

GARTER: A HARMLESS SMALL TO MEDIUM SIZED SNAKE

by

MICHAEL LAYZELL

A thesis submitted to the
Department of Computer Science
in conformity with the requirements for
the degree of Bachelor of Computing

Queen's University
Kingston, Ontario, Canada

March 2016

Copyright © Michael Layzell, 2016

Abstract

Languages used for teaching need to be easy to use, help new programmers find and correct the errors which they make in the program, and are ideally well equipped for developing actual software, to allow the student to explore outside of class with their own projects. In this work, we design and implement a subset with 'training wheels' of the dynamically-typed programming language Python [5] for use in teaching, in order to create a language for learning which better fits those criteria.

Dynamic programming languages such as the Python Programming Language can be powerful tools for experienced developers due to their high productivity potential. Unfortunately, when learning, the error messages which they produce can seem arcane or arbitrary. Garter is a new language, based on the existing dynamic language Python, which aims to produce high quality error messages, and provide a comfortable learning environment for new developers, while exposing them to syntax and semantics which are used in the real world.

Contents

Abstract	i
Contents	ii
Chapter 1: Introduction	1
1.1 Motivation	1
1.2 Problem Overview	3
1.3 Thesis Contributions	4
1.4 Thesis Outline	4
Chapter 2: Background and Related Works	6
Chapter 3: Design	9
Chapter 4: Prototype Implementation	16
Chapter 5: Conclusions and Future Work	18
5.1 Summary of Conclusions	18
5.2 Limitations and Future Work	18
Bibliography	20
Appendix A: Specification	23
A.1 Lexical analysis	23
A.1.1 Logical and Physical Lines	23
A.1.2 Line joining rules	24
A.1.3 Comments	24
A.1.4 Indentation	24
A.1.5 Identifiers	25
A.1.6 Literals	26
A.1.7 Integer Literals	28
A.1.8 Floating Point Literals	28

A.1.9	Operators and Delimiters	28
A.2	Data Model	29
A.2.1	Type Subsumption	29
A.2.2	Types	29
A.2.3	Unwritable Types	32
A.3	Program Start Points	33
A.4	Definitions and Scoping	34
A.4.1	Function Definition	34
A.4.2	Class Definition	35
A.5	Statements	37
A.5.1	Small Statements	37
A.5.2	Compound Statements	40
A.6	Expressions	41
A.6.1	Atoms	41
A.6.2	Primaries	44
A.6.3	Comparison Operators	45
A.6.4	Arithmetic Operators	46
A.6.5	Boolean Operators	48
A.6.6	If Expression	48
A.7	Builtin Functions	49
Appendix B: Grammar		50
Appendix C: Error Glossary		53
C.1	Operator Type Mismatch	53
C.2	Parameter Type Mismatch	54
C.3	Parameter Count Mismatch	54
C.4	No Such Attribute	54
C.5	Attribute Already Defined	55
C.6	Variable Already Defined	55
C.7	Invalid Variable	56
C.8	Incomplete Type	56
C.9	Invalid Typecast Source	56
C.10	Mismatched Branch Types	57
C.11	Not In Loop	57
C.12	Return Outside Function	57
C.13	Invalid Len Argument	57
C.14	Invalid Return Type	58
C.15	Invalid Conditional	58
C.16	Mismatched List Type	58

C.17 Mismatched Dict Type	59
C.18 Invalid Print Line End	59
C.19 Invalid Index Type	59
C.20 Unsupported Index	59
C.21 Unsupported Slice	60
C.22 Invalid Assign Target	60
Appendix D: Example Programs	61
Appendix E: Alternative Implementations	62

Chapter 1

Introduction

1.1 Motivation

Modern software development has moved toward the increased usage of dynamic programming languages. These languages do not contain a complete semantic analysis phase during compilation, and instead handle and report any type errors at run-time, as incorrect code is executed. This serves as an advantage for programmers due to the ability to easily write generic code (as all code is, by default, generic over all compatible types), as well as not have to design a complete data layout before beginning to write code. This allows for projects to be written quickly and easily. In program contexts where the optimizations only available to statically typed languages are not necessary, these can be significant benefits.

Unfortunately, this dynamic execution doesn't only have run-time costs. It also affects the ease of understanding program errors, usually quite negatively. As an example, the same basic programming errors have been introduced into two programs: we will compare the error outputs of two similar incorrect problems, one written in Python, which is a dynamic language, and the other written in C [\[20\]](#), which is a

static language. The python programs are executed with the cpython 2.7.10 [10] interpreter, and the C programs are compiled using clang [1], with the `-Werror` flag enabled.

<pre> 1 # Python 2 print("running") 3 x = 10 4 y = 20 5 print("x+y is", xx + y) </pre>	<pre> 1 /* C */ 2 #include <stdio.h> 3 int main() { 4 printf("running\n"); 5 int x = 10; 6 int y = 20; 7 printf("x+y is %d\n", 8 xx + y); 9 } </pre>
--	--

When the python program on the left is run with cpython, the program begins executing and reaches line 5 before producing a run-time error: `NameError: name 'xx' is not defined`. In contrast, the C program fails to compile, emitting the compile-time error `error: use of undeclared identifier 'xx'`. If the python code was to refer to the incorrectly named code in an infrequent code path, the error could never be caught and reported, even with testing.

<pre> 1 # Python 2 def add(a, b): 3 return a + b 4 5 a = 5 6 b = 10 7 c = "string" 8 9 add(a, c) # oops! </pre>	<pre> 1 /* C */ 2 int add(int a, int b) { 3 return a + b; 4 } 5 6 int main() { 7 int a = 5; 8 int b = 10; 9 char *c = "string"; 10 add(a, c); 11 } </pre>
---	--

These programs, much like the earlier programs, also demonstrate a common error: passing values with incorrect types into a function. In the python code, the error is reported at run-time on line 3 - *within* the function. This function could theoretically be located within a library which a developer is using, meaning that the error is

reported at a code location which is unrelated to the actual programming error. In contrast, The strongly-typed C program reports the error on line 10, which is the call site where the actual mistake occurred.

1.2 Problem Overview

The goal of this thesis was to design, implement, and demonstrate a new teaching programming language: Garter. Garter is intended to be a language which both makes teaching programming concepts to new developers easy, and also provides a smooth path for moving away from the learning environment to implementing and solving real problems. We consider such a language to have to fit the following requirements:

1. The programmer should not be required to use syntax or language features until they understand its meaning, yet should be able to start writing code before they have been taught about syntax.
2. The language should act as a stepping stone to a production programming language, such that eager learners can use what they have learned in class to write more complex programs and springboard into learning a language which is used by professionals in both Scientific Programming and Application Programming.
3. Confusing edge cases should be eliminated, such that students and teachers are able to safely reason about a program without having to spend time studying the language's particulars.
4. The language should prevent incorrect programs from being written, and provide useful explanations as to how to correct the program being run, rather

than breaking in a confusing manner at runtime.

1.3 Thesis Contributions

The main contributions of this thesis are as follows:

- We designed a new teaching programming language, Garter, which aims to bring together the simplicity of other teaching programming languages, like Turing, and the more modern dynamic programming language Python.
- We created a prototype implementation of Garter, based on the cpython runtime, including using the cpython IDE, IDLE, and adapting it to work with Garter code.
- We implemented a series of example programs, in Garter, demonstrating that it can be used to solve simple problems like those solved in classrooms, and showing how to migrate Turing style programs to Garter.

1.4 Thesis Outline

The remainder of this thesis is organized as follows:

Chapter 2, Background: An exploration of the other languages and tooling which exist in this space

Chapter 3, Design: A discussion of the design decisions made for the Garter language to make it fit the 4 requirements.

Chapter 4. Prototype Implementation: An exploration of the prototype implementation of Garter

Chapter 5, Conclusions and Future Work: Where Garter has succeeded and failed, as well as the path forward to a better educational programming language

Chapter 2

Background and Related Works

Garter aims to simplify and restrict the dynamic programming language Python into a learning-friendly dialect. This is done by adding a type system, and restricting the types of programs which can be written to make writing code simpler.

There are other projects which have aimed to add a type system to a dynamic language, with varying levels of success.

The most similar project to this thesis, is the mypy project [3]. Mypy is an optional static type checker for Python, which is designed to enable Python developers with large code bases to take advantage of static typing to make better, more maintainable software. The goal of mypy is to enable as many python structures and patterns to be described with its type system, which means that it possesses a relatively complex system, which includes bidirectional type inference, generics, function types, abstract base classes, multiple inheritance and more.

This thesis proposes a language which is much simpler and smaller than that provided by mypy. The reason for this is that the most important and useful part of a learning language is its error messages. As a type system gets more complex and tries to cover all types of software which a professional programmer would try

to write, it can become unwieldy and produce unclear error messages. This thesis intentionally maintains a small type system footprint such that it can detect errors quickly, and provide useful errors, with suggestions for how the student can improve their software. Mypy also attempts to provide an easy migration path for developers with large existing python code bases: namely it is dynamic by default, and only becomes statically typed when type annotations are added. This is undesirable for a learning language, as we want the safer option to be the default option, otherwise students will accidentally write unsafe code, and not benefit from static typing and improved error detection.

Another similar project is Microsoft's Typescript project [13], [11]. Much like mypy, Typescript aims to extend the ECMAScript language [17] to have optional gradual typing. Many of the same complaints which apply to mypy also apply to Typescript: namely that it's goal is not teaching, and it ends up as a complex system in order to work for real production projects. By making covering all of the capabilities of dynamically typed languages a non-goal for this thesis, we hope to enable better error messages, and a simpler introduction to programming for students learning with it.

The Turing programming language is a teaching programming language which heavily inspired Garter's type system and decisions [18]. It has a simple nominal type system, and aims to provide good error messages. Turing is used throughout Ontario as a teaching language for new students in high school due to its simplicity. Unfortunately, Turing fails in a few areas. Firstly, its syntax no longer feels like the syntax of many modern programming languages. It's more low level than modern

programming languages, not taking advantage of modern programming language features which are popular like Garbage Collection, and can help relieve new developers from thinking of details, and it doesn't follow modern programming conventions of using growable arrays, maps, and the use of objects in order to encapsulate state.

Garter aims to take inspiration from Turing's simplicity, while extending on it with a modern programming lens in order to design a language which helps new programmers write good code, while also making them familiar with Python.

Chapter 3

Design

Made decision to use Python because it is popular in industry.

++ Making Python Type Safe

We want to prevent writing incorrect programs

```
x := 5 s := 'x = ' + x
```

And guide people away from mistakes

```
arr := [1, 2, 3, 4, 5, 6, 7, 8, 9] arr.push(input('Enter a number: '))
```

Simple Type system from languages like pascal. Fitted onto Python. Limits what is allowed (e.g. homogenous arrays). Good thing

Types like arrays and dicts are special, no concept of generic types Too complex and likely to make confusing errors for beginning. Unnecessary syntax which would have to be explained.

++ Objects

Values are objects. int, float, str, bool and functions are immutable, and are passed by value. Other types are passed as a pointer to shared memory.

```
arr := [1, 2, 3] arr2 := arr arr.push(4) print(arr2) [1, 2, 3, 4]
```

int and float numeric types are kept separate to prevent problems with array

indexing

arr[2] OK arr[2.0] Not OK arr[4/2] Not OK arr[4//2] OK

+++ Methods

Methods are just functions as attributes on objects:

class C: def f(self): print('hi') x := C() y : None() = x.f Valid y() prints hi

++ Expressions

Expressions are taken from Python. Eagerly evaluated for simple reasoning. Type annotations unnecessary in expressions.

+++ Literals

supports array [1, 2, 3] object 'a': b numeric 5, 5.5 bool True, False and string 'foo' literals Requires less creation of data for lookups etc. manually with for example init functions.

+++ Function Calls

Performed with standard f(x) syntax functions take fixed number of arguments, with fixed types, and produce fixed type result def f(x : int): print('hi', x) f(5) prints 'hi 5'

Functions are values, and can be passed around def f(): print('hi') x := f x() prints 'hi'

++++ Magic Functions

Some 'function'-like expressions are magic and built in. They don't follow the rules of taking a fixed number of arguments with fixed types:

len(), conversion operators (int(), str(), float()), input()

(SEE ALSO print() magic function-like statement)

+++ Arithmetic

Supports `+` `-` `*` `/` `//` `%` `**` (plus, minus, mult, divide, floor divide, modulus, and exponentiation) on numbers (int/float)

`+` is also supported on arrays and strings - represents array/string concatenation. `+` on arrays produces an new array with a new identity, mutations on new array don't affect old and vice versa.

`+` and `-` are also unary operands on numbers

Fairly standard mathematical operators. Easy to learn

+++ Comparison Operations

`==` is done on value, `'asd' == 'asd'` is always true `['a', 's', 'd'] == ['a', 's', 'd']` is also true, even though have diff identities and mutable *More intuitive idea of equality : has same value*

Exceptions: classes, and functions

```
class Foo:
    x := 5
    a := Foo()
    b := Foo()
    a == b  False
```

```
def a():
    print('hi')
def b():
    print('hi')
a == b  False
```

+++ Attributes + Subscription

Can access attributes of classes defined. Done with `.` operator

```
class Foo:
    x := 5
    a := Foo()
    a.x = 10
print('a.x =', a.x)
```

Also can subscript into arrays/dicts

```
x := 'a'
5 y := [5, 10]
```

```
x['a']  5 y[0]  5 i- 0 indexing is standard in programming languages today
```

++ Statements

+++ Functions

defined with `def`. Can be annotated with types.

Type annotation syntax defined in Python in RFC ref. Used by other project

mypy Re-used in Garter, as it fits well into Python.

```
def foo(x : int, y : str) -> bool: ...
```

functions are defined as objects have their own scope where variables are defined. can refer to enclosing variable scope but cannot assign to it without 'global' or 'non-local' (for nested functions) statement.

Can leave function early with return statement. Return statement is required for function with result to provide the result, on all code paths.

Function declarations are immutable.

+++ Control Flow

for loops go over lists, or ranges, rather than having manual increments Simpler, frequently what you were trying to do

```
for x in range(10): x is 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. Declares x only in scope ...
```

```
for x in [1, 2, 3]: x is 1, 2, 3. Declares x only in scope ...
```

while loop vary basic loop - loops while condition is true

break and continue statements can be placed within loop. Aborts the innermost loop

if statement will check condition. If True, do first branch, else continue onto other branches. Standard construct, powerful and simple to understand. Used everywhere

+++ Assignments Assignments to values are done with `x = 5` Cannot assign to a variable which has not been declared.

Variables are declared with `x := 5` or `x : int = 5`. They are scoped to their function. Cannot shadow a binding from within the same function.

```
def foo(): if cond: x := 5 else: x := 10 return x
```

not OK - x is defined within the if's branches, not at the root

`def foo(): x := 0 if cond: x = 5 else: x = 10 return x` OK - x defined in same scope as return, only assigned to in if statements

`def foo(): x := 0 if cond: x := 5` Oops! This is an error, and catches a problem caused by probably unintentional shadowing

`x := 0 def foo(): x := 10` allowed, as x is in a different function. Doesn't change x

`x := 0 def foo(): global x` global statement is required to change x `x = 10`

A function defined inside another function can see variables defined in the root of the enclosing function, thanks to closures.

`def foo() -λ int(): x := 5 def bar() -λ int: return x return bar foo()()` 5

Nested functions are nice for teaching basic functional style programming which is becoming more popular

Accessing non root values is not OK

`def foo(a : [int]) -λ [int()]: out : [int()] = [] for x in a: def bar() -λ int: return x`
 NOT OK - x is not bound in root of function and thus cannot be seen from enclosed function the value x may not have the same value as you are expecting by the time the function is done (only 1 x value is used, so if this was allowed, all functions would return the last value) `out.push(bar) return out`

With `:=` statement, type is inferred.

This is only syntax addition to python. Declarations have same semantics at runtime as assignment in python. Only meaning is for Garter.

+++ Classes

Classes are custom types. Defined with

`class C: field := initialvalue`

def method(self): self is implicitly available here as the instance of C!

Created by calling the name of the class:

x := C()

C itself is a function and a type name.

Methods and fields are accessed with the . operator

Initial values for fields are evaluated once as the class is defined. Can lead occasionally to confusing results:

class C: arr := [] x := C() x.arr.push(5) !!! Changing the value of arr for all instances of C() which haven't changed their value of arr! y := C() y.arr Contains 5

This is because of the behavior of python. In python normal way to do this would be:

```
class C: def init(self):self.arr=[]
```

This violates requirements to not have weird kooky syntax for classes. Seriously considered making the declaration syntax in classes instead act like *init* *it is already new syntax. However the language*

+++ Expressions

Expressions can also just be written as a statement, and they will be evaluated for their side effects.

++ Programs

A program is just a series of statements. They are executed from top to bottom. It is also possible to interact with a REPL, which evaluates the statements one at a time as they are received.

Functions don't have their bodies typechecked until either: a) their binding's name is referenced (for example, to be called), or b) The end of the current program's input is reached.

This allows mutual recursion, for example:

```
def foo(b: bool): if b: print('b is true') else: print('b is false') bar()  foo is defined
before bar is defined
```

if foo was called here, it would be a lookup error, but it isn't called until after so it's OK

```
def bar(): foo(True)
```

```
foo(False)
```

++ Modules

Only support for including basic modules provided by language like random, and turtle. Eventual plans to expand to allow user or teacher defined modules (SEE Future work). Additional screen module provided for Python to enable basic interaction with terminal windows. Would be nice to have simple graphics API for teaching students. Outside the scope of this thesis (SEE future work). (Hence why copied screen from Turing - existing language for teaching students)

Chapter 4

Prototype Implementation

In order to validate the claims of simplicity and feasibility, a prototype implementation of the Garter language was devised.

Many options were considered for how to implement this prototype, mostly focusing on compiling the resulting program back down to Python logic, while performing the validation. See appendix (Alternative Implementation Strategies)

++ Program Structure

Inserted a phase `cpython (lexer) -i cpython (parser) -i garter (validator) -i cpython (compiler) -i cpython (interpreter) (DIAGRAM)`

The strategy which was selected was to modify the `cpython` interpreter, adding the extra syntax required by the Garter programming language to Python's AST (see language design assignment for info on extra syntax). Validation pass was written. Takes python's pre-compilation AST and typechecks it. Written in python for ease of development (over C)

New compiler entry point for python programs, like the built in `compile` function provided by Python itself. Has same functionality, but also performs typechecking

Modify programs such as IDLE (the Python IDE) to instead use this compile

function rather than the built-in one. Means that code run in IDLE is now valid Garter code.

Also created command line REPL which uses this entry point - acts like the Python executable.

+++ Why not replace python completely

Cannot completely replace the standard compile pathways with Garter ones - python code is used internally in python and would need to be re-written.

++ REPL Support

REPLs require maintaining scope and type information across compilations. This is done by exposing an opaque scope object to calling code of compile function. Can be passed to compile function on a future pass to make the logic run in the same scope as previous scope.

+++ Modifications to REPL mode

Modifications needed to be made to the repl runners such as IDLE in order to ensure that if runtime error killed statement when it is being executed: changes to types in scope are not recorded.

Otherwise, garter validator might think that x has been initialized, when it actually has not yet.

Chapter 5

Conclusions and Future Work

5.1 Summary of Conclusions

Garter is a new teaching programming language, based on the Python Programming Language. It provides the developer with a safe environment in which they can learn to write programs, while familiarising themselves with Python syntax and idioms, making the transition into real programming languages used in production around the world much easier.

5.2 Limitations and Future Work

Garter unfortunately currently has many limitations, mostly due to the short time-frame in which it was designed and implemented. We hope to shore up these problems in a future version of Garter.

1. Garter lacks a module system.

Garter piggybacks on python, but doesn't currently support module loading, outside of a small pre-defined set of libraries, due to the complexity of figuring out

2. Garter lacks a mechanism for exposing Python libraries to Garter code

Occasionally a teacher may want to allow their students to perform actions which are not possible without interacting with other libraries written in Python, such as interacting with a graphics, windowing, high performance math, networking or similar library. Garter currently provides no mechanism for exposing that library other than modifying the distribution directly. Ideally a mechanism for doing this should be integrated into the language proper.

3. Garter lacks a class inheritance structure

To enter and participate in modern programming, you often need to learn the mechanics of OOP, specifically single-inheritance with interfaces. No mechanism was implemented for doing this in Garter (See ??), which means that teaching of OOP concepts will have to occur in a different language. A future Garter may perform the design effort to add support for inheritance.

Bibliography

- [1] "clang" c language family frontend for llvm. <http://clang.llvm.org/>. Accessed: 2015-10-29.
- [2] The julia language. <http://julialang.org/>. Accessed: 2015-10-29.
- [3] mypy - optional static typing for python. <http://mypy-lang.org/>. Accessed: 2015-10-29.
- [4] Pep 3107 – function annotations. <https://www.python.org/dev/peps/pep-3107/>. Accessed: 2015-10-29.
- [5] The python language reference. <https://docs.python.org/3/reference/>. Accessed: 2015-10-29.
- [6] Rascal mpl. <http://www.rascal-mpl.org/>. Accessed: 2015-10-29.
- [7] The spoofax message workbench. <http://metaborg.org/spoofax/>. Accessed: 2015-10-29.
- [8] Stratego program transformation language. <http://strategoxt.org/>. Accessed: 2015-10-29.
- [9] Txl home page. <http://www.txl.ca/>. Accessed: 2015-10-29.

-
- [10] Welcome to python.org. <https://www.python.org/>. Accessed: 2015-10-29.
 - [11] Welcome to typescript. <http://www.typescriptlang.org/>. Accessed: 2015-10-29.
 - [12] Jeff Bezanson, Stefan Karpinski, Viral B Shah, and Alan Edelman. Julia: A fast dynamic language for technical computing. *arXiv preprint arXiv:1209.5145*, 2012.
 - [13] Gavin Bierman, Martn Abadi, and Mads Torgersen. Understanding typescript. In Richard Jones, editor, *ECOOP 2014 Object-Oriented Programming*, volume 8586 of *Lecture Notes in Computer Science*, pages 257–281. Springer Berlin Heidelberg, 2014.
 - [14] James R Cordy, Charles D Halpern-Hamu, and Eric Promislow. Txl: A rapid prototyping system for programming language dialects. *Computer Languages*, 16(1):97–107, 1991.
 - [15] R Kent Dybvig. The scheme programming language. 2009.
 - [16] R Kent Dybvig, Robert Hieb, and Carl Bruggeman. Syntactic abstraction in scheme. *Lisp and symbolic computation*, 5(4):295–326, 1993.
 - [17] ECMA ECMAScript, European Computer Manufacturers Association, et al. Ecma-script language specification, 2011.
 - [18] Richard C Holt and James R Cordy. The turing programming language. *Communications of the ACM*, 31(12):1410–1423, 1988.

-
- [19] Lennart C. L. Kats and Eelco Visser. The Spoofox language workbench. Rules for declarative specification of languages and IDEs. In Martin Rinard, editor, *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, October 17-21, 2010, Reno, NV, USA*, pages 444–463, 2010.
- [20] Brian W Kernighan, Dennis M Ritchie, and Per Ejeklint. *The C programming language*, volume 2. prentice-Hall Englewood Cliffs, 1988.
- [21] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akit and Satoshi Matsuoka, editors, *ECOOP’97 Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer Berlin Heidelberg, 1997.
- [22] Paul Klint, Tijs Van Der Storm, and Jurgen Vinju. Rascal: A domain specific language for source code analysis and manipulation. In *Source Code Analysis and Manipulation, 2009. SCAM’09. Ninth IEEE International Working Conference on*, pages 168–177. IEEE, 2009.

Appendix A

Specification

A.1 Lexical analysis

Garter's syntax and lexical analysis are derived from the syntax and lexical rules from Python. For that reason, many parts of this section are derived directly from the Python 3 Language Reference [5].

Garter programs are described as a series of Unicode code points, and all lexical analysis is performed on the basis of these Unicode code points.

A.1.1 Logical and Physical Lines

A Garter program is formed of logical lines. The end of a logical line is represented by the `NEWLINE` token. A logical line is formed by one or more physical line, joined by explicit or implicit joins.

A physical line is a sequence of unicode code points followed by a newline sequence. This sequence is any of `\r`, `\n`, or `\r\n` where `\r` represents the ASCII Carriage Return (CR) character, and `\n` represents the ASCII Linefeed (LF) character.

A.1.2 Line joining rules

If the last character in a physical line is the backslash character (`\`), and the character is not part of another token (such as a string token), the current physical line is joined with the next physical line, and no `NEWLINE` token is emitted. Lines which are explicitly joined this way may not carry comments

If the end of a physical line is reached while inside a pair of parentheses (`()`), square brackets (`[]`), or curly braces (`{}`), the physical line is implicitly joined with the next line. Lines joined this way may carry comments.

A.1.3 Comments

A comment starts with the hash character (`#`) which is not part of a string literal, and ends at the end of a physical line. Comments are ignored by syntax, and do not cause tokens to be emitted.

A.1.4 Indentation

Leading whitespace at the beginning of a logical line is used to determine the indentation level of the line, which is used for blocks in the Garter programming language.

Whitespace measurements are done on spaces, with tabs being replaced by one to eight spaces, such that the total number of characters is a multiple of 8.

The lexical analysis uses these indentation levels to produce `INDENT` and `DEDENT` tokens, using a stack. The following excerpt from the Python reference explains the algorithm used:

Before the first line of the file is read, a single zero is pushed on the stack; this will never be popped off again. The numbers pushed on the stack will always be

strictly increasing from bottom to top. At the beginning of each logical line, the line's indentation level is compared to the top of the stack. If it is equal, nothing happens. If it is larger, it is pushed on the stack, and one INDENT token is generated. If it is smaller, it *must* be one of the numbers occurring on the stack; all numbers on the stack that are larger are popped off, and for each number popped off a DEDENT token is generated. At the end of the file, a DEDENT token is generated for each number remaining on the stack that is larger than zero.

Whitespace

Whitespace between tokens is ignored, unless it contributes to the indentation rules explained above.

A.1.5 Identifiers

Garter identifiers may be of arbitrary length, and may be formed as follows:

```
1 identifier ::= start continue*
2 start      ::= <'a'-'z', 'A'-'Z', '_'>
3 start      ::= <start, '0'-'9'>
```

Implementations may also support additional unicode characters from outside the ASCII range, following Unicode standard annex UAX-31, like Python.

As an additional restriction, names beginning and ending with two ASCII underscores (`--`), such as `--ADD--`, `--init--`) are reserved for implementation use, and may not be written in a Garter program.

Keywords

Keywords are a set of identifiers which cannot be used as ordinary identifiers. Unlike Identifiers, they do not produce a NAME token, but instead produce a token specific

to the keyword. For any keyword `Foo`, the token generated will be written as `'Foo'` in this document.

Some keywords in `garter` are included for compatibility with the Python programming language, while others are keywords used only within the `Garter` programming language.

The following are the set of keywords in `Garter`:

1 <code>False</code>	<code>class</code>	<code>finally</code>	<code>is</code>	<code>return</code>
2 <code>None</code>	<code>continue</code>	<code>for</code>	<code>lambda</code>	<code>try</code>
3 <code>True</code>	<code>def</code>	<code>from</code>	<code>nonlocal</code>	<code>while</code>
4 <code>and</code>	<code>del</code>	<code>global</code>	<code>not</code>	<code>with</code>
5 <code>as</code>	<code>elif</code>	<code>if</code>	<code>or</code>	<code>yield</code>
6 <code>assert</code>	<code>else</code>	<code>import</code>	<code>pass</code>	<code>print</code>
7 <code>break</code>	<code>except</code>	<code>in</code>	<code>raise</code>	<code>range</code>
8 <code>int</code>	<code>float</code>	<code>bool</code>	<code>str</code>	<code>len</code>

A.1.6 Literals

String Literals

String literals takes the following form

```

1 stringliteral: (shortstring | longstring)
2 shortstring: ("'' shortstringitem* ''' |
3             ''' shortstringitem* ''')
4 longstring: ("''' longstringitem* ''' |
5             """ longstringitem* """)
6 shortstringitem: shortstringchar | stringescapeseq
7 longstringitem: longstringchar | stringescapeseq
8 shortstringchar: <any source character except "\""
9                  or newline or the quote>
10 longstringchar: <any source character except "\"">
11 bytesescapeseq: "\"" <any ASCII character>
```

Escape sequences in string and bytes literals are interpreted according to rules similar to those used by Standard C. The recognized escape sequences are:

Escape Sequence	Meaning
<code>\newline</code>	Backslash and newline ignored
<code>\\</code>	Backslash (\)
<code>\'</code>	Single quote (')
<code>\"</code>	Double quote (")
<code>\a</code>	ASCII Bell (BEL)
<code>\b</code>	ASCII Backspace (BS)
<code>\f</code>	ASCII Formfeed (FF)
<code>\n</code>	ASCII Linefeed (LF)
<code>\r</code>	ASCII Carriage Return (CR)
<code>\t</code>	ASCII Horizontal Tab (TAB)
<code>\v</code>	ASCII Vertical Tab (VT)
<code>\ooo</code>	Character with octal value ooo
<code>\xhh</code>	Character with hex value hh
<code>\N{name}</code>	Character named name in the Unicode database
<code>\uxxxx</code>	Character with 16-bit hex value xxxx
<code>\Uxxxxxxxx</code>	Character with 32-bit hex value xxxxxxxx

All unrecognized escape sequences are left in the string unchanged, including leaving the backslash in the result.

Multiple consecutive string literals delimited by only whitespace are allowed, and will be concatenated into a single combined string literal. This allows writing, for example:

```
1 x := ("string_1" "string_2")
```

Which assigns the value "string 1string 2" to x.

A.1.7 Integer Literals

The following is the lexical definition for integer literals:

```

1 integer      ::= decimalinteger | octinteger |
2               hexinteger | bininteger
3 decimalinteger ::= nonzerodigit digit* | "0"+
4 nonzerodigit ::= "1"..."9"
5 digit        ::= "0"..."9"
6 octinteger   ::= "0" ("o" | "O") octdigit+
7 hexinteger   ::= "0" ("x" | "X") hexdigit+
8 bininteger   ::= "0" ("b" | "B") bindigit+
9 octdigit     ::= "0"..."7"
10 hexdigit    ::= digit | "a"..."f" | "A"..."F"
11 bindigit    ::= "0" | "1"

```

Note that decimal integer literals may not have leading 0 characters. This is to cause c-style octal literals to be an error, as otherwise they would have unexpected behavior.

A.1.8 Floating Point Literals

Floating point literals expand on integer literals to allow the definition of non-integral numbers. The following is the lexical definition:

```

1 floatnumber  ::= pointfloat | exponentfloat
2 pointfloat   ::= [intpart] fraction | intpart "."
3 exponentfloat ::= (intpart | pointfloat) exponent
4 intpart      ::= digit+
5 fraction     ::= "." digit+
6 exponent     ::= ("e" | "E") ["+" | "-"] digit+

```

A.1.9 Operators and Delimiters

The following are additional special operator and delimiter tokens used by Garter:

1 +	−	*	**	/	//	%
2 <	>	<=	>=	==	!=	+=
3 -=	*=	/=	//=	%=	**=	->
4 ,	:	.	;	=	()
5 []	{	}			

A.2 Data Model

Values in garter are objects. Objects have a type, which defines the type's fields, and what other operations which may be performed on it. Objects have identity, and are stored by reference in variables or fields. Assignment doesn't mutate the object inside of the variable/field, but instead replaces the reference. A variable or field is immutable if its value cannot be changed.

A.2.1 Type Subsumption

A type `T` can be said to subsume another type `Q` if a value of type `Q` can be used anywhere that a value of type `T` can be used. For example: an `int` is a more specific type than a `float`, and it would be legal to use a `int` anywhere that a `float` would be required. Thus we can say `float` subsumes `int`. In addition, we can say the inverse, that `int` is a subtype of `float`.

A.2.2 Types

```

1 type: 'int' | 'float' | 'str' | 'bool' | NAME |
2       '{' type ':' type '}' | '[' type ']' |
3       (type | 'None') '(' [typelist] ')',
4 typelist: type (',' type)* [',']
```

Numbers (`int` and `float`)

`float` is an floating point number. An object of this type may assume the value of a double precision floating point value. `float` exposes no attributes. `float` is ordered, and can be compared with the comparison operators `<`, `>`, `<=` and `>=`.

`int` is an integer. An object of this type may assume the value of an arbitrary integer value. `int` exposes no attributes. `int` is a subtype of `float`. `int` is ordered,

and can be compared with the comparison operators `<`, `>`, `<=` and `>=`.

Strings (`str`)

`str` is a unicode string. It may assume the value of an arbitrary length sequence of unicode codepoints. `str` is ordered, and can be compared with the comparison operators `<`, `>`, `<=` and `>=`. `str` exposes the following attributes:

`join` : `str([str])`: The `join` attribute is a function object which concatenates the elements of the argument array, using the implicit self argument as the separator. The `join` attribute is immutable.

Booleans (`bool`)

`bool` is a boolean. It may assume either the value `True` or `False`. `bool` exposes no attributes.

Dictionaries (`{K: V}`)

`{K: V}` is a one-to-many map from values of type `K` to values of type `V`. `K` and `V` may be arbitrary types. `{K: V}` exposes the following attributes. All attributes of `{K: V}` are immutable:

`pop` : `V(K, V)`: If the first parameter is a key in the map, Mutates the map, removing the key from the mapping, and returns the associated value. Otherwise, returns the second parameter.

`setdefault` : `V(K, V)`: If the first parameter is a key in the map, returns its associated value. Otherwise, mutates the map, inserting the key-value pair of the parameters to the map, and returns the second parameter.

get : $V(K, V)$: If the first parameter is a key in the map, returns the associated value. Otherwise, returns the second parameter.

clear : $\text{None}()$: Mutates the map, removing all key-value pairs.

copy : $\{K: V\}()$: Returns a new $\{K: V\}$, with a shallow copy of the key-value pairs from the map.

update : $\text{None}(\{K: V\})$: Mutates the map, adding all key-value pairs from the first parameter, overriding any existing conflicting key-value pairs.

Lists ($[T]$)

$[T]$ is a list of values of type T . An object of this type may assume the value of an arbitrary length list of values of type T . $[T]$ exposes the following attributes. All attributes of $[T]$ are immutable:

append : $\text{None}(T)$: Mutates the list, adding the first parameter to the end of the list.

extend : $\text{None}([T])$: Mutates the list, adding the elements of the first parameter to the end of the list.

insert : $\text{None}(\text{int}, T)$: Mutates the list, adding the second parameter at the index specified by the first parameter, shifting all elements after the insertion point one element.

remove : $\text{None}(T)$: Mutates the list, removing the first element for which `it == arg` (where `it` is the element, and `arg` is the argument) evaluates to `True`.

pop : $T(\text{int})$: Mutates the list, removing the element at the index specified by the first parameter, and returning it. Elements with indexes greater than the first parameter are shifted down one index.

`index : int(T)`: Returns the index of the first element in the list for which `it == arg` (where `it` is the element, and `arg` is the argument) evaluates to `True`.

`count : int(T)`: Returns the number of elements in the list for which `it == arg` (where `it` is the element, and `arg` is the argument) evaluates to `True`.

`reverse : None()`: Mutates the list, reversing the order of its elements.

`sort : None()`: Mutates the list, sorting it in ascending order. This attribute is only available on the list types `[int]`, `[float]`, and `[str]`.

Functions (T(P, ...))

Function objects may have a return type (T), or if `None` is written instead of a type, the function returns no value. In addition, they have an arbitrary-length list of parameter types. Function objects do not expose any attributes.

User-defined Types

Users may define their own types using the `class` statement (Section [A.4.2](#)). These classes have the attributes as defined by their class definition. They are unordered, and can be compared for identity with the equality (`==` and `!=`) operator.

For more details on user-defined types, see Section [A.4.2](#).

A.2.3 Unwritable Types

A type is said to be an unwritable type if it cannot be written using the type syntax. Values with unwritable types may not be assigned to variables or used as the type of fields, parameters, or return values. These types may only be used as the type of a temporary value within an expression. Unwritable types are the types of expressions

such as `None`, `[]` and `{}`.

The expression `None` has the unwritable type `__NoneType__`. This type is subsumed by all class types. Thus, the `None` value can be used in place of any class type due to subsumption rules (Section A.2.1).

The expression `[]` has the unwritable type `__EmptyListType__`. This type is subsumed by all list types. Thus, the `[]` value can be used in place of any list type due to subsumption rules (Section A.2.1). `__EmptyListType__` exposes the same methods as a normal list, except for those which require knowing the element type.

The expression `{}` has the unwritable type `__EmptyDictType__`. This type is subsumed by all dict types. Thus, the `{}` value can be used in place of any dict type due to subsumption rules (Section A.2.1). `__EmptyDictType__` exposes the same methods as a normal dict, except for those which require knowing the key/value types.

All compound list or dictionary types which have keys, values, or item types which are unwritable are also unwritable. They can be subsumed by any type for which all literal types match, and the unwritable keys, values, and/or items are subsumed by the corresponding keys, values, and/or items in the subsuming type.

A.3 Program Start Points

```

1 single_input: (NEWLINE | simple_stmt |
2               compound_stmt NEWLINE |
3               defn NEWLINE)
4 file_input: (NEWLINE | stmt)* ENDMARKER
```

A Garter input may either consist of a self-contained program, consisting of a series of newlines and statements; or a single line of interactive input coming from,

for example, a REPL.

A.4 Definitions and Scoping

```
1 defn: funcdef | classdef | vardef
```

Definitions may occur wherever a statement is permitted. They bind a value to the given name in the current scope. Binding is performed on a per-block level. Shadowing of variables from outside of the current function is permitted, but shadowing within a function is not allowed. Redeclarations are also prohibited.

When referring to a name, first a check is made to see if the variable is local. A variable is local if there is a statement declaring that variable within the current function (whether that variable is currently in scope is ignored). If the name is nonlocal, enclosing scopes are checked for local variables. If a variable with the name is not found, it is a validation time error. Once a variable is found, the variable's validity is checked. A local variable is valid if it has been declared within the current block or a direct parent. A nonlocal variable is valid if it is declared above the enclosing function definition in the root of a function or module. Other references are prohibited as they may no longer be valid when the function is called.

Variables may not be mutated if they are defined using an immutable declaration form. Nonlocal variables may not be mutated, although this may be bypassed with the `varfwd` forms described in [Section A.4.1](#).

A.4.1 Function Definition

```
1 funcdef: ('def' NAME '(' [paramlist] ')') ['->' type] ':'
2         funcbody)
3 paramlist: param (',' param)* [' ','']
4 param: NAME ':' type
5
6 funcbody: (simple_stmt | NEWLINE INDENT varfwd* stmt+ DEDENT)
7 varfwd: globaldef | nonlocaldef
8 globaldef: 'global' NAME (',' NAME)*
```

```
9 nonlocaldef: 'nonlocal' NAME (',' NAME)*
```

A function definition defines a new immutable variable with the given name. It has the type of the corresponding function object, with the return type specified after the `->` token, or `None` if no return type is specified. The parameter types are defined as the types written after the `:` token in each parameter.

For example, the function definition:

```
1 def foo(a: int, b: float, c: str) -> bool:
2     ....
```

Would define an immutable variable of type `bool(int, float, str)`.

When the function is invoked, the statements in the body of the function are executed sequentially.

The variable forward forms, `globaldef` and `nonlocaldef` accept names as arguments. They state that for name resolution rules, the global, or nonlocal (but in another function) variables should be treated as though they are local. This means that they may be mutated, even though they exist in an enclosing nonlocal scope.

Calls to functions are described in more detail in Section [A.6.2](#).

The body of the function is not validated until either the end of the current input's validation phase, (either an interactive command or program file), or the validation of its first reference to the variable within the current input. This means that functions may refer to other functions or variables which have not been declared yet, as long as they are declared before the function is used.

A.4.2 Class Definition

```
1 classdef: 'class' NAME ':' class_body
2 class_body: fielddef | NEWLINE INDENT class_stmt+ DEDENT
3 class_stmt: fielddef | mthddef
4
5 fielddef: NAME ':' [type] '=' expr
```



```
6 mthddef: ( 'def' NAME '(' mparamlist ')' ['->' type] ':' '  
7         funcbody)  
8 mparamlist: NAME ( ',' param)* [ ',' ]
```

A class definition defines a new type to the Garter type system. User-defined classes may not be referred to in a Garter program before their definition.

When the garter class definition for a type `T` is written, an immutable variable of type `T()` is defined with the name `T`. This function can be called to get a new instance of the type `T`. In addition, the initializer statements for the class are run when a class definition is executed as a statement.

If a class is declared within a function, each time that function is run the initializer statements for the class will be re-run, and will be used for new instances of that class within the given scope.

All field and method definitions declare attributes on the class type.

All class types also accept, as a possible value, the value `None`. This represents the absence of a meaningful value. Any attempt to access a field on a `None` value results in a runtime error.

Field Definitions

A field definition defines a new mutable attribute with the written type on the class type. For example, the field definition `x : int = 10` would define a mutable attribute `x` on the class type, with type `int`. Like with variable declarations, the type of the field may be inferred if the type is not written.

Method Definitions

A method definition defines a new mutable attribute with a function type on the class type. The function type will be the same as if the method definition was written as a

function definition (Section A.4.1), except that the first parameter need not be typed, and will be implicitly passed a reference to the instance of the class the function is called on.

Calls are described in more detail in Section A.6.2.

Variable Definition

```
1 vardef: NAME ':' [type] '=' expr
```

A variable definition defines a new mutable variable in the current scope. If the statement is invoked at the global scope, then it defines a new variable in the global scope. If the statement is invoked within the root of a function, it defines a new variable inside of that function's scope. The expression is then evaluated, and assigned to that variable.

If the type is not provided, it is inferred from the type of the expression. This is occasionally not possible, and a type annotation must be provided (such as with the `None` literal).

A.5 Statements

```
1 stmt: simple_stmt | compound_stmt | defn
2 simple_stmt: small_stmt ( ';' small_stmt ) * [ ';' ] NEWLINE
```

Control flow in Garter is controlled by statements. These statements can be categorized into two groups: the single-line small statements, and the multi-line compound statements.

Definitions are also statements, but are described in Section A.4.

A.5.1 Small Statements

```
1 small_stmt: (expr | assign_stmt | pass_stmt |
2             break_stmt | continue_stmt |
3             return_stmt | print_stmt)
```

Expr Statement

The `expr` statement allows for an expression to be invoked for its side effects, discarding the resulting value. Usually this is done with a function or method call. More details on expressions in Section [A.6](#).

Assignment Statement

The basic assignment operator `lhs = rhs` assigns the value in `rhs` to `lhs`.

Compound assignment operators, `+=` `-=` `*=` `/=` `%=` `**=` `//=` perform their operation (`+` `-` `*` `/` `%` `**` `//` respectively), and then assign the result to `lhs`, evaluating `lhs` only once.

The `lhs` of the operator must be an assignable target. This is either a mutable variable, field, array or dictionary index, or array slice. Other expressions are not legal on the left of an assignment operator. If a field or variable is immutable (such as is the case with fields corresponding to methods, and variables corresponding to functions and classes), then it is not a legal assignable target. If a variable is nonlocal, it is also not a legal assignment target. To bypass this, the `nonlocal` and `global` forms may be used (Section [A.4.1](#)).

Pass Statement

```
1 pass_stmt: 'pass'
```

The `pass` statement does nothing. It exists to enable the definition of empty blocks.

Break Statement

```
1 break_stmt: 'break'
```

The `break` statement may only be placed lexically within the `for` or `while` statements. It causes control flow to immediately continue to the next statement after the `for` or `while` statement, not executing the remainder of the current iteration, and ignoring the normal loop exit conditions.

Continue Statement

```
1 continue_stmt: 'continue'
```

The `continue` statement may only be placed lexically within the `for` or `while` statements. It causes control flow to immediately continue to the next iteration of the loop, not executing the remainder of the current iteration.

Return Statement

```
1 return_stmt: 'return' [expr]
```

The `return` statement may only be placed lexically within a function declaration. It causes control flow to immediately exit the current function, not executing the remaining statements in the function. If the `return` statement is passed an expression, then it must have the same type as the return type of the function, and that value is used as the return value of the function. If the `return` statement is not passed an expression, then the function must not have a return type.

Print Statement

```
1 print_stmt: ('print' '(' expr ',' expr)*  
2           [' ','end' '=' expr] [' ',' '])
```

Prints out the result of casting each of the arguments to a `str` to the screen. If this operation would fail, instead prints out a useful debug representation of the object. By default, each argument's representation is separated by an ASCII space character (`' '`), and the `print` statement's output is terminated with an ASCII newline character (`'\n'`).

(`'\n'`). This newline character can be replaced by passing the `end` 'named argument', which will instead terminate the string. The `end` named argument must be a `str`.

A.5.2 Compound Statements

```
1 compound_stmt: (if_stmt | while_stmt | for_stmt)
```

If Statement

```
1 if_stmt: ('if' expr ':' suite ('elif' expr ':' suite)*
2         ['else' ':' suite])
```

The `if` statement is used for conditional execution. It selects exactly one of the suites to execute by evaluating the expressions one by one until one is found to evaluate to the boolean value `True`. If all expressions evaluate to `False`, then the suite of the `else` clause, if present, is executed.

All of the expression arguments must be of type `bool`.

While Statement

```
1 while_stmt: 'while' expr ':' suite
```

The `while` statement is used for repeated execution while an expression evaluates to `True`. It will repeatedly test the expression, and if it is `True`, executes the suite. Whenever the expression evaluates to `False`, the loop terminates.

A `break` statement (Section [A.5.1](#)) executed in the suite terminates the loop immediately. A `continue` statement (Section [A.5.1](#)) executed in the suite skips the rest of the suite and goes back to testing the expression.

For Statement

```
1 for_stmt: ('for' NAME 'in' (expr | range) ':' suite)
2 range: 'range' '(' expr [',' expr [',' expr]] [',' ] ')'
```

The `for` statement is used to iterate over the elements of a list. The expression is evaluated once, and must have a `[T]` type. The suite is then executed once for

each element of the iterator, with the value bound to the name. This name binding is a local declaration, much like an the assignment declaration, however it is not valid outside of the body of the for statement (Section [A.4.2](#).

If the **range** form is used instead of the expression, then all of the expressions must have type **int**. In the 1 argument case, the suite is executed N times, where N is the first argument, with the name bound to each integer in the range $[0, N)$. In the 2 argument case, the suite is executed $M - N$ times, where N is the first argument, and M is the second argument, with the name bound to each integer in the range $[N, M)$. In the 3 argument case, it acts much like the 2 argument case, except the step is I instead of 1, where I is the 3rd argument.

A **break** statement (Section [A.5.1](#)) executed in the suite terminates the loop immediately. A **continue** statement (Section [A.5.1](#)) executed in the suite skips the rest of the suite and proceeds to the next item in the list.

A.6 Expressions

A.6.1 Atoms

```

1 atom: ( '(' expr ')' |
2         '[' [exprlist] ']' |
3         '{' [dictmaker] '}' |
4         'len' '(' expr ')' |
5         ('str' | 'int' | 'float') '(' expr ')' |
6         NAME | INTEGER | FLOATNUMBER | STRING+ |
7         'None' | 'True' | 'False' )
8 exprlist: expr (',' expr)* [' ','\n']
9 dictmaker: expr ':' expr (',' expr ':' expr)* [' ','\n']

```

Atoms are the building blocks of expressions, and are the base cases for the recursive definitions of the other expressions.

Identifiers (names)

See Section [A.1.5](#) for lexical information on identifiers. This identifier evaluates to the value of the variable with the given name in the current scope. The type of this expression is the type of the variable with the given name.

If there is no variable in the current scope with the given name, it is a validation-time error.

Literals

Literals evaluate to a literal of the given type. The **STRING** literal evaluates to a **str** object, The **INTEGER** literal evaluates to a **int** object, the **FLOAT** literal evaluates to a **float** object, the **'True'** literal evaluates to the **bool** value **True**, and the **'False'** literal evaluates to the **bool** value **False**.

The **'None'** literal evaluates to the common **None** value of user-defined types. For more information see Section [A.4.2](#).

Parenthesized Expression

Parenthesized Expressions evaluate to the value of the contained expression, and primarily exist as a mechanism for expression precedence and operation orders.

List literals

```
1 list_literal: '[' [exprlist] ']'
2 exprlist: expr (',' expr)* [',']
```

List literals evaluate their contained expressions in order. All expressions must be of a single type **T**. They evaluate to a **[T]** value with the results of the contained expressions as the list elements.

Dictionary literals

```
1 dict_literal: '{' [dictmaker] '}'  
2 dictmaker: expr ':' expr (',' expr ':' expr)* [',']
```

Dictionary literals evaluate their contained expressions in order. All expressions to the left of the colon must be of a single type **K**, while expressions to the right of the colon must be of a single type **V**. They evaluate to a $\{K:V\}$ value with the results of the expressions to the left of the ':' as the keys, and the ones to the right as values. Duplicate keys produce a runtime error.

Len Expression

```
1 len: 'len' '(' expr ')'
```

If the **expr** is a **str**, produces the length of the string as an **int**. If the **expr** is an **[T]**, produces the number of items in the list. If the **expr** is an $\{K: V\}$, produces the number of key-value pairs in the dictionary. Otherwise, causes a validation time error.

Typecast Expressions

```
1 typecast: ('int' | 'str' | 'float') '(' expr ')'
```

The typecast expressions attempt to convert their argument to their type.

All typecast expressions only accept **int**, **float**, and **str** arguments.

The **str** typecast will convert **int** and **float** objects to their **str** representation. For example, the **int** 5 will be converted to the **str** "5". It will return **str** arguments unchanged.

The **int** typecast will truncate **float** types to their closest integer representation, rounded toward 0. It will also attempt to parse **str** types as an **int**, causing a runtime error if this fails. It will return **int** arguments unchanged.

The `float` typecast will truncate `int` types to their closest floating point representation, which may be less precise than the `int`'s original representation. It will also attempt to parse `str` types as an `float`, causing a runtime error if this fails. It will return `float` arguments unchanged.

A.6.2 Primaries

```
1 primary: fieldref | subscription | call
```

Field Reference

```
1 fieldref: primary '.' NAME
```

A field reference accesses an attribute of the given object. The type of this expression is the type of the attribute on the object. If the object lacks an attribute with the given name, it is a validation-time error.

Call

```
1 call: primary '(' [exprlist] ')'
```

A call calls the function object with the passed arguments. The primary must evaluate to a function object, and the passed arguments must have types which correspond to the argument types of the function object.

The result type is the return type of the function object. If the function object has no return type, then this expression has no result.

This is also used to call methods, as methods are defined as immutable fields with a function object type.

Subscription

```
1 subscription: primary '[' (expr | [expr] ':' [expr]) ']'
```

If the first form of subscription is used, the primary must evaluate to a `[T]` or `{K:V}`. If the primary is a `[T]`, then the `expr` must be a `int`, and the expression will

yield the n th element of the list, where n is the value of the expr. If the primary is a $\{K:V\}$, then the expr must be a K , and the expression will yield the V which is associated with the given key.

If the second form of subscription is used, then the primary must evaluate to a $[T]$, and both expressions, if provided, must be a `int`. The first argument defaults to 0, and the second defaults to the value of `len(primary)`. The expression yields a new list, containing the values in the list starting at the index of the first argument, and ending before the element at the index of the second argument.

Negative integer indexes on $[T]$ are offset from the end of the list, so -1 is the index of the last element in the list.

A.6.3 Comparison Operators

```
1 comparison: arith (comp_op arith)*
2 comp_op: '<' '|' '>' '|' '==' '|' '>=' '|' '<=' '|' '!=' '|' 'in' '|' 'not' '|' 'in'
```

These operators are not parsed as left-associative, unlike the Arithmetic Operators below. Instead, a sequence of comparison operators such as `a < b == c > d` will be resolved like `(a < b)` and `(b == c)` and `(c > d)`, except that each expression will only be evaluated once.

These operators all produce an expression of type `bool`, and will be written with only two operands. Their semantic meaning in sequence is as is written above. Their valid types will be written in place of their operands. As `int` is subsumed by `float`, it may be used whenever a `float` is required.

`float < float`: True if $lhs < rhs$, False otherwise.

`float > float`: True if $lhs > rhs$, False otherwise.

`float <= float`: True if $lhs \leq rhs$, False otherwise.

`float >= float`: True if $lhs \geq rhs$, False otherwise.

`float == float`: True if *lhs* = *rhs*, False otherwise.

`str == str`: True if *lhs* and *rhs* represent the same unicode sequence, False otherwise.

`bool == bool`: True if *lhs* and *rhs* have the same value, False otherwise.

`[T] == [T]`: True if *lhs* and *rhs* have the same number of elements, and for all elements *i*, `lhs[i] == rhs[i]`.

`{K: V} == {K: V}`: True if *lhs* and *rhs* have the same set of keys, and for each key *k*, `lhs[k] == rhs[k]`.

`T == T`: True if *lhs* and *rhs* have the same identity.

`T != T`: True if `T == T` is False, False otherwise.

`T in [T]`: True if there is an element *i* in *rhs* such that `lhs == rhs[i]`. False otherwise.

`K in {K: V}`: True if the key *lhs* is in *rhs*, False otherwise.

`T not in [T]`: False if there is an element *i* in *rhs* such that `lhs == rhs[i]`. True otherwise.

`K not in {K: V}`: False if the key *lhs* is in *rhs*, True otherwise.

A.6.4 Arithmetic Operators

```
1 arith: mult (('+'|'-') mult)*
2 mult: unary (('*'|'/'|'%'|'//') unary)*
3 unary: ('+'|'-') unary | power
4 power: primary ['**' unary]
```

These operators will be written with their valid types in place of their operands. Any unlisted type-operand combinations are validation-time errors. With any operators which accept `float` also accept `int` in place of their operands, unless a more specialized operator is available. For example the `-` binary operator accepts any combination of `int` and `float` operators, and produces an `int` if both operands are `int`,

and a `float` otherwise, but only has two entries on this list, one for `int - int`, and one for `float - float`.

`int + int`: Returns the value $lhs + rhs$ as an `int`.

`float + float`: Returns the value $lhs + rhs$ as a `float`.

`str + str`: Returns a `str` containing the concatenation of the two strings.

`[T] + [T]`: Returns a new `[T]` containing the concatenation of the two lists.

`int - int`: Returns the value $lhs - rhs$ as an `int`.

`float - float`: Returns the value $lhs - rhs$ as a `float`.

`int * int`: Returns the value $lhs \times rhs$ as an `int`.

`float * float`: Returns the value $lhs \times rhs$ as a `float`.

`float / float`: Returns the value $\frac{lhs}{rhs}$ as a `float`.

`int % int`: Returns the remainder from the division $\frac{lhs}{rhs}$ as an `int`. The sign of this remainder will match the sign of the second operand.

`float % float`: Returns the remainder from the division $\frac{lhs}{rhs}$ as a `float`. The sign of this remainder will match the sign of the second operand.

`int // int`: Returns the value $\frac{lhs}{rhs}$, rounded down to the nearest integer as an `int`.

`float // float`: Returns the value $\frac{lhs}{rhs}$, rounded down to the nearest integer as a `float`.

`float ** float`: Returns the value lhs^{rhs} , as a `float`.

`+ int`: Returns its operand unchanged.

`+ float`: Returns its operand unchanged.

`- int`: Returns the negation of its operand.

`- float`: Returns the negation of its operand.

Unary Operators

+: If the operand is `int` or `float`, returns it unmodified. Other operand types cause a validation time error.

-: If the operand is `int` or `float`, returns its negation with the same type. Other operand types cause a validation time error.

A.6.5 Boolean Operators

```
1 or_expr: and_expr ( 'or' and_expr ) *  
2 and_expr: not_expr ( 'and' not_expr ) *  
3 not_expr: 'not' not_expr | comparison
```

Boolean Operators, unlike the other operators, do not necessarily evaluate all of their operands, they may short-circuit their execution if the value of their left operand is sufficient to compute the result.

All Boolean operators require all operands to have type `bool`.

The **or** operator evaluates its left operand. If it produces the boolean value `True`, the result is `True`. Otherwise, it evaluates its right operand, and produces its result.

The **and** operator evaluates its left operand. If it produces the boolean value `False`, the result is `False`. Otherwise, it evaluates its right operand, and produces its result.

The **not** operator evaluates its operand. If it produces the boolean value `False`, the result is `True`. Otherwise, the result is `False`.

A.6.6 If Expression

```
1 expr: or_expr [ 'if' or_expr 'else' expr ]
```

The first and third operands must have a common type `T`. The second operand must have type `bool`.

The `if` expression evaluates its second operand. If it produces the boolean value `True`, the first operand is evaluated, and its result is produced. Otherwise, the third operand is evaluated, and its result is produced.

A.7 Builtin Functions

Garter defines only one function in the global scope at the start of the program:

`input : str()`: Reads in a line of textual input from the user, producing a `str` containing the read-in text.

`ord : int(str)`: Returns the ordinal number for the passed-in character.

Appendix B

Grammar

```

1 # Grammar for Garter
2
3 # Start symbols for the grammar:
4 #     single_input is a single interactive statement;
5 #     file_input is a program or sequence of commands
6 #     read from an input file;
7
8 # Start Symbols #
9 single_input: (NEWLINE | simple_stmt |
10               compound_stmt NEWLINE |
11               defn NEWLINE)
12 file_input: (NEWLINE | stmt)* ENDMARKER
13
14 # Types #
15 type: 'int' | 'float' | 'str' | 'bool' | NAME |
16       '{' type ':' type '}' | '[' type ']' |
17       (type | 'None') '(' [typelist] ')'
18 typelist: type (',' type)* [',' ]
19
20 # Statements #
21 stmt: simple_stmt | compound_stmt | defn
22 simple_stmt: small_stmt (',' small_stmt)* [',' ] NEWLINE
23
24 small_stmt: (expr | assign_stmt | pass_stmt |
25             break_stmt | continue_stmt |
26             return_stmt | print_stmt)
27 assign_stmt: expr assignop expr
28 assignop: ('=' | '+=' | '-=' | '*=' |
29           '/=' | '%=' | '**=' | '//=')
30 pass_stmt: 'pass'
31 break_stmt: 'break'
32 continue_stmt: 'continue'
33 return_stmt: 'return' [expr]

```

```

34 print_stmt: ('print' '(' expr ',' expr)*
35             [',' 'end' '=' expr] [',' ''])
36
37 compound_stmt: (if_stmt | while_stmt | for_stmt)
38 if_stmt: ('if' expr ':' suite ('elif' expr ':' suite)*
39          ['else' ':' suite])
40 while_stmt: 'while' expr ':' suite
41 for_stmt: ('for' NAME 'in' (expr | range) ':' suite)
42 range: 'range' '(' expr [',' expr [',' expr]] [',' ''])
43 suite: simple_stmt | NEWLINE INDENT stmt+ DEDENT
44
45 # Definitions #
46 defn: funcdef | classdef | vardef
47
48 # Variable Definition #
49 vardef: NAME ':' [type] '=' expr
50
51 # Function Definition #
52 funcdef: ('def' NAME '(' [paramlist] ')') ['->' type] ':'
53         funcbody)
54 paramlist: param '(' ',' param)* [',' '']
55 param: NAME ':' type
56
57 funcbody: (simple_stmt | NEWLINE INDENT varfwd* stmt+ DEDENT)
58 varfwd: globaldef | nonlocaldef
59 globaldef: 'global' NAME '(' ',' NAME)*
60 nonlocaldef: 'nonlocal' NAME '(' ',' NAME)*
61
62 # Class Definition #
63 classdef: 'class' NAME ':' class_body
64 class_body: fielddef | NEWLINE INDENT class_stmt+ DEDENT
65 class_stmt: fielddef | mthddef
66
67 fielddef: NAME ':' [type] '=' expr
68 mthddef: ('def' NAME '(' mparamlist ')') ['->' type] ':'
69         funcbody)
70 mparamlist: NAME '(' ',' param)* [',' '']
71
72 # Expressions #
73 expr: or_expr ['if' or_expr 'else' expr]
74 or_expr: and_expr ('or' and_expr)*
75 and_expr: not_expr ('and' not_expr)*
76 not_expr: 'not' not_expr | comparison
77 comparison: arith (comp_op arith)*
78 comp_op: '<' | '>' | '==' | '>=' | '<=' | '!=' | 'in' | 'not in'
79 arith: mult (('+' | '-') mult)*
80 mult: unary (('*' | '/' | '%' | '//') unary)*
81 unary: ('+' | '-') unary | power
82 power: primary ['**' unary]
83 primary: fieldref | subscription | call
84 fieldref: primary '.' NAME

```



```
85 subscription: primary '[' (expr | [expr] ':' [expr]) ']'
86 call: primary '(' [exprlist] ')'
87 atom: ((' expr ') |
88        '[' [exprlist] ']' |
89        '{' [dictmaker] '}' |
90        NAME | INTEGER | FLOATNUMBER | STRING+ |
91        'None' | 'True' | 'False')
92 exprlist: expr (',' expr)* [',' ]
93 dictmaker: expr ':' expr (',' expr ':' expr)* [',' ]
```

Appendix C

Error Glossary

The error messages are an important part of Garter. Garter programs will be written by inexperienced programmers, and the error messages which occur when they write incorrect programs will have to be useful and provide a mechanism for correcting their errors.

Sometimes, an error will be connected with 'notes', which are intended to provide additional information for the coder, directing them towards advice. For this reason, the error messages will occasionally be long, spanning multiple lines. To avoid scaring away developers with pages of errors, only one error should be shown at a time.

Only validation-time errors are described here.

C.1 Operator Type Mismatch

```

1 Line X, Column X
2     ....
3     ^
4 OperatorTypeMismatch:
5 The operands to operator '+' (int + str), are invalid.
```

This error appears when a user attempts to use a unary or binary operator on two types which are not supported.

If the operator is '+', and at least one of the operands is **str**, then we can predict that the user meant to perform string concatenation, and provide a fixit note.

```
1 NOTE: Can only concatenate str and str. Consider casting the left
   argument
2 to a str for concatenation with 'str(...) '.
```

C.2 Parameter Type Mismatch

```
1 Line X, Column X
2     ....
3     ^
4 ParameterTypeMismatch:
5 The 3rd parameter to this function is invalid. Expected int, instead
   found bool.
6
7 NOTE: Function expects parameters (int, int, float)
```

This error appears when a user attempts to call a function, either internal or user-defined, with the incorrect type parameter.

The expected parameters are listed to help the user correct their logic quickly.

C.3 Parameter Count Mismatch

```
1 Line X, Column X
2     ....
3     ^
4 ParameterCountMismatch:
5 This function expects 3 parameters, instead found 4 parameters.
6
7 NOTE: Function expects parameters (int, int, float)
```

This error appears when a user attempts to call a function, either internal or user-defined, with the incorrect number of arguments.

The expected parameters are listed to help the user correct their logic quickly.

C.4 No Such Attribute

```
1 Line X, Column X
2     ....
3     ^
4 NoSuchAttribute:
```

```
5 This object of type 'str' has no attribute named 'jion'.
6
7 NOTE: 'str' has attributes: 'join'.
```

This error appears when a user attempts to access an attribute of an object which is not available. The list of available attributes is listed to help them find the attribute they meant to access.

C.5 Attribute Already Defined

```
1 Line X, Column X
2     ....
3     ^
4 AttributeError:
5 This class has a duplicate definition of the attribute 'foo'.
6
7 NOTE: The previous definition of this attribute is here:
8 Line Y, Column Y
9     ....
10    ^
```

This error occurs when a user attempts to define an attribute on an object which has previously been defined. It identifies the location of the previous definition, so that the user can quickly correct the offender.

C.6 Variable Already Defined

```
1 Line X, Column X
2     ....
3     ^
4 VariableAlreadyDefined:
5 The variable 'foo' has already been defined in this scope.
6
7 NOTE: The previous definition of this variable is here:
8 Line Y, Column Y
9     ....
10    ^
```

This error occurs when a user attempts to define a variable in a scope where it has previously been defined. It identifies the location of the previous definition, so

that the user can quickly correct the offender.

C.7 Invalid Variable

```

1 Line X, Column X
2     ....
3     ^
4 InvalidVariable:
5 The variable 'foo' may not have a valid value when it is accessed here.
```

This error occurs when a user attempts to access a variable which may not be valid when it is accessed. This might happen if, for example, a function is declared in a loop, and the function attempts to access the iterator value.

C.8 Incomplete Type

```

1 Line X, Column X
2     ....
3     ^
4 IncompleteType:
5 Unable to infer the type of this value. Try annotating this declaration
  with the desired type.
```

This error occurs when the user attempts to assign an unwritable type to a variable or field. For example, the declaration `x := []`, for which the item type of the expression cannot be determined.

If it is a simple error like `x := []`, `x := {}`, or `x := None` then we also add one of these notes:

```

1 NOTE: Annotate this declaration to specify the item type of the list
2 NOTE: Annotate this declaration to specify the key and value types of
  the dict
3 NOTE: Annotate this declaration to specify the class type
```

C.9 Invalid Typecast Source

```

1 Line X, Column X
2     ....
3     ^
4 InvalidTypecastSource:
5 Cannot cast to type 'float' from type 'bool'.
```

This error occurs when a user attempts to typecast from an invalid type. It identifies both of the types.

C.10 Mismatched Branch Types

```
1 Line X, Column X
2     ....
3     ^
4 MismatchedBranchTypes:
5 The types of the branches of the if expression must match.
6 Instead found 'int' and 'bool'
```

This error occurs when an if expression with mismatched types on the then and else branches is written.

C.11 Not In Loop

```
1 Line X, Column X
2     ....
3     ^
4 NotInLoop:
5 The 'break' and 'continue' statements may only be written in loops.
```

This error occurs when the user writes a **break** or **continue** statement outside of a loop.

C.12 Return Outside Function

```
1 Line X, Column X
2     ....
3     ^
4 ReturnOutsideFunction:
5 The 'return' statement may only be written within functions.
```

This error occurs when the user writes a **return** statement outside of a function or method.

C.13 Invalid Len Argument

```
1 Line X, Column X
2     ....
3     ^
```

```

4 InvalidLenArgument:
5 The len expression only accepts lists , dicts , and strs . Instead found a
  'bool' .

```

This error occurs when an invalid argument is passed to the len magic function.

C.14 Invalid Return Type

```

1 Line X, Column X
2     ....
3     ^
4 InvalidReturnType:
5 This function must return type 'bool' , instead found a 'str' .

```

This error occurs when a returns statement in a function has an invalid type.

If the function shouldn't return any type, the error will instead be:

```

1 This function doesn't return a type , instead found a 'str' .

```

If the function should return a type but there is not one written

```

1 This function must return type 'bool' , instead found an argument-less
  return statement

```

C.15 Invalid Conditional

```

1 Line X, Column X
2     ....
3     ^
4 InvalidConditional:
5 The type of the conditional in an if or while must be 'bool' . Instead
  found 'int'

```

This error occurs when a non-bool value is used as the conditional in an if or while.

C.16 Mismatched List Type

```

1 Line X, Column X
2     ....
3     ^
4 MismatchedListType:
5 The type of elements in a list must be consisitent .
6 The first element in this list literal is 'bool' , while the 3rd element
  is 'str'

```

This error occurs when a list literal is used with inconsistent item types.

C.17 Mismatched Dict Type

```
1 Line X, Column X
2     ....
3     ^
4 MismatchedDictType:
5 The type of keys and values in a dict must be consistitent.
6 The first key in this dict literal is 'bool', while the 3rd key is 'str'
```

This error occurs when a list literal is used with inconsistent key/value types.

C.18 Invalid Print Line End

```
1 Line X, Column X
2     ....
3     ^
4 InvalidPrintLineEnd:
5 The 'end' named parameter to the print statement must be 'str', instead
   found 'bool'
```

This error occurs when the `end` named parameter to the `print` function is the incorrect type.

C.19 Invalid Index Type

```
1 Line X, Column X
2     ....
3     ^
4 InvalidIndexType:
5 Only 'int' may be used to index into '[bool]'. Instead found 'float'
```

This error occurs when the incorrect type is used for indexing into `list`, `dict` or `str` types with the subscription syntax.

C.20 Unsupported Index

```
1 Line X, Column X
2     ....
3     ^
4 UnsupportedIndex:
5 Cannot index into type 'bool'.
```


This error occurs when the indexing form of the subscription syntax is used on a type which doesn't support it.

C.21 Unsupported Slice

```
1 Line X, Column X
2     ....
3     ^
4 UnsupportedSlice:
5 Cannot slice into type 'bool'.
```

This error occurs when the slicing form of the subscription syntax is used on a type which doesn't support it.

C.22 Invalid Assign Target

```
1 Line X, Column X
2     ....
3     ^
4 InvalidAssignTarget:
5 This expression is immutable, and thus cannot be assigned to.
```

This error occurs when the user attempts to assign to an expression which isn't a valid assignment target.

Appendix D

Example Programs

The following are a series of example programs written in the Garter programming language. The majority of these example programs were adapted from the series of example programs to be shown to teachers interested in using the Turing programming for education.

Appendix E

Alternative Implementations

The first mechanism for implementing this system would be to follow in the footsteps of mypy. Mypy uses Python 3's Function Annotation syntax [4] to allow code written for mypy to also be valid python 3 code. On one hand, this is an advantage, as it means that no translation work needs to be done. Unfortunately, to make typing more explicit and reduce programmer error, we also want to add annotations to variable declarations, and other places which are not covered by the function annotation syntax. However, we will likely re-use the function annotation syntax for this thesis as a standard way to write function types and return values, as it already fits with the rest of python's syntax, and is an advanced enough feature that it is reasonable to remove from the dialect created for this thesis.

Languages based very closely on the semantics of other languages, such as TypeScript mentioned above, are often implemented through source-to-source compilers, or transpilers. The transpiler acts like a compiler, parsing and analyzing the code written in the dialect. However, instead of performing translation to assembly or machine code directly, transpilers emit code in the base language. For example, the

TypeScript compiler reads in TypeScript code, analyzes it, and then emits valid ECMAScript code. This has an awesome advantage of eliminating the requirement of implementing a full language implementation of the base language, as the semantics can be derived from an existing implementation of that language. We plan to use that technique for this thesis in order to avoid re-implementing much of python, and to ensure that we maintain accurate semantics.

One approach to the development of this dialect would be to take the same approach as Microsoft. The dialects transpiler would be written, much like any other compiler, from scratch using standard compiler technologies. The code generation stage would be the only difference between the transpiler and any other compiler, in that instead of generating machine code, it would be aiming to generate code in the base language which as closely maps to the original code as possible, such that errors generated by the interpreter map back accurately to the source code for debugging purposes. However, writing a complete compiler from scratch is often unnecessary as people have developed awesome tooling for implementing language dialects already.

An interesting and useful tool for defining new languages without writing a complete compiler is the Spoofax workbench [19], [7]. Spoofax is a complete tool-chain for defining the syntax and semantics of a language. Syntax is defined using the Syntax Definition Formalism (SDF3) specification format, which allows for complete language grammars to be described using a declarative formation. Semantic analysis is handled using a variety of tools, including Name Binding Language (NaBL), which handles semantic analysis of name binding, such as namespaces, declarations, and references; and Type Specification Language (TS), which allows for a declarative

definition of a language's type system. Finally, language developers translate the complete, type-checked and analyzed language into a base language, such as Java, using the Stratego Transformation Language [8], which declaratively maps AST structures in the new language to structures in an existing language, such as Java. Spoofox also provides a comprehensive testing story, with the Spoofox Testing Language (SPT) which can be used to test parsing rules, error and warnings in language productions, and transformation outputs. Finally, Stratego also uses these analyses to generate IDE tooling for the languages in question - languages built with Stratego get syntax highlighting, go to definition, and other useful tools almost for free.

The biggest, and most impressive, advantage of the Spoofox toolbox for language definition is its polish and integration with the Eclipse IDE. Languages defined in spoofox are developed in a complete development environment, which includes features such as inspection and debugging of intermediate data structures, as well as IDE integration for the generated language. The IDE integration is especially useful for a learning project, as it gives new developers the tools they need to explore and learn about their new programming language without having to memorize function and type names from libraries.

The Turing Extender Language (TXL) [14], [9] is another system for defining new languages in terms of base languages. TXLs processing is split into two distinct steps. First, in the syntactic phase, the dialect language is parsed, and an AST of the dialect language is generated. This dialect AST is then passed to the semantic phase, which describes the semantic meaning of the dialect language through a series of transformations into the base language AST. This AST is then written out, and can be compiled or executed.

Much like Spoofax, TXL is intended to be used with known syntax definitions for a base language, such as the Turing programming language. TXL supports tooling for modifying portions of a base languages syntax, allowing for dialect languages to be easily extended off of the syntactic structures of the base language. However, TXL is also capable of describing complete new languages, with completely different syntactic properties. In addition, by using multiple passes which annotate expressions with their type, it is possible for TXL to perform complex semantic type analysis. In the case of a dynamic language, a TXL definition of the base language, such as python, would be developed, followed by the addition of the typing syntactic structures in the dialect.

Much like Spoofax, and despite the use of Turing in its name, TXL is a generic framework for extending languages, and is not specific to the Turing Programming Language. With TXL it is possible to manipulate and generate code in many different programming languages.

Unlike Spoofax, TXL doesnt provide an immediate tool for integration into an IDE, but with a language like Python, an extension to the open-source IDE IDLE to transform the language before running it shouldnt be too difficult to add, and could create a very integrated experience which will be familiar when the student moves away from the statically typed subset to taking advantage of the entire language.

The Rascal meta-programming Language [22], [6] is an attempt to develop a standard language for all forms of program analysis, parsing, and code generation. It provides a single set of abstractions for creating static analyses for existing programming languages, performing automated code refactorings with understanding of the semantic meaning of the underlying code, and generating DSLs (Domain Specific

Languages) which compile to an underlying language. Like Spoofox, the programs written in Rascal can also be connected to an IDE, to provide auto-completion and features like go-to-definition to developers in the IDE, which is a very useful feature for developing a teaching language. The unified structure provides an advantage for the development of

In addition Rascal comes with parsers for existing languages, such as Java, which can allow for the easy creation of extra tooling or language features which build on top of existing language parsers, limiting the amount of development which is necessary in order to add a new feature to a programming language. This means that new features can be added to a language like Java with relative ease, by defining the new language feature in terms of features in the base language. While there are existing libraries for parsing many languages, Python is supported, which means that a new library will have to be implemented in Rascal for this project if Python is chosen as the target language, and Rascal as the transpilation framework.

If were OK with building the analysis upon other languages, we might want to take a look at the Scheme programming language [15]. The Scheme programming language takes a different approach to language extension then the approaches taken by Spoofox and TXL: namely, instead of starting from a complex language syntax and extending it, it starts with an extremely simple syntax - that of S-expressions, and defines the entire language in terms of them. In scheme, the statement 'if' looks like '(if THEN ELSE)' which is identical to the appearance of a function call with two arguments '(func ARG1 ARG2)'. This uniform syntax allows for very easy to use macros. Scheme defines macros like a special function type which is evaluated outside-in at compile-time. When the macro is invoked, the AST of the arguments

is provided to the macros implementation, which is then given the opportunity to emit arbitrary code as output. In addition, Scheme comes with a powerful macro-by-example system called `define-syntax`, which allows for syntactic patterns to be defined and matched against, along with simple syntax for defining the resulting meaning in a declarative manner. In addition, scheme macros implement hygiene, which means that they can internally use identifiers without the risk of the identifiers conflicting with identifiers found at the use-sites of the macro [16].

This is a very powerful approach, as it allows defining brand new language constructs which look nearly identical to the built-in language constructs directly within the language itself. Unfortunately, it is a relatively poor tool for defining brand new languages. While it is possible to define many new language concepts, the syntax is constrained to use s-expressions, and certain keywords are defined by the core language implementation, and cannot be re-defined by macros. Thus, schemes macro system is a very powerful system for language extension, but is incapable of defining arbitrary new languages.

However, we are not aiming to create an arbitrary new language, we are looking to extend the language with type checking. Unfortunately Scheme doesnt come with the best tools for the job here, as the program as a unit is not implicitly wrapped in a macro, which means that it is not possible in the generic case to perform full-program transformations. However, if an implicit outer macro invocation was created, it would be possible to define a macro transformation which would generate valid Scheme, and which would perform type checking. Unfortunately, however, much of the benefits of the modular scheme approach would not be able to be taken, and the macro would effectively amount to a partial compiler implementation.

In addition, Scheme has the disadvantage of, while it has a somewhat significant library collection, lacking the industry presence which we are looking for to make the language more appealing to students, and a better stepping stone to real projects. These factors make Scheme a poor choice for this project.

The Julia Programming Language [12], [2], much like in the Scheme programming language, allows programmers to manipulate the AST of Julia programs. However, unlike Scheme, Julia doesn't limit its program syntax to only take the form of s-expressions, rather the syntax takes the form of arbitrary Julia syntactic structures. The syntax for Julia's macro expressions, however, are limited. Macro names must be prefixed by the @ sign, distinguishing them from native language constructs. In addition they must take one of two forms, either '@name a1 a2 ' or '@name(a1, a2,)'. Much like Scheme's macros, Julia's macros are hygienic, and do not pollute the call sites identifier space. The arguments which are passed to the the Julia function take the form of Julia expression AST nodes.

In addition to Julia's macros, Julia also supports generated functions. Generated functions are formed using the @generated macro, which annotates a function declaration. Wherever a generated function is called, the generated functions body is called, and passed the computed types of the arguments. The function is then able to generate custom logic which will be executed when the function is called at run-time. This is very similar in concept to generic functions in other languages such as C++, except instead of writing the code generation logic in a declarative system such as C++'s template syntax, generated function code generation logic is instead written directly in Julia. This is extremely useful for writing complex generic functions which

need to have custom functionality based on types which would be difficult or impossible to express under a more limited system. Combined with Julias macros, this provides Julia with a very powerful meta-programming story. However, as is the case with Scheme, Julia is limited in that it is not possible to define a completely distinct language with only these constructs, as the syntactic transformations which may be performed are limited.

Julia is also a type-safe language, and has remarkably good error messages. Many of the type-checking semantics and error messages which Julia provides could be used as guides for the target behavior of whatever mechanism is chosen. Unfortunately, we do not believe that Julia does a good job of filling this role, as it lacks the libraries and production usage of other languages like Python due to its young age. However, we could see Julia potentially being used in the future for teaching programming concepts.

Another area of CS research related to program transformation is the area of Aspect Oriented Programming. In the Aspect Oriented Programming paradigm, programs are described in two parts: the main program, which describes the primary business logic, and a set of cross-cutting concerns or aspects [21]. The aspect interpreter or compiler then executes the main program, interleaving the aspect logic whenever certain pointcuts (language structures) are reached in execution. For example, an aspect may bind to method calls on a database object, causing a logging operation to take place whenever one of these calls is made. This technology can also be used by language developers which wish to make changes to the basic semantics of the target language. By binding and executing different instructions before and after pointcuts in the program using an aspect oriented interleaver, the semantics

of individual instructions can be modified without needing to change the languages structure, allowing new languages to be defined with different semantics, but the same syntax, as other languages.

This form of transformation, however, is not the type of language transformation that we need for this project. Our goal is to avoid changing the semantics of the language while providing additional analysis at compile time, which takes advantage of new syntax. For this reason, the technologies used by aspect oriented programming will be unlikely to be very useful in this project.