WORKSHOP 10 – CODE GENERATION PART 2

PURPOSE OF THE WORKSHOP

The purpose of this workshop is to finalise the development of the code generation component of the triangle Compiler. This week we will look at the implementation of the runtime entities needed to represent the language components in the target machines instructions.

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

PART 1: THE CODE GENERATION FRAMEWORK

In this week's lecture I showed you how the assign command should be implemented based on the code template

```
execute [V:= E] = evaluate[E]
                  assign [V]
```

You will see in your Encoder-Commands.cs file that you were working with last week that this method (VisitAssignCommand)has not yet been implemented.

If we look at the code semantic rules used for our source language, we see that we should evaluate the expression and then assign a variable name / identifier to that value. So for example if we were to write x:= 42 we would evaluate the 42 to a IntegerLiteral and assign the identifier x to it.

This is exactly what our code template above suggests we should do in the code generation. Evaluate the expression E (and put its value on top of the stack) then assign the v-name V to whatever value is on top of the stack.

To do this in code we would first visit the expression `ast.Expression.Visit` however because we need to keep track of the memory being used by the declaration we need to also pass the frames being used.

step1) So `var valSize = ast.Expression.Visit(this, frame);` would allow us to visit that expression, including the current frame and returning the size of the value (valsize) from the expression itself.

Step2) Now from our set of rules last week an assign code generation method should use the EncodeAssign pattern.

```
EncodeAssign(ast.Vname, frame.Expand(valSize), valSize);
```

This method, which sits in the runtime entity classes (implemented in the next section) takes the variable name, the size of the stack frame needed and the size of the value to be stored.

So our first line (step 1), places the value of the expression on top of the stack, the second uses the vname to remove that value from the stack and assign it to a memory address linked to the vname provided. Exactly what our semantic rules and code template state we should do.

PART 2: RUNTIME ENTITIES

The classes representing the runtime entities are held in your CodeGenerator/ Entities folder. You will see that there are separate classes for a number of entities.

KnownAddress – entities that have a known memory address at runtime

KnownValue – entities that have a value associated with them at runtime

UnknownAddress - Entities with an address not known at runtime

UnknownValue – Entities with a value unknown at runtime.

All of these classes will use one or more of the following TAM instructions in their encode methods

RUNTIME INSTRUCTION CODES

LOAD	Load an object from the provided address and push it on the top of the stack
LOADA	Push the provided address onto the stack
LOADL	Push the literal value provided onto the stack
LOADI	Pop a data address from the stack, fetch an object from that address and push it onto the stack
STORE	Pop an object from the stack and store it the address provided
STOREI	Pop an address from the stack then pop an object from the stack and store it at that address

KNOWNADDRESS

Open the KnownAddress.cs file, you should see that it contains the 3 methods discussed in the lecture, EncodeAssign, EncodeFetch and EncodeFetchAddress.

Encode assign is used to assign a value to a variable by 'popping a value from the top of a stack and storing it in a variable'

The code it's self is quite simple and many of the parameters we need are already passed in from the other parts of the compiler, mainly the Annotated AST.

```
emitter.Emit(OpCode.STORE, size, frame.DisplayRegister(Address),Address.Displacement);
```

We use the emitter, just as we did last week, to emit the instruction needed.

- in this case STORE
- next we tell the machine how big the variable is SIZE
- then we need to send in the current stack frame to store the variable, this is calculated for us from the address
- finally, we send in a displacement to represent where in the frame the variable will sit

And that's it, one line of code allows us to generate the code to store a variable and its value.

You will see that the method for EncodeFetch and EncodeFetchAddress are incomplete. These follow a very similar pattern to the EncodeAssign but will use different instructions to performs the necessary tasks.

EncodeFetch should load the value from an address onto the stack.

EncodeFetchAddress should load an address onto the stack.

Given the runtime instructions above you should now attempt to implement the code in these methods following the same pattern as the EncodeAssign above.

KNOWNVALUE

Open the KnownValue.cs file, you will see that the EncodeFetch implementation is incomplete. Again the implementation here uses the instruction to load a literal value onto the stack. In this case all we need to do is provide the value and the correct Opcode. From the table above you should be able to input the correct instruction to load the literal value.

```
emitter.Emit(xxxxxx, 0, 0, _value);
```

UNKNOWNVALUE

Open the UnknownValue.cs file, you will see that, again, the EncodeFetch implementation is incomplete. This method is required for values that is not be known at runtime, so we can only work with the addresses. In this case we want to load the value stored at a particular address to the top of the stack.

```
emitter.Emit(OpCode.LOAD, size, frame.DisplayRegister(_address
),_address.Displacement);
```

The information about the address comes ultimately from the annotated AST and the other parts of the encoder that have executed up to this point. Hence why we pass the current stack frame through each visit in the encoder pass.

UNKNOWNADDRESS

The unknownAddress.cs is a slightly more complex entity. It primarily deals with when an argument is bound to a variable (var) argument.

Starting with the EncodeAssign methods highlighted in the lecture.

First we use the LOAD instruction to push the object at the address created from the annotated AST at that point in the execution to the top of the stack

```
emitter.Emit(OpCode.LOAD, Machine.AddressSize, frame.DisplayRe
gister(_address),_address.Displacement);
```

we can then use the STOREI instruction to pop that address and retrieve the object and store it.

```
emitter.Emit(OpCode.STOREI, size, 0, 0);
```

The EncodeFetch method is designed to produce the opposite effect. It should push an object to the top of the stack, then use that address to fetch an object and finally push that object onto the stack. This could be similar to the getint() function where a unknown value is generated at runtime from the user and could be passed to a constant with an unknown address at runtime. The code is provided below but you should use the table above to determine the correct instruction (opcode) to send to the emitter.

```
emitter.Emit(xxxxxxxx, Machine.AddressSize, frame.DisplayRegis
ter(_address),

emitter.Emit(xxxxxxxx, size);
```

Finally the EncodeFetchAddress method should only simply LOAD an address onto the top of the stack

```
emitter.Emit(OpCode.LOAD, Machine.AddressSize, frame.DisplayRe
gister(_address),_address.Displacement);
```

You should also have two procedure entities files KnownProcedure and UnknownProcedure, as I have not assked you to develop the generation for procedures I have provided completed versions of these files on moodle.

PART 3: TESTING THE COMPILER

In the original code package, I gave you there is a project called Triangle.AbstractMachine.Interpreter. This project allows you to run and receive compiled sourcefiles against an implementation of the Triangle Abstract Machine. If the code is correct it will run and produce output.

The code below is a small triangle program that allows the user to enter a value between 0 and 100. This value is then used as a starting point for a loop that repeatedly removes 5 from the current total outputting the result. It’s a pretty useless program, but demonstrates scope, userinput, output and a number of constants and variables.

```
let
  const MAX ~ 100;
  var y: Integer;
  var x: Integer
in
begin
  y := 5;
  put('?');
  getint(var x);
  if (x>0) /\ (x<=MAX) then
    while x > 0 do
            begin
                    x := x-y;
                    putint(x);
                    put(',')
            end
    else
end
```

Create a file for this code called sub.tri ,save it in your compiler directory, then compile it using **dotnet run sub.tri** from the command prompt.

If successful this should create a file called obj.tam, this is the compiled object code for the Triangle Abstract Machine. Copy this file to your Triangle.AbstractMachine.Interpreter folder.

Navigate to this folder from the command prompt and execute it against the Interpreter using **dotnet run obj.tam** if your compiler has worked correctly you should get a ? printed to the screen, you should be able to enter a number and see an output.

Run you other test files against your compiler, particularly files that output values, e.g. hi.tri, print.tri and printer.tri and then execute these against your interpreter to see the result.

CREATING YOUR OWN FILES

Finally, you have been using Triangle for around 8 weeks now and should be aware of its structure, syntax and semantics. Based on this you should be able to write your own programs in Triangle (both correct and incorrect) that you can use to test your compiler and then run against the interpreter.