Rattlesnake: An attempt to create a Python-to-Rust compiler

Toby Collier

Contents

Abstract

This project, Rattlesnake, was an attempt to create a Python-to-Rust compiler. While the attempt was ultimately unsuccessful, there are meaningful lessons to be learned from the experience.

This report will discuss the motivation behind this project, the expected challenges, the unexpected challenges, and the lessons learnt. It will also discuss the artefacts produced, and the reasons why their approaches would not scale to a full compiler.

1 Introduction

1.1 Background

Python is a popular programming language, known for its ease of use and flexibility. However, it is also known for its speed; or rather, lack thereof. CPython, the official and most commonly used implementation of Python, is an interpreter, a much slower model of code execution than the traditional compile-execute model used by languages such as C and Rust. Alternatives, such as PyPy or Cython, have been created to bridge the gap, but each has its own trade-offs (addressed in section 2.7, *Why a Python-to-Rust compiler would be useful*). Rattlesnake was an attempt to provide a different set of trade-offs, by translating Python code into Rust code, which can then benefit from Rust's rich ecosystem.

1.2 Aim

The aim of Rattlesnake was to provide a compiler from Python to Rust, allowing Ahead-Of-Time (AOT) compilation and optimisations to be performed on Python code, while not requiring the programmer to learn new syntax. This would have been achieved by translating Python source code directly into Rust, deferring most of the type-checking stage to the Rust compiler.

The eventual goal of Rattlesnake was to infer types from Python source code, and allow using Python types from within the compiled Rust output. This would allow Rattlesnake to be usable on (almost) any Python code base with no changes to the source code, and would require minimal changes to be made to the source code in most other cases.

Additionally to the core compiler, Rattlesnake would have provided a run-time Python library poly-filling types, functions, and macros from Rust's standard library, allowing writing optimised Rust code from within Python, before translating it to Rust. This would have the primary advantage of allowing REPL iteration, while still providing the speed of Rust once compiled.

1.3 Objectives

The objectives of the project were as follows:

1. Create the core Rattlesnake compiler, which would take Python source code as input, and produce Rust source code as output.
2. Create the run-time Rattlesnake library for Rust, which would provide various standard Python types and functions, enabling most code to be translated without changes.
3. Create the run-time Rattlesnake library for Python, which would provide access to various standard Rust types, functions, and macros, and act as a poly-fill for interpreted Python code, as well as allowing type-checkers to understand Rattlesnake code.
4. Create the associated documentation for the Rattlesnake compiler and run-time libraries.
5. Create these documents as part of the dissertation project.

1.4 Outcome

In all, 3 working prototypes of the Rattlesnake compiler were produced, along with an experiment which attempted to leverage the `py2erg` project alongside the Erg compiler to provide type information to the Rattlesnake compiler. Each of these will be discussed further in their own sub-sections of the Artefacts section.

1.5 Product Overview

1.5.1 Scope

The scope of Rattlesnake was to take as input arbitrary Python 3 source code, and translate that source code into Rust source code - or raise an error if the input contains unsupported features. The output Rust code would be able to be

compiled into a native executable, which could then be run without a Python interpreter. The output code would depend on the library `rattlesnake`, which would have been a Rust library providing various standard Python types and functions.

There would also have been a Python library `rattlesnake`, which would provide various standard Rust types, functions, and macros, and could be used by Python code in order to produce more optimised or efficient Rust code.

1.5.2 Audience

Rattlesnake was intended to be a tool for Python developers who want the speed and memory-efficiency benefits of Rust. It would also serve as a rapid prototyping tool for Rust projects, and a way to simplify the distribution of Python projects as a single executable which would reduce the effort required for deployment.

2 Background Review

2.1 What is a compiler?

At its core, a compiler is a program which takes in code written in one language, and outputs code written in a different language. Most commonly, a compiler will either translate code into machine code (such as in C, C++, and, Rust, to name a few), in order for the program to run natively on a particular machine, or will translate code into machine-agnostic byte-code (such as in Java, Python, and C#), in order for the program to run on a virtual machine, an intermediate program which can run natively on many machines and which interprets the byte-code.

Another common use of compilers is to translate code between two high-level languages, for example Vala and V, which both compile to C, or TypeScript, which compiles to JavaScript. This is also known as a 'transpiler'.

2.1.1 Aside: Transpilers

A transpiler is a specialised term for a compiler which translates between two languages which are at a comparable level of abstraction (Fenton, 2012). For example, a compiler from C to assembly would not be considered a transpiler, as assembly is a much lower-level language than C, but a compiler from C++ to C could be considered a transpiler, as both C and C++ allow for many operations to be performed in a single statement. While Rattlesnake fits within this category (as Python and Rust are both considered high-level languages), the term 'compiler' will be preferred over 'transpiler' throughout this document, except when referring to sources which use the term 'transpiler'.

2.2 How compilers work

Typically, a compiler will work in several stages, each of which transforms the input code in some way. While the exact stages vary from compiler to compiler, the most common steps are:

1. Parsing: The input code is read and transformed into a more easily manipulable form, typically an Abstract Syntax Tree (AST).

   There are several approaches to this process, but the two most common are:

   - Tokenisation/Parsing
     - Often performed using the tools Lex (from 'Lexical analysis',) and Yacc, leading to this form often being referred to more generally as Lex/Yacc parsing
     - This parsing method first uses a tokeniser (or 'lexer') to transform the input code into tokens - the fundamental building blocks of the language - and then a token stream parser to transform the tokens into the internal representation. In the tokenisation ('lexing') step, the input code is split into tokens using regular expressions; in the parsing step, a Context-Free Grammar (CFG) - in the case of Yacc, specifically an $LALR(1)$ ('look-ahead (1 token), left-to-right, right-most derivation') grammar (Johnson, 1978) - is used to transform the token stream into the AST.
   - Parsing Expression Grammars (PEGs), which are a more modern approach to parsing, created by Ford (2004), where no tokenisation step is required. In PEG parsing, the grammar defines an unambiguous parse order, which means that implementing a PEG parser based on a grammar is much simpler than implementing a Lex/Yacc parser. In particular the use of ordered-choice operators in PEGs, combined with the fact that backtracking is disallowed, means that PEGs are very closely related to the actual code necessary to execute the parse for that language.

2. Transformation: The AST is transformed in various ways, typically being translated into an intermediate representation (IR) which can be manipulated outside the confines of the syntax of the input language.

3. (Optional) Optimisation: The IR is optimised through various methods, including dead code elimination, constant pre-computation, function inlining, and instruction reordering to take advantage of the cache.

4. Code generation: The IR is transformed into the output language. For machine languages this is often as simple as iterating over the IR and outputting the corresponding object code, but for higher-level languages it may involve reconstructing an AST for the output language, and then outputting that AST as code.

2.3 Classifications of grammars

In formal language theory, grammars are classified into one of four types, based on the simplest automaton which can recognise them. This classification is known as the Chomsky hierarchy, and is as follows:

Table 1: The Chomsky hierarchy

Classification	Language	Recognising automaton
Type 0	Recursively enumerable	Turing machine
Type 1	Context-sensitive	Linear-bounded non-deterministic Turing machine
Type 2	Context-free	Non-deterministic push-down automaton
Type 3	Regular	Finite-state automaton

The Chomsky hierarchy is a strict hierarchy; all type 3 grammars are type 2 grammars, all type 2 grammars are type 1 grammars, and all type 1 grammars are type 0 grammars.

2.4 What is a Parsing Expression Grammar?

A Parsing Expression Grammar (PEG) is a style of grammar created in 2004 by Bryan Ford. PEGs do not fit neatly into the Chomsky hierarchy; according to Ford (2004),

> These properties [that 1. PEGs can express LL(k), LR(k), and 'many others, including some non-context-free' languages and 2. *all* PEGs can be parsed in linear time] strongly suggest that CFGs and PEGs define incomparable language classes.

Additionally, Loff, Moreira and Reis (2020) have shown that there can be no pumping lemma for PEGs, and that there exists a PEG for a P-complete language (under logspace reductions), in contrast to context-free languages, which cannot be P-complete under logspace reductions 'unless $P \subseteq NC_2$'.

The pumping lemma is a lemma which states that, for all regular or context-free languages, there exists a 'pumping length' $p \geq 1$ such that any string of that length or longer can be 'pumped' - that is, split into a number of sub-strings (5 for context-free languages, 3 for regular languages); for context-free languages $s = uvwxy$ where s is a string in the language of at least p symbols, vx is not empty, vwx is at most p symbols long. Given these properties, the Pumping Lemma states that uv^iwx^iy is also in the language for all $i \in \mathbb{N}^0$.

That no pumping lemma can exist for PEGs shows that they are fundamentally different from context-free grammars, and that they are able to express languages which context-free grammars cannot.

NC_2, or 'Nick's Class', is a complexity class which is often considered to be a strict subset of P (the class of problems which can be solved in polynomial time), and is related to problems which are efficiently solvable on a parallel computer (Arora and Barak, 2009).

PEGs are less common than Context-Free Grammars (CFGs), but have a number of advantages over CFGs when parsing machine languages:

- PEGs always have a single, unambiguous parse for any input. This is a significant advantage over CFGs, as it means there is no need to learn ambiguity resolution rules, and the parse tree can (theoretically) be used directly as the AST.
  - This also serves as a speed advantage - 'Ambiguity in CFGs is difficult to avoid even when we want to, and it makes general CFG parsing an inherently super-linear-time problem' (Ford, 2004).
- PEGs are recognition-based, rather than generation-based. This is in contrast to most of language theory, but directly in line with most practical language applications within computer science. This also means that PEGs can be translated into parsers almost directly; Ford notes that 'A PEG may be viewed as a formal description of a top-down parser'.

2.4.1 A small PEG example

Before continuing, it may be useful to illustrate the use of PEGs with a small example.

In table-top games, it is common to represent dice rolls with a textual notation indicating which dice to roll, and how to modify the result (if applicable). For example, the simplest notation is `d6`, which simply means 'roll (one) six-sided die'.

This notation can get arbitrarily complex (I have created a moderately-complete grammar for this, which is included in Appendix A), but here are some more simple examples:

Table 2: Examples of dice expressions

Dice expression	Meaning	Image	Shown result
`d6`	Roll one six-sided die		4
`2d6`	Roll two six-sided dice, and sum them (very common in board games)		7 (1 + 6)

Dice expression	Meaning	Image	Shown result
`4d6`	Roll 4 six-sided dice, and sum them		18 (5 + 3 + partially-obscured 4 + partially-obscured 6)
`3d6 + 2`	Roll 3 six-sided dice, sum them, and add 2		16 (6 + 2 + 6 + modifier 2)
`2d4`	Roll two four-sided dice		7 (3 + 4)

Such expressions can be parsed with a PEG, such as that in Appendix A (in the PEG syntax used by the `pest` parser generator). Under this grammar, the expression `3d6 + 2` would be parsed as follows:

```
(Expression
  (Expr
    (MinMaxTerm
      (AddTerm
        (MulTerm
          (Repeat
            (Number "3")
            (DieRoll
              (Number "6"))))
        (AddOp "+")
        (MulTerm
          (Repeat
              (Number "2"))))))
  (EOI ""))
```

Notice that there is no explicit tokenisation step in a PEG parse; because PEG is recognition-oriented, rather than the traditional generation-oriented grammars, tokenisation can be performed at precisely the point where it is needed, rather than as a separate step. This has the additional advantage of allowing for 'soft keywords' in programming languages, a concept where a keyword is only a keyword in certain positions, and in all other positions is free to be used as an identifier.

For example, in Python, the switch to a PEG parser (introduced optionally in Python 3.9, but made the sole parser in 3.10) allowed for the `match` construct to be introduced - `match` is a soft keyword, meaning that it is only a keyword when it is part of a match statement. This was important for backwards compatibility, as `match` was already a common identifier in Python code, particularly when dealing with regular expressions - and thus it would not be reasonable to make `match` a 'hard' keyword.

2.5 Python

Python is 'the most popular language in Machine Learning' (Lunnikivi, Jylkkä and Hämäläinen, 2020), due largely to the fact that '[it] tends to be readable and concise, leading to a rapid development cycle' (Behnel *et al.*, 2011).

Its ease-of-use also makes it a popular choice for scripting and other short programs. For example, approximately 40% of respondents to the Advent of Code surveys from 2018 to 2023 used Python as their primary language for the event (Heijmans, 2023).

Python, however, is not without its drawbacks. The most significant of these is its speed; as an interpreted language, Python is significantly slower than compiled languages such as C or Rust. Efforts have been made to improve its speed, such as the PyPy project, but it is still a significant way behind the performance of compiled languages.

2.6 Rust

Rust, the most loved programming language in the StackOverflow Developer Survey every year since 2016 and the third most loved language in the year of its release, 2015 (Stack Overflow, 2015, 2016, 2017, 2018, 2019, 2020, 2021, 2022, 2023), is 'a systems programming language meant to supersede languages like C++' (Bugden and Alahmar, 2022). It is designed around enforcing memory safety without the use of a garbage collector, but its data model is also well-suited to most other kinds of safety, including thread-safety and type-safety, as well as Software Fault Isolation and Information Flow Control analysis (Balasubramanian *et al.*, 2017), all without sacrificing performance.

Additionally, Rust has a strong focus on ergonomics, having a powerful macro system and a standard build tool, Cargo, which also handles dependency management (from a centralised repository, crates.io) and testing.

2.7 Why a Python-to-Rust compiler would be useful

CPython (the official Python implementation, and the most commonly used one), being an interpreter, is significantly slower than Ahead-of-Time (AoT) compiled binaries such as those produced by the compilers for languages such as C or Rust. This is due in part to the fact that the interpreter must maintain its own state in addition to that of the program being run, and also to the fact that while AoT-compiled binaries are optimised before they begin to run, CPython must instead attempt to optimise the code during execution, and must do so again every time the code is run.

Alternatives to CPython, such as PyPy or Cython, have been created in an attempt to improve the speed of Python programs; while these are certainly faster than CPython, they each have their own trade-offs. PyPy, for example, is a Just-In-Time (JIT) optimising interpreter, meaning that it can compile so-called 'hot paths' - those which are run more frequently than others - either directly to machine code, or (more commonly) to specialised instructions within the interpreter. Unfortunately, these optimisations take time to perform, and so PyPy is not suitable for short-lived programs. Cython, on the other hand, is a

hybrid between Python and C, and can be used to generate C extension modules and even convert a source file directly into C code. However, Cython-generated code, by default, does not greatly optimise over CPython - as Cython aims to provide 100% compatibility with CPython, and will not infer specific types except in certain patterns, it is up to the programmer to explicitly declare functions and variables using either `cdef` or `cpdef` , meaning that the source code is no longer valid Python.

A Python-to-Rust compiler would allow for the best of both worlds: the ease-of-use and rapid iteration cycle of Python, and the performance and optimisations of Rust. This would be especially useful in the context of Rapid Application Development (RAD), where the ability to quickly prototype and iterate is crucial, but performance may also be a consideration.

2.8 Why Rust is a viable compile target

Rust, unlike C or C++, has a rich type system with plenty of expressive power, and its standard library and ecosystem share this expressiveness. Additionally, many of Rust's core features, such as its iterator-based for-loops and its pattern matching, are similar to those found in Python, and so a lot of Python's expressiveness can be translated directly into Rust in a way that would not be possible with many other languages.

3 Methodology

3.1 Software development process

The software development model I chose for Rattlesnake was the Rapid Application Development (RAD) model of development.

RAD is a methodology which focuses on rapid prototyping and iteration. It is a form of agile development, conforming strictly to the Agile Manifesto (Beck *et al.*, 2001), and is particularly suited to projects where the requirements are not fully known at the start of the project, or may change as the project progresses.

I chose RAD primarily because it has a significantly reduced overhead compared to other methodologies, such as the mountain of documentation associated with scrum, or the strict requirements of waterfall. It also allows for a more flexible timescale, as each iteration can be as long or as short as necessary.

Over the course of the project, I produced three compilers (and one experiment):

- Version 1, in Python, using the built-in module `ast` (section 4.1)
- Version 2, in Rust, using a custom PEG grammar (section 4.2)
- Version 3, in Rust, using the `rustpython_parser` library (section 4.3)
- What would have been version 4, in Rust, attempting to use the `py2erg` and `erg_compiler` libraries (section 4.4).

3.2 Technologies used

The primary technologies used in the development of Rattlesnake were Python and Rust, along with the libraries `pest` and `rustpython_parser` . Addi-

tionally, argument parsing was performed using the `clap` library, and the project was built using `cargo`.

This document was written in Markdown, and converted to PDF using Pandoc (via PDFLaTeX). The bibliography is managed using BibTeX, and the citations are formatted according to a Brookes-style CSL file. The presentation slideshow was created using Obsidian (with the Advanced Slides plugin), and exported to PDF using Google Chrome. the poster was created with Figma, and then resized to the correct size using `pdfjam`.

The resulting PDFs were combined into the final document using `pdftk-java`, and the metadata was corrected using `exiftool`.

3.3 Version management

The version management system used for Rattlesnake, and its prose documents, was `git`. The repository is hosted on GitHub at https://github.com/Starwort/py-rattlesnake (private repository) and includes all the code, as well as the prose documents.

4 Artefacts

The initial plan for the project was to first create the core compiler in Python, then to translate the compiler into Rust, and finally to create the run-time libraries for both Python and Rust. However, for various technical reasons, the Rust version of the compiler ended up being 2 versions of the compiler and a failed experiment; the `pest` version, the `rustpython_parser` version, and the `erg_compiler` experiment. A full comparison of the features supported by each version of the compiler is available at the end of the section.

4.1 Compiler v1: Python and `ast`

The first attempt at creating the compiler aimed to test the feasibility of the project by using Python's built-in `ast` module to parse the code, rather than starting in Rust and writing a parser from scratch.

Because `ast.NodeVisitor` is a class, all the code for the compiler had to be placed within a single class, which made the code difficult to read and maintain. Additionally, the lack of a strongly-typed `match` statement in Python made writing the translation code more difficult, as I could not be sure that all the cases were handled (either by compiling correctly, or by raising an error).

These issues meant that, once I had a prototype that was functional enough to convince me of the feasibility of the general approach, I discontinued development of this version of the compiler, in favour of a more robust and easily-maintained compiler in Rust.

At the time of discontinuation, the compiler supported:

- If-statements

- Assignments (including augmented assignments and annotated assignments, which were in fact *required* for a variable to be declared)
- Dictionary literals (depending on a run-time library feature which had not yet been written)
- Formatted strings (f-strings)
- Expression statements (where the expression was already supported)
- String, integral, floating-point, boolean, and `None` literals
- List literals (where the items within the list literal are assumed to be of the same type)
- For-loops (where the iterable is assumed to implement `Iterator` or `IntoIterator`)
- Attribute access (where the access is performed dynamically using a library feature which had not yet been written)
- Function calls (including macro calls where the Python name for the macro starts with `MACRO_` , and keyword arguments within macro calls only)
- Import and import from statements (where the imported module is assumed to exist, unless the imported module is `rattlesnake` , in which case the import is translated to `std`)
- Function definitions (where un-annotated arguments are translated to be of type `()`)
- Return statements, with or without a value
- Binary operations (other than matmul, `@` , pow, `**` , and floor division, `//`)
- The `pass` statement (which is translated to the block comment `/* pass */`)
- Storage classes and visibility modifiers (using Rattlesnake type primitives)

Example Python code

```
from rattlesnake.prelude import *

for n in [3, 5, 7]:
    MACRO_println("Multiplication Table:")
    for j in [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]:
        result: usize = n * j
        MACRO_println("{n} * {j} = {result}")
```

Example Rust code, as output by the compiler given the example Python code

```
use rattlesnake::prelude::*;

fn main() {
    for n in [3, 5, 7] {
        println!("Multiplication Table:");
        for j in [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] {
            let result: usize = n * j;
            println!("{n} * {j} = {result}");
        }
    }
}
```

Overall, while somewhat successful, this version of the compiler was not sustainable for further development.

4.2 Compiler v2: Rust and `pest`

The second attempt at the compiler used Rust and `pest`, a PEG (Parsing Expression Grammar) parser generator.

The grammar used for this compiler is in Appendix B.

This attempt was discontinued because maintaining a grammar for Python is a considerable amount of work, and the grammar would need to be updated every time support for a new feature was added to the compiler. Additionally, I had just discovered that `rustpython_parser` existed and was usable as a library, and so I realised that using that would allow me to focus on the challenges of the actual translation of the code, rather than the parsing of it. Furthermore, the general architecture of the compiler had some immediate issues:

- Due to the way strings were being concatenated, the output of the compiler was very badly formatted, and would not have been easy to manipulate to be better formatted
- Because `pest` returns a parse tree of rules rather than a typed AST, all the transformation code needed to be written in a single recursive function, which causes the same issues as the first version of the compiler's single-class architecture
  - To be more precise, `pest` parsers return a single recursive type, `Pairs` - an iterator of `Pair`s containing a `Rule` and an inner `Pairs`. Because of this single-type return, the transformation all happens within a single `match` statement, which has to manage all the different cases of the grammar, even when certain rules can only be reached in certain contexts
- Because Python uses significant whitespace, the grammar was incredibly brittle in terms of the input it would accept; while this would be fixed by `pest` version 3, which allows the use of two separate whitespace rules, that version is not yet stable, and so was not used in this project.

At the time of discontinuation, the compiler supported:

- Return statements (despite the fact that functions were not yet supported)
- Assignments, including augmented assignments and annotated assignments
  - Annotated assignments were required for a variable to be declared, although Rattlesnake type primitives were not supported. Additionally, since subscript is not implemented in the grammar, generic types could not be instantiated.
  - Augmented assignments do not support the matmul (`@`), pow (`**`), or floor division (`//`) operators
- Import and import from statements
  - The imported module was assumed to exist; there was no special support for `rattlesnake` imports
- Binary arithmetic operations (other than matmul, `@`, pow, `**`, and floor division, `//`)

- Binary boolean operations
- Unary arithmetic and boolean operations
- The `pass` statement (which is translated to the block comment `/* pass */`)
- For-loops (where the iterable is assumed to implement `Iterator` or `IntoIterator`)
- If-statements (where the condition is assumed to be a boolean)
- Function calls (basic support, but no grammar-level support for keyword arguments, and no support for macro invocations)
- List literals (where the items within the list literal are assumed to be of the same type)
- Number, string, boolean, and `None` literals
  - Decimal literals may be either integers or floats; hex, octal, and binary literals are only allowed to be integers (at the grammar level). As all supported numeric literals are also valid Rust literals, they are copied directly.
  - `None` literals are translated to `()`

Example Python code (trailing newline is required by this version of the compiler)

```
from rattlesnake.prelude import MACRO_println, usize
x: usize = 0
MACRO_println("x = {x}")
x: usize = x+1
MACRO_println("x + 1 = {}",x+1)
```

Example Rust code, as output by the compiler given the example Python code

```
fn main() {
use rattlesnake::prelude::{MACRO_println,usize};
let x: usize = 0;
MACRO_println("x = {x}");
let x: usize = x+1;
MACRO_println("x + 1 = {}",x+1);
}
```

4.3 Compiler v3: Rust and `rustpython_parser`

The third attempt at the compiler used Rust with `rustpython_parser`, the parser from the RustPython project. This attempt was the most generally feature-complete of the three, which enabled me to run into the largest road-block of the project: type information.

4.3.1 Why type checking is a fundamental requirement

Type-checking, at first, does not seem like a fundamental requirement for the Rattlesnake compiler. After all, the Rust compiler is able to infer types in almost all cases - so it would seem that there would be no need for Rattlesnake to handle or apply any type information other than that supplied by annotations on functions or explicitly annotated variables.

Unfortunately, this is not the case. Python's namespacing system is inherently ambiguous, using the `.` operator to access items from modules, static items from classes, and instance items from objects. This means that, in order to correctly translate the code into Rust, Rattlesnake would need to be able to know the type of access being performed in order to use the correct operator in Rust; `::` for module or associated item access, and `.` for instance item access. This is further complicated by the fact that static access is allowed on instances, and therefore the compiler would need to transform the access to be performed on the struct, using path syntax, rather than on the instance, using attribute syntax.

Additionally, the lack of type inference in the compiler means that every initial assignment would need a type declaration on it (to indicate to the compiler that it is a new variable), thus significantly reducing the applicability of the compiler.

4.3.2 The other subtle issue

Another issue looming over the compiler was the growing issue of the architecture. Because the translation was done directly into strings, rather than into a Rust AST, it was difficult to further manipulate the output. This would be necessary in order to perform transformations such as lifting `use` statements, functions, and constants to module level. Since those transformations weren't yet in scope, I had not considered this issue, but if the project had progressed to that point, this would have been a significant hurdle to overcome.

At the time of discontinuation, the compiler supported:

- Storage classes and visibility modifiers (using Rattlesnake type primitives)
- Function definitions (where un-annotated arguments are translated to be of type `()`)
  - Function modifiers are applied by annotating the return type of the function
- Return statements, with or without a value
- Assignments, including augmented assignments and annotated assignments
  - Annotated assignments were required for a variable to be declared, and could use Rattlesnake primitives to declare storage classes and visibility modifiers
- For-loops (where the iterable is assumed to implement `Iterator` or `IntoIterator`)
- If-statements (where the condition is converted to a boolean using a Rattlesnake library function)
- Assert statements (where the condition is assumed to be a boolean, and the message is assumed to be in macro format)
- Import and import from statements
  - Imports from `rattlesnake.prelude` are ignored; other imports from `rattlesnake` are translated to `std`
  - All other imports are assumed to exist
- Boolean operations
- Binary arithmetic operations
  - Matmul (`@`), pow (`**`), and floor division (`//`) are transformed into function calls

        * The augmented assignment operators use function calls to the corresponding `_in_place` versions of those functions
- Lambda expressions (where the argument types are assumed to be inferred)
- Call expressions (where the function is assumed to exist)
  - Keyword arguments are not supported outside of macro invocations
- List literals
- Constants
  - `None` literals are translated to `PyNone`
  - Boolean literals are translated to Rust boolean literals
  - String constants are translated to Rust string literals
    * This technically causes a memory leak in the compiler, as the string is never deallocated, but this should be fine for most code as the allocated memory will be deallocated when the compiler exits; in the translated code, the strings are constants
  - Bytes literals are translated to Rust arrays of bytes
  - Integer literals are translated to Rust integer literals
    * This also causes a memory leak in the compiler, for the same reason as for strings. This is also unlikely to be an issue, as integers are (usually) small and so the leaked strings will also be small
  - Tuples are translated directly into Rust tuples

Example Python code

```python
from rattlesnake.prelude import *

def generate_multiplication_table(n: usize):
    MACRO_println("Multiplication Table:")
    for j in [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]:
        result: usize = n * j
        MACRO_println("{n} * {j} = {result}")

for n in [3, 5, 7]:
    generate_multiplication_table(n)
```

Example Rust code, as output by the compiler given the example Python code

```rust
use rattlesnake::prelude::*;

fn main() {
    fn generate_multiplication_table(n: usize) {
        println!("Multiplication Table:");
        for j in [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] {
            let result: usize = n * j;
            println!("{n} * {j} = {result}");
        }
    }
    for n in [3, 5, 7] {
        generate_multiplication_table(n);
    }
}
```

4.4 The experiment: Rust, `py2erg`, and `erg_compiler`

The fourth attempt at the compiler was an attempt to leverage type information generated by the Erg compiler. Erg is a programming language which compiles into Python, and has Rust-like static type checking. The idea for this approach came from the `pylyzer` Python linter, which uses its own `py2erg` crate to transform Python code into Erg code, which is then analysed by the Erg compiler, the output of which is reported back to the user.

While this approach seemed promising in theory, in practice there are two major problems:

- `py2erg` depends on an outdated version of `rustpython_parser`, which will not compile with the latest version of Rust; downgrading Rust to use `py2erg` is not an option, as `erg_compiler` depends on a feature which was not stabilised until a later version of Rust.
- Even after vendoring a patched copy of `py2erg`, `erg_compiler` is not designed to be used as a type-checking library, and so it cannot output type information about specific items, unless there is a type error in the code.

As a result of these problems, the experiment never made it to the point of being able to parse Python code to a suitable level of detail, and so was discontinued.

4.5 Summary

Table 3: Comparison of supported features

Feature	Compiler v1	Compiler v2	Compiler v3
Annotated assignments	Yes[1]	Yes[1]	Yes[1]
Assert statements	No	No	Partial[2]
Assignments	Yes	Yes	Yes
Attribute access[3]	Partial[4]	Partial	Partial
Augmented assignments	Partial[5]	Partial[5]	Yes[6]

[1] Annotated assignments are required for a variable to be declared

[2] Assert statements assume their condition is already a boolean, and their format string must be `format!` -compatible

[3] Attribute access is an inherently ambiguous operation, as the same syntax is used for namespace lookup, static (class variable) lookup, and instance attribute lookup. Thus, without type information, it is impossible to determine which operator to use in Rust - the prototypes all assume instance attribute access (i.e., the `.` operator)

[4] The attribute lookup is performed dynamically, using a run-time library interface which was not yet written

[5] Matmul (`@`), pow (`**`), and floor division (`//`) are not supported

[6] Matmul (`@`), pow (`**`), and floor division (`//`) are transformed into function calls, provided by a (yet-unwritten) run-time library

Feature	Compiler v1	Compiler v2	Compiler v3
Binary arithmetic operators	Partial[5]	Partial[5]	Yes[6]
Binary logical operators	Yes	Yes	Yes
Constant literals	Yes[7]	Yes[8,9]	Yes[7,9]
Dictionary literals	Yes[10,11]	No	No
Expression statements	Yes	Yes	Yes
For-loops	Yes[12]	Yes[12]	Yes[12]
Formatted strings (f-strings)	Yes	No	No
Function calls	Partial[13]	Partial[14]	Partial[13]
Function definitions	Partial[15]	No	Partial[15]
If-statements	Yes[16]	Yes[17]	Yes[16]
Import and import from statements	Yes[18]	Yes[19]	Yes[18]
Lambda expressions	No	No	Yes[20]
List literals	Yes[11]	Yes[11]	Yes[11]
`pass`	Yes	Yes	Yes

[7] `None` literals are translated to `PyNone` in Rust

[8] `None` literals are translated to `()` in Rust

[9] The compiler leaks memory when translating string and integer literals

[10] Dictionary literals are translated to use a run-time library feature which had not yet been written

[11] Items inside a collection literal are assumed to be of the same type, causing a compile error in the Rust code if they are of incompatible types

[12] For-loops assume that the translated iterable implements `Iterator` or `IntoIterator`

[13] Keyword arguments are not supported outside of macro invocations; any call to a function starting with `MACRO_` is translated into a macro invocation

[14] Function calls are supported, but do not support keyword arguments

[15] Unannotated arguments are translated into `()`; visibility modifiers and `const` can be applied to the function by applying Rattlesnake type primitives to the function's return value

[16] If-statements use a run-time library feature, `bool()`, to coerce the condition of an if-statement to a boolean

[17] If-statements assume that the condition is a boolean

[18] Imports from `rattlesnake` are translated into imports from `std`; all other imports are assumed to exist. Imports from `rattlesnake.prelude` are excluded entirely

[19] All imports are assumed to exist

[20] Lambda expressions assume that the argument types are inferable from context

Feature	Compiler v1	Compiler v2	Compiler v3
Return statements	Yes	Yes	Yes
Storage classes and visibility modifiers	Yes	No	Yes
Unary operators	No	Yes	Yes

5 Professional issues

If I were to finish and release Rattlesnake, I would need to consider several professional issues:

5.1 Legal

The largest issue I would need to consider is licence compatibility. I would need to ensure that the way Rattlesnake would be released conformed to both its own licence, as well as the licences of the libraries it used. This would be particularly important if I were to use my originally-planned licensing scheme, where the compiler would be dual-licensed under GPL-3.0 (a viral copy-left licence) and customised business licences. Because many Rust libraries are licensed as one of MIT, Apache-2.0, or dual-licensed as either, I would need to ensure either that those libraries' licences were compatible with my chosen licences, or that the way I distributed the project was compatible with those libraries' licences.

It could also be necessary to ensure that the licence of the compiler had a provision which allows the compiler's output to be licensed under a different licence; while this would be a non-issue for the business licence, as it would be a custom licence for each instance, it could be an issue for the GPL-3.0 licence, as it is a viral licence and so could cause disruption for other open-source projects which wished to use the compiler.

5.2 Ethical

Another important issue I would face is that I would have a moral and ethical responsibility for supporting the compiler and its output - I would need to ensure that the output code was correct (i.e., that it doesn't contain any bugs which are not present in the input code), and that as much of Python as possible would be supported. Were Rattlesnake to introduce a bug into the output code, it could have serious consequences for the users of the compiler, and I would be responsible for causing it.

5.3 Environmental

Were the project to be successful, I believe it would have a positive impact on the environment. By allowing Python code to be compiled into Rust, it would allow large Python code-bases to be compiled into more efficient machine code,

which would run faster and use less memory. This would be a net energy saving, as fewer computational resources would be required to run the code. This effect compounds as the frequency or lifespan of the executions increases, as the initial energy cost of compilation and optimisation is amortised over more time.

6 Future work

To create a useful artefact from the experiences of this project, there are two primary avenues which could be pursued:

1. Create a type-checking library in Rust, which outputs an AST annotated with type information. This would allow leveraging the existing parts of the third version of the compiler (i.e., the Rust implementation under `compiler/`), and extending it to handle the type information. However, the third version of the compiler has other, previously mentioned, architectural issues which may make this option unfavourable, such as the fact that all translation is done directly into strings, rather than into a Rust AST which could then be manipulated further before the final output.
2. Create a new compiler completely from scratch, built around an existing type-checking library. This is probably the most viable option, as utilising an existing type-checking library would allow for more exotic transformations to be performed on the code, due to the type information available. This would also allow a new architecture to be designed, which could avoid the issues which would be faced if continuing with the third version of the compiler.

In terms of architectural decisions, a major improvement would be to use an AST as the intermediate Rust representation, rather than handling strings as the intermediate format. This would allow for better and less brittle output formatting, as well as allowing for contextual transformations to be performed on the code. Additionally, using an AST would allow for more intelligent transformations, such as lifting `use` statements, functions, and constants to module level, or using `Result` in functions not tagged with `@no_except`.

7 Glossary

- AoT (adj., acr.) (also AOT): *A*head-*o*f-*t*ime (AoT compiler, AoT compilation)
  - A compilation model where all transformations occur prior to running the code. This is the traditional compilation model, used by languages such as C or Rust, and is the alternative to JIT.
- AST (n., acr.): *A*bstract *s*yntax *t*ree
  - A representation of source code which is easy to manipulate. It is often used as an intermediate representation of code in compilers as it represents the ideas written in the code without the complicating details such as whitespace.
    For example, the following pseudocode:
    ```
    func foo(bar) do
        return bar + 1
    end
    ```
    Could be represented with the following AST (in Lisp syntax):
    ```
    (function-definition
        (name "foo")
        (parameters
            (name "bar"))
        (body
            (return-statement
                (+ (name "bar")
                    (constant 1)))))
    ```
- Cargo, `cargo` (n.): The standard Rust package manager and build tool
  - Cargo is the standard tool used to generate, build, and test Rust projects. It is also used to manage dependencies, and is the primary way of distributing Rust projects.
- CFG (n., acr.): *C*ontext-*f*ree *g*rammar
  - A style of grammar which is used to describe the syntax of a language. CFGs are widely used, and there are many tools available for working with them.
  - See also: PEG.
- Compiler (n.): A program which takes as input code in one form, and returns as output code in another form.
  - Typically, this is translating code from a high-level language into a lower-level language - such as transforming C or Rust into machine code - but it can also be translating code between two high-level languages - such as transforming TypeScript into JavaScript, or (in this case) transforming Python into Rust.
  - See also: transpiler.
- CPython (n.): The official Python interpreter
  - https://www.python.org/
  - The reference implementation of Python, and the most widely used. As its name suggests, it is written in C.
  - Not to be confused with Cython, a separate project which compiles a superset of Python into C equivalent to the code that would be run by CPython.

- crates.io (n.): The official Rust package registry
  - https://crates.io/
  - Rust projects are distributed as crates, which can be published to crates.io. The `cargo` tool can then be used to install these crates, either globally (installing their associated binaries) or locally (as a dependency of the current project).
  - See also: docs.rs, PyPI.
- docs.rs (n.): The official Rust documentation host
  - https://docs.rs/
  - All crates published to crates.io have their documentation built automatically, and hosted on docs.rs. This allows easy access to documentation for Rust projects, without having to install the project locally, or needing to visit arbitrary third-party websites. Additionally, Rust's standard library documentation is also hosted on docs.rs, meaning that it is the single point of reference for all Rust documentation.
  - See also: crates.io.
- Exotic transformation (n.): A transformation which significantly changes the structure of the code.
  - Exotic transformations are transformations or optimisations made by a compiler which significantly change the structure of the code. The most extreme examples of exotic transformations are found in functional languages, where lazy evaluation and immutability can be used to transform the source code into almost-unrecognisable object code.
- IR (n., acr.): *I*ntermediate *r*epresentation
  - A representation of code which is easy to manipulate, but is not tied to the syntax of the input language. It is often used in compilers which output machine language as it represents the operations written in the code in a way which is more amenable to exotic transformations.
  - See also: AST.
- JIT (adj., acr.): *J*ust-*i*n-*t*ime
  - A compilation model where transformations and optimisations occur during runtime. Popularised by Self and Java, this is the compilation model used by most modern JavaScript engines, and is the alternative to AoT.

- LALR(k) (adj., acr.): *L*ook-*a*head (k tokens), *l*eft-to-right, *r*ight-most derivation (of a grammar or parser)
  - A style of grammar which is used to describe the syntax of a language. LALR(k) grammars are a subset of LR(k) grammars, and are used by tools such as Yacc to transform a token stream into an AST.
  - LALR(k) grammars, while they would first appear to be equivalent to LR(k) grammars, are actually based on LR(0) grammars - LALR(k) can also be written as LA(k)LR(0). The advantage LALR(k) parsers have over LR(k) parsers is that they are more memory-efficient for languages that are LALR(k).
  - Look-ahead: The number of future tokens the parser may inspect before choosing a production to apply to the current token. This is necessary to disambiguate between different productions which may be valid at a given point in the parse.
  - Left-to-right: The parser starts at the beginning of the stream, and travels only towards the end.
  - Right-most derivation: The parser/generator always attempts to resolve the right-most non-terminal first.
  - See also: CFG, LL(k), LR(k).
- Linter (n.): A program which analyses source code for potential issues
  - Linters are typically used to enforce a coding style, or to catch common errors before they become a problem. They are often used in conjunction with the compiler, typically integrated into the developer's editor to provide real-time feedback, but can also be run as a separate step in the build process.
  - Linters take their name from the idea of finding small presentational issues in the code, metaphorical pieces of lint which need to be removed, but have since evolved to also act as real-time verifiers.
- LL(k) (adj., acr.): *L*eft-to-right, *l*eft-most derivation, with k tokens of lookahead (of a grammar or parser)
  - A style of grammar or parser which is used to describe the syntax of a language. LL(k) grammars are a subset of CFGs.
  - According to Waite and Goos (1984), LL(k) grammars were introduced by Stearns and Lewis (1969).
  - Left-to-right: The parser starts at the beginning of the stream, and travels only towards the end.
  - Left-most derivation: The parser/generator always attempts to resolve the left-most non-terminal first.
  - Look-ahead: The number of future tokens the parser may inspect before choosing a production to apply to the current token. This is necessary to disambiguate between different productions which may be valid at a given point in the parse.
  - See also: CFG, LALR(k), LR(k).

- LR(k) (adj., acr.): *L*eft-to-right, *r*ight-most derivation, with k tokens of lookahead (of a grammar or parser)
  - A style of grammar or parser which is used to describe the syntax of a language. LR(k) grammars are a subset of CFGs.
  - Left-to-right: The parser starts at the beginning of the stream, and travels only towards the end.
  - Right-most derivation: The parser/generator always attempts to resolve the right-most non-terminal first.
  - Look-ahead: The number of future tokens the parser may inspect before choosing a production to apply to the current token. This is necessary to disambiguate between different productions which may be valid at a given point in the parse.
  - See also: CFG, LALR(k), LL(k).
- Monkey-patch (n.): The result of monkey-patching
- Monkey-patch (v.): To modify a global namespace, typically to create additional global functions or data
  - This is generally considered bad practice in most languages, as doing so can lead to conflicts between different libraries which both monkey-patch the same namespace.
  - It is often used in languages like JavaScript, where poly-fills use monkey-patching to (conditionally) update the global namespace to add support for features from newer specifications into older run-times.
- MSRV (n., acr.): Minimum Supported Rust Version
  - The oldest version of the Rust compiler supported by a project
- PEG (n., acr.): Parsing Expression Grammar
  - A style of grammar which is used to describe the syntax of a language. Less common than CFGs, but more powerful and easier to write.
  - See also: CFG.
- Poly-fill (n.): Code which poly-fills a specific feature
- Poly-fill (v.): To provide an implementation of a feature which is not natively supported by (this version of) the platform
  - The most common occurrence of this is in web development, where poly-fills can be used to provide support for newer features and APIs when running in older browsers.
  - Poly-fills refer specifically to monkey-patching the namespace being poly-filled, rather than providing a separate namespace for the feature, which will be implemented either by delegating to the native implementation (if available) or by providing a custom implementation (if not) - this alternative method is known as a *pony-fill*.
- PyPI (n., abbr.): Python Package Index
  - https://pypi.org/
  - The official Python package registry. It is used by the `pip` tool, and all related tools, to install Python packages.
  - See also: crates.io.
- RAD (n., acr.): Rapid Application Development
  - A software development methodology which focuses on rapid prototyping and iteration.
- REPL (n., acr.): *R*ead-*e*val-*p*rint *l*oop
  - A program which prompts the user to enter code, which is then

evaluated and (if applicable) the result is printed by the run-time. This is often used for rapid ephemeral prototyping, as it allows the programmer to see the result of their code immediately.

- Transpiler (n.): A term to refer to a compiler which translates specifically from a high-level language to another high-level language
  - See also: compiler.
- Vendor, vendoring (v.): Including a copy of a library in the source code of a project
  - This is often done to ensure that the project will always be able to build, even if the library is no longer available, or if the library is updated in a way which breaks compatibility with the project.
  - It is also necessary if the library needs to be patched, or if the library cannot be installed through the standard package manager (for example, when building the package manager itself).

Appendix A: A grammar for parsing dice expressions

```
// SOI and EOI refer to Start and End Of Input, respectively.
// They are necessary to enforce that `pest` recognises the
↪  entire
// string, rather than simply stopping on locating a syntax
↪  error.
// A brief overview of `pest`'s syntax:
// - Rules are defined with the syntax `RuleName = { ... }`
// - Rules can be concatenated with the `~` operator
// - Rules can be repeated with the `*` (zero-or-more) or `+`
↪  (one-or-more)
//   operators
// - Rules can be made optional with the `?` (one-or-zero)
↪  operator
// - Subexpressions can be grouped with brackets
// - Different rules can be matched with the `|` (ordered choice)
↪  operator
// - Literal strings can be matched by placing them in double
↪  quotes
Expression  = { SOI ~ Expr ~ EOI }
Expr        = { MinMaxTerm ~ ((Gt | Lt | Ge | Le | Eq) ~
↪  MinMaxTerm)* }
CompOp      = { ">" | "<" | ">=" | "<=" | "=" }
MinMaxTerm  = { AddTerm ~ (MinMaxOp ~ AddTerm)* }
MinMaxOp    = { "v" | "^" }
AddTerm     = { AddOp? ~ MulTerm ~ (AddOp ~ MulTerm)* }
AddOp       = { "+" | "-" }
MulTerm     = { Repeat ~ (MulOp ~ Repeat)* }
// `//` and `/v` are both equivalently floor division;
// `/` is true division and `/^` is ceiling division
MulOp  = { "*" | "/" | "//" | "/v" | "/^" | "%" }
Repeat = { Atom ~ (Keep? ~ Atom)* }
Keep   = { "k" ~ LowHigh ~ Atom }
// parse low first, because high will always match due to the ?
↪  operator
LowHigh = { "l" | "h"? }
// _ makes a rule 'silent'; its inner rules are parsed as normal,
↪  but
// the rule itself does not appear in the parse result
Atom         = _{ Number | "(" ~ Expr ~ ")" | DieRoll |
↪  FunctionCall }
FunctionCall =  { FunctionName ~ "(" ~ Expr ~ ("," ~ Expr)* ~
↪  ","? ~ ")" }
FunctionName =  {
    "min"
  | "max"
  | "avg"
```

```
  | "equal"
  | "floor"
  | "ceil"
  | "round"
  | "trunc"
  | "sum"
}
// @ makes a rule 'atomic'; its inner rules are parsed without
↪  handling
// WHITESPACE and all parsed content is saved only as the string
↪  value
// for this rule. ASCII_DIGIT is a built-in rule which matches
↪  any of the
// characters 0-9
Number = @{ ASCII_DIGIT+ ~ ("." ~ ASCII_DIGIT+)? }
// ^ before a string makes that string match case-insensitively
// in other words, d6 and D6 are equivalent
DieRoll = { ^"d" ~ Atom }

// a magic rule which is automatically parsed and ignored
↪  wherever a ~
// is used within a (non-atomic) rule
WHITESPACE = _{ " " }
```

Appendix B: Grammar used in the second version of the compiler

```
Program       = _{
    SOI ~ BlankLine* ~ (PEEK_ALL ~ JoinedStatements ~
↪  BlankLine*)* ~ EOI
}
Statement     = _{
    Return
  | Assign
  | AugAssign
  | AnnAssign
  | For
  | If
  | Import
  | ImportFrom
  | StatementExpr
  | Pass
}
StatementExpr =  { Expr }
Return        =  { "return" ~ (" "+ ~ Expr)? }
Assign        =  { Expr ~ " "* ~ "=" ~ " "* ~ Expr }
AugAssign     =  { Expr ~ " "* ~ AugOp ~ " "* ~ Expr }
AnnAssign     =  {
    Expr ~ " "* ~ ":" ~ " "* ~ Expr ~ (" "* ~ "=" ~ " "* ~ Expr)?
}
Import        =  { "import" ~ " "+ ~ (Path ~ ("," ~ Path)*) }
ImportFrom    =  {
    "from" ~ " "+ ~ Path ~ " "+ ~ "import" ~ " "+ ~ Name ~ " "* ~
↪  (
        "," ~ " "* ~ Name
    )*
}
Pass          =  { "pass" }
For           =  {
    "for" ~ " "+ ~ Expr ~ " "+ ~ "in" ~ " "+ ~ Expr ~ " "* ~ ":"
↪  ~ " "* ~ (
        Statement | NEWLINE ~ Block
    )
}
If            =  {
    "if" ~ " "+ ~ Expr ~ " "* ~ ":" ~ " "* ~ (
        Statement | NEWLINE ~ Block
    ) ~ Elif* ~ Else?
}
Elif          =  {
    "elif" ~ " "+ ~ Expr ~ " "* ~ ":" ~ " "* ~ (Statement |
↪  NEWLINE ~ Block)
}
```

```
Else          =  { "else" ~ " "* ~ ":" ~ " "* ~ (Statement |
↪  NEWLINE ~ Block) }

Path = { Name ~ ("." ~ Name)* }

Atom        = _{ BracketExpr | UnaryExpr | Constant | Name | List
↪  }
BracketExpr =  { "(" ~ Expr ~ ")" }
UnaryExpr   = _{ UnaryOp ~ Atom }
CallExpr    = _{ Atom ~ Call* }
Call        =  { ("(" ~ (Expr ~ ("," ~ Expr)*)? ~ ","? ~ ")") }
BinOpExpr   = _{ CallExpr ~ (BinOp ~ CallExpr)* }
Expr        =  { BinOpExpr ~ (BoolOp ~ BinOpExpr)* }

List         =  { "[" ~ (Expr ~ ("," ~ Expr)*)? ~ ","? ~ "]" }
Constant     = _{ Number | String | True | False | None }
True         =  { "True" }
False        =  { "False" }
None         =  { "None" }
Number       = _{ DECIMAL | HEX | OCTAL | BINARY }
DECIMAL      = @{
    DECIMAL_NUMBER ~ ("_"? ~ DECIMAL_NUMBER)* ~ (
        "." ~ DECIMAL_NUMBER ~ ("_"? ~ DECIMAL_NUMBER)*
    )? ~ (
        ("e" | "E") ~ ("+" | "-")? ~ DECIMAL_NUMBER ~ ("_"? ~
↪  DECIMAL_NUMBER)*
    )?
}
HEX          = @{ "0x" ~ HEX_NUMBER ~ ("_"? ~ HEX_NUMBER)* }
HEX_NUMBER   = _{ DECIMAL_NUMBER | 'a'..'f' | 'A'..'F' }
OCTAL        = @{ "0o" ~ ASCII_OCT_DIGIT ~ ("_"? ~
↪  ASCII_OCT_DIGIT)* }
BINARY       = @{ "0b" ~ ("0" | "1") ~ ("_"? ~ ("0" | "1"))* }
String       =  {
    PUSH("'") ~ StringChar* ~ POP
  | PUSH("\"") ~ StringChar* ~ POP
  | PUSH("'''") ~ StringChar* ~ POP
  | PUSH("\"\"\"") ~ StringChar* ~ POP
}
StringChar   =  { !"\\" ~ !PEEK ~ ANY | StringEscape }
StringEscape =  { "\\" ~ ("\\" | "r" | "n" | "t" | "b" | "'" |
↪  "\"" | "0") }

Block = _{
    NEWLINE* ~ PEEK_ALL ~ PUSH(" "+) ~ JoinedStatements ~ (
        BlankLine* ~ PEEK_ALL ~ JoinedStatements ~ (";" |
↪  NEWLINE) ~ NEWLINE*
    )* ~ POP
}
```

```
BinOp   = { "+" | "-" | "*" | "/" | "%" | "&" | "|" | "^" | ">>"
↪  | "<<" }
BoolOp  = { "and" | "or" }
UnaryOp = { "+" | "-" | "~" | "not" }
AugOp   = {
    "+=" | "-=" | "*=" | "/=" | "%=" | "&=" | "|=" | "^=" | ">>="
↪  | "<<="
}
Name    = { (ASCII_ALPHA | "_") ~ (ASCII_ALPHANUMERIC | "_")* }

BlankLine        = _{ " "* ~ NEWLINE }
JoinedStatements = _{ (Statement ~ " "* ~ ";" ~ " "*)* ~
↪  Statement ~ NEWLINE }
```

Appendix C: Software and other projects mentioned within this report

Projects are listed in alphabetical order by name.

Project (link)	Licence	Repository
Advanced Slides for Obsidian	MIT	MSzturc/obsidian-advanced-slides
Advent of Code	(custom)	(none)
`ast`	PSF Licence v2	python/cpython/Lib/ast.py
`cargo`	MIT/Apache-2.0	rust-lang/cargo
`clap`	MIT/Apache-2.0	clap-rs/clap
CPython (Python)	PSF Licence v2	python/cpython
Cython	Apache-2.0	cython/cython
`erg_compiler`	MIT/Apache-2.0	erg-lang/erg/crates/erg_compiler
`exiftool`	GPL-1.0	https://sourceforge.net/p/exiftool/code/ci/master/tree/
Figma	(unknown)	(none)
`git`	GPL-2.0	https://git.kernel.org/pub/scm/git/git.git/
Google Chrome	(unknown)	(none)
Obsidian	(custom)	(none)
`pandoc`	GPL-2.0-or-later, optionally BSD-3-Clause (templates only)	jgm/pandoc
`pdfjam`	GPL-2.0	rrthomas/pdfjam
`pdflatex`	Unclear - see `LICENSE.TL`	svn://tug.org/texlive/trunk/Build/source (mirror)
`pdftk-java` (no link)	GPL-2.0-or-later	https://gitlab.com/pdftk-java/pdftk
`pest`	MIT/Apache-2.0	pest-parser/pest
`py2erg`	MIT	mtshiba/pylyzer/crates/py2erg
`pylyzer`	MIT	mtshiba/pylyzer
PyPy	MIT	pypy/pypy
Rust	MIT/Apache-2.0	rust-lang/rust
`rustpython_parser`	MIT	RustPython/Parser
TypeScript	Apache-2.0	microsoft/TypeScript
V (language)	MIT	vlang/v
Vala	LGPL-2.1	https://gitlab.gnome.org/GNOME/vala

Appendix D: Bibliography

Arora, S. and Barak, B. (2009) *Computational complexity: A modern approach*. Cambridge University Press, p. 118.

Balasubramanian, A. *et al.* (2017) 'System Programming in Rust: Beyond Safety', *Proceedings of the 16th Workshop on Hot Topics in Operating Systems* [Preprint]. Available at: https://doi.org/10.1145/3102980.3103006.

Beck, K. *et al.* (2001) 'Agile manifesto'. Online. Available at: https://agilemanifesto.org/.

Behnel, S. *et al.* (2011) 'Cython: The Best of Both Worlds', *Computing in Science & Engineering*, 13(2), pp. 31–39. Available at: https://doi.org/10.1109/MCSE.2010.118.

Bugden, W. and Alahmar, A. (2022) 'Rust: The Programming Language for Safety and Performance'. Available at: https://arxiv.org/abs/2206.05503.

Fenton, S. (2012) 'Compiling vs Transpiling'. Available at: https://www.stevefenton.co.uk/blog/2012/11/compiling-vs-transpiling/.

Ford, B. (2004) 'Parsing expression grammars: A recognition-based syntactic foundation', in N.D. Jones and X. Leroy (eds) *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on principles of programming languages, POPL 2004, venice, italy, january 14-16, 2004*. ACM, pp. 111–122. Available at: https://doi.org/10.1145/964001.964011.

Heijmans, J. (2023) 'Advent of Code (Unofficial) Survey Results'. Available at: https://jeroenheijmans.github.io/advent-of-code-surveys/.

Johnson, S.C. (1978) *Yacc: Yet another compiler-compiler*. Bell Laboratories (Computing science technical report). Available at: https://books.google.co.uk/books?id=FrDeHAAACAAJ.

Loff, B., Moreira, N. and Reis, R. (2020) 'The computational power of parsing expression grammars'. Available at: https://arxiv.org/abs/1902.08272.

Lunnikivi, H., Jylkkä, K. and Hämäläinen, T. (2020) 'Transpiling Python to Rust for Optimized Performance', in A. Orailoglu, M. Jung, and M. Reichenbach (eds) *Embedded Computer Systems: Architectures, Modeling, and Simulation*. Cham: Springer International Publishing, pp. 127–138.

Stack Overflow (2015) 'Stack Overflow Developer Survey 2015'. Available at: https://insights.stackoverflow.com/survey/2015#tech-super.

Stack Overflow (2016) 'Stack Overflow Developer Survey 2016'. Available at: https://insights.stackoverflow.com/survey/2016#technology-most-loved-dreaded-and-wanted.

Stack Overflow (2017) 'Stack Overflow Developer Survey 2017'. Available at: https://insights.stackoverflow.com/survey/2017#most-loved-dreaded-and-wanted.

Stack Overflow (2018) 'Stack Overflow Developer Survey 2018'. Available at: https://insights.stackoverflow.com/survey/2018#most-loved-dreaded-and-wanted.

Stack Overflow (2019) 'Stack Overflow Developer Survey 2019'. Available at: https://insights.stackoverflow.com/survey/2019#most-loved-dreaded-and-wanted.

Stack Overflow (2020) 'Stack Overflow Developer Survey 2020'. Available at: https://insights.stackoverflow.com/survey/2020#most-loved-dreaded-and-wanted.

Stack Overflow (2021) 'Stack Overflow Developer Survey 2021'. Available at: https://insights.stackoverflow.com/survey/2021#technology-most-loved-dreaded-and-wanted.

Stack Overflow (2022) 'Stack Overflow Developer Survey 2022'. Available at: https://survey.stackoverflow.co/2022/#section-most-loved-dreaded-and-wanted-programming-scripting-and-markup-languages.

Stack Overflow (2023) 'Stack Overflow Developer Survey 2023'. Available at: https://survey.stackoverflow.co/2023/#section-admired-and-desired-programming-scripting-and-markup-languages.

Stearns, R.E. and Lewis, P.M. (1969) 'Property grammars and table machines', *Information and Control*, 14(6), pp. 524–549. Available at: https://doi.org/10.1016/S0019-9958(69)90312-X.

Waite, W.M. and Goos, G. (1984) *Compiler construction*. Heidelberg: Springer (Texts and monographs in computer science), pp. 121–127.