This is the final lecture 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 in F#
No Lecture on Tuesday

Next Lecture

There will be no lecture on Tuesday 21 October.

The next APL lecture is on Friday 24 October.
Outline

1. Metaprogramming
2. F#
3. Examples of metaprogramming in F# with LINQ
The LINQ framework for .NET integrates database queries into a host programming language. The integration goes deep: queries become meaningful data structures in the host language, not just raw strings of syntax.

This provides a more reliable interface for the programmer, as well as rich possibilities for manipulation and optimization by the compiler.

However, to do so requires several language extensions, including:

- Lambda expressions
- Extension methods
- Structural datatypes, type inference
- Expression trees

This last introduces *metaprogramming*, where code itself is exposed to programmatic manipulation.

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

```
... added to pre-existing classes
```

```
var v = new{left=50,right=100}
```

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

2 F#

3 Examples of metaprogramming in F# with LINQ
The term *metaprogramming* covers almost any situation where a program manipulates code, either its own or that of some other program. This may happen in many ways, including for example:

- Textual manipulation of code as strings
- Code as a concrete datatype
- Code as an abstract datatype
- Code generation at compile time or run time
- Self-modifying code
- Staged computation

Strictly speaking, any compiler or interpreter would qualify. However, the idea of metaprogramming usually indicates specific language features, or especially close integration between the subject and object programs.
Metaprogramming Examples

Macros

```c
#define geometric_mean(x,y) sqrt(x*y)

float size = geometric_mean(length,width)

#define BEGIN {
#define END }

#define LOOP(var,low,high) \
    for (int var=low; var<high; var++) BEGIN

int i, total = 0; LOOP(i,1,10) total=total+i; END
```

Here `geometric_mean` is an inlined function; while the *non-syntactic LOOP* macro builds code at compile time.
This template describes a general routine for adding vectors of arbitrary dimension. Compile-time specialization can give custom code for fixed dimensions if required. The C++ Standard Template Library does a lot of this kind of thing.
Metaprogramming Examples

Java reflection

```java
import java.io.*;
import java.lang.reflect.*;

Class c = Class.forName("java.lang.System"); // Fetch System class
Field f = c.getField("out"); // Get static field
Object p = f.get(null); // Extract output stream
Class cc = p.getClass(); // Get its class
Class types[] = new Class[] { String.class }; // Identify argument types
Method m = cc.getMethod("println", types); // Get desired method
Object a[] = new Object[] { "Hello, world" }; // Build argument array
m.invoke(p, a); // Invoke method
```

Reflection of this kind in Java and many other languages allows for programs to indulge in runtime *introspection*. This is heavily used, for example, by toolkits that manipulate Java *beans*. 
Metaprogramming Examples

**Javascript eval**

 eval("3+4"); // Returns 7

 a = "5-"; b = "2";
 eval(a+b); // Returns 3, result of 5-2

 eval(b+a); // Runtime syntax error

 a= "5-"; b = "1"; c = "a+a+b";
 eval(c); // Returns the string "5-5-1"
 eval(eval(c)); // Returns the number −1

Any language offering this has to include at least a parser and interpreter within its runtime.
Metaprogramming Examples

Lisp eval

```
(eval `( + 3 4)) ; Result is 7
```

```
(eval `(+ ,x ,x ,x)) ; Result is 3*x, whatever x is
```

```
(eval--after--load "bibtex"
  `(define--key bibtex--mode--map
     [(meta backspace)] 'backward--kill--word))
```

Unlike Javascript `eval`, code here is structured data, built using quote `'( ... )`, with no runtime syntax errors. The backquote or *quasiquote* `'( ... )` allows computed values to be inserted using the *antiquotation* comma `,(' ...')`. 
Arbitrary OCaml code can be quoted .< >., antiquoted with .~ and executed with .!. All these can be nested, giving a multi-stage programming language with detailed control over exactly what parts are evaluated when in the chain from source to execution.
Metaprogramming Examples

MetaOCaml

```ocaml
# let x = .< 4+2 >. ;;
val x : int code = .< 4+2 >.

# let y = .< .~x + .~x >. ;;
val y : int code = .< (4+2)+(4+2) >.

# let z = .! y ;;
val z : int = 12
```

Various research projects have implemented multi-stage versions of Scheme, Standard ML, Java/C# and so on.
This is *homogeneous* metaprogramming: the language at all stages is OCaml. There is a version of MetaOCaml that supports *heterogeneous* metaprogramming, with final execution of the code *offshored* into C.

(pun)
Outline

1. Metaprogramming

2. F#

3. Examples of metaprogramming in F# with LINQ
F# is a succinct, expressive and efficient functional and object-oriented language for .NET which helps you write simple code to solve complex problems.  

http://research.microsoft.com/fsharp, 2010-11-01

Easy F#

```fsharp
let rec fact n = match n with 0 -> 1 | n -> n * fact (n-1)

let build first last = System.String.Join( " ", [|first; last|] )

let name = build "Joe" "Smith"
```

Very roughly, F# is OCaml syntax with .NET libraries. That grew. A lot.
F# is a mature, open source, cross-platform, functional-first programming language which empowers users and organizations to tackle complex computing problems with simple, maintainable and robust code.

http://fsharp.org, 2014-10-17

Easy F#

```fsharp
let rec fact n = match n with 0 -> 1 | n -> n * fact (n-1)

let build first last = System.String.Join( " ", [|first; last|] )

let name = build "Joe" "Smith"
```

Very roughly, F# is OCaml syntax with .NET libraries. That grew. A lot.
F# at Microsoft Research

F# originated at Microsoft Research, Cambridge. The Microsoft Research team continues to partner with teams across Microsoft and with external open-source organizations, researchers, companies and users to break new ground in programming language systems, including F#. Microsoft Research staff continue to contribute to F#, its ecosystem and research based around F#.

F# is open source under an OSI-approved license (Apache 2.0) and is available across multiple platforms through the F# Software Foundation. You can contribute to F# in many ways, including through that organization.

http://research.microsoft.com/fsharp, 2014-10-17
Interoperability with the .NET framework and other .NET languages is central to F#.

- Core syntax is OCaml: with higher-order functions, lists, tuples, arrays, records, ...
- Objects are as in C#: with classes, inheritance, dot notation for field and method selection, ...
- .NET toys: extensive libraries, concurrent garbage collector, install-time/run-time (JIT) compilation, debuggers, profilers, ...
- Creates and consumes .NET/C# types and values; can call and be called from other .NET languages.
- Generates and consumes .NET code: can exchange first-class functions with other languages.
F# Timeline

- Developed by Don Syme at Microsoft Research Cambridge (MSR).
- Started as Caml.NET, with a first preview release of F# compiler in 2002/2003.
- 2005: MSR release V1.0, with basic Visual Studio integration.
- September 2008: Official Microsoft Community Technology Preview (CTP) release.
- April 2010: Visual Studio 2010 and .NET 4.0 releases with C#, VB, C++ and F# as its core languages.
- August 2014: F# on Windows, Linux, Mac, iOS, Android, BSD, HTML5, CUDA/GPU, ...
F# Timeline

- Developed by Don Syme at Microsoft Research Cambridge (MSR).
- Started as Caml.NET, with a first preview release of F# compiler in 2002/2003.
- 2005: MSR release V1.0, with basic Visual Studio integration.
- September 2008: Official Microsoft *Community Technology Preview* (CTP) release
- April 2010: Visual Studio 2010 and .NET 4.0 releases with C#, VB, C++ and F# as its core languages.
- August 2014: F# on Windows, Linux, Mac, iOS, Android, BSD, HTML5, CUDA/GPU, ...

“This is one of the best things that has happened at Microsoft ever since we created Microsoft Research over 15 years ago”

S. Somasegar, Head of Microsoft Developer Division, 2007-10-17
Outline

1 Metaprogramming

2 F#

3 Examples of metaprogramming in F# with LINQ
D. Syme
Leveraging .NET meta-programming components from F#: Integrated queries and interoperable heterogeneous execution.

Now the basis of several F#-to-GPU frameworks, with applications in large-scale asset management and deep machine learning.

Recall again that LINQ→SQL passes on 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.

Expression trees are built as required, and may include details of C# source code. For example:

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

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

This is quotation, but implicit: rather than having syntax to mark quotation of “(id => (id<max))”, the compiler deduces this from its type “Expression”. F# is different: code quotation is explicit, as in LISP or MetaOCaml.
Simple quote

> open Microsoft.FSharp.Quotations

> let a = <@ 3 @>;;
val a : Expr<int>

> a;;
val it : Expr<int> = <@ (Int32 3) @>

F# provides explicit code quotation markers. Here the interactive response exposes the internal structure of an expression.
Code Quotation in F#

Larger quote

```fsharp
> <@ "Hello " + "World" @>;;

val it : Expr<string>

= <@
    (App (App (Microsoft.FSharp.Core.Operators.op_Addition)
           ((String "Hello")))
      ((String "World")))
@>
```

A more complex quotation gives a more complex expression. Although verbose, the structure is exactly that of the original expression.
An expression of function type includes details of the function body. Here \texttt{x#39844.4} is a variable name chosen by the expression printer.
Quotation Templates

Quote with hole

> let f = <@ 5 + _ @>;;
val f : (Expr<int> -> Expr<int>)

> f a;;       // Remember that a is <@ 3 @>
val it : Expr<int>
= <@
(App (App (Microsoft.FSharp.Core.Operators.op_Addition) ((Int32 5)))
  ((Int32 3)))
@>

A quotation with one or more holes gives a function mapping expressions to expressions. An function “lift : 'a -> Expr<'a>” allows antiquotation, plugging in runtime values.
Quotation Templates

Splicing into a quotation

> let f x y = <@ %x + %y @>;;
val f : (Expr<int> ⇒ Expr<int> ⇒ Expr<int>)

> f a (lift (2+5));;  // Remember that a is <@ 3 @>
val it : Expr<int>
  = <@
    (App (App (Microsoft.FSharp.Core.Operators.op_Addition) ((Int32 3)))
      ((Int32 7)))
@>  // The expression <@ 3 + 7 @>

Quotation holes are one-off: the code splicing operator “%” helps to write more complex functions that build large expressions from smaller ones.
The query function will inspect an in-memory datastructure `db.Employees`, filtering those working in Edinburgh and projecting out their name and address.

Here `where` and `select` are versions of `filter` and `map` acting on records from the `db.Employees` data type.
Quoting the internals now gives a query function that will inspect an external database instead.
The SQL function takes a quoted expression and passes it to LINQ; which compiles it to SQL and then hands it off to the database engine as:

**SELECT** Name, Address **FROM** Employees **WHERE** City = "Edinburgh"
Query via SQL

```fsharp
val ( |> ) : 'a -> ('a -> 'b) -> 'b

let query = SQL
<@ fun db ->
    db.Employees
    |> where (fun e -> e.City = "Edinburgh")
    |> select (fun e -> (e.Name, e.Address)) @>
```

Notice that the SQL function is working with an expression representing F# source, including lambdas and first-class functions “where” and “select”.
Application: F# to SQL by LINQ

Query via SQL

```fsharp
val ( |> ) : 'a -> ('a -> 'b) -> 'b

let query = SQL
<@ fun db ->
    db.Employees
    |> where (fun e -> e.City = "Edinburgh")
    |> select (fun e -> (e.Name, e.Address)) @>
```

This heterogeneous metaprogramming leads to some mismatches between F# and SQL semantics: for example, SQL date/time is rounded to 3msec, less precise than .NET, and the definition of Math.Round is different.
Powers of x

> let rec power (n,x) = if n = 0 then 1 else x*power(n-1,x);;
val power : int * int -> int

> let power4 = fun x -> power (4,x);;
val power4 : int -> int

> power4 5;;
val it : int = 625

Although power4 always calls power with the fixed value 4, this will still run the general-purpose code which uses a loop and a counter.
Powers of x

> let rec metapower (n,x) =
| – if n = 0
| – then <@ 1 @>
| – else <@ _ * _ @> (lift x) (metapower(n−1,x)) ;;
val metapower : int * int -> Expr<int>

> let metapower4 = fun x -> metapower (4,x) ;;
val metapower4 : int -> Expr<int>

This metapower function computes $x^n$ as an expression rather than a value.
Powers of x

```fsharp
> metapower4 5;;
val it : Expr<int> = <@
  (App (App (Microsoft.FSharp.Core.Operators.op_Multiply) (5))
   (App (App (Microsoft.FSharp.Core.Operators.op_Multiply) (5))
    (App (App (Microsoft.FSharp.Core.Operators.op_Multiply) (5))
     (App (App (Microsoft.FSharp.Core.Operators.op_Multiply) (5))
      ((Int32 1)))))) @>
```

Executing `metapower4` runs the loop over F# code, not values, giving us the expression that would compute $5^4$.
Powers of x

> metapower4 5;;

val it : Expr<int> = <@
  (App (App (Microsoft.FSharp.Core.Operators.op_Multiply) (5))
  (App (App (Microsoft.FSharp.Core.Operators.op_Multiply) (5))
  (App (App (Microsoft.FSharp.Core.Operators.op_Multiply) (5))
  (App (App (Microsoft.FSharp.Core.Operators.op_Multiply) (5))
  ((Int32 1)))))) @>

This expression can be passed to LINQ, for appropriate compilation and then execution as .NET bytecode.
Application: F# Runtime Code Generation

Powers of $x$

```fsharp
> metapower4 5;;

val it : Expr<int> = <@
(App (App (Microsoft.FSharp.Core.Operators.op_Multiply) (5))
 (App (App (Microsoft.FSharp.Core.Operators.op_Multiply) (5))
  (App (App (Microsoft.FSharp.Core.Operators.op_Multiply) (5))
   (App (App (Microsoft.FSharp.Core.Operators.op_Multiply) (5))
    ((Int32 1)))))) @>
```

LINQ provides lightweight code generation: at runtime the code is built, JIT compiled, run, and then garbage collected away.
Conway’s Game of Life

1. A cell with exactly three neighbours comes to life.
2. A live cell with two or three neighbours stays alive.
3. All other cells die.

let matrix f = Array2.init x y f  // Build x*y array filled with f x y
...

let neg a = matrix (fun i j -> - a.(i,j))
let (+) a b = matrix (fun i j -> a.(i,j) + b.(i,j))
let (&&) a b = matrix (fun i j -> a.(i,j) && b.(i,j))

let rotate a dx dy = matrix (fun i j -> a.((i+dx)%x,(j+dy)%y))
let count a = matrix (fun i j -> int_of_bool a.(i,j))

let nextGeneration(a) =  // Take one step in Conway’s Life
  let N dx dy = rotate (count a) dx dy in
  let sum = N (-1) (-1) .+ N (-1) 0 .+ N (-1) 1
          .+ N 0 (-1) .+ N 0 1
          .+ N 1 (-1) .+ N 1 0 .+ N 1 1 in
  (sum .= three) .| | (sum .= two) .&& a);;
Application: Accelerating F# by Outsourcing

open Microsoft.ParallelArrays // Use e.g. GPU pixel shader

let shape = [| x; y |] // Fixed dimensions x,y

let And (a:FPA) (b:FPA) = FPA.Min (a, b) // Built-in operations on floating-point arrays

let Or (a:FPA) (b:FPA) = FPA.Max (a, b)

let Rotate (a:FPA) i j = a.Rotate([| i; j |])

let nextGenerationGPU (a:FPA) = // Take one step in Conway's Life

let N dx dy = Rotate a dx dy in

let sum = N (-1) (-1) .+ N (-1) 0 .+ N (-1) 1
     .+ N 0 (-1) .+ N 0 1
     .+ N 1 (-1) .+ N 1 0 .+ N 1 1 in

Or (Equals sum three) (And (Equals sum two) a);;
Using the *Accelerator* data-parallel library to drive an alternative computing engine is neat, but we did have to rewrite the code.
Using the Accelerator data-parallel library to drive an alternative computing engine is neat, but we did have to rewrite the code.

As an alternative to writing new code for this particular application, we can write a general GPU translator that works over any expression:

```
val accelerateGPU : ('a[,] -> 'a[,] expr -> 'a[,] -> 'a[,]]
```
Using the *Accelerator* data-parallel library to drive an alternative computing engine is neat, but we did have to rewrite the code.

As an alternative to writing new code for this particular application, we can write a general GPU translator that works over any expression:

```
val accelerateGPU : ('a[,] -> 'a[,] ) expr -> 'a[,] -> 'a[,] 
```

All we need do to run life on the GPU is then:

```
let nextGenerationGPU' = accelerateGPU <@ nextGeneration @> 
```

Caveat: The semantic mismatches are now more serious — actual floating-point arithmetic on GPU and CPU may not be bit-identical.
Using the *Accelerator* data-parallel library to drive an alternative computing engine is neat, but we did have to rewrite the code.

As an alternative to writing new code for this particular application, we can write a general GPU translator that works over any expression:

\[
\text{val accelerateGPU : ('a[,] \rightarrow 'a[,] \rightarrow 'a[,])}
\]

All we need do to run life on the GPU is then:

\[
\text{let nextGenerationGPU' = accelerateGPU <@ nextGeneration @>}
\]

This idea has been taken up and commercialised by *QuantAlea* in their *Alea.Cubase* product.
WELCOME
We are a software and solution provider for high performance and GPU computing. Our strength is a unique blend of expertise, combining first class software engineering and implementation skills with specific industry knowledge.

PRODUCTS
Alea.cuBase V1 1.3.914
The new version 1.3 of Alea.cuBase is a major step adding full support for debugging and several new features.

LATEST NEWS
Language Integrated Compiler
March 30, 2014
View: These are the latest developments of our company.

LATEST BLOGS
Alea.cuBase 1.3 released
Aug. 5, 2014
We are very pleased to announce the long awaited new version 1.3 of...
The interaction of the different components and modules is schematically explained in the next illustration.

Example

The following example illustrates the usage of Alea.cuBase by means of a simple kernel adding together two large arrays. First we write the parallel template and the launch logic.

```csharp
open Alea.CUDA

let pfunct = cuda {
    let! kernel =
        <@ fun (C:DevicePtr<float>) (A:DevicePtr<float>)
            (B:DevicePtr<float>) ->
        let tid = threadIdx.x
    |> defineKernelFunc

    return PFunc (fun (m:Module) (A:float[]) (B:float[]) ->
        let length = A.Length
        use A = m.Worker.Malloc(A)
        use B = m.Worker.Malloc(B)
        use C = m.Worker.Malloc<float>(length)
        let lp = LaunchParam(1, length)
        kernel.Launch m lp C.Ptr A.Ptr B.Ptr C.ToHost())
}
```

Launching logic with device memory management

GPU kernel function
Metaprogramming ranges from syntactic expansion through hygienic macros to staged computation and runtime code generation.

F# is an ML for .NET, with an emphasis on interlanguage working.

Quotations and templates bring metaprogramming to F#.

F# can use LINQ to generate SQL . . .

. . . or native code at runtime . . .

. . . or to outsource execution wherever seems best.
Homework

The next topic will be on language-based methods for checking and certifying correct code. In anticipation, you should revise what you know about *propositional logic* and *predicate logic*.

Read the following:

- Slides from *Discrete Mathematics and Mathematical Reasoning*
  - Lecture 2: Review of Propositional Logic
  - Lecture 3: Predicate Logic
    http://www.inf.ed.ac.uk/teaching/courses/dmmr/schedule.html

You might also find this helpful, on the same topic:

  http://is.gd/mitcslogic1