PROGRAMMING CIL FOR THE .NET PLATFORM

G. C. MUGANDA
DEPARTMENT OF COMPUTER SCIENCE
NORTH CENTRAL COLLEGE

1. Introduction

CIL is the Common Intermediate Language for the .NET platform. CIL is the language for a stack oriented virtual machine much like the Java Virtual Machine. This article will introduce CIL.

CIL is actually an object-oriented assembly language: it supports classes, inheritance, polymorphism, exception handling, and provides access to the entire class library that comes with .NET. We will start by listing a number of .NET classes and methods for doing IO, and then show how those methods can be called from CIL.

2. Standard Output Methods

The .NET System namespace has a Console class that has several Write and WriteLine methods and several Read and ReadLine methods. These methods can be used to perform output and input, respectively. We mention only a few of them: for the full story, consult the MSDN documentation online

(1) Console.Write(int32)
(2) Console.Write(String)
(3) Console.Write(Object)
(4) Console.Write(String, Object)
(5) Console.Write(String, Object, Object)
(6) Console.Write(String, Object, Object, Object)

The first three methods write a 32 bit integer, a string, an object (by calling its `ToString()` method. THe last three methods take a string that specifies formatting information, as well as one or more objects as parameters. The string is used to specify how the objects should be formatted. Look online for more informatin.

All these methods have `WriteLine` versions that send a newline delimiter to standard output after writing their arguments.

3. Standard Input Method

We will use the

```
String Console.ReadLine()
```

which returns a string that corresponds to the next available line in the input. To read an integer number from the console, first read the number as a string using `ReadLine()` and then parse the number using the method

```
Int.Parse(String s)
```

This method is similar to the `Integer.parseInt(String s)` method in Java.

4. An Overview of CIL Instructions

As already mentioned, the .NET CLR (Common Language Runtime), the virtual machine that executies CIL, is a stack machine. All instructions look for and remove their arguments from the stack, perform on operation on the arguments, and push the result back onto the stack. Functions (methods) likewise, when called, take their parameters from the stack and place their result on the stack before returning. Thus a method replaces the arguments it finds at the top of the stack with its result.

Selected instructions are as follows.

4.1. **Load Instructions.**

The load family of instructions push a value onto the stack. A subfamily of the load instructions has the form

```
ldc.i4  value
```

loads a 4-byte (32 bit) compile time integer constant onto the top of the stack. For example

```
ldc.14 100
```

loads 100 onto the stack.

There is a special form used to load small integers in the range 0..8. For example,

```
ldc.i4.0
ldc.i4.1
ldc.i4.8
```

load 0, 1, and 8 onto the top of the stack respectively. In addition

```
ldc.i4.m1
```

loads minus 1 (-1).

There is a family for loading values of local variables of a method onto the stack. Each local variable is assigned a position or offset, such as 0, 1, 2, ..., and so on. Local variables are declared by a statement such as

```
.locals init (int32 a, int32 b, int32 c)
```

which allocates local variable `a, b, c`, assigning them the respective positions 0, 1, and 2. The values can then be loaded on the stack as follows:

```
ldloc.0          // load a
ldloc.2          // load c
```

Likewise there is a family for loading values of method parameters onto the top of the stack. Parameters are assigned positions according to their order in the parameter list. For example a method

```
.method int32 myMethod(int32 a, int32b)
```

has two parameters a and b at positions 0 and 1 respectively. The `ldarg` subfamily loads values of parameters onto the stack:

```
ldarg.0   // load argument a
ldarg.1   // load argument b
```

Finally, a reference to a string can be loaded onto the stack using

```
ldstr  string
```

4.2. **Store Instructions.** A `stloc` store instruction removes a value from the top of the stack and stores it in a local variable.

```
stloc.0   // store into local variable at position 0
stloc.4   // store into local variable at position 4
```

4.3. **Branching Instructions.** The branching instructions specify jumps, both conditional and unconditional. The unconditional jump is specified by

```
br  target
```

Such an instruction causes an unconditional transfer to the targetted instruction. The conditional branches are

```
beq target
bne target
bgt target
bge target
blt target
ble target
```

All six take two values off of the top of the stack, compare them, and jump if the the first value is equal, not equal, greater than, greater or equal, less than, or less or equal.

Two other conditional jumps pop a single value off of the stack and jump according to whether the value is true or false:

```
brfalse target
brtrue target
```

5. Arithmetic Instructions

The following instructions add, subtract, and multiply two values. The values are at the top of the stack are removed and replaced by the sum, difference, or product.

```
add
sub
mul
```

Finally,

```
neg
```

replaces the top of the stack with its negative.

6. Miscellaneous

Two more instructions are of interest:

```
box int32
pop
```

The first instruction removes a 32 bit integer from the stack and boxes it up (forms a wrapper object containing the integer). It then pushes the wrapper object back onto the stack. This is useful when you need to pass an integer to a method that is expecting an object.

`pop` removes the value at the top of the stack and discards it.

You can find more complete listings of CIL instructions on line. See, for example:

`http://en.csharp-online.net/CIL_Instruction_Set`

7. Hello World

Here is a simple program in CIL. It is the traditional Hello World program.

```
.assembly extern mscorlib{}

.assembly HelloWorld
{
    .ver 1:0:1:0
}

.method static void main()
{
    .entrypoint
    .maxstack 1

    ldstr "Hello world!"
    call void [mscorlib]System.Console::WriteLine(string)
    ret
}
```

This simple program illustrates a number of concepts.

The directive `.entrypoint` indicates that this is the method at which execution of the program will start. The `.maxstack` indicates that the method will use a single slot on the stack. Each CIL method must declare in this way the number of stack slots that it will use.

On the .NET platform, an assembly is an executable module or a library module The first `.assembly` directive references an external assembly, the `mscorlib` library, which contains, among other things, the methods for doing input and output. The second `.assembly` directive specifies a name for the assembly of which this module will be a part, as well as a version number for the assembly being created. Put these two directives at the top of your CIL files.

8. Running CIL Programs

Let us run the above program. Microsoft provides a commandline tool, `ilasm`, that can be used to assemble CIL programs and produce executables. The easiest way to use `ilasm` is to open up a Visual Studio Command Prompt. Follow these steps:

(1) From your Windows Programs menu find the following sequence of entries:

```
Microsoft Visual Studio 2008
Visual Studio Tools
Visual  Studio 2008 Command Prompt
```

and click on the last item listed to open up the command prompt. Now the `cd` command to navigate to the directory you will use to store your `cil` files. For example, you might use the command

```
C:\Users\gcm\Ncc\Classes\Winter10\220\ilasm>
```

to navigate to a diectory named `ilasm` that has been created for this purpose.

(2) Using a text editor, create a file with an `il` extension, say `Hello.il` containing the above program. Save the file.

(3) Assemble the program using the command

```
C:\Users\gcm\Ncc\Classes\Winter10\220\ilasm>ilasm hello.il
```

(4) Execute the program by invoking its name on the commandline:

```
C:\Users\gcm\Ncc\Classes\Winter10\220\ilasm>hello
```

9. Local variables

The following method, (found on an online site) is uses three local variables. It prompts the user for two numbers, computes the sum, and prints the result. It uses the the version of `System.Console.Write` that takes a format string and three additional objects. The format string is

```
"{0} + {1} = {2}"
```

It uses placeholders much as C printf format strings do, except the placeholders identify the arguments to be inserted by their zero-based positions.

`.locals init` declares local variables and stipulates that they be initialiazed to their default values. It establishes indentifiers that will be used to access those local variables.

`box int32` removes an integer from the stack, boxes it into a wrapper object, and pushes the reference to the wrapper object onto the stack.

```
.assembly extern mscorlib{}
.assembly LocalVars
{
    .ver 1:0:1:0
}

.method static void main()
{
    .entrypoint
    .maxstack 4
    // declare loca variables
```

```
    .locals init (int32 first, int32 second, int32 result)
    // read first number and store in first local variable
    ldstr "First number: "
    call void [mscorlib]System.Console::Write(string)
    call string [mscorlib]System.Console::ReadLine()
    call int32 [mscorlib]System.Int32::Parse(string)
    stloc first
    // read second number and store in second local variable
    ldstr "Second number: "
    call void [mscorlib]System.Console::Write(string)
    call string [mscorlib]System.Console::ReadLine()
    call int32 [mscorlib]System.Int32::Parse(string)
    stloc second
    // add the two numbers and store in third local variable
    ldloc first
    ldloc second
    add
    stloc result
    // load the format string
    ldstr "{0} + {1} = {2}"
    // load the three objects to print
    ldloc first
    box int32
    ldloc second
    box int32
    ldloc result
    box int32
    // call method to print
    call void [mscorlib]System.Console::WriteLine(string, object,
                                                   object, object)
    ret
}
```

The results of an example run are

```
C:\Users\gcm\Ncc\Classes\Winter10\220\ilasm>localvars2
First number: 2
Second number: 3
2 + 3 = 5
```

Note that the calls to the various methods include type information to resolve ambiguities resulting from method overloading.

The following is a simpler version of the above program: it reads in two numbers and prints the sum.

```
.assembly extern mscorlib{}

.assembly sumprog
{
    .ver 1:0:1:0
}

.method static void main()
```

```
{
    .entrypoint
    .maxstack 4
    .locals init (int32 first, int32 second)

    ldstr "First number: "
    call void [mscorlib]System.Console::Write(string)
    call string [mscorlib]System.Console::ReadLine()
    call int32 [mscorlib]System.Int32::Parse(string)

    ldstr "Second number: "
    call void [mscorlib]System.Console::Write(string)
    call string [mscorlib]System.Console::ReadLine()
    call int32 [mscorlib]System.Int32::Parse(string)

    add
    call void [mscorlib]System.Console::Write(int32)
    ret
}
```

10. CIL Programs With Methods

A simple method that takes two integers as parameters and returns the sum is written as follows.

```
.method static int32 sum(int32, int32)
{
   .maxstack 2
   ldarg.0
   ldarg.1
   add
   ret
}
```

It pushes its two parameters, the values to be added onto the stack and then performs and `add` before returning. A full program using `sum` follows.

```
.assembly extern mscorlib{}
.assembly HelloWorld
{
    .ver 1:0:1:0
}

.method static void main()
{
    .entrypoint
    .maxstack 4
    .locals init (int32 first, int32 second)

    ldstr "First number: "
    call void [mscorlib]System.Console::Write(string)
    call string [mscorlib]System.Console::ReadLine()
    call int32 [mscorlib]System.Int32::Parse(string)
```

```
    ldstr "Second number: "
    call void [mscorlib]System.Console::Write(string)
    call string [mscorlib]System.Console::ReadLine()
    call int32 [mscorlib]System.Int32::Parse(string)

    call int32 sum(int32, int32)
    call void [mscorlib]System.Console::Write(int32)
    ret
}

.method static int32 sum(int32, int32)
{
    .maxstack 4
    ldarg.0
    ldarg.1
    add
    ret
}
```

11. Branching Examples

The binary branching opcodes are

`beq, bne, bgt, bge, blt, ble,`

and the unary opcodes ones are

`brfalse, brtrue`

Here is an example of putting these to work writing a function to compute the maximum of two integers:

```
.assembly extern mscorlib{}
.assembly max
{
    .ver 1:0:1:0
}
.method static void main()
{
    .entrypoint
    .maxstack 4

    .locals init (int32 first, int32 second)

    ldstr "First number: "
    call void [mscorlib]System.Console::Write(string)
    call string [mscorlib]System.Console::ReadLine()
    call int32 [mscorlib]System.Int32::Parse(string)

    ldstr "Second number: "
    call void [mscorlib]System.Console::Write(string)
    call string [mscorlib]System.Console::ReadLine()
    call int32 [mscorlib]System.Int32::Parse(string)
```

```
    ldstr "The maximum is "
    call void [mscorlib]System.Console::Write(string)

    call int32 max(int32, int32)
    call void [mscorlib]System.Console::Write(int32)
    ret
}

.method static int32 max(int32 a, int32  b)
{
    .maxstack 2

     ldarg.0
     ldarg.1
     bge   firstBigger
     ldarg.1
     ret
firstBigger:
     ldarg.0
     ret
}
```

12. Recursion

Here is an example of a recursion in a CIL program. The program uses a recursive method to compute the combination of n things taken k at a time according to the algorithm illustrated in the following C code.

```
  int comb(int n, int k)
  {
     if (k == 0) return 1;
     if (n == k) return 1;
     return comb(n-1) + comb(n-1, k-1);
  }
```

The CIL implementation of this recursive algorithm can be found in the following program.

```
.assembly extern mscorlib{}

.assembly comb
{
    .ver 1:0:1:0
}

.method static void main()
{
    .entrypoint
    .maxstack 4

    .locals init (int32 first, int32 second)
```

```
    ldstr "First number: "
    call void [mscorlib]System.Console::Write(string)
    call string [mscorlib]System.Console::ReadLine()
    call int32 [mscorlib]System.Int32::Parse(string)
    stloc.0

    ldstr "Second number: "
    call void [mscorlib]System.Console::Write(string)
    call string [mscorlib]System.Console::ReadLine()
    call int32 [mscorlib]System.Int32::Parse(string)
    stloc.1

    ldstr "comb({0},{1}) is "
    ldloc.0
    box int32
    ldloc.1
    box int32
    call void [mscorlib]System.Console::Write(string,
                                      object, object)

    ldloc.0
    ldloc.1
    call int32 comb(int32, int32)
    call void [mscorlib]System.Console::Write(int32)

    ret
}

.method static int32 comb(int32 n, int32  k)
{
     .maxstack 4

     ldc.i4.1      //default return value of 1

     // if k = 0  return 1
     ldarg.1
     ldc.i4.0
     beq  comb_is_done

     // if n == k  return 1
     ldarg.0
     ldarg.1
     beq  comb_is_done

     pop            //remove default value of 1 from the stack

     //compute comb(n-1, k)
     ldarg.0
     ldc.i4.1
     sub
     ldarg.1
```

```
    call int32 comb(int32, int32)

    //compute comb(int32, int32)
    ldarg.0
    ldc.i4.1
    sub
    ldarg.1
    ldc.i4.1
    sub
    call int32 comb(int32, int32)

    // return comb(n-1, k) + comb(n-1, k-1)
    add
comb_is_done:
    ret
}
```

13. Looping

Loops can be implemented using CIL. Consider a program to output a list of the first 10 positive integer squares:

```
1
4
9
16
25
36
49
64
81
100
```

We can use a main method with two local variables k and n initialized to 1 and 10 respectively. The strategy is to step k through the range 1..10, printing k^2 at each iteration of the loop. We need to compare k to n at each iteration, and we can do that by pushing k and n onto the stack and applying the

```
bgt  endOfloop
```

instruction to detect when to terminate the loop. The `br` clears k and n from the stack. If the jump to the end does not take place we load two copies of k onto the stack and apply `mul`. The produce is left on the stack to be removed and printed. Details of implementation can be seen in the following code.

```
.assembly extern mscorlib{}

.assembly HelloWorld
{
    .ver 1:0:1:0
}


.method static void main()
```

```
{
    .entrypoint
    .maxstack 2

    .locals init (int32 k, int32 n)

    ldc.i4.1       //k = 1
    stloc.0
    ldc.i4 10      //n = 10
    stloc.1
topOfLoop:
    ldloc.0        //k
    ldloc.1        //n
    bgt  done      //if k > n we are done
    //print k*k
    ldloc.0
    ldloc.0
    mul
    call void [mscorlib]System.Console::WriteLine(int32)
    //k = k + 1
    ldloc.0
    ldc.i4.1
    add
    stloc.0
    br  topOfLoop
done:
    ret
}
```

14. Assignment 4

Write a CIL program that prompts the user for a positive integer n and prints the first n terms of the fibonacci sequence

`1  1   2  3, 5, 8, 13, \ldots,`

Your program must use a `fibonacci` method separate from main, and the method must be recursive.

15. Assignment 5

Write a CIL program that prompts the user for a positive integer n and prints the first n terms of the fibonacci sequence

`1  1   2  3, 5, 8, 13, ....`

Your program must use a `fibonacci` method separate from main, and the method must be not be recursive.

16. Optional Extra Credit Assignment

This is for 2 percent extra credit: if you get a 100 % on this you get 2 points added to your average for the entire course!

The number of ways to partition a set of n objects into k disjoint non-empty subsets is called a Stirling number of the second kind. For example, the set $\{1, 2, 3\}$ can be partitioned into 1 subset in only one way:

$$\{1, 2, 3\}$$

but can be partitioned into 2 non-empty subsets in 3 ways:

$$\begin{array}{ll} \{1\} & \{2, 3\} \\ \{2\} & \{1, 3\} \\ \{3\} & \{2, 3\} \end{array}$$

and into 3 non-empty subsets in only one way:

$$\{1\} \quad \{2\} \quad \{3\}$$

Stirling numbers are usually denoted by $s(n, k)$. It can be shown that Stirling numbers of the second kind can be defined using the function

```
int stirling(n, k)
{
   if (k = 1 || n == k)
     return 1;
   else
     return stirling(n-1, k-1) + k * stirling(n-1, k);
}
```

Write a CIL program that prompts the user for n and k, where $n \geq k \geq 1$, and prints the stirling number $s(n, k)$.