# CM4106 Languages & Compilers

Workshop 3 – The Scanner

## Purpose of the Workshop

The purpose of this week is to give you your first experience of the Target language and framework you will be using for the module. You will also explore the first stage of compilation, the scanner.

## Part 1: Building the Framework

I have created a zip file on Moodle which contains a number of code files that you will need for this lab and your coursework.

Download this file now and unzip it to the location where you want to work, (remember on the UNI machines this should be your H Drive). Do not to unzip the file into the root of a drive, remember to make a directory first.

Once you have extracted the file you need to restore the project. Firstopen the command prompt**, make absolutely sure you are in the directory you have just extracted the zip file to.** (use **cd** to change directory and **dir** to list what is in a directory. If you need to change drives just type the name of the drive at the prompt eg **H: )**

If you are using one of the university computer, you should now be in something likeH:/Triangle.NetCore.Student

Now we need to restore the project using the **dotnet restore** this ensures that the framework has been restored to your system with all its dependencies.

Finally, as we are here we should do a quick test build, **dotnet build,** this will attempt to build the project and will tell you if any dependencies are missing (if anything goes wrong at this point, let me know).

Ok, you can now open the main solution TriangleToolsDOtNetCoreStudent.sln file in Visual Studio. *(Note: that if you are using your own machine you may get an error here, if this happens simply create a new solution in your version of visual studio and import the separate projects in each of the directories)*

## Part 2: THe Skeleton Framework

The zipfile contains a Visual Studio solution (.sln file) which you can open directly in visual studio. You will see that there are 4 separate projects within this solution

Projects within **TriangleToolsDotNetCoreStudent** solution –

**Triangle. AbstractMachine** – this project is a library that contains some definitions that are used by the Abstract Machine. You can have a look at the source files in here but you shouldn’t need to change anything.

**Triangle.AbstractMachine.Dissassembler** – this project acts as a disassembler for TAM targeted code. It is designed to take previously compiled code written in a language suitable for the Abstract Machine and turn it back in to the source language. This will be useful for testing later when your compiler produces code that does not execute as expected.

**Triangle.AbstractMachine.Interpreter**- this project represents the working Triangle Abstract Machine (TAM) and is your target platform for this module. The TAM can take compiled “.tam” files and execute them in its own special way. The “.tam” files are what your own compiler will eventually produce from “.tri” source files. These Because this is your target obviously you shouldn’t change anything in here, that would be cheating, but have a look and you should start to build up an understanding of how the abstract machine works internally.

**Triangle.Compiler** – This project is the starting point for your own compiler. It already includes a skeleton scanner class along with the Sourcecode, Token and TokenKind classes as we discussed in the lecture.

## Part 3: Using the Triangle Abstract Machine

The first, practical part of this lab is getting used to using the Triangle Abstract Machine.

In Visual Studio click on the **Triangle.AstractMachine.Interpreter** project. You will see a single file called interpreter.cs, this contains all the workings of the abstract machine.

As above the Interpreter acts as your Target Machine, it will execute compiled files with the extension .tam and produce an output as if it was a real platform you were targeting. Think of this like the Java Virtual Machine, or the C# CLR, it’s an application that sits on top of the Operating System and can execute code.

Because the Interpreter is a command line application it is best to run it from the command line.

Open the Command Prompt (CMD or Terminal) and navigate to the **Triangle.AstractMachine.Interpreter**

The Interpreter will, by default, run a file called obj.tam in the same directory. So if you type **dotnet run** you should see the output from this default file!

\*\*\*\*\*\*\*\*\*\* TAM Interpreter (.NET CORE) \*\*\*\*\*\*\*\*\*\*

Hi!

Program has halted normally.

As you can see the default program is pretty simple.

I’ve written a small program here in Triangle, it simple asks the user for a number and then prints out a countdown. As you can see the syntax is pretty simple.

let

const MAX ~ 10;

var n: Integer

in

begin

put('n');put('u');put('m');put('b');put('e');put('r');put('?');

getint(var n); !ask user for input

if (n>0) /\ (n<=MAX) then

while n > 0 do begin

putint(n); puteol();

n := n-1

end

else

end

As you haven’t built your compiler yet I have compiled this file for you, it’s called test.tam in your interpreter folder. If at the command prompt you run **dotnet run test.tam** you should see that it asks you foar a number and then prints the countdown.

## Part 2: THe Scanner

In the Compiler project you will first of all see the Compiler class. This is the main entry point to the application, it creates instances of the source file and the scanner. At the minute it is set up run the scanner and print each token to the console. As we do not have a parser at the minute (next week’s lecture and lab) this is enough for now.

Again the compiler is a command line application so it is best to run it from the command line. If you navigate to the **Triangle.Compiler** folder and type **dotnet run test.tri** you will see that the Compiler attempts to compiler our test application.

Kind=Error, spelling="l"

Kind=Error, spelling="e"

Kind=Error, spelling="t"

You will see that this does not really do what we want and the compiler gets stuck after the first few characters. You may have to press CTRL-C to exit your application.

This is happening as I have only given you part of a scanner, at the minute it returns every Token as an error and does not deal with white space, hence it gets stuck when it hits the first whitespace.

Have a look at the scanner.cs file in SyntacticAnalysis folder. You should recognise much of this from the lecture. You will see that the ScanSeparator and ScanToken methods have very little code in them. This is why the scanner cannot deal with the source file fully.

First you should populate the ScanSeparator method to deal with whitespace and new lines. When you have completed and you run the compiler you should see that it is able to get through the test.tri file without getting stuck.

Next you should populate the ScanToken method to deal with the characters in the source. Again look at the lecture to see how this was done for digits, you should be able to replicate this to deal with the other tokens of Triangle, i.e Operators, Identifiers and the reserved words.

case '0':

case '1':

case '2':

case '3':

case '4':

case '5':

case '6':

case '7':

case '8':

case '9':

TakeIt();

while (IsDigit(\_source.Current))

{

TakeIt();

}

return TokenKind.IntLiteral;

Once completed your Compiler should be able to use the scanner to run though the test file and output the tokens and the token types.

## Part 3: Location

For the final part of the scanner, you will want to identify the position or location of the tokens so that this can be included in any error message.

You will see in the scanner class’s GetEnumerator() function that I have commented out three lines of code surrounding the ScanToken call.

//var startLocation = \_source.Location;

var kind = ScanToken();

//var endLocation = \_source.Location;

//var position = new SourcePosition(startLocation, endLocation);

\_source.Location should ask the SourceFile class for the current characters location in the source file. The SourcePosition class would hold a representation of the tokens current position. Unfortunately, this functionality does not yet exist.

You should add a Location class that represents a Location in the source file e.g (line number and line index) .

You should then add a SourcePosition class that represents the position of a token , eg the two locations representing the start and finish of the token.

Finally you should amend the token class so that the SourcePosition is added to the token when it is created, and printed out when the token’s toString() method is called.

Once you have completed these steps you should have a working scanner that can run through the test files and output the tokens, their types and their position.

## Part 4: Writing in Triangle

If you have finished creating your scanner, have a go a writing your own mini-triangle programs based on the language definition in the lecture. Save them as .tri files and have a go at scanning them.