a new **var** keyword instructs the compiler to infer an appropriate type from the value provided.

```
var i = 42          // i is an int
var s = "Foo"       // s is a string
var v = new { left = 50, right = 100 } // v has an anonymous type
```

This means that later uses of the object v can be typechecked correctly.

Metaprogramming

In a final programming technology twist, LINQ to SQL and LINQ to XML pass on full details of how a query was constructed, to help with efficient evaluation. This is in the form of an *expression tree*, which can also include details of C# source code. For example:

```
Expression<Func<int,bool>> accept = (id => (id<max));
```

Now accept is not an executable function, but instead a data structure representing the given lambda expression.

LINQ presents the information needed to evaluate a query as an expression tree. By analyzing this, a complex expression combining several query operations might be executed in a single SQL call to the database.

This is a limited form of structured *metaprogramming*, where a program may inspect and work with code itself in a type-safe way.

Summary

- .NET is a large application development framework, with a common virtual machine, type system, and support for interlanguage working.
- LINQ manages queries from within the programming language, not as strings but as first-class entities.
- This uses a number of programming language features new to .NET.
- LINQ also introduces first-class expressions, the beginnings of structured reflection and metaprogramming.

This language/query integration goes deep: queries become meaningful data structures in the host language, not just raw strings of syntax.

Homework

Read this

Read the "Introduction to LINQ" piece from this MSDN Library text:

- Visual Studio 2010: LINQ (Language-Integrated Query)
  http://msdn.microsoft.com/en-gb/library/bb397926.aspx

Post this

Tuesday's lecture will be about metaprogramming in F#:

- Find an online tutorial about F#
- Post its URL to this lecture's entry on the blog

References

The MSDN documentation on LINQ is extensive and informative. These sections cover material particularly relevant to this lecture.

- C# Features That Support LINQ
  http://msdn.microsoft.com/en-gb/library/bb397909.aspx
- Getting Started with LINQ in C#
  http://msdn.microsoft.com/en-us/library/bb397933.aspx

Oracle have various Java tutorials covering JDBC. This section covers some of the material used in the lecture.

- Prepared Statements
  http://download.oracle.com/javase/tutorial/jdbc/basics/prepared.html