Lecture 7

August 11, 2016

Computational complexity, Algorithms and data structures for text processing

Reminder: start the recording

Announcements

- Reminder: Graduate Non-Matriculated (GNM) status
  - If you think you might convert to CLMS at some point, you *must obtain GNM status before starting your fall classes!*
- Today's lecture
  - Computational complexity
  - Specialized algorithms for text processing

Project 3: Encoding issues

1. Your text editor
   – Can your editor create, display and save a file with wide literals (typ. UTF-8)?
2. Language support for wide characters
   – Does your compiler support wide literals in source code files?
   – If so, does it need/expect/reject a BOM? Or do you perhaps need to specify the source file's encoding on the compiler command line?
   – If not, does it support wide characters through via ASCII escape?
3. Utility programs:
   – Does your FTP program correctly understand UTF-8 files when it tries to convert line-endings from Windows to Unix? (try binary mode)
4. At runtime:
   – Does your environment have functions for reading unicode files?
   – Which input file are you trying to read? { UTF-8, UTF-16LE, TIS-620 }
   – Does your environment have functions for writingunicode files?
   – What output encoding are you trying to write? { UTF-8, UTF-16LE, TIS-620 }

Using HTML <meta> tag to specify encoding

```
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
```

Additional step for reading and writing:

- Save the file using the specified encoding
- Make sure your editor is set to read or write the encoding
- This requires using an editor that is capable of doing so

Additional step for using the page on a web server

- Configure the web server to send a matching HTTP header
- This is an out-of-band mechanism for specifying the content encoding of every single web page

```
HTTP/1.1 200 OK
Date: Mon, 23 May 2005 22:38:34 GMT
Server: Apache/1.3.3.7 (Unix)  (Red-
Hat/Linux) Last-Modified: Wed, 08 Jan 2003
23:11:55 GMT Etag: "3f80f-1b6-3e1cb03b"
Accept-Ranges:
bytes Content-
Length: 438
Connection: close
```

Programming language support for encodings

```
String s = "สวัสดีครับ";        // C# strings are always unicode
int i = s.Length;              // number of code points: 10
Char ch = s[0];                // always a 16-bit character.
                               // In this case the value is 3626 (U+0E2A)

Byte[] bytes = Encoding.GetEncoding("TIS-620").GetBytes(s);
i = bytes.Length;              // number of bytes: 10
byte b = bytes[0];             // a byte. In this case, the value is 202 (\xCA)

bytes = Encoding.UTF8.GetBytes(s);
i = bytes.Length;              // number of bytes: 30

s = Encoding.UTF8.GetString(bytes);     // back to the original string
```

Using C#

With your UW-NetID, you can download and install the full version of Microsoft Visual Studio 2013 Professional for free: http://www.dreamspark.com

To compile a C# program on patas:

```
/home2/joe-student$ gmcs project2-b.cs
```

To run it:

```
/home2/joe-student$ mono project2.exe
```

Algorithmic Complexity

- Algorithms can be characterized according to how their use of a resource scales with the size of the input(s).
- We typically look at two resources
  - space (memory consumption)
  - time (execution time) How long for the program to run

we don't think about developer time?

Technically, these are ind<u>e</u>pendent *per algorithm*, but we often think of a family of algorithms that represent a spectrum of trade-offs between space and time

Example: space/time trade-off

$$ff(xx) = \begin{cases} 0 & iiff\ xx = 'T' \\ 1 & iiff\ xx = 'C' \\ 2 & iiff\ xx = 'A' \\ 3 & iiff\ xx = 'G' \\ -1 & 0000o00000oo ii0000 \end{cases}$$

Example: space/time trade-off

4 time steps
0 memory

```
int f(char ch)
{
    if (ch == 'T')
        return 0;
    if (ch == 'C')
        return 1;
    if (ch == 'A')
        return 2;
    if (ch == 'G')
        return 3;
    return -1;
}
```

1 time step
256b "extra" memory

```
// one-time initialization:
int[] map = Enumerable.Repeat<int>(-1, 256).ToArray();
map['T'] = 0;
map['C'] = 1;
map['A'] = 2;
map['G'] = 3;
// ...

int f(char ch)
{
    return map[ch];
}
```

Best/average/worst case

- If an algorithm's performance depends on more than just the size of the input, what is the possible range of complexity for a given input?
- Examples:
  - a sort algorithm may perform differently depending on the input ordering
    - intuitively, shouldn't sorting have complexity that's inversely proportional to the entropy of the input?
    - Xiaoming Li et al. (2007) *Optimizing Sorting with Machine Learning Algorithms*. IEEE
  - inserting into a balanced tree may perform differently depending on the current geometry

Asymptotic notations

- Express resource use in terms of input size
- Discard factors that become asymptotically irrelevant

Name	Notation	Expresses
Big-O	$f(n) = O(g(n))$	Upper bound
Big Omega	$f(n) = \Omega(g(n))$	Lower bound
Big Theta	$f(n) = \Theta(g(n))$	Upper and lower bound

- Usually when people say $O(n^2)$ "O of n-squared" etc. they actually mean $\Theta(n^2)$ "Theta of n-squared"
  - because usually, analysis of best case, worst case, and average case must be done separately
  - I will continue this tradition of abuse (i.e. on the next slide)

Complexity Classes

$OO\ kk$	constant	hash table
$OO(\log nn)$	logarithmic	binary search
$OO(nn)$	linear	naïve search (best and worst case)
$OO(nn \log nn)$	log-linear	quicksort (best case)
$OO(nn^2)$	quadratic	naïve sort (best and worst case)
$OO(nn^3)$	cubic	parsing context-free-grammar (worst case)
$OO(nn^{kk})$	polynomial	2-SAT (boolean satisfiability)
$OO(kk^{nn})$	exponential	traveling salesman DP
$OO(nn!)$	factorial	naïve traveling salesman

worst case

Sorting?

Complexity scaling

- Consider an algorithm containing ~~8 million~~ instructions
- State-of-the-art processor 2010 (1.48×10^{11} instructions per second)

Problem size $nn = 1111$ → $nn = 333333$ Problem size

$OO(1)$	54 $\mu\mu$s.
$OO(\log nn)$	58 $\mu\mu$s.
$OO(nn)$	649 $\mu\mu$s.
$OO(nn \log nn)$	700 $\mu\mu$s.
$OO(nn^2)$	7.8 ms.
$OO(nn^3)$	93 ms.
$OO(nn^{kk}), kk = 8$	23 sec.
$OO(kk^{nn}), kk = 8$	1032 hours
$OO(nn!)$	2,231,000 centuries

$OO(1)$	54 $\mu\mu$s.
$OO(\log nn)$	137 $\mu\mu$s.
$OO(nn)$	19 ms.
$OO(nn \log nn)$	48 ms.
$OO(nn^2)$	6.6 sec.
$OO(nn^3)$	38.6 min.
$OO(nn^{kk})$	387 million years
$OO(kk^{nn})$	age of universe $\times 10^{294}$
$OO(nn!)$	age of universe $\times 10^{718}$

Improving best case complexity: example?

- Quicksort is $O(n \log n)$ best case
- Checking that an array is already sorted always obtains its worst case $O(n)$ when successful
- Let's add this step to improve quicksort:

$$O(n \log n) + O(n) = O(n \log n)$$

- So why don't we do this in practice?

Hashing

- Hashing is crucial for indexing very large datasets
- HashSet: $O(1)$ test for membership
- "Dictionary" or "HashMap" or "Associative array" etc. $O(1)$ lookup
  - directly access one item based on the value of another
- All modern programming languages have hashing support
  - you usually won't need to write your own hash functions

Hash function

A hash function ff directly computes the index uu of an element xx in array gg:

Find signature of an item.

Summarize with signature

$$uu = ff(xx),\ uu \in \mathbb{Z}$$

$$gg_{uu} := xx$$

Signature always the same for each item

The ideal function should have the following properties:

- $OO(1)$ to compute
  - in practice, $ff(xx)$ is often $OO(nn)$ in the size of a single input
- Generate unique uu for every xx
- uu is uniformly distributed over some range for all possible values of xx

Hashing issues

- Collisions – gives same hash for duplicate items
- Quality of Hash function
- Perfect hash functions – No collisions

- Might have different hashes to account for different properties.
-

Memory address space of your computer (all in same linear array)
can not change the index number

Hash function example in C

```c
unsigned int HashWord(char *p)
{
    union
    {
        struct  // composition of the 32-bit hash:
        {
            unsigned int cch :  5;          // number of characters
            unsigned int fc :   6;          // first character value
            unsigned int lc :   6;          // last character value
            unsigned int maxc : 6;          // max character value
            unsigned int cks :  9;          // checksum
        };
        unsigned int raw;
    } h;
    h.raw = 0;
    h.fc = *(unsigned char *)p - 'A';
    char *p0 = p;
    while (*p)
    {
        h.lc = *(unsigned char *)p - 'A';
        h.cks += h.lc;
        if (h.lc > h.maxc)
            h.maxc = h.lc;
        p++;
    }
    h.cch = p-p0;
}
```

4 billion possibilities
32 bit

produce hash for string
computes a check sum over the string
guarantees every character has opportunity to add to hash.

Workhorse data structures

All modern languages have a number of off-the-shelf collection objects. For example C# offers the following, all strongly-typed:

T[] (Array<T>)	Fixed-size, ordered list of objects or values of type T
List<T>	Dynamically-sized, ordered list of objects or values of type T
Stack<T>	LIFO stack of objects or values of type T
Queue<T>	FIFO queue of objects or values of type T
LinkedList<T>	Doubly-linked list of objects or values of type T
HashSet<T>	Hash table of objects or values of type T
Dictionary<TKey,TValue>	Table of objects of type TValue hashed by objects of type TKey
SortedDictionary<TKey,TValue>	Sorted table of objects of type TValue hashed by objects of type TKey
ILookup<Tkey,Telement>	Multimap (an immutable dictionary that allows duplicate keys)

Optimization fever

"Early optimization is the root of much evil."
- Donald Knuth (or, misattributed to Sir C. Anthony R. Hoare, developer of quicksort)

"Make everything as simple as possible, but not simpler."
- Albert Einstein

The best—and uncontroversial—advice:
measure, measure, measure

Measuring performance

- Theoretical knowledge of your algorithms
  - Big-O complexity classes
  - Combining classes
- Actual stopwatch measurements

```
using System;
using System.IO;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Diagnostics;

class MyProgram
{
    static void Main(string[] args)
    {
        Stopwatch stopw = Stopwatch.StartNew();
        // ... do stuff ...
        Console.Error.WriteLine(stopw.ElapsedMilliseconds);
    }
}
```

Time yourself

Multithreading?

- This can be a real boost, but how well do you know your hardware?
- The 8-processor machines in the treehouse cluster – each support 8 Condor nodes
- To do 50-way multiprocessing on Condor, issue 50 single-threaded jobs

Example

- For Ling573, our LSI/SVD-based document summarization system needed to do intensive processing of 50 documents for each system tuning trial
- We wrote a program that automatically issues each document as a separate Condor job
- We were able to run over 200 trials, 140 documented experiments
- More trials → better tuning → better system

method	time
Single-threaded Condor job	28:05
50 Condor jobs	4:45
3.2 GHz 8-way Windows Server	2:58

Immutable strings

- For modern programming languages, a key performance bottleneck is garbage collection
- Immutable strings provide many benefits, but GC activity can soar if you're not careful

Disaster

```
String s = File.ReadAllText("chr1.dna");
s = s.ToUpper();
```

& its 8-bit on the server.

Read mass bytes

- If you're going to be modifying a string, use mutable character arrays or special mutable "StringBuilder" objects instead

C# in ten slides or more

- C# primitive data types
  - `byte, sbyte`
  - `int, uint`
  - `long, ulong`
  - `float, double`
  - `Char, String`
  - `array[T]`
- User-defined value types
- User-defined objects (reference types)

Extending C# programs

- Like java, C# functions must be in classes
- If you don't need (or want) to define object classes, you can use static functions in a static class

```
static class Program
{
    static void Main(string[] args)
    {
        int q = MyFunc("cheeze", 3.14, new int[] { 1, 3, 5, 6 });
        Console.WriteLine(q); // prints 14
    }

    static int MyFunc(String s, double d, int[] arr_of_int)
    {
        return arr_of_int.Where(i => i > 2).Sum();
    }
}
```

C# main ‘Program’ class

- Creating an instance of the class that implements your program

```
class Program
{
    static void Main(string[] args)
    {
        var p = new Program();

        var tokens = p.scan(Console.In.ReadToEnd());

        /// ...
    }
```

Defining a class

```
class person
{
      public person(String name, int age)
      {
            this.name = name;
            this.age = age;
      }
      public String name;
      public int age;
};
```

```
person p = new person("นงนุช", 23);
```

Console I/O

- Reading from and writing to the console

```
using System;
using System.IO;
using System.Collections.Generic;
using System.Linq;
using System.Text;

/// ...
String txt_in = Console.In.ReadToEnd();

Console.WriteLine(txt_in);
```

Formatting text output

- Writing formatted text to the console

```
int i =  4;
String s = "foo";
Console.WriteLine("'i' equals {0}, and 's' equals {1}", i, s);
```

- Creating a formatted string
  - this example shows 8-digit hexadecimal format

```
int i = 1234;
String s2 = String.Format("{0:X8}", i);
```

C# data types

- C# operates in a garbage-collected environment
  - use ‘new’ to create instances
  - you do not have to free() any instances that you create

```
int x = 1234;
int[] array_of_ints = new int[x];

array_of_ints[999] = 3;
```

C# Dictionary<TKey,TValue>

```
Dictionary<String, int> dict = new Dictionary<String, int>();

dict.Add("number one", 1);    /// add a new value
dict["number four"] = 4;      /// add or change
dict["number four"] = 4444;   /// change existing value
int x = dict["number_one"];   /// retrieve existing value

int y;
if (dict.TryGetValue("number six", out y))
    Console.Write("found it");
else
    Console.Write("did not find it");

if (!dict.ContainsKey("number five"))
    dict.Add("number five", 5);

Console.WriteLine("There are {0} items.", dict.Count);
```

C# List<T>

```
List<double> radii = new List<double>();

radii.Add(12.2);

double q = Math.PI * Math.Pow(radii[0], 2);

radii.Remove(7.0);      // returns 'false' if not found
radii.RemoveAt(0);      // remove at index zero

radii.Sort();
```

IEnumerable, yield, and deferred execution

- Before describing the trie data structure, let's look at iterators which enumerate a sequence of elements

> Examples in C#. If you use another language, it will be instructive to think about how to adapt the solutions to your language

- Enumeration is obvious when the data is at hand and you want to use it all:

```
String[] data = { "able", "bodied", "cows", "don't", "eat", "fish" };

foreach (String s in data)
    Console.WriteLine(s);
```

We can pass (a reference to) the array around too, no problem

```
String[] data = { "able", "bodied", "cows", "don't", "eat", "fish" };
// ...
ProcessSomeStrings(data);
// ...

void ProcessSomeStrings(String[] the_strings)
{
    foreach (String s in the_strings)
        Console.WriteLine(s);
}
```

What if we only want to "process" the four-letter words?

```
String[] data = { "able", "bodied", "cows", "don't", "eat", "fish" };
// ...

List<String> filtered = new List<String>();
foreach (String s in data)
    if (s.Length == 4)
        filtered.Add(s);
ProcessSomeStrings(filtered);
// ...

void ProcessSomeStrings(List<String> the_strings)
{
    foreach (String s in the_strings)
        Console.WriteLine(s);
}
```

This doesn't seem very nice. For one thing, we have to use more memory and waste time copying the elements we care about to a new list.

Is there a way to pass this function enough information to filter the *original list* itself, where it lies?

- Remember the non-filtered example for a second

```
void ProcessSomeStrings(String[] the_strings)
{
    foreach (String s in the_strings)
        Console.WriteLine(s);
}
```

- The processing function doesn’t really care about the fact that the data is in an array
- This violates an important programming maxim:

A flexible interface *demands the least* and *provides the most*:

- *Inputs* are as general as possible (allowing clients to supply any level of specificity, i.e. be lazy)
- *Outputs* are as specific as possible (allowing clients to capitalize on work products, i.e. be lazy).

```
void ProcessSomeStrings(String[] the_strings)
{
    foreach (String s in the_strings)
        Console.WriteLine(s);
}
```

The extra (unused) demands this function is making by asking for String[]:

- That the strings all be in memory at the same time
- That the strings be randomly accessible by an index
- That the number of strings be known and fixed before the function starts

- To modify this to comply with the maxim, we first ask:
- Q: What is the absolute minimum that this function actually needs to accomplish it's work?
- Answer: a way to iterate strings

Interfaces

- `IEnumerable<T>` is one of many system-defined interfaces that a class can elect to implement

An interface is a named set of zero or more function signatures with no implementation(s)

- To implement an interface, a class defines a matching implementation for every function in the interface
- Interfaces are sometimes described as contracts
- You can define and use a reference to an interface just like any other object reference

Contrived Example

```
interface IPropertyGetter
{
    String GetColor();
}

class Strawberry : IPropertyGetter
{
    public String GetColor() { return "red"; }
}

class Ferrari : IPropertyGetter
{
    public String GetColor() { return "yellow"; }
}
```

- This looks like C++ class inheritance
  - yes, but it's more ad-hoc
  - C# classes can have single inheritance of other classes, and multiple inheritance of interfaces
  - Interfaces can inherit from other interfaces (not shown)

IEnumerable<T>

- This is one of the simplest interfaces defined in the BCL (base class libraries)
- This interface provides just one thing: a way to iterate over elements of type T
- All of the system arrays, collections, dictionaries, hash sets, etc. implement `IEnumerable<T>`
  - Implementing IEnumerable<T> on your own classes can be very useful, but you don't need to worry about that
  - For now, what's important is that you get to use it, because it's available on all of the system collections

IEnumerator<T>

- IEnumer**able**<T> has only one function, which allows a caller or caller(s) to obtain an *enumerator* object which is able to iterate over elements
  - The actual enumerator object is an object that implements a different interface, called IEnumera**tor**<T>
  - This "factory" design allows a caller to initiate and maintain several simultaneous iterations if needed
  - The enumerator object, IEnumerator<T> can only:
    - Get the current element
    - Move to the next element
    - Tell you if you've reached the end
  - Note: There's no count
    - ICollection inherits from IEnumerable to provide this

Interfaces as function arguments

- Using interfaces as function arguments allows you to require the absolute minimum functionality the function actually needs
- In this way, the ad-hoc nature of interfaces allows us to comply with the maxim

```
void ProcessSomeStrings(IEnumerable<String> the_strings)
{
    foreach (String s in the_strings)
        Console.WriteLine(s);
}
```

Now, this function is exposing the weakest (most general) requirement possible for the processing it has to do. This provides more flexibility to callers since they can choose whatever level of specificity is convenient. The function can be used in the widest possible variety of situations.

Example

```
String[] d1 = { "able", "bodied", "cows", "don't", "eat", "fish" };
ProcessSomeStrings(d1);

List<String> d2 = new List<String> { "clifford", "the", "big", "red", "dog" };
ProcessSomeStrings(d2);

HashSet<String> d3 = new HashSet<String> { "these", "must", "be", "distinct" };
ProcessSomeStrings(d3);

Dictionary<String,int> d4 =
        new Dictionary<String, int> { { "the", 334596 }, { "in", 153024 } };
ProcessSomeStrings(d4.Keys);

void ProcessSomeStrings(IEnumerable<String> the_strings)
{
    foreach (String s in the_strings)
        Console.WriteLine(s);
}
```

Python users might not be impressed, but the difference is that this is all 100% strongly typed

Iteration is efficient

- That’s cool, IEnumerable<T> lets a function not care about where a sequence of elements is coming from
  - We don’t copy the elements around
  - Iterators let us access elements right from their source
- All of those examples iterate over elements that already exist somewhere
- Is there a way to iterate over data that’s generated on-the-fly, doesn’t exist yet, or is never persisted at all?
- Yes!

Iterating over on-the-fly data

```
IEnumerable<String> GetNewsStories(int desired_count)
{
    for (int i = 0; i < desired_count; i++)
        yield return RealtimeNewswireSource.GetLatestStory();
}
```

see next slide

```
// ...
IEnumerable<String> d5 = GetNewsStories(7);
ProcessSomeStrings(d5);
// ...
```

This is exactly the same as before, but this time there's no "collection" of elements sitting anywhere

```
void ProcessSomeStrings(IEnumerable<String> the_strings)
{
    foreach (String s in the_strings)
        Console.WriteLine(s);
}
```

This function doesn't care. In fact, it can't even tell.

yield keyword

- The `yield` keyword makes it easy to define your own custom iterator functions
- Any function that contains the `yield` keyword becomes special
  - It must be declared as returning an IEnumerable<T>
  - Deferred execution means that the function's body is not necessarily invoked when you "call" it
  - It must deliver zero or more elements of type T using:

    ```
    yield return t;
    ```
  - Sometime later, control may continue immediately after this statement to allow you to yield additional elements
  - It may signal the end of the sequence by using:

    ```
    yield break;
    ```

Custom iterator function example

```
IEnumerable<String> GetNewsStories(int desired_count)
{
    for (int i = 0; i < desired_count; i++)
        yield return RealtimeNewswireSource.GetLatestStory();
}

// ...
IEnumerable<String> d5 = GetNewsStories(7);
ProcessSomeStrings(d5);
// ...

void ProcessSomeStrings(IEnumerable<String> the_strings)
{
    foreach (String s in the_strings)
        Console.WriteLine(s);
}
```

code from this custom iterator function is *not* executed at this point.

d5 refers to an iterator that "knows how" to get a certain sequence of strings when asked

This finally demands the strings, causing our custom iterator function to execute–interleaved with this loop!

Closures

- Lambda expressions automatically capture local variables that they reference, which are then passed around as part of the lambda variable
  - Caution: languages do this differently with respect to reference (the lambda expression will modify the original value) versus value (the lambda expression has a snapshot of the value)
- This can lead to interesting scoping issues

Lambda expressions

```
// recall Select(ch => ('a' <= ch && ch <= 'z') || ch == '\'' ? ch : ' ');

String s = "Al's 20 fat-ish oxen.";
Func<Char, Char> myfunc = (ch) => ('a' <= ch && ch <= 'z') || ch == '\'' ? ch : ' ';
IEnumerable<Char> iech = s.Select(myfunc);
// iech is now a deferred enumerator for the characters in: " l's    fat ish oxen "

Func<Char, Char> myfunc = (ch) =>
    {
        if ('a' <= ch && ch <= 'z')
            return ch;
        if (ch == '\'')
            return ch;
        return ' ';
    };

Func<String, int, bool> string_is_longer_than = (s, i) => s.Length > i;

bool b = string_is_longer_than("hello", 3);     // true
```

Closure example

```
int x = 3;
Action a = () =>
    {
        Console.WriteLine(x);
    };
a();          // prints 3
x = 5;
a();          // prints 5
```

State machine example

Project 3

```
using System;
using System.Collections.Generic;
using System.Linq;

static class Program
{
    static class MainClass
    {
        enum State { Zero, One, Two };

        static Dictionary<State, Func<Char, State>> machine = new Dictionary<State, Func<Char, State>>
        {
            {
                State.Zero, (ch) => { return State.Two; }
            },
            {
                State.One, (ch) => { return State.One; }
            },
        };

        static void Main(String[] args)
        {
            String s = "the string to parse";
            int i = 0;

            State state = State.Zero;
            while (i < s.Length)
                state = machine[state](s[i++]);
        }
    }
}
```

lambda

A dictionary of lambda functions

Given a state Id, returns a piece of code you can run.

The state machine

LINQ in C#

- Sequences: IEnumerable<T>
- Deferred execution
- Type inference
- Strong typing
  - despite the ‘var’ keyword
  - (C# 4.0 now has the `’dynamic’` keyword, which allows true runtime typing where desired)

LINQ operators

- Filter/Quantify:
  `Where, ElementAt, First, Last, OfType`
- Aggregate:
  `Count, Any, All, Sum, Min, Max`
- Partition/Concatenate:
  `Take, Skip, Concat`
- Project/Generate:
  `Select, SelectMany, Empty, Range, Repeat`
- Set:
  `Union, Intersect, Except, Distinct`
- Sort/Ordering:
  `OrderBy, ThenBy, OrderByDescending, Reverse`
- Convert/Render:
  `Cast, ToArray, ToList, ToDictionary`

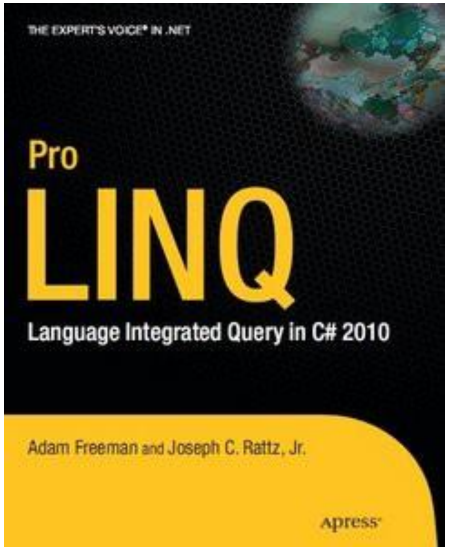
Example

```
Dictionary<String, int> pet_sym_map =
    File.ReadAllLines("/programming/analytical-grammar/erg-funcs/pet-symbol-key.txt")
    .Select(s => s.Split(sc7, StringSplitOptions.RemoveEmptyEntries))
    .Where(rgs => rgs.Length == 3)
    .Select(rgs => new { id = int.Parse(rgs[0]), sym = rgs[1].Trim().ToLower() })
    .GroupBy(a => a.sym)
    .Select(grp => grp.ArgMin(a => a.id))
    .ToDictionary(a => a.sym, a => a.id);
```

C# LINQ (Language Integrated Query)

- Declarative operations on sequences
- Recommendation:

Joseph C. Rattz, Jr. (2007) *Pro LINQ: Language Integrated Query in C# 2008*. Apress.



Programming Paradigms: Procedural

- Procedural ("imperative") programming
  - FORTRAN (1954) grew out of hardware assembly languages, which are necessarily procedural
  - We explicitly specify the (synchronous) steps for doing something (i.e. an algorithm)

```
int Factorial(int n)
{
        int f = 1;
        for (int i = n; i > 1; i--)
                f *= i;
        return f;
}
```

Functional Programming

- A type of declarative programming
  - Constraint-based syntax formalisms such as unification grammars (LFG, HPSG, …) are also declarative
- Like function definitions in math, the "program" describes asynchronous relationships
- Functions are stateless and should have no side-effects
- Immutable values
  - As a bonus, this really facilitates concurrent programming
- Scheme, Haskell, F#

F# Example

- F# interactive on patas:

```
$ mono /opt/fsharp/bin/fsi.exe --gui-
          (you must use an ANSI terminal)
```

```
> let rec factorial = function
    | 0 -> 1
    | n -> n * factorial(n -
1);; val factorial : int -> int
> factorial 5;;
val it : int =
120
> #quit;;
```