**COMP 496**

Project Report

Pie 0.2: Dynamic Typing, Integrated Unit Tests, and Test Coverage Analysis

Jason Bell 3078931

February 21, 2016

**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 was 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 experience of creating a concise language with test driven development motivated me to build upon that accomplishment. In researching many languages, I developed a great appreciation for the simplicity of Python, mostly due to its dynamic typing and white-spaced scoping. Pie is already white-spaced. Why not make it dynamic typed as well? However, there are pros and cons to consider.

Dynamic typing has a number of advantages:

1. It allows an extremely concise syntax as types do no need to be explicitly declared.
2. Types can be modified at runtime. The ability to add fields at any time is extremely flexible.
3. No need for generics, as all types are already essentially generic.
4. No need for interfaces, as class contracts go against the point of using a dynamic typed language in the first place. Two objects, even if they do not share an interface, can be swapped with each other as long as they both implement the required fields and methods.

Dynamic typing has the following disadvantages:

1. Dynamic typing requires late binding: the location of members of a class are looked up at runtime rather than compile time. This look up every time a member is accessed results in a significant performance penalty [1].
2. Late binding also results in the compiler not detecting errors until the program is running. While a compiler for a static typed language may detect a call to an invalid member of a class at compile time, the dynamic typed language will not trigger an error until that member is accessed at runtime [1].

It is clear that it is important to carefully consider when to use a static or dynamic typed language. Particularly, dynamic typing may not be the best choice when performance is a consideration.

**Goal**

This second iteration of Pie added the following goals:

1. Dynamic typing, in furtherance of the goal of a concise syntax.
2. Unit testing support built into the language. Microsoft Visual Studio’s unit testing library requires that the developer keep unit tests in a separate project from the code being tested [2]. During the development of the first iteration, this was found to be cumbersome once the project had grown past a trivial size. While having unit tests embedded alongside the code being tested is a violation of separation of concerns, it may be that the benefits outweigh this.
3. Unit test coverage analysis. The compiler is able to determine what percentage of the source code is covered by unit tests.
4. Implementation of the compiler in the Pie language, since it is a notable milestone when a compiler is able to “compile itself”.

**Design and Methods**

The overall process of compilation in Pie 0.2 is the same as in Pie 0.1:

1. The pie source code is parsed into a parse tree representing the grammar of the code.
2. The parse tree is converted into a tree of language expressions.
3. The language expression tree is converted to C# code.
4. The C# code is compiled by the C# compiler.

Using C# as a bridging language has an important benefit: the Pie compiler takes advantage of C# compiler errors. Prototyping early in COMP 495 revealed that creating a full suite of compiler errors for Pie would be a monumental task worthy of being one of these projects by itself. In these early versions of the language it also allows the developer to focus on developing a robust grammar, rather than trying to do too much at once. Instead of worrying about catching all compiler errors, they are allowed to fall through to be caught by the C# compiler. Once work on Pie compiler errors begins, the C# compiler errors can be used for unit testing: a unit test will determine whether the Pie compiler catches an error successfully before it reaches the C# compiler. Using a bridging language for early versions of a compiler has precedent: the first C++ compiler, CFront, compiled to C [3].

In designing this iteration, I took a different approach from that of the previous. In Pie 0.1, the compiler was divided into loosely coupled parser, validator, and generator modules so that in future iterations they could be swapped out with new versions. One consequence of this was that the parsing, validation, and generation of language expressions was distributed across three modules. In other words, if an issue was found with a language expression, one had to investigate multiple files.

In the spirit of unit tests being embedded in source files, in Pie 0.2 most or all of the code required for a language expression is found in one file per expression. Each expression manages its own creation and lifetime. While this results in a less modular design, I wished to see if this would make testing easier. The typical lifetime of a language expression is as follows:

1. While the compiler is initializing, each expression defines its own grammatical rules so that the compiler knows how to recognize them. Many of the simpler or more fundamental rules are defined by the compiler itself, but more complex expressions have a DefineRules() method where they define their own rules.
2. When the compiler is walking through the parse tree generated by Irony.NET and recognizes an expression rule, the current parse node is passed to the expressions Build() method. The expression inspects the node and constructs an instance of itself and attaches it to a growing expression tree.
3. Once the parse tree has been fully processed, it is discarded. Now, it is the expression tree that the compiler walks through. As each expression is visited, it is told to refine itself via its Refine() method where it performs any actions necessary before generating C# code. For example, an assignment operation must determine whether the variable being assigned to already exists or not. This is also the step where compiler error detection will be performed by future versions.
4. Once all expressions in the tree have been refined, it is again walked through and each has it’s Emit() method invoked. This method appends the C# representation of that expression to the growing set of C# code.
5. Once all expressions have emitted their C# representations, the C# code is passed to the C# compiler for compilation.

This design is not modular in the same sense as the previous version, where the modules were larger components with smaller numbers of responsibilities. Instead, each expression is essentially responsible for itself, and is expected to follow a clearly defined set of steps over its lifespan. Each expression is its own module: this was found to make adding new expressions much faster and simpler than it was in the previous version, where adding a new expression meant major additions across multiple files.

**Implementation**

This version of Pie was implemented with Visual Studio 2015 Enterprise [4]. As with the previous version, it uses the Irony.NET [5] library as the language parser. Unlike the previous version, it does not use the CodeDOM [6] library to generate C# code, as it was found to be surprisingly buggy and incomplete. Instead, Pie 0.2 generates C# manually in code: early prototyping found this to be faster and simpler overall as one has full control over the generated code.

Creating a self-compiling compiler in Pie was by far the largest task. This required creating a “scaffolding” of C# code to get the basic compiler established. As each section of C# was implemented it was translated to Pie until the Pie version of the compiler was mature enough to stand on it’s own; the point where the Pie compiler was able to compile new language expressions into itself. After that point it was a matter of rebuilding the compiler repeatedly to incorporate new code until it reached its current state.

Integrated unit testing was implemented through the addition of new test and assert constructs. The tests are implemented as static methods with a signature that allows the compiler to recognize it as such. The assert keyword is simply a wrapper around an if conditional that throws an exception if the condition being tested fails. To execute tests, the compiler uses reflection to locate all unit tests in the assembly and invokes each in turn. If an exception is thrown in the test, whether by an error or a failed assertion, it is caught by the compiler and logged as a failed test.

Code coverage analysis consists of determining what percentage of the code in an assembly is covered by its unit tests. This analysis has two requirements: the presence of unit tests and that the assembly has been built with the infrastructure required for the analysis. This infrastructure consists of sensors inserted at every code fork. When unit tests are run, the code for these sensors are activated if the test executes them. The activated sensors log their activation in a file, since runtime state is not kept between unit tests. Once all unit tests have executed, measuring coverage is a simple matter of determining what percentage of all sensors were activated.

Pie declares variables and instantiates types as in Python, without the var and new keywords found in C#:

X = SomeClass()

This required that the compiler be able to determine two things:

1. Is X a variable that already exists and is visible from the current scope?
2. Is SomeClass() a type to instantiate, or a method invocation?

The solution to this problem was the implementation of a “scope stack” system. Each language expression that generates a scope, such as type bodies, method bodies, if conditionals, or for loops, owns a scope. Any variables or types declared inside that body are stored in it’s scope. During the refine process, each expression pushes its scope onto the stack while it’s refining, and pops it from the stack when it’s done. The result is that an expression is aware of everything that its parent expressions are aware of, as their scopes also exist on the stack. In the above code example, if that line of code is inside a method which is inside a class with an X field, the assignment expression will realize that X should not be declared as a new variable: it assigns to the X field of the class. If none of the parent scopes contain an X variable, the assignment expression knows to declare it as a new variable.

**Challenges**

The challenges in this project were many and diverse, to say the least; significantly more so than the first iteration created for COMP 495. While that first iteration was far from trivial, most of the challenge consisted of learning to use the tools and how to unit test. The challenges for this second iteration were as follows:

1. Lack of IDE tools for developing a complex application with the Pie language, such as runtime debugging, real-time display of errors, and code completion and refactoring. The majority of the programming was essentially done with no IDE assistance. While this was good experience, the workload was increased significantly.
2. Working with a dynamic typed language means late-binding, which means that errors are often not discovered until the application is running: the compiler simply does not know about all members of an object, nor is it supposed to. For example, I spent several hours chasing after one bug, only to have it turn out that I had used “=” instead of “==” as the equality conditional in an if statement. Because Pie is not aware of the types of objects, it does not (nor should it) do any sort of checks ahead of time in these situations.
3. Dynamic typed languages are slower than static typed languages, also due to late-binding. This is an unavoidable trade-off one has to consider when using dynamic typed languages. While the performance of the compiler is acceptable, it is still noticeably slower than the previous version.
4. It can be difficult to create grammatical rules for complex structures in a white-spaced language. For example, I had wanted to include anonymous methods or multi-line lambda statements. However, it seems to be impossible to nest these inside another statement, as may be the case if you’re passing a lambda statement to a method. This is why Python does not allow you to do this: anonymous methods must be declared by themselves. Python lambda expressions can be nested, but can not be multi-line.
5. A major challenge was an explosion of code coverage sensors. The first version of the compiler with these sensors built in compiled and tested extremely slowly and generated many megabytes of sensor activation records. I wondered at first if the compiler was slow due to it being the first version fully implemented in Pie rather than from the C# scaffolding. Once I saw the number of sensor records being produced I realized that I was unit testing the compiler with a version of the compiler that also had the full set of sensors. In other words, not only were sensor records for the tested compiler being generated, but for every unit test run, the testing compiler was also generating it’s own sensor records. Therefore, there was now an exponential increase in the number of sensor records being generated. If my description is not that clear, I hope that helps to convey how hard the problem was to diagnose. The solution to the problem is to run the unit tests on a test compiler with another compiler that does not have its own sensors.

The obvious conclusion to be reached from these challenges is that I was overly ambitious. I am very pleased at what was accomplished in this iteration, but it would have been more sensible to wait a few more iterations before writing a complex project in the Pie language: particularly waiting until there is good IDE support for the language.

**Results**

**see deliverables document for details**

1. Self compiling and testable Pie compiler written in the Pie language.
2. Pie language specification document.
3. Some simple example pieces of code.
4. A simple syntax highlighter for Visual Studio 2015.

**Self Assessment**

While I was too ambitious in this project, my goals were accomplished and the experience gained is invaluable. I consider the quality of the project to be excellent, for the following reasons:

1. If you’ll forgive me making a personal comment: the first time that the compiler “compiled itself” was exquisite. I had an “IT LIVES!!” moment, like I had just created Frankenstein’s monster. Getting the language to the point where it was able to compile itself from a very complex set of code was extremely gratifying.
2. The grammatical rules of the language are much better defined than they were in the previous version, owing both to the experience gained previously and to the simpler syntax of Pie 0.2.
3. The code is, in my opinion, beautiful. Particularly when viewed in Visual Studio with the syntax highlighter extension installed.
4. Unit testing works well, as does the code coverage analysis.
5. Having each language expression manage its own creation and lifetime in its own source file, along with its own unit tests, worked very well. Adding new expressions and unit tests was much easier than it was in Pie 0.1.

I do of course have criticisms:

1. As said, I was too ambitious, though this is hindsight.
2. Code coverage analysis could be more sophisticated. At the moment you are only presented with the percentage coverage. It would be more useful if it also listed forks in the code that were not covered by the tests. With this information one could determine the set of states required to reach that fork, and add an appropriate test to cover it.
3. I am not satisfied with the level of test coverage obtained. It was higher in the previous version, although that version used Microsoft’s sophisticated unit testing tools. It is able to tell the user which sections of code are not covered, which made it easier to get higher coverage. This first version of Pie’s coverage analysis does not have this ability. At the same time, covering critical sections of code with tests is more important than covering non-critical portions. All of the Expression classes are covered by tests, and those tests were invaluable when detected and locating bugs. Clearly there is a trade-off in having to decide what degree of test coverage is “good enough”. Given that each Expression class is in some way covered by a test, I am confident that the most critical pieces of code are covered.
4. The number of unit tests included in the project falls far short of the number that I claimed it would have in the project proposal. The reason for this is the same as the previous criticism: more sophisticated code coverage feedback is necessary. I found that the project was at the point where any new tests were added blindly, with no real idea of what, if anything, the new test covers. A later version of the coverage analyzer that pinpoints sections of uncovered code will make it much easier to implement new tests for uncovered code.

**Future Plans**

I plan to continue working on the language as a personal project, albeit at a more sedate pace. I have two goals for the next version:

1. Determine whether to continue with white-spacing. Using curly braces may make the language more palatable for some users, and will make it easier to define more complex constructs such as nested lambda statements.
2. Implement a complete compiler error checker, so that the Pie compiler does not need to fall back on the C# compiler for error messages: this will be a monumental task. The error checker will be implemented as a rule-based system: each compiler error will have a set of rules attached to it. As the compiler walks through the parse tree, it will check each rule against the current parse node. If that node and its neighbouring nodes match a rule, that compiler error is presented to the user. For example, namespaces can not be nested inside a class. When inspecting a namespace parse node, there will be a rule that is activated if it has a class node as a child, and the error is triggered.

Later versions will add complete IDE support for the language. Unfortunately, Visual Studio’s extension API is surprisingly esoteric and undocumented.

Once there is good IDE support, and the compiler no longer needs to rely upon C# to catch errors, then it will be time to rewrite the compiler to generate .NET byte code rather than C#. This will significantly speed up the compiler, as it will have moved from using C# as a bridging language between Pie and byte code, to directly generating byte code.

I have from the start pondered alternative environments on which to implement Pie. .NET is and was the logical choice given my extensive experience with it, it’s excellent standard library, and support for reflection and runtime compilation. However, my experience is that Java has superior platform support compared to .NET. It’s runtime compilation and reflection abilities are also comparable to those of .NET, so porting to the Java runtime is a viable option if it’s found to be worthwhile.

There may be lack of public interest in the language. In fact, this is a near certainty given the huge number of languages that come and go. If there is no public interest in the language, I may consider approaching the language’s future from the perspective of entertainment rather than practicality. I have experience with writing Assembly, so creating a native version of Pie is also an option. Obviously this would be a huge task, which is why I would only consider it if Pie became a fun hobby rather than a practical project of public interest. I find the idea of a dynamic typed language that executes natively to be very intriguing, assuming that it’s even possible.

**Citations**

[1] Wikipedia, (2015). *Late Binding* [Online] Available: <https://en.wikipedia.org/wiki/Late_binding#Criticism>

[2] Microsoft MSDN, (2015). *Create a unit test project* [Online] Available: <https://msdn.microsoft.com/en-us/library/hh598957.aspx>

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

[4] Microsoft, (2015). *Overview of Visual Studio 2015 products* [Online] Available: <https://www.visualstudio.com/en-us/products/vs-2015-product-editions.aspx>

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

[6] Microsoft MSDN, (2015). *Using the CodeDOM* [Online] Available: <https://msdn.microsoft.com/en-us/library/y2k85ax6(v=vs.110).aspx>