WORKSHOP 9 – CODE GENERATION PART 1

PURPOSE OF THE WORKSHOP

The purpose of this workshop is to start the process of Code Generation from your annotated AST. You will start to implement the code templates to create a viable Encoder that will translate your AST into Triangle Abstract Machine code.

YOU MUST COMPLETE LAST WEEK'S WORK BEFORE ATTEMPTING THIS WORKSHOP

PART 1: THE CODE GENERATION FRAMEWORK

The process of code generation actually follows the same basic approach as our semantic analysis last week. We will be using the visitor design pattern again, remember how we said it lets you "visit" objects without damaging the objects themselves? It is exactly this property that we want to sue again. Our encoder will be designed to visit the node of the AST and use execute the correct encoding (code generation) method for that type of node.

As shown in the lecture each of our types of phrase will have a code template that describes how it should be represented in machine code. In each encoding method we will implement the code template for that type of phrase.

For today's labs and the future coursework submission I have provided you with a partly functional Code Generator, this file package contains a skeleton structure and some helper methods. Not all of the code here will make sense until we look at the final code generation and memory management components in lab 10.

You should down load the source package zip now. The package contains a single folder CodeGenerator, you should copy this folder into your Triangle.Compiler folder.

The folder contains a number of files which form the structure for the Code Generator.

The Entities folder contains files which represent the entities of the source language. These classes handle the call, assign and fetch methods describes in the lecture, to manage the memory used by the source program. Do not worry about these files just yet, we will look at these in more detail when we cover memory addressing next week.

The file emitter.cs contains the emitter class, this class deals with actually writing out the target program. You will see from the code within this file it is a set of overloaded Emit methods. Each of these Emit methods takes a different set of instruction components and generates the required components for the main Emit method on line 80. This method creates an instruction from the components (the operator, the length of the instruction, the address, or register and the operand, or value) and adds it to the instruction set.

You will see the final method in this file, SaveObjectProgram, actually (tries) to turn this instruction set into the final obj file that we can run against the Abstract Machine.

The file Frame.cs represents a stack frame, as discussed in the lecture this is a representation of the memory used for each level of scope in the program. There are 2 main fields and 4 main methods. The frame has a size representing the current number of words (spaces in the stack) assigned to it. It has a level, representing how deep into a nested structure this frame sits. The methods are more important;

> Expand – allows the frame size to be extended to allow for new variables etc. This method receives a new size and returns a new frame which is that much better than the original.
>
> Replace – allows the frame to be replaced by an empty one of a set (provided) size at the same level
>
> Push – creates a new frame of the given size at the next nested level.

The final set of files, with the prefix Encoder, comprise the

Now to get the new files working with the setup you already have you will need to make some changes to your compiler.cs file.

First add the declaration

```
Encoder _encoder;
```

Just below where your checker is declared.

Then in your compiler's constructor instantiate the encoder with an instance of your error reporter.

```
 _encoder = new Encoder(ErrorReporter);
```

Finally, in your CompileProgram method you need to add our 3rd pass to the compiler.

```
// 3rd pass
ErrorReporter.ReportMessage("Code Generation ...");
_encoder.EncodeRun(program);
if (ErrorReporter.HasErrors)
{
 ErrorReporter.ReportMessage("Compilation was unsuccessful.");
 return false;
}
```

You can also add the following line directly afterwards but leave it commented for now

```
// finally save the object code
//_encoder.SaveObjectProgram(ObjectFileName);
```

Your code for today's lab will compile if correct but will not yet produce a viable object file.

PART 2: STARTING THE ENCODING PROCESS

As already mentioned the Encoder works in a very similar way to the Checker that we implemented in the previous lab. The point of the Encoder is to implement the Code Templates defined for the language.

In the lecture we saw the following table, which outlines the steps in the code templates

Phrase Class	Code Function	Effect of Generated Code
Program	*run* P	Run the program P and then Halt, starting and finishing with an empty stack

Command	*execute* C	Execute the command C, possibly updating variables, but not expanding or contracting the stack
Expression	*evaluate* E	Evaluate the expression E, Pushing its result onto the stack, but having no other effect
V-name	*fetch* v	Push the value of the constant or variable named V on to the stack top
V-name	*assign* V	Pop a value from the stack top and store it in the variable named V
Declaration	*elaborate* D	Elaborate the declaration D, expanding the stack to make space for any constants.

Because we are using a visitor pattern we can accomplish some of the tasks here using the visit method we already have in our Syntax Tree Structure.

Phrase Class	**Visitor Function**	**Behaviour of visitor function**
Program	*visitProgram*	Generate code specified by 'run P'
Command	*visit…Command*	Generate code specified by 'execute C'
Expression	*visit…Expression*	Generate code specified by 'evaluate E'
V-name	*visit…Vname*	Return an entity description of the given value or variable name (more on this next week)
Declaration	*visit…Declaration*	Generate the code specified by 'elaborate D'
Type-Denoter	*visit…TypeDenoter*	Return the size of the given type

Using this we can start to flesh out the visit methods needed by our Encoder. Let's start with the simplest

The code template for a Triangle Program is

run Program

HALT

So we need to create and call a EncodeRun method in our encoder that takes a representation of a program. If you have a look in Encoder – Common you will see that the very first method after the constructor is exactly this. The method receives an instance of our AST, which at this point will represent out entire program as generated by the Parser and then Annotated by the Checker.

At the minute there is no code in here so we should implement the code template. Looking at our second table above visitProgram represents 'run p', so we need to visit the program node. From last week, we can visit the program by calling the visit method of the AST at that point and passing it the current AST node. We also need to pass this program the current stack frame, as we are at the beginning of the program we create a new one.

```
ast.Visit(this, Frame.Initial);
```

The next part of the code template requires us to halt (or end) the execution. There is a helper method in the emitter that generate this for you, all we have to do to call it is

```
    _emitter.Emit(OpCode.HALT);
```

And that is it, our EncodeRun method can now generate the code required for the first stage of the source program.

Let's look at the next stage, when we call VisitProgram we will be calling the VisitProgram method code in Encoder-Program.cs, this partial class deals with encoding programs. A program consists of a Command, so all we need to do here, just like we did in the Checker, is to visit the Command node, again passing it the current stack frame.

```
return ast.Command.Visit(this, frame);
```

This will now call the VisitCommand method in the Encoder – Commands.cs file, where the specific visit method for that command will be called.

Let's look at the code template for an If command:

Execute[if E then C1 else C2] =

(1)	**Evaluate**	**E**	***evaluate the expression***
(2)	**JUMPIF (0)**	**g**	***if the answer is false, jump to address g***
(3)	**Execute**	**C1**	***otherwise execute true command (C1)***
(4)	**Jump**	**h**	***then jump to address h***
(5) g:	**Execute**	**C2**	***if we arrive at address g – execute false command (C2)***
(6) h:			***if we arrive at h: carry on.***

So how do we implement this?

There are a number of properties of our emitter that help us with this.

1) Whenever we call Emit the emitter will return an **int** which is the current address in the stack where that instruction has been inserted
2) The emitter has a method called Patch, that allows an address to be pushed into the stack.

Step 1: So first we can evaluate the expression, from table 2 above, we do this by just visiting it (and passing the frame)

```
ast.Expression.Visit(this, frame);
```

Step 2: Now from the template we need to do a jump to the false command, to do this we need the address of the false command. Remember that our emitter can provide this when we supply it the required components of the instruction.

```
var jumpFalseAddress = _emitter.Emit(OpCode.JUMPIF, Machine.Fa
lseValue, Register.CB);
```

Here we are setting up a variable to hold the falseAddress from the emitter. The instruction we are emitting contains the Operator Code JUMPIF as per our code template. We also pass in a statement Machine.FalseValue defining this as 0 this comes from the machine itself. Finally we tell the machine were to store the code in memory, in this case we put it in the CodeBlock (Register.CB)

Step 3: Next we execute the true command, again from table 2 all this means is we perform a visit

```
ast.TrueCommand.Visit(this, frame);
```

Step 4: Now we must implement the jump to the end of the code if the if expression is true. If we look at the code template (address h) there is no value in the code we are jumping to, so we can just emit the JUMP operator and the CodeBlock register

```
var jumpTrueAddr = _emitter.Emit(OpCode.JUMP, Register.CB);
```

Step 5: So from our code template we are now needing to set the addresses of the false command, we do this using the emitter's patch method, with the address of our false command.

```
_emitter.Patch(jumpFalseAddress);
```

If we arrive here we must execute the false command, again just by visiting it.

```
ast.FalseCommand.Visit(this, frame);
```

Step 6: Finally we need to patch in the address where we will go if the expression is true, again using the emitter's patch method.

```
_emitter.Patch(jumpTrueAddr);
```

And that is the IF command complete, you can leave the return null in place.

Again your code won't do anything new this week, but should build successfully if the code is correct. Let me know if you get any errors.

You can see from this example that a lot of the hard work is done for us. The OpCodes and Registers are all stored in your Triangle.AbstractMachine folder in the root of the solution, you can have a look at the files in here to see what other instructions are available, you should recognise some of these from the lecture.

PART 3: WHILE, CALL AND SEQUENTIAL COMMANDS

Now let's look at a while command code template

Execute[while E do C] =

(1)	**JUMP**	**h**	***jump to the condition initially (address h)***
(2) g:	**execute**	**C**	***execute the command at address g***

(3) h:	**Evaluate**	**E**	***evaluate the condition at address h***
(4)	**JUMPIF(1)**	**g**	***if the condition is true go back to g***

The process is very similar to the IF statement once we break it down into steps.

Step (1): get the jump address for the condition and emit the JUMP instruction

```
var jumpAddr = _emitter.Emit(OpCode.JUMP, Register.CB);
```

Next we need to store the current address so we can loop back to it later. The emitter gives us this if we ask it for NextInstrAddr, so

```
var loopAddr = _emitter.NextInstrAddr;
```

Step (2): we need to execute the command and then patch in the jumpAddress

Follow the instruction in table two and the example for the IF statement to work out what2 lines of code need to go in at this stage.

Step (3) we need to evaluate the expression

Again you should be able to follow the instruction in table to work out what code to put here.

Step (4): we need to create the jump instruction to check the value of the condition and jump the code back if it is true. So we need to give the emitter the instruction, the Machine.TrueValue (i.e 1), the register to the code memory stack (Register.CB) and then the loop address we created in step 1 to jump back to.

```
_emitter.Emit(OpCode.JUMPIF, Machine.TrueValue, Register.CB, loopAddr);
```

Again we can just leave the return null and our while command is finished.

The **Call Command** is a little different. Remember that this command is used when we are calling a built in function, like putInt() that receive parameters. At the point when we call the command we don't know how many parameters there are, so we don't know how big the memory needed is going to be. An added issue is that the parameters used will only be available within the scope of that functions, so we need to create a new stack frame.

So first we need to visit the Actual Parameters to get the size of stack frame needed for the parameters.

```
int argsSize = ast.Actuals.Visit(this, frame);
```

in triangle our functions are just identifiers followed by a set of parameters. So to build the code for one we need to visit the Identifier.

```
ast.Identifier.Visit(this, frame.Replace(argsSize));
```

The main difference in this code is that instead of simply passing the same frame, we create a new stack frame set to the size of the parameters we are using. Again we can leave the return null to complete the call command.

The **sequential command** should be easy for you to complete**,** remember what a sequential command is.

Execute[C1;C2] =

(1) Execute C1

(2) Execute C2

Go back and look at table 2 and work out what two commands should go in your code to implement this code template.

PART 4: SOME EXPRESSIONS

Again I have completed most of the expressions for you, and you should now be able to look at the code and work out what is happening at the different stages.

The first two expressions are special, they deal with our main terminals, Character and Integer. These are important as they are going to return values as opposed to another part of the AST. The value needs to be loaded onto the stack correctly to be used.

Look at the VisitCharacterExpression method, you will see that the code need some work to be fully implemented.

It's an expression so its template is just evaluate [CharacterLiteral] which isn't that helpful. But we should know how to deal with it. The first thing we need to do is visit the terminals type to get an idea of the memory size its value is going to take up.

```
var valSize = ast.Type.Visit(this, null);
```

Even though we know that all our types are 1 word long it is best practice to still ask.

Next we use the machines LOADL instruction (load literal, as shown in the lecture) to push the terminals value onto the stack.

```
_emitter.Emit(OpCode.LOADL, (short)ast.CharacterLiteral.Value)
;
```

finally we need to return the memory size needed `return valSize;` so that our program can maintain the memory required.

Have a go yourself following the same pattern to deal with the VisitIntegerLiteral method

PART 5: WHAT NEXT

Keep working on this week's and last week's material to build up the course work. We will be adding the very last section, entities, next week and by then you should all have a working compiler that is able to produce a triangle object file that you can execute against the Triangle interpreter.