Languages and Compilers (SProg og Oversættere)

Bent Thomsen

Department of Computer Science

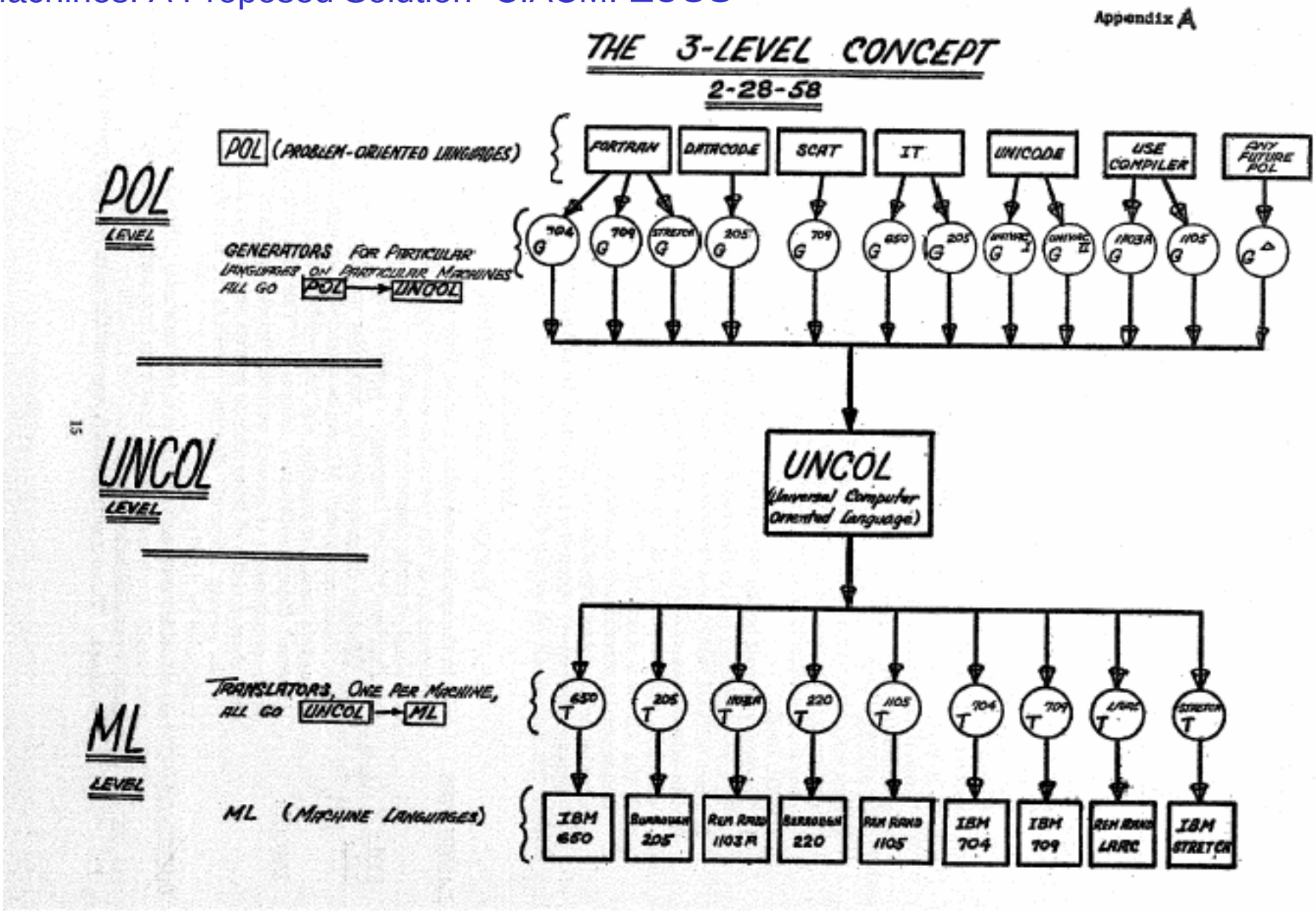
Aalborg University

With acknowledgement to Microsoft, especially Nick Benton, whose slides this lecture is based on.

The common intermediate format nirvana

- If we have n number of languages and need to have them running on m number of machines we need m*n compilers!
- If we have one common intermediate format we only need n front-ends and m back-ends, i.e. m+n
- Why haven't you taught us about the common intermediate language?

Strong et al. “The Problem of Programming Communication with Changing Machines: A Proposed Solution” C.ACM. **1958**



Quote

This concept is not particularly new or original. It has been discussed by many independent persons as long ago as 1954. It might not be difficult to prove that “this was well-known to Babbage,” so no effort has been made to give credit to the originator, if indeed there was a unique originator.

“Everybody knows that UNCOL was a failure”

- Subsequent attempts:
  - Janus (1978)
    - Pascal, Algol68
  - Amsterdam Compiler Kit (1983)
    - Modula-2, C, Fortran, Pascal, Basic, Occam
  - Pcode -> Ucode -> HPcode (1977-?)
    - FORTRAN, Ada, Pascal, COBOL, C++
  - Ten15 -> TenDRA -> ANDF (1987-1996)
    - Ada, C, C++, Fortran
  - ....

Sharing parts of compiler pipelines is common

- Compiling to textual assembly language
- Retargetable code-generation libraries
  - VPO, MLRISC
- Compiling via C
  - Cedar, Fortran, Modula 2, Ada, Scheme, Standard ML, Haskell, Prolog, Mercury,...
- x86 is a pretty convincing UNCOL
  - pure software translation (VirtualPC)
  - mixed hardware/software (Transmeta “code morphing”)
  - pure hardware (x86 -> Pentium RISC “micro-ops”)

Compiling high-level languages via C as a “portable assembler”

- Everybody does it, but it’s painful
  - Interfacing to garbage collection
  - Tail-calls
  - Exceptions
  - Concurrency
  - Control operators (call/cc & friends)
- Hard to achieve genuine portability (gcc only plus lots of ifdefs is common)
- Performance often unimpressive
  - C is difficult to optimise well, and it’s hard or impossible for the front-end to communicate invariants (e.g. alias information) to the backend
- Often can’t use what little it seems to give you (e.g. GHC doesn’t use the C stack at all)

Interlanguage Working

- Smooth interoperability between components written in different programming languages is another dream with a long history
- Distinct from, more ambitious and more interesting than, UNCOL
  - The benefits accrue to users not to compiler-writers!
- Interoperability is more important than performance, especially for niche languages
  - For years we thought nobody used functional languages because they were too slow
  - But a bigger problem was that you couldn't really write programs that did useful things (graphics, guis, databases, sound, networking, crypto,...)
  - We didn't notice, because we never tried to write programs which did useful things...

Interlanguage Working

- Bilateral or Multilateral?
- Unidirectional or bidirectional?
- How much can be mapped?
- Explicit or implicit or no marshalling?
- What happens to the languages?
  - All within the existing framework?
  - Extended?
  - Pragmas or comments or conventions?
- External tools required?
- Work required on both sides of an interface?

Calling C bilaterally

- All compilers for high-level languages have some way of calling C
  - Often just hard-wired primitives for implementing libraries
    - Extensibility by recompiling the runtime system ☹
  - Sometimes a more structured FFI
  - Typically implementation-specific
- Issues:
  - Data representation (31/32 bit ints, strings, record layout,...)
  - Calling conventions (registers, stack,..)
  - Storage management (especially copying collectors)
- It’s a dirty job, but somebody’s got to do it

Is there a better way?

- Well we saw the JVM before the break …
- But there are problems with languages which are not ”Java”-like
- What then? …

Microsoft's .NET

Support major standards initiatives such as XML, SOAP, UDDI, WDSL and … to make it ready for developers who want to take advantage of the Services on Demand vision

What is Microsoft .NET?

- .NET is Microsoft's vision of a world connected using open standards
- .NET is the name given to the latest Microsoft technology, products and development framework
- .NET is the basis for Microsoft's current and future roadmap

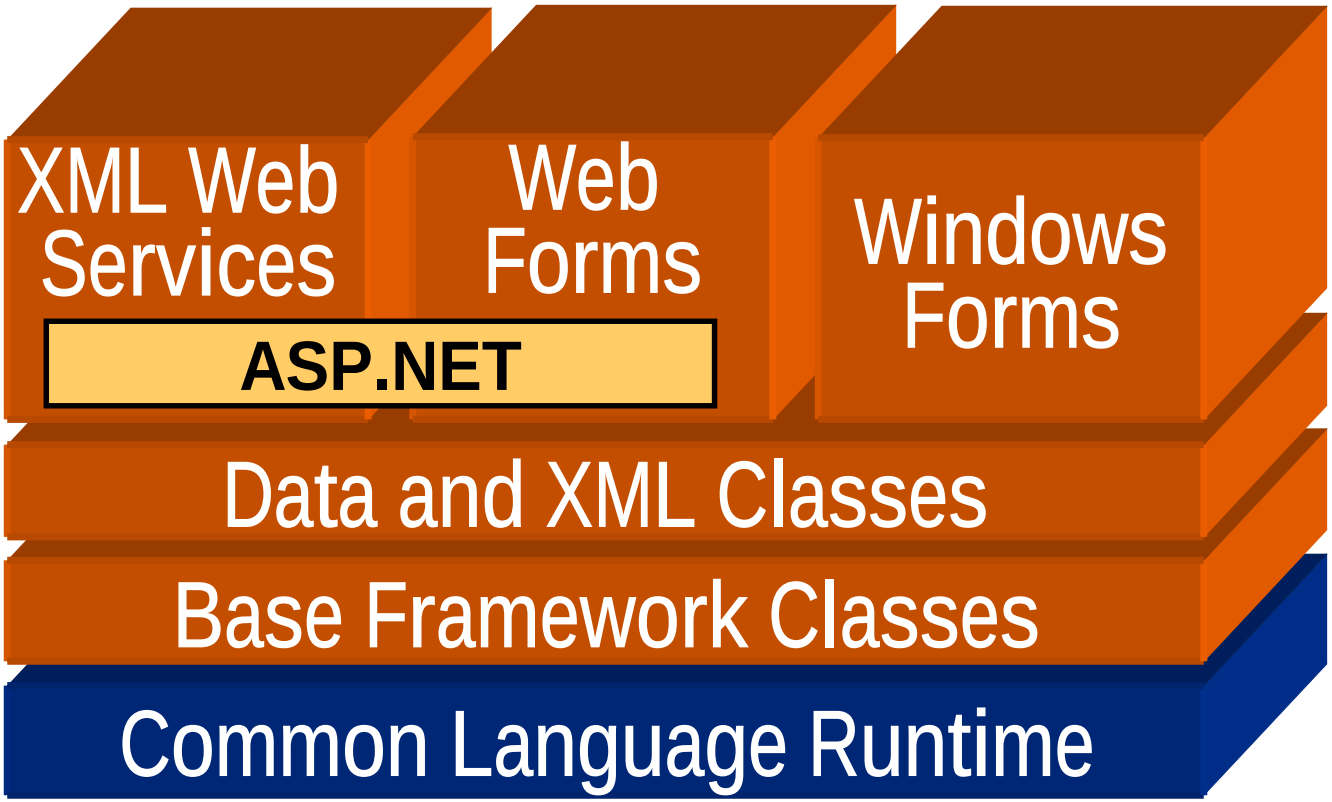
- '.NET is Microsoft's platform for a new computing model built around XML Web Services'
  Microsoft Corporation Annual Report, 2001

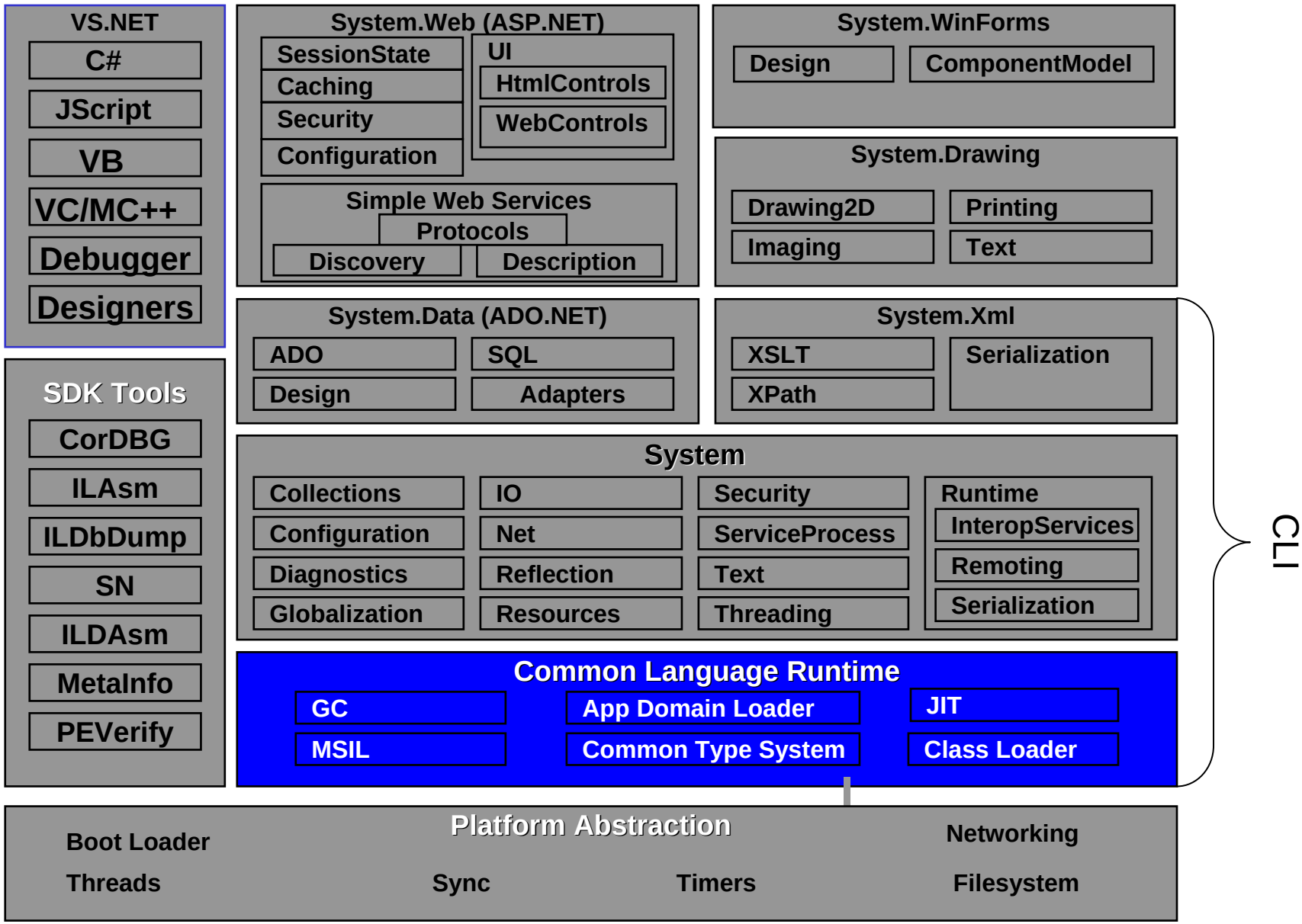
- .Net is Microsoft's core business Strategy

Programming for .Net

- Common Language Runtime + class libraries
- ADO.NET
- ASP.NET
- Web services – XML, SOAP, UDDI, WSDL …
- Programming languages
  - C++, VB, C#, (J#)
  - APL, COBOL, Eiffel, Forth, Fortran, Haskel, SML, Mercury, Mondrian, Oberon, Pascal, Perl, Python, RPG, Scheme, SmallScript, …
- Visual Studio .Net
  - Professional
  - Server Edition
  - Mobile Internet Toolkit
  - Academic

Common Programming Model - .NET




VS.NET
C#
JScript
VB
VC/MC++
Debugger
Designers
SDK Tools
CorDBG
ILAsm
ILDbDump
SN
ILDAsm
MetaInfo
PEVerify
System.Web (ASP.NET)
SessionState
Caching
Security
Configuration
UI
HtmlControls
WebControls
Simple Web Services
Protocols
Discovery
Description
System.WinForms
Design
ComponentModel
System.Drawing
Drawing2D
Printing
Imaging
Text
System.Data (ADO.NET)
ADO
SQL
Design
Adapters
System.Xml
XSLT
Serialization
XPath
System
Collections
IO
Security
Runtime
InteropServices
Configuration
Net
ServiceProcess
Remoting
Diagnostics
Reflection
Text
Serialization
Globalization
Resources
Threading
Common Language Runtime
GC
App Domain Loader
JIT
MSIL
Common Type System
Class Loader
CLI
Platform Abstraction
Boot Loader
Networking
Threads
Sync
Timers
Filesystem

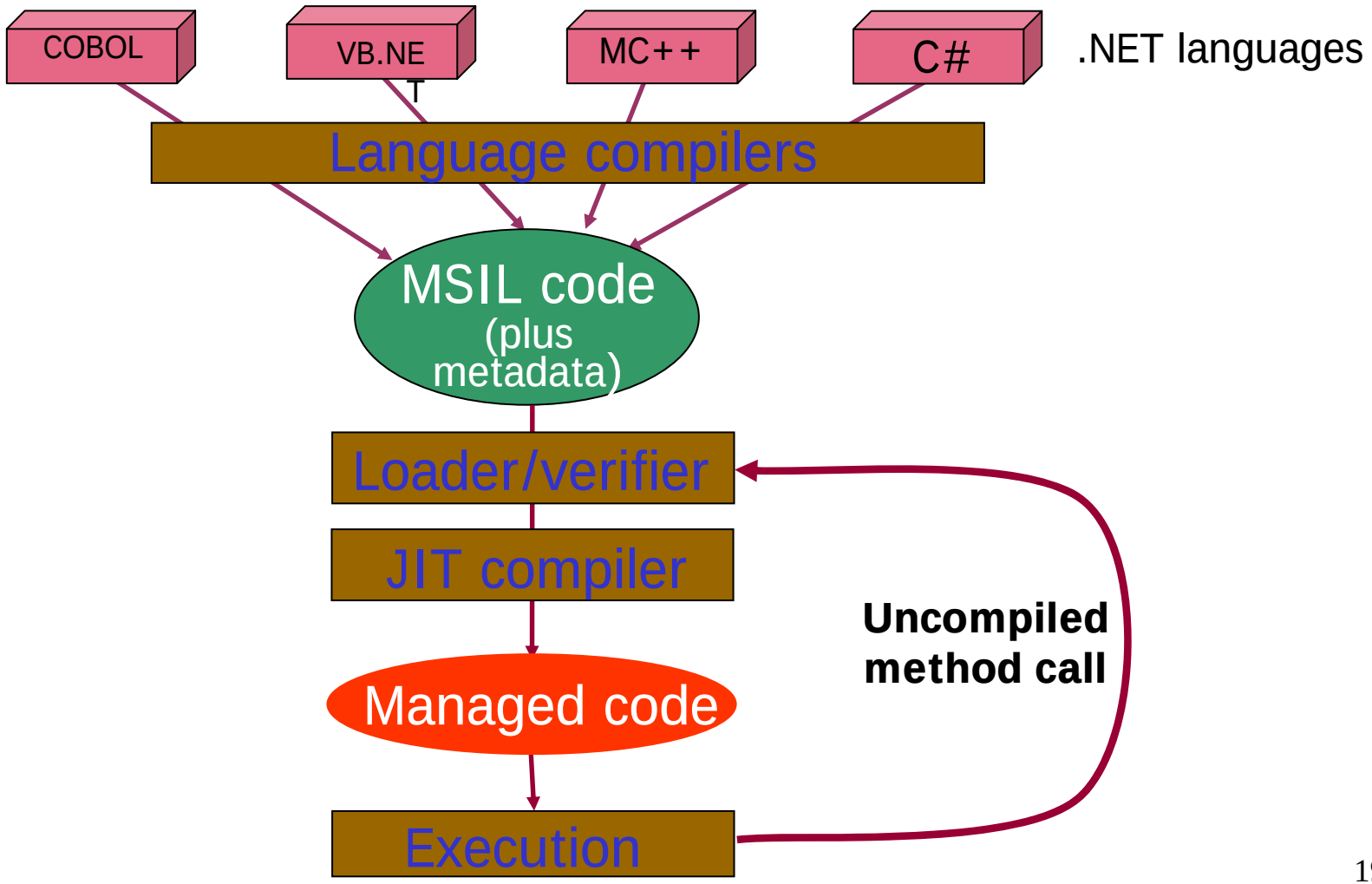
Overview of the CLI

- A common type system…
  …and a specification for language integration (**CLS**)
  - Execution engine with garbage collector and exception handling
  - Integral security system with verification
- A factored class library
  - A “modern” equivalent to the C runtime
- An intermediate language
  - CIL: Common Intermediate Language
- A file format
  - PE/COFF format, with extensions
  - An extensible metadata system
- Access to the underlying platform!

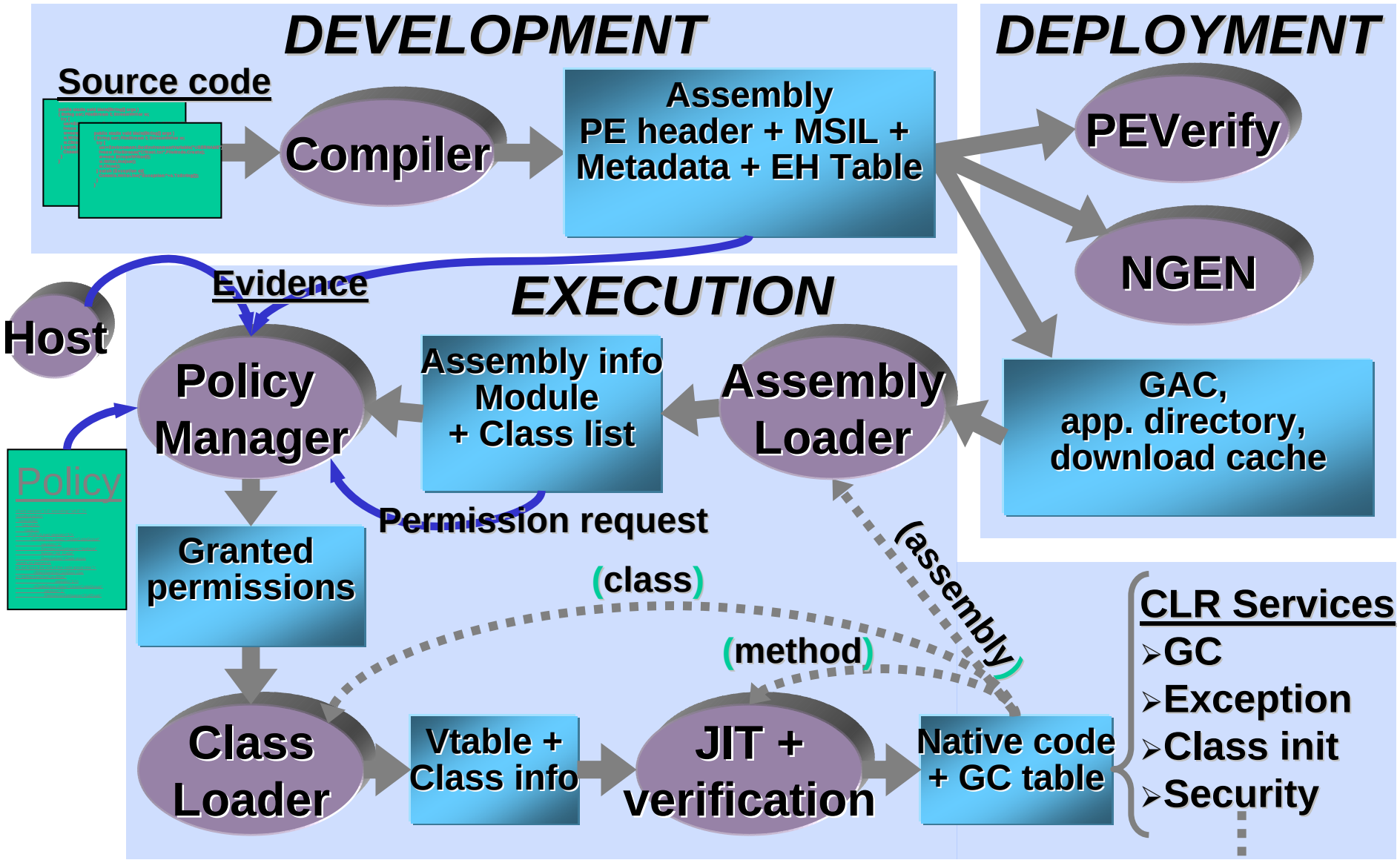
Terms to swallow

- CLI (Common Language Infrastructure)
- CLS (Common Language Specification)
- CTS (Common Type System)
- MSIL (Microsoft Intermediate Language)
- CLR (Common Language Runtime)
- GAC (Global Assembly Cache)

Execution model



Managed Code Execution



What is the Common Language Runtime (CLR)?

- The CLR is the execution engine for .NET
- Responsible for key services:
  - Just-in-time compilation
  - heap management
  - garbage collection
  - exception handling
- Rich support for component software
- Language-neutral

The CLR Virtual Machine

- Stack-based, no registers
  - All operations produce/consume stack elements
  - Locals, incoming parameters live on stack
  - Stack is of arbitrary size; stack elements are “slots”
  - May or may not use real stack once JITted
- Core components
  - Instruction pointer (IP)
  - Evaluation stack
  - Array of local variables
  - Array of arguments
  - Method handle information
  - Local memory pool
  - Return state handle
  - Security descriptor

- Execution example

```
int add(int a, int b)
{
   int c = a + b;
   return c;
}
```

Offset	Instruction	Parameters
IL_0000	ldarg	0
IL_0001	ldarg	1
IL_0002	add	
IL_0003	stloc	0
IL_0004	ldloc	0
IL_0005	ret	

CIL Basics

- Data types
  - `void`
  - `bool`
  - `char`, `string`
  - `float32`, `float64`
  - [`unsigned`] `int8`, `int16`, `int32`, `int64`
  - `native` [`unsigned`] `int`: native-sized integer value
  - `object`: System.Object reference
  - Managed pointers, unmanaged pointers, method pointers(!)
- Names
  - All names must be assembly-qualified fully-resolved names
    - [*assembly*]*namespace*`.`*class*`::`*Method*
    - `[mscorlib]System.Object::WriteLine`

CIL Instructions

- Stack manipulation
  - `dup`: Duplicate top element of stack (pop, push, push)
  - `pop`: Remove top element of stack
  - `ldloc`, `ldloc.`*n*, `ldloc.s` *n*: Push local variable
  - `ldarg`, `ldarg.`*n*, `ldarg.s` *n*: Push method arg
    - "this" pointer arg 0 for instance methods
  - `ldfld` *type class::fieldname*: Push instance field
    - requires "this" pointer on top stack slot
  - `ldsfld` *type class::fieldname*: Push static field
  - `ldstr` *string*: Push constant string
  - `ldc.`<*type*> *n*, `ldc.`<*type*>`.`*n*: Push constant numeric
    - <type> is i4, i8, r4, r8

CIL Instructions

- Branching, control flow
  - `beq`, `bge`, `bgt`, `ble`, `blt`, `bne`, `br`, `brtrue`, `brfalse`
    - Branch target is label within code
  - `jmp` <*method*>: Immediate jump to method (goto, sort of)
  - `switch (`*t1*, *t2*, … *tn*`)`: Table switch on value
  - `call` *retval Class::method(Type, …)*: Call method
    - Assumes arguments on stack match method expectations
    - Instance methods require “this” on top
    - Arguments pushed in right-to-left order
  - `calli` *callsite-description*: Call method through pointer
  - `ret`: Return from method call
    - Return value top element on stack

CIL Instructions

- Object model instructions
  - `newobj` *ctor*: Create instance using ctor method
  - `initobj` *type*: Create value type instance
  - `newarr` *type*: Create vector (zero-based, 1-dim array)
  - `ldelem`, `stelem`: Access vector elements
  - `isinst` *class*: Test cast (C# “is”)
  - `castclass` *class*: Cast to type
  - `callvirt` *signature*: Call virtual method
    - Assumes “this” in slot 0--cannot be null
    - vtable lookup on object on *signature*
  - `box`, `unbox`: Convert value type to/from object instance

CIL Instructions

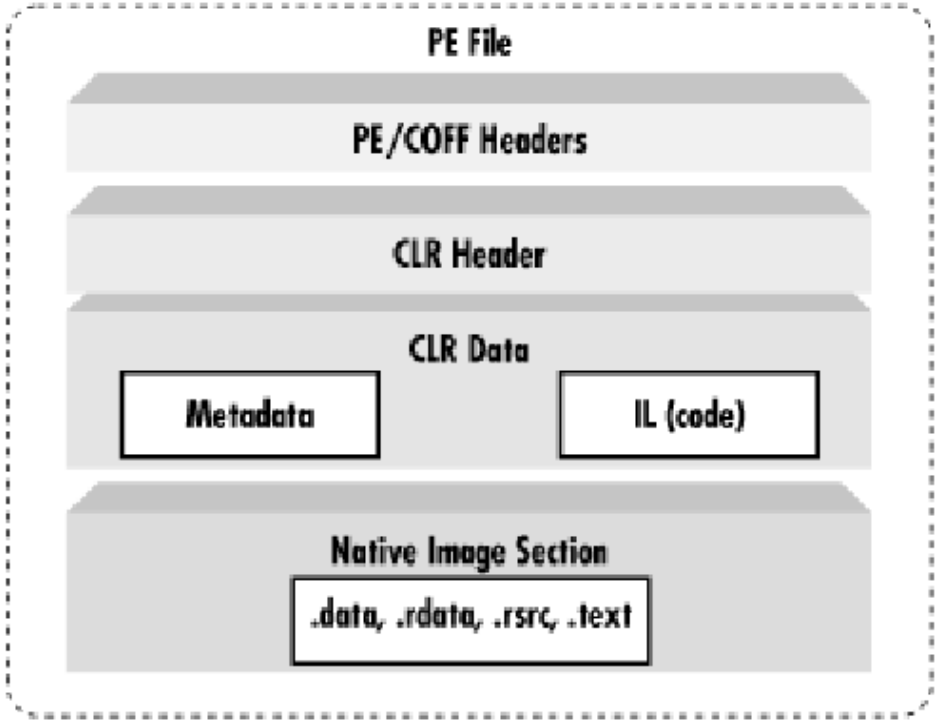
- Exception handling
  - `.try`: Defines guarded block
  - Dealing with exception
    - catch: Catch exception of specified type
    - fault: Handle exceptions but not normal exit
    - filter: Handle exception if filter succeeds
    - finally: Handle exception and normal exit
  - throw, rethrow: Put exception object into exception flow
  - leave: Exit guarded block

CIL assembler

- ILAsm (IL Assembly) closest to raw CIL
  - Assembly language
    - CIL opcodes and operands
    - Assembler directives
    - Intimately aware of the CLI (objects, interfaces, etc)
  - ilasm.exe (like JASMIN for Java/JVM)
  - Ships with FrameworkSDK, Rotor, along with a few samples
  - Creates a PE (portable executable) file (.exe or .dll)

PE File

- Windows Portable Executable (PE) Standard
- File extension: EXE, DLL



Parts of a Managed Module

- PE Header
  - Indicates type of file (DLL, GUI, etc.)
  - Info on Native CPU
- CLR Header
  - Version of Common Language Runtime (CLR)
  - Location of Metadata, Resouces, etc.
- Metadata
  - Description of Type and Members
  - Description of References
- Intermediate Language (IL) Code
  - Code to be compiled to Native CPU Instructions

ILAsm

- Some ILAsm directives
  - `.assembly extern`: referencing another assembly
  - `.assembly`: declaring local assembly, version, hash, etc
  - `.module`: optional declaration of module (file) name
  - `.namespace`: declare a lexical namespace for types
  - `.entrypoint`: marks method as entrypoint (“Main()”)
  - `.maxstack`: optimization; how many stack slots required?
  - `.locals`: declares local variables
    - May auto-initialize if `init` is present

ILAsm

- .class
  - `interface`: class is actually an interface
  - Access control: `public`, `private`
  - `explicit`, `auto`, `sequential`: Field layout (value types)
  - `implements`, `extends`: inherits interface or base class
  - `abstract`, `sealed`: as with C#
  - Nested classes must use `nested` modifier
  - String handling: `ansi`, `autochar`, `unicode`
  - `beforefieldinit`: don't type-init on static method calls

```
.class private auto ansi beforefieldinit App
       extends [mscorlib]System.Object
{
  // . . .
} // end of class App
```

ILAsm

- .field
  - Access control: `public`, `assembly`, `family`, `famandassem`, `famorassem`, `private`
  - `initonly`: constant field (“readonly” in C#)
  - `literal`: constant value; inline replacement when used
  - `static`: one instance for all type instances

```
.class private auto ansi beforefieldinit App
       extends [mscorlib]System.Object
{
  .field private string message
  .field private static object[] cachedValues
} // end of class App
```

ILAsm

- .method
  - Access control: as for .field
  - Method name:
    - `.ctor`, `.cctor`: Special names for constructors
  - `instance`, `static`, `abstract`, `final`: as with C#
  - `virtual`: late-bound, but doesn't indicate slot consumption
  - `newslot`: method takes new slot in vtable
    - C# "virtual" == CIL "virtual newslot"
    - C# "overrides" == CIL "virtual"
  - `pinvokeimpl`: P/Invoke binding to native method
  - `specialname`, `rtspecialname`: Name is important
    - `.ctor`, `.cctor`, property `get_`/`set_` methods, etc.

ILAsm

- .method (continued)
  - Method hiding
    - `hidebyname`: hides base class method(s) of name
    - `hidebysig`: hides base class method(s) of exact signature
  - Implementation attributes
    - `cil`, `native`, `runtime`: CIL, native, or runtime-provided?
    - `managed`, `unmanaged`: somewhat redundant
    - `synchronized`: Acquire lock before executing

ILAsm

- .property
  - two directives: .get and .set
  - correspond directly to methods to invoke
  - unlike C#, methods can be named anything
  - no concept of indexer
    - C# language construct
    - property named “Item” taking an int32 parameter
- .event
  - Like .property, binds methods to event targets
    - .addon
    - .removeon
    - .fire
  - Like .property, methods can be named anything

Example 1

- Hello, CIL!

```
.assembly extern mscorlib { }
.assembly Hello { }

.class private auto ansi beforefieldinit App
       extends [mscorlib]System.Object
{
  .method private hidebysig static void Main() cil managed
  {
    .entrypoint
    .maxstack  1
    ldstr      "Hello, CIL!"
    call       void [mscorlib]System.Console::WriteLine(string)
    ret
  } // end of method App::Main
} // end of class App
```

Example 1

- Compiling, Running

```
C:\Prg\Demos>ilasm Hello.il

Microsoft (R) .NET Framework IL Assembler.  Version 1.0.3705.0
Copyright (C) Microsoft Corporation 1998-2001. All rights reserved.
Assembling 'Hello.il' , no listing file, to EXE --> 'Hello.EXE'
Source file is ANSI

Assembled method App::Main
Creating PE file

Emitting members:
Global
Class 1 Methods: 1;
Writing PE file
Operation completed successfully

C:\Prg\Demos>hello
Hello, CIL!
```

Reflection Emit

Abstractions correspond closely to the CTS that underlies the CLR:

- Assemblybuilder
- ConstructorBuilder
- CustomAttributeBuilder
- EnumBuilder
- EventBuilder
- FieldBuilder
- ILGenerator
- Label
- LocalBuilder
- MethodBuilder
- ModuleBuilder
- ParameterBuilder
- PropertyBuilder
- TypeBuilder

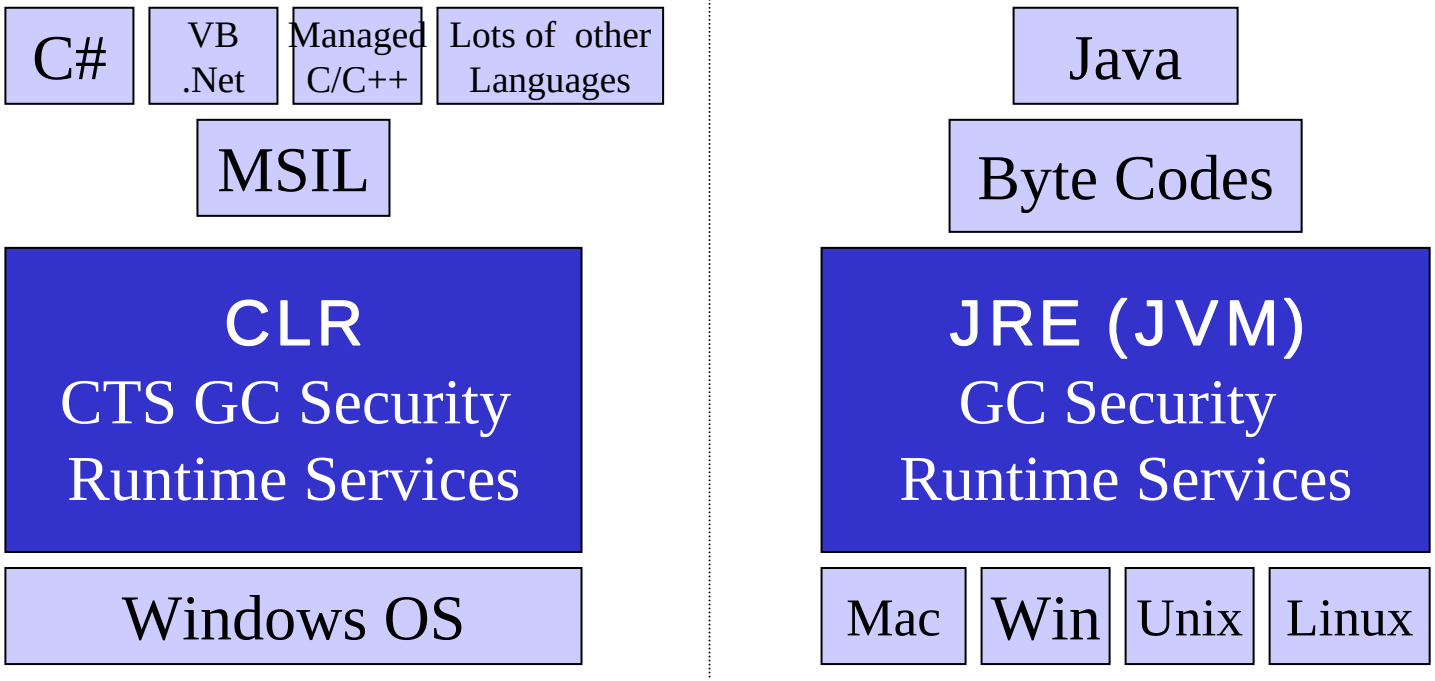
Generating IL code

```
ILGenerator generator = simpleMethod.GetILGenerator( );
// Emit the IL
// Push zero onto the stack. For each 'i‘ less than 'theValue',
// push 'i' onto the stack as a constant
// add the two values at the top of the stack.
// The sum is left on the stack.
generator.Emit(OpCodes.Ldc_I4, 0);
for (int i = 1; i <= theValue; i++) {
    generator.Emit(OpCodes.Ldc_I4, i);
    generator.Emit(OpCodes.Add);
  }
// return the value
generator.Emit(OpCodes.Ret);

computeSumInfo = typeof(IComputer).GetMethod("ComputeSum");
myType.DefineMethodOverride(simpleMethod, computeSumInfo);

// Create the type.
myType.CreateType( );
return newAssembly;
}
```

Generated IL Code

```
Ldc_I4 0
Ldc_I4 1
Add
Ldc_I4 2
Add
Ldc_I4 3
Add
Ldc_I4 4
Add
Ldc_I4 5
Add
Ret
```

CLR vs JVM



Both are ‘middle layers’ between an intermediate language & the underlying OS

JVM vs. CLR at a glance

	JVM	CLR
Managed execution environment	X	X
Garbage Collection	X	X
Metadata and Bytecode	X	X
Platform-abstraction class library	X	X
Runtime-level security	X	X
Runs across hardware platforms	X	?

Java Byte Code and MSIL

- Java byte code (or JVML) is the low-level language of the JVM.
- MSIL (or CIL or IL) is the low-level language of the .NET Common Language Runtime (CLR).
- Superficially, the two languages look very similar.

JVML:

```
iload 1
iload 2
iadd
istore 3
```

MSIL:

```
ldloc.1
ldloc.2
add
stloc.3
```

- One difference is that MSIL is designed only for JIT compilation.
- The generic add instruction would require an interpreter to track the data type of the top of stack element, which would be prohibitively expensive.

JVM vs. CLR

- JVM’s storage locations are all 32-bit therefore a e.g. a 64-bit int takes up two storage locations
- The CLR VM allows storage locations of different sizes

- In the JVM all pointers are put into one reference type
- CLR has several reference types e.g. valuetype reference and reference type

JVM vs. CLR

- CLR provides ”typeless” arithmetic instructions
- JVM has separate arithmetic instruction for each type (iadd, fadd, imul, fmul...)

- JVM requires manual overflow detection
- CLR allows user to be notified when overflows occur

- Java has a maximum of 64K branches (if...else)
- No limit of branches in CLR

JVM vs. CLR

- JVM distinguish between invoking methods and interface (invokevirtual and invokeinterface)
- CLR makes no distinction

- CLR support tail calls (iteration in Scheme)
- Must resort to tricks in order to make JVM discard stack frames

JVM vs. CLR

- JVM designed for platform independence
  - Single language: Java (?)
  - A separate JVM for each OS & device
- CLR designed for language independence
  - Multiple languages for development
    - C++, VB, C#, (J#)
    - APL, COBOL, Eiffel, Forth, Fortran, Haskel, SML, Mercury, Mondrian, Oberon, Pascal, Perl, Python, RPG, Scheme, SmallScript, …
    - Impressive usage of formal methods and programming language research during development
    - Impressive extensions for generics and support for functional languages underway
  - Underlying OS:
    - Windows 2000/XP, special version for WindowsCE
    - FreeBSD Unix, Mac OS X
    - Linux

Why is .Net so interesting from a programming language research point of view?

- CIL is more powerful than JVML and allows the compiler writer more freedom in data representation and control structures
- The runtime provides services (execution engine, garbage collection, security…) which make producing a good implementation of new languages easier
- The freedom to choose language
  - All features of .NET platform available to any .NET programming language
  - Application components can be written in multiple languages
  - Multi-language component-based programming makes it much more practical for other people to use your language in their own projects
  - The frameworks & libraries mean you can actually do useful things with your new language (graphics, networking, database access, web services,…)
- Over 26 languages supported in the CLR
  - APL, COBOL, Pascal, Eiffel, Haskell, ML, Oberon, Perl, Python, Scheme, Smalltalk or you could write your own!