**COMP 496**

**Project Proposal**

**Jason Bell – 3078931**

**Nov 2, 2015**

**Pie 0.2: Dynamic Typing and Unit Testing**

**Introduction**

My COMP 495 project consisted of creation of a custom .NET language, called Pie, along with it's compiler. The primary goal of the language was a more concise syntax than that of other high level languages such as C# and Java. The name Pie was chosen both as whimsy and as a nod to Python as an inspiration: the language created for that course had a hybrid of C# and Python syntax.

A secondary goal was to employ and assess test driven development throughout the creation of the compiler. This secondary goal was extremely successful: not only did test driven development result in robust code, it also had the unexpected benefit of greatly boosting productivity. I had assumed that writing unit tests would increase development time, but experienced a decrease in development time due to the reduction in debugging. A common rule of thumb is that development is 20% writing code and 80% debugging. By employing test driven development, I found those numbers became reversed: instead of spending hours hunting for a bug, the unit tests point the way.

**The Proposal**

My informal proposal for COMP 496 described two goals:

1) Write a new version of the compiler in Pie, as it is a note-worthy milestone when a programming language is mature enough to “compile itself”. This would also reveal bugs not covered by the existing unit tests. When experimenting with writing the compiler in Pie, use cases were encountered that hadn't been considered when writing the unit tests. This demonstrates that while unit testing is excellent, it isn't necessarily a substitute for testing through use of the system. Unit tests only cover the use cases that the programmer thinks of, and few programmers will think of every use case ahead of time.

2) Add unit testing as a feature integrated into the language. Unit testing suites typically have the unit tests separate from the code being tested: for example, Visual Studio unit testing requires storing the tests in a separate project. I found having to jump between the test and the code being tested to be somewhat cumbersome, and came upon the idea of tests being a language feature, and existing alongside the code in the same file. This is a violation of the principle of separation of concerns, but I am unconcerned by this for two reasons. First, in .NET languages it is simple to flag code as “debug only”: meaning that those sections of code are ignored by the compiler when building a release version. Thus, the unit testing constructs will only exist in debug builds. Secondly, I'm experimenting with new ideas and so don't need to be bound by traditional concepts, especially if there's potentially an advantage to be gained.

These two goals still stand in my formal proposal, with the addition of another:

3) Convert Pie to a dynamic typed language. Until recently I did not have much experience with dynamic typing, but gained an appreciation for it's flexibility and conciseness while experimenting with Python. This flexibility and conciseness comes with the price of reduced performance, but this is an unavoidable trade-off inherent in dynamic typed languages [1]. Dynamic typing furthers the goal of a concise language, and I am quite excited by the syntax as it is developing.

I initially had the overly ambitious goal of making Pie a language that supports both dynamic and static typing, just as it supports both object oriented and functional programming. I spent several weeks vacillating about what such a language would look like, and realized that incorporating both forms of typing is a poor idea for two reasons. First, it violates the primary goal of the language: conciseness. Any syntax that supported both was messy, with a very complicated grammar. Second, why support static typing at all, when the dominant two .NET languages, C# and VB.NET, are both static typed? Just as Python allows one to offload code that demands performance to C, when using Pie one could offload such code to another .NET language or C.

**Tools and Methods**

The tools required for this project will be largely the same as those for the COMP 495 project:

1) Visual Studio 2015 rather than 2013, to take advantage of new language features found in C# 6.0 such as static using statements [2].

2) Irony.NET [3], a .NET shift-reduce parser, to build the grammar required to form a parse tree representation of Pie source code.

Unfortunately I will not be re-using code from COMP 495: the conversion to dynamic typing is a drastic change, and regardless I wish to rewrite the grammatical rules with the benefit of experience gained. This is not as large a concern as it sounds, as I'll have the benefit of experience and dynamic typed Pie will have a much easier to define grammar.

Two versions of the compiler will be developed in parallel: one in C# and one in Pie. The C# version of the compiler will be necessary to compile the Pie code until the Pie version is mature enough to take over. As before, this isn't as bad as it sounds: prototyping has shown that the C# code doesn't need to be comprehensive: it only needs to be “good enough” to compile Pie code that replaces it. Additionally, I won't be writing unit tests in both C# and Pie: all unit tests will be in Pie using it's built in unit test constructs, which has also been proven with prototyping.

As in COMP 495, this new version of Pie will compile using C# as a bridging language. Using bridging languages in early compilers has precedence: for example Cfront, the first C++ compiler, generated C code. While generating .NET byte code directly is fairly straight-forward for someone who has experience with assembly languages, using C# as a bridging language allows one to take advantage of the many compiler errors generated by the C# compiler. Giving the Pie compiler the ability to catch all of these errors would be a monumental task far outside the scope of this project: this will be a personal project after the completion of this course, and will likely take a year or more. This method also takes advantage of code optimizations performed by the C# compiler.

Unlike in COMP 495, I will not be using .NET's CodeDOM library to generate the C# code: this library was found to be surprisingly incomplete and buggy. For example, events can not be declared as static, and unary operators are unsupported. For COMP 496 I will instead be generating the C# code manually. Counter-intuitively, prototyping has shown this to be easier than working with the CodeDOM library, with the added bonus of having direct control over how the code is generated.

The new dynamic typed syntax and unit testing construct can be demonstrated with a simple example:

type Cow:

new(name):

this.Name = name

test Name:

c = Cow(“Alfalfa”)

assert c.Name == “Alfalfa”

Types are defined with a simple “type” keyword, rather than “class” or “struct”. A Pie type is equivalent to a C# class, and Pie will not support structs. Structs are allocated on the stack and are not tracked by the garbage collector, and so provide significant performance advantages over classes [5]. However, in the context of dynamic typing, these performance advantages will likely be cancelled out by the reduction in performance caused by late binding. Even if that was not the case, making Pie types dynamic requires that they inherit the System.Dynamic.DynamicObject type [6], while structs are only able to inherit interfaces. Therefore, Pie will not allow declaration of structs. In the event that one is required, the user would be encouraged to implement it in C# or VB.NET.

Variables and method parameters do not have a type defined: in fact, in the generated C# code they are all of type “dynamic” [7].

Thanks to dynamic typing, members can be added to a type at run-time. Note that the constructor adds a “Name” field despite there not being one declared in the type. One will still be able to define such a field for clarity or as a contract guaranteeing it's existence:

type Cow:

Name

new(name):

Name = name

Object instantiation is as in Python: there are no “var” or “new” keywords. This adds complexity to the compiler, as it must determine two things. First, is there already something (such as a local variable, class field, or static import) named c? If so, it must be treated as a new variable declaration. Secondly, is the method invocation on the right of the assignment a call to a method, or is Cow a type? The compiler must be able to keep track of all types that are visible within that scope to determine this. For COMP 496 this validation will be provided by a system that has already been prototyped: each code expression has a “Scope” object that contains all the information that is visible from within it's scope. Scopes are inherited by and propagate to child expressions. When a variable is referenced, the scope is first checked to see if something of that name already exists. If it does not, it is added to the scope, and propagated down to each child scope. The result is that expressions outside that scope are not aware of the new variable, while expressions that are inside that scope are aware of it. While these steps add complexity, they will provide a very important basis for future code validation and error detection systems.

The unit test will be a construct unique to Pie. When compiled, it becomes a static method with an added prefix that allows it to be recognized as a unit test by the Pie compiler. The addition of this prefix has the added benefit of allowing a test to have the same name as a method or field. This avoids the awkward scenario of having a method called “Foo” and having to declare it's test as “test TestFoo” to avoid name collision. When requested, the compiler will use reflection to find all unit tests in the compiled assembly and invoke each in turn. When the conditional being tested by an assertion resolves to false, an exception is thrown and the unit test is reported to have failed. These tests will use the conditional attribute to ensure that they are only compiled in debug builds [8].

Finally, test coverage analysis will be an exciting feature that determines how many of the total number of code paths or blocks are covered by one's tests. I had initially thought that this would be tremendously difficult, but came upon a simple solution. Before coverage analysis is done, the compiler will do a simple depth-first search through the parse tree and at each fork in the code add a listener that reports whether it was invoked during the tests. The compiler will then list how many of the total number of listeners were invoked during the tests, which would be the degree of coverage. It will then list the line numbers of listeners that were not invoked by tests, allowing the user to inspect that code and create tests that reach it. Consider the following Pie code, and hypothetical C# code generated with inserted listeners.

type Foo:

act DoStuff(what):

if what:

Console.WriteLine(“true”)

else:

Console.WriteLine(“false”)

class Foo

{

void DoStuff(dynamic what)

{

listener1.Invoke();

if(what)

{

listener2.Invoke();

Console.WriteLine(“true”);

}

else

{

listener3.Invoke();

Console.WriteLine(“false”);

}

}

}

While generating the C# code, the compiler has inserted listeners at forks in the code. When tests are fired, the listeners will report whether they were invoked or not. A single test that passes “true” to this method will invoke listeners 1 and 2, and result in 2/3 or 66% coverage. Another single test that passes “false” will invoke listeners 1 and 3, with the same amount of coverage (though of different code). Both tests being executed will invoke all three listeners, resulting in 100% coverage. This is untested with prototype code, but seems like a very simple and effective solution.

**Schedule and Milestones**

While the COMP 495 project was completed in five weeks, this will not be the case in COMP 496. During COMP 495 I was only registered in the one course, while this time I am registered in four. Therefore, I expect to take all or most of the remaining course time, which ends in February 2016. Creating the compiler in C# will be a very rapid process due to prior experience, but the other tasks will be considerably more time consuming.

Basic supporting compiler written in C#:

November 30, 2015

Pie compiler written in Pie with use of the language's unit testing:

January 15, 2016

Completion of and validation of test code coverage analysis (ie, validating that the coverage results are accurate):

January 30, 2016

Documents prepared and all deliverables polished and ready for submission:

February 29, 2016

**Deliverables**

Visual Studio Pie syntax highlighter extension.

Self compiling Pie compiler written in Pie.

200-300 unit tests, as part of the above.

Language specification document.

“How to” document on use of the compiler and running unit tests with coverage analysis.

**Citations**

[1] Vanderplas, Jake. (2014). *Why Python Is Slow: Looking Under The Hood* [Online] Available: <https://jakevdp.github.io/blog/2014/05/09/why-python-is-slow/>

[2] Michaelis, Mark. (2015). *A C# 6.0 Language Preview* [Online] Available: <https://msdn.microsoft.com/en-us/magazine/dn683793.aspx>

[3] rivantsov, (2015). *Irony - .NET Language Implementation Kit* [Online] Available: <https://irony.codeplex.com/>

[4] Wikipedia, (2015). *Cfront* [Online] Available: <https://en.wikipedia.org/wiki/Cfront>

[5] Microsoft MSDN, (2009). *Choosing Between Class and Struct* [Online] Available: <https://msdn.microsoft.com/en-us/library/ms229017(v=vs.110).aspx>

[6] Microsoft MSDN, (2009). *DynamicObject class* [Online] Available: https://msdn.microsoft.com/en-us/library/system.dynamic.dynamicobject(v=vs.110).aspx

[7] Microsoft MSDN, (2009). *Using Type Dynamic* [Online] Available: https://msdn.microsoft.com/en-us/library/dd264736.aspx

[8] Microsoft MSDN, (2009). *The Conditional Attribute* [Online] Available: https://msdn.microsoft.com/en-us/library/aa664622(v=vs.71).aspx