COMP 495
Project Report
Jason Bell 3078931
July 7, 2015

Introduction

After some timing spent writing code both as a research assistant and a student, one comes to realize how much typing is unnecessary when programming. A list of examples from the C# language include:

1) Inability to list multiple namespaces after a single using statement.

2) Open and closing braces.

3) Semi-colon delimiters.

4) Explicit variable declarations and casting.

5) Inability to import class names as if they are namespaces: allowing one to type “WriteLine” instead of “Console.WriteLine”. C# 6.0 will add this feature to the language [1].

It should be mentioned that these features of the syntax can also assist readability. There is clearly a trade-off between readability and conciseness, and where an individual's preference lies is a matter of personal taste.

An example of a programming language with a more concise syntax is Python. However, upon researching Python's object orientated paradigm one comes to realize that Python, at it's core, is a beautiful and concise language, but one that had new features added with little planning. Some examples include:

1) Python constructors are declared as __init__. There are many other tokens similar to this one, such as __len__ and __str__. It seems strange that a language that is so clean and concise would use such messy tokens.

2) There is no access control in classes: everything is public. The programmer can superficially make a member private or protected by prefixing the member name with underscores, but the member is still visible and can be accessed if the programmer is aware of how the interpreter mangles the member name [2].

3) There are two ways to define properties: defining getter and setter functions, then bringing them together with a third member, or attaching decorators to a method declaration. Thus, properties are not a fully or cleanly integrated feature of the language, at least not when compared to C#.

4) Global variables can not be accessed inside a scope without declaring it in the scope with the global

keyword. It can be argued that global variables are dangerous, but this is a choice that programmers should be free to make, and requiring programmers to explicitly acknowledge that they are using a global variable seems like an unnecessary obstruction.

In addition to these criticisms of the syntax of the language, performance is also a major concern: because it is interpreted, Python code typically runs at single digit percentages of the speed of native or just-in-time compiled code [3].

A worthy attempt was made to overcome these concerns with the “Boo” programming language. Boo was a .NET language with Python syntax, but with the performance of just-in-time compiled code. Unfortunately, the main website for the language closed shortly before the time of this writing, but the language's GIT repository is still extant [4]. Regardless, the project is long abandoned.

Goal

The goal of this project was to implement a compiler for a custom .NET language, called Pie, with white-spaced scoping and concise syntax. The name of the language is both whimsy in the tradition of names such as Python (named after Monty Python's Flying Circus) and Hadoop (named after a child's toy elephant), and as a nod to Python as a white-spaced language.

Pie is statically and strongly typed, as in C#. It employs white-spaced scoping, and has a syntax that could be described as a hybrid of Python and C-like languages. It supports both the object oriented and functional programming paradigms. Every language construct was designed with conciseness as the first priority. A common criticism raised when colleagues have been shown the language is that it could be more readable: this is an entirely valid criticism. However, this language is deliberately designed with readability being secondary to conciseness: a niche that this project aims to fill where the dominant two .NET languages, C# and VB.NET, are more verbose.

A secondary goal of this project was to experience test-driven development in action. The fact that a compiler has obvious and easily interpreted inputs and outputs made it an ideal choice for attempting test-driven development for the first time.

Design and Methods

The Pie compiler was designed with three components: parser, validator, and generator. This is simpler than is usual for compilers, which may also have preprocessor and optimizer components: these are features to add in a future iteration. Optimization of byte code is a particularly advanced subject that is well outside of the scope of this project. As will be seen shortly, it is also unnecessary at this stage.

The first component, the parser, receives source code as it's input and produces a tree of expressions that represent that code. This tree is made up of the child/parent relationships of expressions: classes are children of namespaces, methods are children of classes, and so forth.

This parser is implemented using Irony: a .NET language implementation kit [5]. This library allows one to define grammatical rules for a language, and uses those rules to parse source code into an expression tree. Defining and testing this grammar is a painstaking process and consumed the majority

of the time required to implement this first iteration. Rigorous testing was required for each new rule to ensure that new rules did not conflict with, and hence break, previous rules.

For this iteration the validator is essentially a stub that passes the parse tree from the parser to the generator. It is unnecessary in the current iteration because of the way that the generator is implemented, but will be expanded in future iterations. The validator will be responsible for walking through the expression tree and validating each expression: it will detect the majority of compiler errors, such as invalid names, missing types, and mismatched method signatures. This is an extremely large task that will be an incremental process: perhaps if the project continues over a few years the validator may become as mature as the C# compiler in regards to detecting errors.

The third and final component is the generator: it emits byte code for each expression in the expression tree. This is accomplished by using the CodeDOM tools provided by the .NET standard library: CodeDOM provides the means of generating C# or VB.NET code from a tree of expressions. Thus this current iteration works by translating Pie to C# and compiling with the C# compiler. This provides a few advantages:

1) Any compiler errors not detected by the validator will be detected by the C# compiler. This will be an important way of measuring the effectiveness of the validator: over time the validator will catch increasing numbers of errors and express them as Pie compiler errors, and fewer will be expressed as C# errors.

2) The developer can focus on developing and testing grammatical rules, already a challenging task, and not implementing arcane and difficult to debug byte code as well. Instead, this is a task that can be handled later once the grammar and validator are considered to be of sufficient completeness. The validator will need to be of particularly high quality before implementing a byte code emitter as the msIL compiler does not provide the safety net that the C# compiler does: the parse tree must be completely validated before byte code is generated for it.

3) This method takes advantage of byte code optimizations already performed by the C# compiler, likely to result in faster code than byte code written by a programmer who is not an expert in optimization.

4) This CodeDOM generator can be used to test future versions of the generator that skip this translation step and emit byte code directly. Byte code produced by CodeDOM and a manual byte code generator can be compared in automated tests for correctness and performance. If there is a discrepancy between the two, the byte code of each can be disassembled and compared.

Compilers translating to another language has precedent: the first C++ compiler, “Cfront”, developed by Bjarne Stroustrup translated C++ code to C [6]. A more modern example is Cython, a version of Python that supports static typing and that also translates to C [7].

.

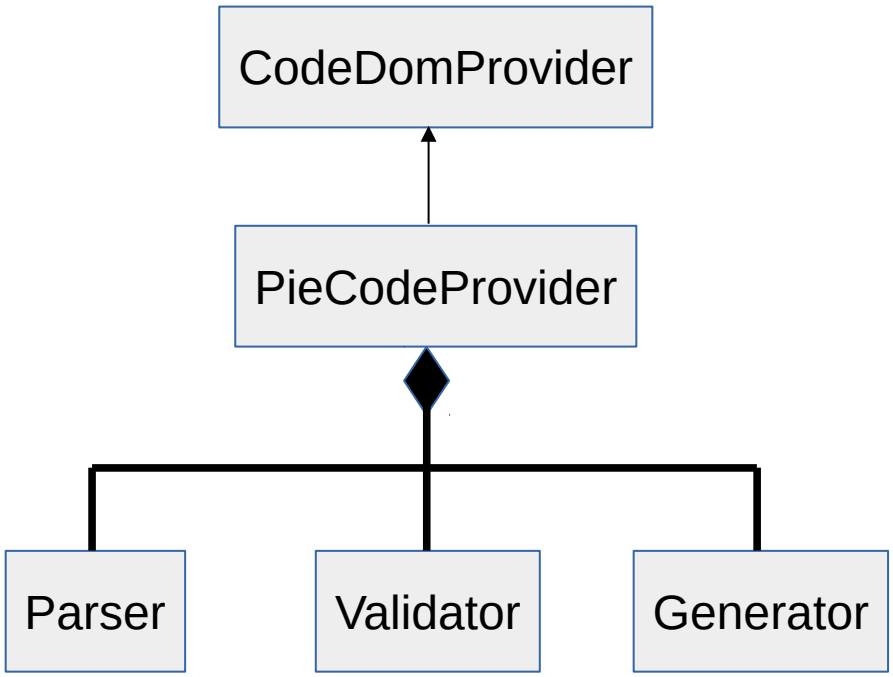

Fig. 1 High level class model of the Pie compiler.

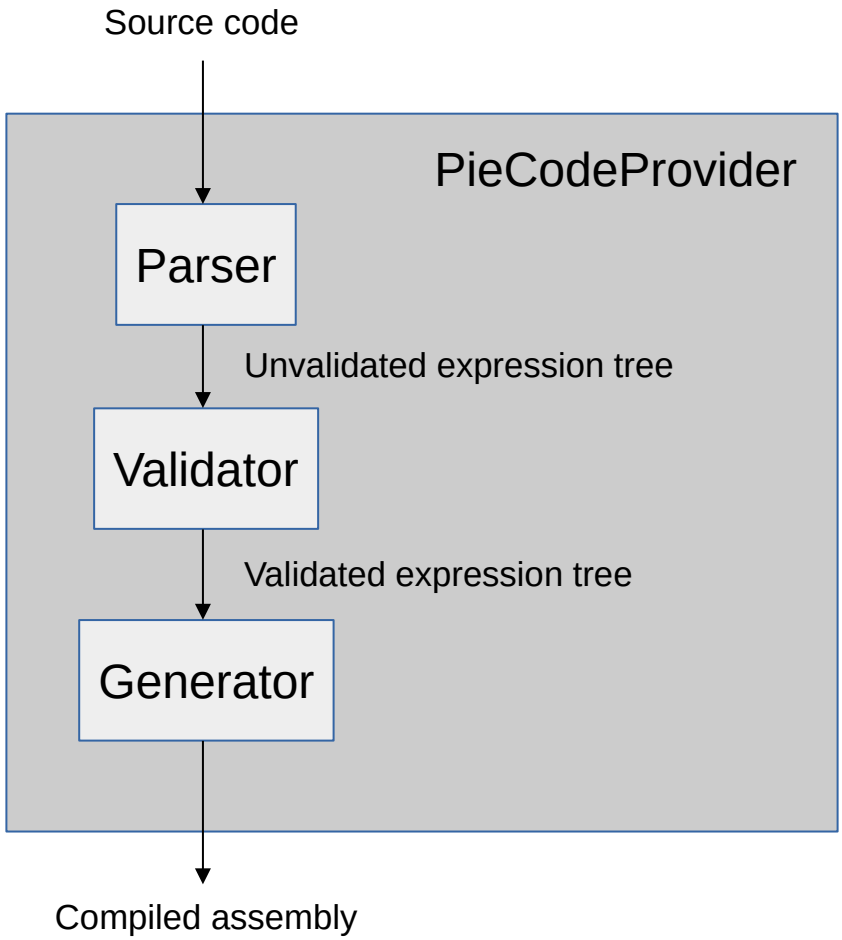

Fig. 2 High level data flow model of the Pie compiler.

Testing and Design Rationale

A secondary goal of this project was to experience and assess test driven development. A compiler is ideal for demonstrating test driven development because of the clear inputs and outputs of each component. For each component unit tests can be written that feed valid or invalid data into the component, and test that the component either produces correct outputs or handles the invalid data gracefully.

This project did not adhere to the strictest definition of test driven development, which requires that unit tests be implemented before the code is written. It was found that while this may be possible to do if one has an expert knowledge of the domain, it is extremely difficult to do if one is engaged in a learning process. That is, the experience was that writing the tests first was simply extra work, as they needed to be rewritten as one's knowledge evolved. Instead, a “working tests” project was maintained in which new tests were prototyped while learning about a concept. Once there was sufficient confidence that the concept was understood and the test would not need to be overhauled, it was graduated into the test suite.

The testing methodology used for the project was “sandwich” testing, by which the system is unit tested starting with the input and output components and testing inwards, until the tests meet and the boundary between them is tested. The parser was tested with different valid or invalid source codes, and the parse tree produced by it was inspected to determine correctness. The generator was tested with parse trees, and the byte code produced was inspected for correctness by using reflection.

This testing methodology is sandwich tested because the validator is untested at this point: it simply passes the parse tree from the parser to the generator. It is “sandwiched” between two tested components and so forms the previously mentioned test boundary. Ideally, when the validator is developed and tested it's neighbouring components will be sufficiently robust that it can be assumed that most newly detected errors will be caused by the fledgling validator.

Implementation

The compiler was implemented in the C# language using Visual Studio 2013 Ultimate. This IDE provides powerful source completion and refactoring tools. More importantly it provides built in unit test visualizations and sophisticated test coverage analysis.

All language constructs are derived from a base Expression class that owns a reference to it's parent expression, a list of child expressions, and the source code token from which the expression was generated. The source code token is needed because the location of the error in the source code is needed when displaying a useful compiler error to the user. These expressions form the parse tree used by the validator and generator: it is simply a tree of parent and child expressions. For example, an implemention of a Expression representing a class will have as it's children Expressions representing events, methods, variables, or properties.

The parser, validator, and generator are implemented in separate namespaces and are fully uncoupled. This will make it possible to replace a component with a new version without breaking the others. For example, if it is decided to replace the Irony parser with another, the validator and generator should still function as long as the expression tree produced by the parser is still in a format that they understand.

Excitingly, the existing parser unit tests can be re-used to determine that this is the case.

The parser is implemented using the Irony library. The first step of this implementation is to define the rules of the grammar by deriving from Irony's Grammar class. Grammatical rules are formed by assembling simpler rules into more complex one in an additive, optional, or multiplicative way. The simplest rules are “terminal” rules: those that do not themselves lead to further rules. The simplest example of a terminal rule is a reserved word:

classTerminal.Rule = ToTerm(“class”);

With this rule defined, any time the parser encounters the token “class”, it will match to this rule. Other examples of terminal rules are identifiers and grammatical symbols. These terminal rules can be aggregated into a more complex non-terminal rule describing a class declaration:

classDec.Rule = classTerminal + identifierTerminal + colonTerminal;

This rule will match any series of tokens starting with the reserved word “class”, followed by an identifier, followed by a colon, such as “class foo:”.

With the grammar defined, the parser uses Irony to create a parse tree. This tree is very complex and is simplified to a tree of Expressions through a depth-first search. For each rule in the Irony parse tree, the appropriate builder method is invoked to create the corresponding Expression implementation. These builder methods are purposefully implemented as static methods to ease rewriting them as classless Pie functions to demonstrate functional programming in the next iteration.

As stated, the current version of the validator simply passes the tree of Expression implementations to the generator.

The generator works in much the same way as the parser: it performs a depth first search of the tree of Expressions, but for each it produces the corresponding CodeDOM expression or statement (CodeDOM defines a “statement” as an expression capable of existing alone). These CodeDOM expression can then be used to generate code for any .NET language with CodeDOM support. In this case it generates C# code that is then compiled by the C# compiler. The emitting methods are also implemented as static methods to facilitate rewriting them as Pie classless functions.

A command line application was also implemented for the compiler, in the Pie language. This command line compiler is capable of generating executables from Pie source code. It functions simply by parsing simple command line options, loading source files, and compiling them with the Pie compiler.

Challenges

There were a number of challenges in this project, the first of which has already been discussed: the difficulty of writing unit tests in advance when one's understanding of the tools is constantly evolving. This was overcome by prototyping tests in an informal working project, then graduating them into the formal test suite once the test was deemed to be satisfactory.

A second challenge of testing was obtaining adequate code coverage, or even assessing what level of coverage qualifies as “adequate”. If one tries to test every minute detail, one winds up with a combinatorial explosion and an unmanageable number of tests. There has to be threshold past which coverage is considered to be “good enough”: it is impossible to find every bug. This was amply demonstrated when implementing the command line compiler in Pie: writing more complex code revealed bugs and missing features not considered when writing the unit tests. It showed that while unit testing is important, testing with production code is also important.

The only IDE support provided for current iteration is a simple syntax highlighter: creating Visual Studio extensions was found to be surprisingly complex and undocumented. Creating a full featured IDE extension with support for refactoring, code completion, unit test visualization, code coverage analysis, and runtime debugging is by itself a large enough task to be worthy of a COMP 495 project.

An attempt was initially made to write the parser with no third party tools, but it too was found to be a monumental task that is worthy of a project itself. Additionally, doing so is a wasted effort when there are already so many solutions available: it is a solved problem. Irony is an excellent parsing tool with a developer who is both active and helpful. However, Irony is high level enough that in some cases it was found to be inflexible. For example, the intention was for generics to be defined as in C#: with opening and closing greater and less than signs (class foo<T>). However, the way in which Irony defines operators means that when it sees those tokens, it assumes an operator, even in contexts inappropriate for operators. This is why the generics are defined in Pie as “class foo{T}”. This works, but is certainly less readable: curly braces are less distinguishable from parentheses. The hope is that in the next iteration the rules can be refined or alterations made to Irony (with the developers assistance) to make C-style generics possible.

Finally, CodeDOM is surprisingly buggy and incomplete: it does not cover all language constructs, even the basic ones. Examples of bugs or missing features are:

1) Event declarations can not be static.

2) No operator overloading.

3) No support for unary operators of any kind, most notably negation and inversion.

4) Missing binary operators such as exclusive OR and bitwise shift.

As a consequence these features are missing from the Pie language. They can be overcome in the next iteration by either dropping CodeDOM and generating C# source code manually, or by inserting custom code snippets into the CodeDOM-generated source code. The former option may be preferable as one has to wonder: if features as fundamental as these are broken or missing, what else will be? Inserting code snippets to shore up missing features also seems like a fragile solution.

Results

The results of this project are the following deliverables:

1) The Pie compiler as an implementation of .NET's CodeProvider class. This is implemented in C# as a DLL and can be used from any other .NET language for runtime compilation of Pie code.

2) A Pie command line application making use of the PieCodeProvider to compile Pie source from the command line. This application is implemented in the Pie language.

3) A test suite of 187 unit tests.

4) Pie language specification document.

5) A simple syntax highlighter extension for all versions of Visual Studio 2013 except for the Express version, which does not support extensions.

Self Assessment

First, this project far exceeded my expectations in terms of learning about test driven development: I can't fathom the idea of coding any other way. I expected a boost in the quality of my code, but did not expect the boost in productivity that I experienced. In fact, I had assumed that writing the unit tests would be extremely time consuming. While writing the tests was indeed time consuming, the trade-off was less time and frustration spent on debugging: instead of spending hours searching for a bug, the unit tests immediately point the way. The unit tests are also a one-time investment of effort: once they're implemented they can be re-run and re-used any number of times. Debugging an application without unit tests is not a one time investment of effort.

This project resulted in the largest and most sophisticated complete product I have made, and feel that the architecture is excellent. The design is very simple, while the components of the compiler are entirely decoupled to make testing and replacement simple.

There are two quality criticisms that I have to make, that reveal that this a) is a first iteration and b) was very much a learning process. The grammatical rules, while adequate, could be far cleaner: the way that they are presently implemented results in a parse tree requiring a lot of redundant code to interpret. For example, there are duplicate rules and duplicate code to interpret declaration modifiers (public, private, shared, final, etc). Different kinds of types such as classes, enums, and interfaces each have their own rules which could be unified. This leads to two of the goals that I will list for my COMP 496 proposal: rewriting the grammatical rules with a more mature understanding of how they work, and rewriting the compiler in Pie. While rewriting the compiler in Pie I will remove the redundant code, and writing a complex project in the language will certainly reveal many bugs and missing features not covered by the unit tests. As these errors are revealed, new unit tests can be created to account for them, ensuring that they are detected if they reappear later.

This project took 5 weeks rather than the projected 4. However, thanks to test driven development, I feel that taking 5 weeks to implement a project of this size is still impressive.

Overall I would assess this project as good to excellent, particularly in terms of accomplishing the desired goals of implementing a custom language and compiler with test driven development, and while openly acknowledging some minor code issues that I plan to correct in the second iteration in COMP 496. In fact, I am purposefully leaving the redundant code in: I see no reason to bother replacing them in this iteration considering that it's essentially a prototype that will be rebuilt from scratch in COMP 496.

Citations

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

[2] Pilgrim, Mark. (2004). *Dive Into python: 5.9 Private Functions* [Online] Available: http://www.diveintopython.net/object_oriented_framework/private_functions.html

[3] Gouy, Isaac. (2015). *The Computer Language Benchmarks Game* [Online] Available: http://benchmarksgame.alioth.debian.org/u64q/python.html

[4] Oliveira, Rodrigo B. de. (2013). *The Boo Programming Language* [Online] Available: https://github.com/bamboo/boo

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

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

[7] Wikipedia, (2015). *Cython* [Online] Available: https://en.wikipedia.org/wiki/Cython