Funky.

Joseph Ellis

Dissertation for BSc Computer Science
Supervised by Harish Tayyar Madabushi
University of Birmingham
Academic Year 2018/2019

Submitted: 7th April 2019

Abstract

This paper introduces a new functional programming language, affectionately named Funky. The project delivers a compiler for converting source files written in Funky to other target languages and a REPL allowing the user to evaluate Funky code interactively.

Leveraging research and literature regarding the theory and implementation of functional programming compilers, the author has developed a robust language that is a balanced blend between the syntaxes of Haskell and Python. The language is designed to be simple and user-friendly so that it is appropriate for introducing new users to the functional paradigm.

Funky's key features include an elegant syntax complete with pattern matching and lazy evaluation. Funky is statically typed, guaranteeing that compiled programs do not produce type-related errors. Funky source code is compiled to a small, yet powerful intermediate language that closely resembles the pure lambda calculus, which combined with the modular nature of the compiler, facilitates the process of creating code generators for different target languages.

Keywords: *Programming Languages, Language Design, Compilers, Functional Programming*

Acknowledgements

I would like to give acknowledgements to those who have helped me throughout the completion of this project. First, I would like to thank my supervisor, Harish Tayyar Madabushi, for his continual guidance and support.

I would like to thank Nicole Webster for her constant reassurance. I am extremely grateful for your unquestioning confidence in me.

I would like to thank my good friend Ben Warren for taking the time to produce a logo for Funky. It looks fantastic, and your artistic talents never cease to impress.

Finally, I would like to thank my family and friends for their patience and encouragement throughout.

Funky.

Contents

1 Introduction

Functional programming has withstood the test of time and proven itself to be a valuable tool in a programmer's arsenal. The functional paradigm often lends itself naturally to complex problems where the imperative style might complicate matters. Despite this, it is a common remark that functional programming feels less 'natural' to new programmers compared to imperative programming. Indeed, human beings are more familiar with sequential instructions and linear constructs than they are with recursion, a staple of functional languages.

It can be argued that, at least in part, the syntax of functional programming languages is an obstacle. Older languages such as Scheme, LISP, and to some extent OCaml, are particularly notorious in this regard; their syntaxes are generally verbose, and often, bracketing is used liberally. Verbose syntaxes increases the cognitive distance between the code and its semantics and can be discouraging to new programmers.

However, functional programming can be made more welcoming. Haskell is a good example of a language with a more beginner-friendly frontend, whilst simultaneously offering the same power and expressiveness of the aforementioned languages.

Inspired by highly-readable languages such as Python and Haskell, this project delivers a new functional programming language called Funky, complete with a compiler and REPL, that is effective as an introductory language for new programmers interested in the functional paradigm. Funky's syntax is a balanced blend between Python and Haskell syntax, emphasising readability.

A brief overview of the structure of the paper is below:

- Section 2 captures the motivation and inspiration for the project, as well as the principles and design choices followed when developing a user-friendly language.
- Section 3 describes the software development processes and tools employed throughout the development of the compiler.
- Section 4 communicates the architecture of the Funky compiler, expanding on the various stages of compilation and the algorithmic challenges encountered in each.
- Section 5 compares the Funky language and compiler against the requirements defined in the project proposal, as well as demonstrating that Funky is performant through a number of benchmarks.
- Section 6 illustrates aspects of the Funky language and compiler that were *not* required by the project proposal, but are still worth mentioning.
- Section 7 reflects on the project as a whole, discussing the project's achievements and known problems.

1.1 Background Reading

Compilers Aho et al. (1988) is considered the standard text for the theory of compilation. Mogensen (2007) and Cooper and Torczon (2004) are also fantastic texts in this regard.

Functional Programming Hughes (1990) is an ode to functional programming and explains why it is useful. It is exceptional background reading on functional programming.

Language Design Marlow et al. (2010) defines the syntax for Haskell programs and their semantics. This text has been indispensable throughout the development of Funky and contains a wealth of information about the design and implementation of functional languages.

Lambda Calculus Barendregt (2014) and Meyer (1982) are both good texts.

2 Design of Funky

This section documents the motivation for Funky, its inspiration, as well as the principles and design choices made while developing a user-friendly language.

2.1 History of Programming

It cannot be denied that programming is significantly easier now than it has been in the past. One reason for this is the wealth of learning resources made easily accessible via the internet. We are in an age where anyone can learn how to create software without specialist training, using only material freely available online.

Another significant reason is that the act of programming itself has become easier. Enormous progress has been made in simplifying the process of writing code, primarily due to the introduction of high-level programming languages. These languages are defined by a strong abstraction from the intricate details of the computer system, allowing the programmer to express their intent in a more direct and readable manner. This contrasts from the meticulous, painstaking art of writing code in low-level languages, where the onus is on the programmer to fully understand the underlying computer system.

Before high-level languages existed, programmers were required to write all of their code in a low-level language, the principal example of which is plain C. Low-level languages offer little abstraction from the machine, and programming in them requires a level of diligence far beyond what is required for high-level languages. Over time, programmers have sought to improve the tools used to create software, which has resulted in new programming languages to further abstract the code from the machine it will run on. In turn, these languages have been used to produce languages even further detached from the details of the machine, and so on.

Programming languages are now very close to plain English. Python is a good example of such a language.

One might assume that higher-level languages have a power or expressivity trade-off, but realistically this is not the case. High-level languages are applicable to most software engineering problems. In fact, a majority of software engineers will never need to think about low-level details of the computer system, such as pointer manipulation and memory management, since it is abstracted away by a high-level programming language. This is tremendously valuable; not only does this make programming accessible to a wider audience by lowering the barrier to entry, but it means software development is easier, less costly, and far less error-prone.

Imperative programming languages have seen immense improvements in this regard. Comparing a sorting algorithm written in C (Listing 1), a low-level programming language, with the exact same algorithm written in Python (Listing 2), a high-level programming language, profoundly illustrates this abstraction.

Functional programming languages have seen an even more pronounced development. Consider the same algorithm (quicksort) implemented in Scheme, a language introduced in 1975, in Listing 3, compared with its implementation in OCaml from 1996 in Listing 4.

Hopefully the reader will agree that these examples prove that programming has generally become more concise and programmer-friendly. Funky aims to continue this pattern by combining syntactic elements from two existing programming languages popular for their readability: Python and Haskell. Funky's syntax is designed to read like plain English as far as possible.

For a taste of what is to come, let us finally take a look at the quicksort algorithm in Funky in Listing 5. Funky bears syntactic resemblance to both Haskell and Python and tries to be as human-readable as possible.

```c
void swap(int *a, int *b) {
    int t = *a;
    *a = *b;
    *b = t;
}

int partition(int arr[], int low, int high) {
    int i = (low - 1);
    int pivot = arr[high];

    for (int j = low; j <= high- 1; j++) {
        if (arr[j] <= pivot) {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return i + 1;
}

void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
```

Listing 1: Quick sorting in C.

```python
def quicksort(arr):
    if len(arr) <= 1:
        return arr

    pivot = arr[len(arr) // 2]
    left   = [x for x in arr if x <  pivot]
    middle = [x for x in arr if x == pivot]
    right  = [x for x in arr if x >  pivot]
    return quicksort(left) + middle + quicksort(right)
```

Listing 2: Quick sorting in Python.

```scheme
(define (split-by l p k)
  (let loop ((low '())
             (high '())
             (l l))
    (cond ((null? l)
           (k low high))
          ((p (car l))
           (loop low (cons (car l) high) (cdr l)))
          (else
           (loop (cons (car l) low) high (cdr l))))))

(define (quicksort l gt?)
  (if (null? l)
      '()
      (split-by (cdr l)
                (lambda (x) (gt? x (car l)))
                (lambda (low high)
                  (append (quicksort low gt?)
                          (list (car l))
                          (quicksort high gt?))))))
```

Listing 3: Quick sorting in Scheme.

```ocaml
let rec quicksort = function
    | [] -> []
    | x::xs -> let smaller, larger = List.partition (fun y -> y < x) xs
               in quicksort smaller @ (x::quicksort larger)
```

Listing 4: Quick sorting in OCaml.

```
import "intlist.fky"

quicksort Nil          = Nil
quicksort (Cons x xs) = (quicksort s) ~concatenate~ unit x ~concatenate~ (quicksort l)
                        with s = filter ((>)  x) xs
                             l = filter ((<=) x) xs
```

Listing 5: Quicksort in Funky.

2.2 Motivation

One motivation for this project has been to make functional programming more beginner-friendly, with the aim having been to develop a functional language which is syntactically similar to Python and Haskell.

Python was chosen here, as its syntax is designed under the provision that "code is read more often than it is written" (van Rossum et al. (2001)), and is specifically intended to be comprehensible. Not only this, but Python is a popular choice for a first imperative programming language, and it follows that many beginner programmers are already acquainted with its syntax. By bridging the gap between imperative and functional languages by way of a similar syntax and philosophy, it is hoped that Funky provides a gentle introduction to the functional paradigm for new programmers.

Haskell is also a motivator simply because it is widely praised for being concise and readable, much like Python, and because of its widespread popularity in the functional programming space.

2.3 Inspiration

Funky, as a complete language (rather than just in terms of syntax), is profoundly inspired by Haskell. Haskell is a mature, open-source programming language with a large team of competent developers and researchers that have been contributing to for almost thirty years. Funky simply cannot compete with a language as intricate and complex as Haskell. That said, many of Funky's aspects and features draw inspiration from Haskell, and the Glasgow Haskell Compiler (GHC) is something of a paragon to the Funky compiler. Marlow et al. (2010), referred to as the Haskell Report, has been an essential reference throughout Funky's development, with its valuable insights on how to parse and compile functional code,

The author deeply praises and respects the work accomplished by the Haskell team.

Funky's *syntax* is inspired by both Haskell and Python. These languages were chosen as inspiration for Funky because of the following reasons:

- 'High readability' is at the heart of the design of the Python programming language, as celebrated in van Rossum et al. (2001), which defines a style guide and set of idioms for the language. Guido's key insight is that "code is read much more often than it is written", so it is sensible to make every effort to ensure code is readable. Python achieves this in a number of ways, most notably by using a syntax that reads like plain English. Funky's choice of keywords and general prose is inspired by this philosophy.
- Haskell employs lots of syntax sugar to facilitate writing clean, yet powerful code. Haskell presents a nice balance in the amount of syntax sugar available so as to maximise readability. This gives the programmer enough freedom to emphasize important parts of the code, while being restrictive enough that readers will know what to expect.
- Both languages employ an indentation-based scoping mechanism, forcing the programmer to format and layout their code uniformly. Violating the 'layout rule', as it is called, will result in a parsing/compilation error. By forcing the programmer to use a consistent code layout, the resulting program is usually more readable.

The Funky language is inspired by all of the ideas above.

2.4 Funky's Design Goals

The design goals kept in mind while developing the language are below:

- Funky must read like plain English. The language should opt for plain English keywords where possible, and bracketing should be kept to a minimum.
- Funky must be effective at introducing the concept of functional programming. Typically, the introduction will be made to programmers who already have experience in an imperative language like Python. Therefore, Funky must facilitate an easy transition by appealing to Python-like syntax – this means high readability and a closeness to plain English.

- It is assumed that users will 'graduate' from Funky to using a more mature language, such as Haskell. Therefore, the Funky language must resemble traditional functional code and use standard features of the functional paradigm so that this graduation is as smooth as possible.
- Funky is intended to 'bridge the gap' between a functional language and a Python-like language. It must exhibit syntax which feels familiar to both Haskell and Python programmers.
- Complex language features (i.e. Haskell monads, Python decorators, etc), whilst important in serious software development, are not appropriate for a language intended to introduce the concept of functional programming and should be omitted.
- Funky should define a minimal set of basic ingredients, such as a small collection of built-in types, type classes, and functions, and little else. This means that data structures, such as lists and trees, and the operations one might perform on them should be defined either by the user *or* defined in the standard library. The author believes that this is beneficial, because concepts such as data structures are introduced and understood in terms of more basic, elementary language features, rather than being a 'black box'.
- The compiler and REPL should expose plenty of options, such as logging configuration and the ability to see the output of each stage. This will be beneficial to the user if they want to gain an understanding of how the compiler works.

2.5 Funky's Syntax

Funky's syntax is designed to be as close to plain English as possible, whilst still maintaining sufficient expressiveness. There are many examples of Funky programs in Appendix F which give a good general overview of the Funky language. This section provides a complete description of the syntax, including justification for certain design choices.

2.5.1 Basics

Comments Comments in Funky operate in the exact same way as they do in Python. They are stripped from the code at lex-time, so they are not present in the compiled source code. Comments are entirely symbolic and are intended to convey information to the reader of the code.

```
# Comments in Funky are denoted with '#'
pi = 3.14 # They span from the '#' to the end of the line.
```

The '#' character is an extremely popular choice for a 'comment character' in modern programming languages. It is natural for Funky to adopt this standard.

Indentation *Spaces* must be used for indentation in Funky. PEP8 (van Rossum et al. (2001)), the Python style guide, states that indentation in Python should consist of spaces. Since Funky is inspired by Python, and tries to appeal to Python's ideology, this standard is carried forward.

The compiler will refuse to process any code that has been indented with tabs, or uses mixed indentation. The reason for this strictness is to enforce consistency. Some languages will attempt to resolve mixed indentation, to various degrees of success. Funky's policy is that it is in the programmer's best interest to catch and amend these inconsistencies early on.

Modules A module in Funky is the minimum requirement for a `.fky` file. Modules group together top-level definitions.

```
module funky_module with
    # imports
    # ...
    # declarations
```

Module declarations in Funky are entirely symbolic. Their purpose is to explicitly signal to the user that 'this file is a Funky source file'.

Specifying an Entrypoint Without an entrypoint, programs are just collections of definitions. This *might* be desirable – for instance, libraries do not generally provide an entrypoint. However, in most programs, there must be a way to explicitly define *what* should be run upon execution, just like `main` methods in languages like C and Java. Following convention, the Funky compiler looks for a variable called `main` at compile-time. The contents of this variable are what is outputted to `stdout` when the compiled program is run. As a trivial example, consider the cherished 'hello world' program written in Funky – all that we have to do is set the special `main` variable to our hello world string, which tells the compiler that this is what we want to output.

```
module hello_world with

    main = "Hello, world!"
```

The `main` variable can be set to any valid expression in Funky. Typically the `main` variable will call functions and use variables declared elsewhere in the module. Since Funky supports out-of-order definitions (Section 4.10.5), `main` does *not* have to be the last definition in a file; it can be placed anywhere, even as the first definition.

2.5.2 Definitions

Variable Definitions Variables in Funky are defined in typical fashion – the variable identifier is followed by an equals (`=`) symbol, followed by the value of that variable. This is exactly the same syntax as in Python, and is quite common among high-level programming languages.

```
integer_variable = 15
float_variable   = 0.2
bool_variable    = True
str_variable     = "hello world"
func_variable    = lambda x -> x / 2
```

Funky's primitive types are `Integer`, `Float`, `Bool`, and `String`. Variables can take the value of any of these primitive types, a function type, or a user-defined algebraic data type.

It was considered whether to use the keyword `is` in place of the equals symbol for definitions. After discussing this option with colleagues, it was decided that while using `is` made the code read more like plain English, it was actually a distraction since it looked foreign. Colleagues reported that it took them longer to understand what the code is doing with the `is` keyword and overwhelmingly preferred the equals symbol.

Named Function Definitions Functions can be defined in Funky using what is referred to as 'function application notation'. Functions are defined using a syntax that resembles function application by juxtaposition. The name of the function is given, followed by a space-separated list of its parameters, followed by an equals sign and an expression. This style of syntax is common in functional languages (e.g. Haskell). Using similar syntax for defining and applying functions makes it clear what the code is doing and eases the understanding of the two concepts for new programmers.

```
# A function with three parameters
f x y z = (x - z) * y
```

Function definitions tend to be slightly more complicated than this. Implicit pattern matching and guards, discussed shortly, mean that functions definitions are rarely this trivial.

Binary Infix Function Definitions Syntax sugar is available in Funky for defining binary infix functions. To define a binary infix function, the first parameter can be placed *before* the function name, with the function name surrounded by two tilde characters (~).

```
x ~has_same_sign~ y = x * y > 0
# equivalent to:
has_same_sign x y = x * y > 0
```

The option of declaring functions using this syntax certainly increases code readability in some contexts. For instance, the infix definition of the function above could be read as '`x` has the same sign as `y` if `x * y > 0`' – it reads more like plain English than the standard definition.

Initially, the character tilde (~) was a backtick (\`), meaning that infix function definitions (and infix function application) were the same syntactically as in Haskell. This was discussed with colleagues, and it was decided that the tilde (~) character is preferable because it appears visually as though it is 'connecting' the two parameters.

Anonymous Functions/Lambdas Anonymous functions are a staple of functional programming, and as such, they are available in Funky. In Funky, the `lambda` keyword is used to create an anonymous function, similar to Python's syntax. The general format is the keyword `lambda` followed by a space-separated list of arguments, followed by an arrow (`->`), followed by an expression representing the body of the lambda.

```
lambda x y z -> (x == y) and (y == z)
```

Note the use of an explicit keyword, `lambda`, as opposed to something like Haskell's syntax of using a single backslash (\) to specify anonymous functions – for example, `\x y -> x`. There was some deliberation about this. Discussion amongst colleagues, to much surprise, revealed a distaste for Haskell's style in this instance. General consensus was that it is too subtle and easily missed, and that the backslash character itself has a dissatisfying 'asymmetry'.

A direct copy of the Python syntax – resembling `lambda x, y:  x` – was also considered. Similarly, colleagues expressed an slight aversion to Python's use of the colon (`:`) here, suggesting that it does not feel like 'functional syntax' and does not visually delimit the parameters of the function from its body sufficiently.

In the end, the style above was decided upon. It is a sensible balance between Python and Haskell syntax, and is readable.

2.5.3 Applying Functions

Function Application Functions are applied by juxtaposition. Function application is a right-associative operation. Functions can be partially applied to create a curried function.

```
f 5 1 7 # = ((f 5) 1) 7
```

Lambdas can be applied in the same way.

```
(lambda x y -> x) 5 10 # = 5
```

Binary Infix Function Application Binary infix functions can be applied in the same way they can be defined: by placing the first argument before the tilde-wrapped function name, followed by the second argument.

```
10 ~has_same_sign~ 20
# equivalent to:
has_same_sign 10 20
```

Again, as mentioned earlier in this section, this syntax for binary infix functions is more readable is some contexts. For example, `"yes" if x ~has_same_sign~ y else "no"` reads more like plain English than `"yes" if has_same_sign x y else "no"`.

Operator Functions Built-in binary infix operators in Funky (such as `+`, `*`, etc) are just infix functions. They can be applied in postfix notation by wrapping them in parentheses. This allows for partial application of built-in operators for currying, and facilitates writing concise code.

```
(+) 10 20
# equivalent to:
10 + 20
```

2.5.4 Conditionals

There are numerous ways to specify conditional execution in Funky.

If-Else Expressions The most basic form of conditional execution in Funky is the if-else expression. The syntax is exactly the same as an if-else *expression* in Python. During design, it was considered whether or not to use the Python-style conditional (`true_part if condition else false_part`) or the Haskell-style conditional (`if condition then true_part else false_part`). In the end, the Python style was chosen: it allows us to maintain consistency with the Python language and also allows us to avoiding the unnecessary introduction of a `then` keyword.

```
true_part if condition else false_part
```

Function Conditionals/Guards Guards are available in Funky as a readable way of declaring multiple function bodies, each of which associated with a predicate. The first guard predicate that evaluate to `True` at runtime is the function body that is executed. Guards are semantically equivalent to if statements, but are far more readable.

In Funky, guards are defined with the `given` keyword. There was some deliberation during design about whether or not a simple character, like a '`|`', might be more appropriate rather than a keyword. It was eventually decided that 'given' reads like plain English, so it was preferred.

```
signum n given n <  0 = -1
         given n == 0 =  0
         given n >  0 =  1
```

Using guards, it is possible to define a partial function – that is, a function which is undefined for some inputs. This occurs when none of the guard clauses evaluate to `True`, and is usually undesirable; an error will be raised at runtime. Guards should generally provide complete coverage over the input arguments – therefore, it is common for the last guard to be just `True` as a catch-all.

```
abs n given n >= 0 = n
      given True   = -n
```

The standard library `stdlib.fky` provides a variable `otherwise` which is defined as `otherwise = True`. This is used to improve readability of guarded functions.

```
import "stdlib.fky" # <- get the otherwise variable
abs n given n >= 0     = n
      given otherwise = -n
```

Pattern Matching The `match` keyword can be used for pattern matching. Pattern matching can be performed on literals, like so:

```
match n on
    0 -> 0
    n -> n + 1
```

It can also be performed on user-defined algebraic data types.

```
match xs on
    Cons x Nil -> True
    _          -> False
```

The `match` keyword makes it possible to embed pattern matching within expressions, although generally, implicit pattern matching (described below) is preferred for readability and robustness.

Haskell uses the keyword `case` to perform pattern matching. Discussing this with colleagues revealed that `match` was preferred, because the keyword is for performing pattern *match*ing – it is a better reflection of what the syntactic construct actually means.

Implicit Pattern Matching Implicit pattern matching is available in Funky. This allows the programmers to define different function bodies for different patterns of arguments in a natural, readable way. To use implicit pattern matching, simply substitute a symbolic argument to a function with a pattern. Implicit pattern matching in Funky is robust and is discussed in detail in Section 4.7.

```
True  ~xor~ False = True
False ~xor~ True  = True
_     ~xor~ _     = False
```

2.5.5 Local Bindings

Local bindings are available in Funky for convenience. They can be used to 'save' sub-expressions to make the code easier to comprehend.

With The `with` keyword in Funky can be used to start a list of bindings to be made locally available in some expression. The `with` keyword is equivalent to `where` in Haskell.

```
energy wavelength = (h * c) / wavelength
                    with h = 6.62 * 10.0 ** (-34.0) # <- Planck's constant
                         c = 299792458.0            # <- Speed of light
```

Originally in the Funky language, the `with` keyword was `where`. It was changed to `with` to more closely match Python's syntax, since Python has a `with` keyword which can be used in a similar manner.

Let The `let` keyword in Funky can be used in the same way as the `with` keyword, except the definitions are given before the expression they appear in. Which one the programmer decides to use is simply a matter of personal taste.

```
energy wavelength = let h = 6.62 * 10.0 ** (-34.0) # <- Planck's constant
                        c = 299792458.0            # <- Speed of light
                    in (h * c) / wavelength
```

The Funky compiler enforces that the `in ...` part of the let statement *must* be on its own line. This results in more readable code.

2.5.6 Data Type Declarations

Data type declarations are available in Funky using the `newtype` keyword. The `newtype` keyword functions in a manner similar to Haskell's `data` keyword – it allows the user to define a *sum type*, whose instances can be initialised to precisely one of a number of possible alternatives. These are referred to in Funky as *algebraic data types*. They are extremely powerful. The `newtype` keyword permits inductive definitions, permitting recursive (or even infinite) data structures.

```
newtype IntList = Cons Integer List | Nil
newtype BinTree = Branch BinTree Integer BinTree | Empty
newtype IntPair = P Integer Integer
newtype FList   = Cons (Integer -> Integer) FList | Nil
```

2.5.7 Fixity Declarations

Fixity declarations can be used to change the fixity of an operator in a Funky program. This is discussed further in Section 4.5.2. This is done using the `setfix` keyword. The `setfix` keyword accepts an associativity, either `nonassoc`, `leftassoc`, or `rightassoc`; a precedence (see Section 4.5 for precedence values); and an operator. The given associativity and precedence are applies to the chosen operator.

```
setfix rightassoc 8 **
x = 2 ** 2 ** 3         # = 256
setfix leftassoc 8 **
y = 2 ** 2 ** 3         # = 64
```

2.5.8 Imports

Funky offers an import system allowing the programmer to include declarations from other source files. Imports must appear at the very top of a module, before any declarations. The import statement requires a string which is a path relative to the file currently being compiled, or relative to a known library directory. The import system is discussed in detail in Section 4.4.

```
import "stdlib.fky"         # <- import standard library
import "some/path/file.fky" # <- import relative path
```

Reading Section 4.4, it is clear that import statements in Funky are comparable to the `#include` preprocessor directive in languages like C. The reason why the keyword `import` was preferred over `include` is because Python uses `import`, along with most other high-level programming languages. Programmers already familiar with a high-level programming language should be able to instantly recognise what the import keywords means – the same could not necessarily be said if the keyword was `include`.

The design choice to specifiy the import path as string (i.e. `import "stdlib.fky"` rather than `import stdlib.fky`) is to make imports more closely resemble a standard function call.

2.5.9 Runtime Error Reporting

Funky offers basic utilities for reporting errors at runtime. A special function `fail` is provided which accepts a error message as a string. The function can be used anywhere in place of an expression to signal an error.

```
import "stdlib.fky"
sqrt x given x > 0      = x ** 0.5
       given otherwise = fail "Cannot sqrt negative number!"
```

Alternatively, if a precise error message is not required, you can instead use the special `undefined` variable to signal that something is not defined. `undefined` literally translates to `fail "undefined"`, but it makes more sense in some contexts.

```
recip 0.0 = undefined
recip x   = 1.0 / x
```

2.5.10 Builtins

Operators/Functions Table 1 lists Funky's built-in operators/functions.

Operator	Type	Description	Notes
`==`	$t \rightarrow t \rightarrow Bool$	Equality operator.	
`!=`	$t \rightarrow t \rightarrow Bool$	Inequality operator.	`!=` reflects Python syntax.
`<`	$Num \rightarrow Num \rightarrow Bool$	Less than check.	
`<=`	$Num \rightarrow Num \rightarrow Bool$	Less than or equal to check.	
`>`	$Num \rightarrow Num \rightarrow Bool$	Greater than check.	
`>=`	$Num \rightarrow Num \rightarrow Bool$	Greater than or equal to check.	
`**`	$Float \rightarrow Float \rightarrow Float$	Floating point exponentiation.	`**` reflects Python syntax.
`^`	$Num \rightarrow Integer \rightarrow Num$	Raising to an integer power.	
`+`	$Num \rightarrow Num \rightarrow Num$	Basic addition.	
`++`	$String \rightarrow String \rightarrow String$	String concatenation.	`++` reflects Haskell syntax.
`-`	$Num \rightarrow Num \rightarrow Num$	Basic subtraction.	Also used for unary negation.
`negate`	$Num \rightarrow Num$	Unary negation.	
`*`	$Num \rightarrow Num \rightarrow Num$	Basic multiplication.	
`/`	$Num \rightarrow Num \rightarrow Num$	Basic division.	
`%`	$Integer \rightarrow Integer \rightarrow Integer$	Modulo operator.	`%` reflects Python syntax.
`and`	$Bool \rightarrow Bool \rightarrow Bool$	Logical and.	`and` reflects Python syntax.
`or`	$Bool \rightarrow Bool \rightarrow Bool$	Logical or.	`or` reflects Python syntax.
`slice_from`	$Integer \rightarrow String \rightarrow String$	String slicing.	
`slice_to`	$Integer \rightarrow String \rightarrow String$	String slicing.	
`fail`	$String \rightarrow t$	Error reporting.	

Table 1: Funky's built-in operators and functions.

Note that there are two operators for exponentiation: `**` and `^`. The difference between the two is their type signature: `**` has type $Float \rightarrow Float \rightarrow Float$ and `^` has type $Num \rightarrow Integer \rightarrow Num$. Exponentiation to an integer power can be done using the `^` operator. Floating point exponentation can be done with `**`.

Type Conversion Functions Funky provides functions for converting between built-in primitive types. These functions are:

- `to_str`, for converting types to a string.
- `to_int`, for converting types to an integer.
- `to_float`, for converting types to a float.

`to_str`, `to_int`, and `to_float` accept instances of the type classes `Stringable`, `Intable`, and `Floatable` respectively[1]. As their names suggest, these typeclasses consist of primitives which can be covert to strings, integers, and floats. For example, the `Intable` typeclass allows `Float`, `String`, and `Bool`, as it is possible to convert all of these to integers.

These functions are useful for converting between types.

```
(to_int   "26"  ) +  (to_int   1.5 )  # = 27
(to_float "82.2") +  (to_float 5   )  # = 87.2
(to_str   100   ) ++ (to_str   12.2)  # = 10012.2
```

3 Software Engineering

The Funky compiler is written in the Python programming language. The development process used while developing the compiler is most similar to **Agile**. The software engineering practices and principles employed throughout the compiler's development are discussed in this section.

General Approach The general methodology used while programming the compiler resembles Agile. Work on the compiler has been done in short, incremental and iterative sequences similar to sprints. At the start of each week, a series of goals were established to be completed by the week's end. At the end of each week, an evaluation took place to identify what went well, what was achieved, and what still requires work. This process is reflected in the weekly logs submitted to Canvas.

This approach was initially chosen due to the relatively loose specification of the project. At the time of writing the proposal, the features of the project were unclear difficulty- and time-wise, and so it was anticipated that some requirements might change as time went on. Taking a less structured approach to development by performing weekly evaluations and making decisions on a short-term basis accommodated this need well.

A complete log of work on the project can be found in Appendix C.

Modularity Every effort has been made to ensure that the compiler code is as modular as possible. Thankfully, compiler development is a highly researched field and the general architecture of a compiler is well understood. Aho et al. (1988), Mogensen (2007), and Cooper and Torczon (2004) were instrumental in becoming acquainted with compiler design, and as a result, the structure of the Funky compiler largely reflects what is presented in these books. In the compiler, each stage of compilation (described in Section 4.1) is separated into its own sub-package within the project. These sub-packages are loosely coupled and are not inter-dependent. They are designed such that they take an input and produce an output with limited side-effects. Precisely what each stage takes as input and produces as output can be understood by examining Figure 2 in Section 4.1.

The result is that the Funky compiler is modular. It is easy to 'plug in' stages of compilation, and the interface for doing so is well documented in the code. For instance, if a programmer wanted to create an extension to the compiler allowing Funky code to be compiled into a different target programming language, it is very easy for them to do so.

The complete code structure of the software is outlined in Appendix A.

[1] note that there is no `Boolable` typeclass, because converting a type to a boolean can be done trivially using logical operators. E.g. `int_to_bool = (!=) 0`

Version Control Git has been used for version control throughout the project. Git has been very useful in tracking changes in the code over time, as well as reverting to a known-good state when there has been problems.

The project code has been pushed to GitHub throughout and will be made freely available to the public after the project is complete and formal marking has been carried out. Figure 1 visualises commits made to the compiler over time, which is a good indicator of when the bulk of the programming work was done.

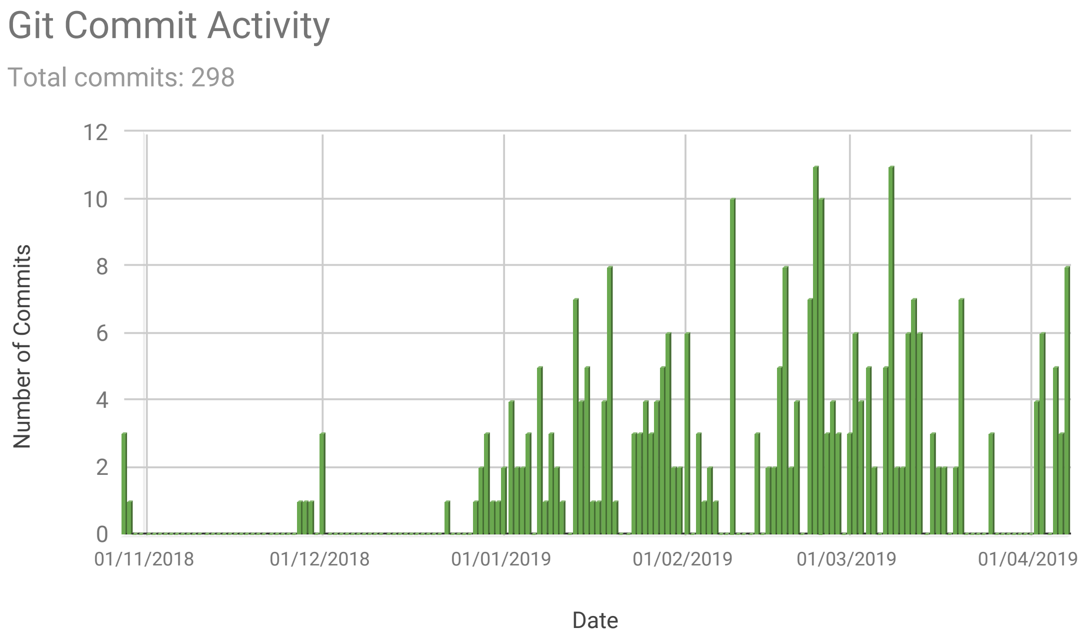


Figure 1: Chart visualising commits made to the compiler repository over time.

Logging Logging is used throughout the compiler. Having logs available has been extremely useful for debugging. Special care has been taken to ensure that all important parts of the compiler have extensive logging, and this has been used for debugging throughout development.

Unit Tests The compiler codebase has good unit test coverage, and these automated tests have been used throughout the development process to detect and fix regressions while making changes to the compiler.

Test-Driven Development There are a number of sample programs bundled with the Funky compiler. Some of these programs were written before a working compiler was even available; they were used as a guideline for what the compiler *should* be able to do when it is finished. This is a form of test-driven development – the expected outcome was defined before the software was produced, and was validated against the compiler afterwards.

Prototype-Driven Development Prototypes have been used throughout development, particular for modelling the different phases of the compiler. For instance, the parser was prototyped many times before a final version was decided.

4 Architecture of the Funky Compiler

4.1 General Architecture

Aho et al. (1988) describes a typical architecture for a compiler. The Funky compiler largely adheres to this structure, with some minor modifications. The following subsections explain each stage of compilation in detail. Broadly speaking, these stages are:

1. Lexing/lexical analysis (Section 4.2), where the raw source is split into basic tokens of meaning and disambiguated.
2. Parsing/syntax analysis (Section 4.3), where the lexed input is parsed according to a context-free grammar to produce a syntax tree.
3. Import handling (Section 4.4), where any imports discovered from the parsing phase are recursively lexed, parsed, and merged with the current syntax tree to make their declarations available.
4. Fixity resolution (Section 4.5), where infix expressions are converted to function applications.
5. Renaming (Section 4.6), where all user-named variables are given machine generated names and basic sanity checks are performed.
6. Desugaring (Section 4.7), where the complex source tree is reduced to a simple and small intermediate language called the core tree. This includes conversion of pattern matching to efficient decision trees (Section 4.8).
7. Optimisation/pruning (Section 4.9), where the core tree is optimised and pruned to produce smaller/more efficient target code.
8. Type inference (Section 4.10), where the core tree is annotated with types.
9. Code generation (Section 4.11), where the core tree is converted to a target language.

Each of these stages presents itself with a number of interesting algorithmic problems to tackle, which are explained in detail under their own subheading in this section. Due to thoughtful consideration when designing the compiler, the stages are modular and mostly self-contained. The output of one stage used as the input to the next.

The flow of data through the Funky compiler is summarised in Figure 2.

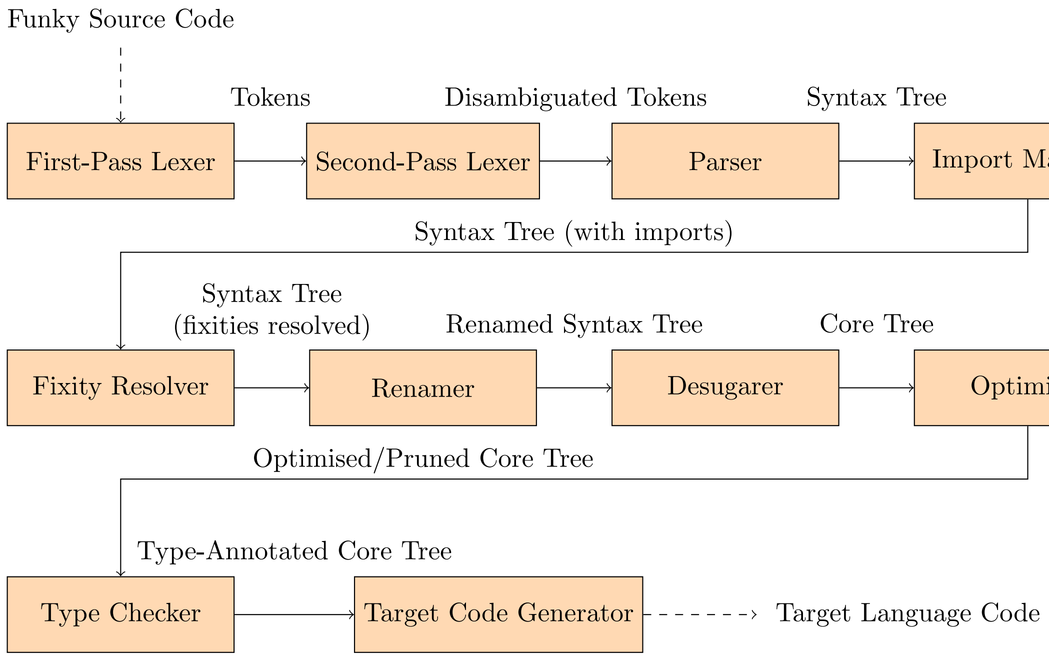


Figure 2: A high-level overview of the flow of data through the compiler.

4.2 Lexing

Lexing, often referred to as lexical analysis, is the process of converting the raw source code of a program into a sequence of tokens with an assigned name and identified meaning. For example, the string `x + 10` might be converted into `<VAR x> <OP +> <INT 10>`, which is much easier to manage. Lexing can be considered the process of transforming the source code into basic units of meaning.

Funky's implementation of lexical analysis is more involved than what would be required by a more simple programming language. The design choice to minimise bracketing in Funky and to create as much similarity with Python as possible implies that it is unacceptable to force the programmer to explicitly define scopes and collections of statements. In other words, the programmer should not have to use characters such as '{' and '}' to delimit compound statements like they might in a language such as Java.

However, if we do not explicitly define where blocks of code begin and end, our code can be ambiguous. This is catastrophic for parsing. To overcome this, the Funky compiler performs an additional step after basic lexing to directly insert explicit braces into the lexed source. We refer to this process as *disambiguation.*

Disambiguation is guided by indentation in the source code. Spacing is used to implicitly delimit groups of statements and definitions. The algorithm used by the disambiguation step is known as the *layout rule.*

The result is that Funky performs lexing in two passes. The first pass creates tokens from meaningful items in the source code, typical of a basic lexer. The second implements our disambiguation step, employing the layout rule to insert explicit braces. These two passes are discussed in detail below.

4.2.1 First Pass – Basic Lexing

Funky's lexer uses PLY to build a lexical analyser. For the first pass, we bind Python functions to the occurrence of particular regexes to represent tokens. This allows us to run a segment of code whenever we encounter matching strings. This behaviour is used to create tokens of the desired type and value.

Outside of trivial regex-based lexing, which is largely handled by PLY, the first-pass lexer must ensure that whitespace is handled correctly. The rule is simple: whitespace is entirely ignored [2], unless it occurs at the start of a line, where it is considered to be indentation. To handle this, the first-pass lexer maintains a flag `at_line_start` which is set to `True` upon matching a newline character, and set to `False` upon matching any other lexeme. If we match `WHITESPACE` and `at_line_start` is `True`, we insert a whitespace token into the token stream whose value equals exactly how much whitespace was consumed. Otherwise, if we match `WHITESPACE` and `at_line_start` is `False`, we simply ignore it, adding nothing to the token stream.

The results of the first lexing pass are best illustrated with an example. The code depicted below is transformed to the following token stream.

```
module lex_example with

    area r    = pi * r ** 2.0
                with pi = 3.14
    main      = area 51.0
```

$\Longrightarrow$

```
MODULE IDENTIFIER=lex_example WITH
WHITESPACE=4 IDENTIFIER=area
IDENTIFIER=r EQUALS IDENTIFIER=pi TIMES
IDENTIFIER=r POW FLOAT=2.0 WHITESPACE=13
WHERE IDENTIFIER=pi EQUALS
FLOAT=3.14 WHITESPACE=4 IDENTIFIER=main
EQUALS IDENTIFIER=area FLOAT=51.0
```

4.2.2 Second Pass – Disambiguation

Upon completing the first pass, we have a collection of tokens representing the source program. Interspersed within are `WHITESPACE` tokens. Careful lexing in the first-pass ensures that these tokens always represent indentation.

[2]'entirely ignored' as in whitespace tokens are not included in the token stream – they are dropped when encountered. Whitespace is still used to delimit other tokens, such as identifiers.

In Funky, indentation is used to implicitly delimit groups of statements. Information about how the program is indented is therefore important for disambiguation.

The problem description is as follows: the second-pass lexer is required to insert explicit braces '{' and '}' into the token stream to at the appropriate location to explicitly group statements. In the worst case[3], the second-pass lexer only has indentation information available to do this.

Section 2.7 of Marlow et al. (2010) provides a fantastic description of the layout rule and its implementation in Haskell. Funky's implementation is similar. Let us consider where statements need to be grouped. In Funky, there are three places:

- Bindings in a `with` body need to be grouped, i.e. `with {x = 12; y = 2}`.
- Bindings in a `let` body need to be grouped, i.e. `let {x = 1; y = 2} in ...`.
- Alternatives in a `match` body need to be grouped, i.e. `match n on {0 -> ...; 1 -> ...; ...}`.

Thus, braces need to be inserted following the keywords `with`, `let`, and `on`. With this insight:

- The layout rule is applied if and only if an explicit brace is omitted following the keywords `with`, `let`, and `on`. If braces are included by the user, no disambiguation is performed.
- When a brace is omitted, the indentation of the next lexeme is recorded and a '{' is inserted. Let the recorded indentaton be denoted `ind`, serving as the 'base indentation level' for the statements.
- For each line that follows:
  - If it either consists of entirely whitespace or its indentation is strictly greater than `ind`, nothing is inserted. This is equivalent to continuing the previous statement.
  - If its indentation is equal to `ind`, a single '`;`' is inserted to delimit the previous statement.
  - If its indentation is less than `ind`, a single '}' is inserted to delimit the whole group.

[3]the user can explicitly insert braces into the source code to bypass the layout rule, but it is not recommended.

The algorithm is perhaps better expressed through pseudocode, shown below.

```
function DISAMBIGUATE(tokens)
    $new\_tokens \leftarrow ind\_stack \leftarrow emptystack$
    $i \leftarrow 0$
    while $i < length(tokens)$ do
        $tok \leftarrow i^{th}$ token
        $nxt \leftarrow (i+1)^{th}$ token
        if $tok$ is “with”, “let”, “on” and $nxt$ is not ‘{’ then
            $x \leftarrow$ next non-whitespace token
            PUSH($ind\_stack$, column of the $x$)
            APPEND($new\_tokens, tok$)
            APPEND($new\_tokens$, ‘{’).
            $i \leftarrow$ index of $x$
        else if $tok$ is whitespace then
            if $ind\_stack$ is nonempty then
                $spaces \leftarrow$ number of spaces in $tok$
                if $spaces ==$ PEEK($ind\_stack$) then
                    APPEND($new\_tokens$, ‘;’)
                else if $spaces <$ PEEK($ind\_stack$) then
                    PUSH($new\_tokens$, ‘}’)
                    POP($ind\_stack$)
                    continue
                end if
            end if
            $i \leftarrow i + 1$
        else
            APPEND($new\_tokens, tok$)
            $i \leftarrow i + 1$
        end if
    end while
    while $ind\_stack$ is nonempty do
        APPEND($new\_tokens$, ‘}’)
        POP($new\_tokens$)
    end while
    return $new\_tokens$
end function
```

These rules allow the user to write clean and concise code whilst simultaneously conveying enough information about groupings to avoid ambiguity in parsing.

Once again, this is best illustrated with an example. The code below, undergoes the following transformation when disambiguated [4]:

[4] Note that the actual output from disambiguation is a stream of tokens – here, we show the output from the second-pass lexer as modified source code for notational convenience.

```
module lex_example with

    sqrt n = n ** 0.5

    euler = e ** (i * pi)
            with
              e  = 2.72
              pi = 3.0
                 + 0.14
              i  = sqrt (-1.0)

    main  = euler
```

$\Longrightarrow$

```
module lex_example with {

    sqrt n = n ** 0.5;

    euler = e ** (i * pi)
            with
            { e = 2.72;
              pi = 3.0 + 0.14;
              i = sqrt (-1.0)
            };

    main = euler }
```

Disambiguation marks the final step of the lexing phase. We are left with a stream of tokens that is ready for parsing.

Summary In this section, we have seen how the process of lexing is divided into two passes: the first pass, which converts the raw source code into a sequence of meaningful tokens, and the second pass, which inserts explicit braces into these tokens to delimit sections of code according to the layout rule. The result of this phase is a stream of tokens which is ready for parsing.

The command line flag `--dump-pretty` can be used to output a textual representation of the first-pass lex, and the command line flag `--dump-lexed` can be used to ouput a textual representation of the disambiguated token stream.

The results of the lexing phase of compilation are then passed to the next phase: parsing.

4.3 Parsing

Parsing involves taking the tokens generated by the lexing stage of compilation and producing a tree to represent the structure of the source code. This tree is referred to as the *source tree*.

4.3.1 Choice of Parser

There are many different kinds of parser, each with their own set of advantages and shortcomings. We discuss these briefly.

LL Parsers It is possible to parse a context-free grammar by hand – that is, the programmer writes code to parse input strings by themselves, without relying on a *parser generator* to mechanically produce some form of ‘code’ (be it a finite state automaton or other) to do it for them. Generally speaking, hand written parsers fall into the category of LL parsers. Such parsers scan the input text left-to-right and produce a left-most derivation. A standard example of a hand-written parser is a recursive-descent parser, which defines functions for each non-terminal symbol in the grammar and performs backtracking as appropriate when no derivation matching the current state of the parser exists. Predictive parsers are more robust, and look ahead in the input string to avoid backtracking.

LL parsing might be preferred where there is a requirement for absolute control over the parsing process. They allow the programmer to handle context sensitivity within the parser exclusively, which can be desirable in some applications.

However, the disadvantages are obtrusive. Unfortunately, hand-writing a parser has a few significant shortcomings: firstly, it is non-trivial for large grammars, and secondly LL parsing is capable of processing a smaller subset of grammars than other parsing techniques. Perhaps more insidious is the fact that because the parser is programmed by a human, it is likely to contain bugs that may go unnoticed without significant testing.

LR Parsers LR parsers are a kind of bottom-up parser, described in Knuth (1965). They scan the input text left-to-right as before, but the parse tree is generated from the *leaves* to the *root*, in the opposite direction as the previously discussed LL parsers. LR parsers are capable of parsing a larger subset of context-free grammars than LL parsers and are better at resolving ambiguity. This makes them desirable for programming languages. In general, LR parsers are generated using a parser generator which creates code to parse the grammar from some formal specification. The generated nature of the parsing code is reassuring, as it is far less likely to contain bugs relative to hand-written parsers. However, the formal specification used to define grammars for these parsers is limiting compared to the freedom achieved by hand-writing a parser. Although this means context sensitivity is impaired, this is not something that is likely to hinder us for most real-world applications, since many parsing libraries implementing LR parsers provide some form of syntax-directed translation to introduce more control into the parsing process.

LALR Parsers LALR is a simplified version of a canonical LR parser, but is less powerful than an LR(1) parser. LALR parsers were invented by Frank DeRemer in his 1969 PhD dissertation, DeRemer (1969). LALR parser generators merge states if the GOTO tables and lookhead sets for reductions agree, resulting in a smaller number of states (and hence a smaller parse table) at the expense of the inability to distinguish certain sequences that LR can. These smaller parse table mean that LALR parsing is less memory intensive.

LALR is the parsing technique of choice in many situations, because the ‘lost power’ can usually be accommodated with small changes to the grammar if necessary.

GLR Parsers A GLR parser is an extension of an LR parser which handles nondeterministic and ambiguous grammars. The theoretical foundation for GLR parsers was established in Bernard Lang’s 1974 paper Lang (1974). GLR parsers are capable of parsing the largest subset of context-free grammars out of all of the previously mentioned techniques, and are also better equipped to handle ambiguity in a grammar. However, this comes at the expensive of time complexity. McPeak and Necula (2004) (pg 86) suggests, in the worst case, the time complexity bound of GLR parsing is $O(n^3)$ – a significant regression on the worst-case time complexity of LR/LALR parsing, $O(n)$.

Where an LR/LALR parser generator would raise a shift-reduce or reduce-reduce conflict (meaning that there is nondeterminism in the grammar), a GLR parser would attempt all available options by branching. If a particular branch fails, it is simply ignored; it cannot be the correct parse. If multiple branches produce successful parses, the onus is on the programmer to specify which was intended.

4.3.2 Implementation

Funky uses PLY to parse the tokens, which implements an LR(1) parser. As we have seen, LR(1) parsing is reasonably efficient and well suited for larger grammars. The justification for using an LR(1) parser is that, for one, there are a wide range of available libraries to choose from, many of which are well maintained and supported. PLY is a good example of such a library – it is a direct Python implementation of the industry-standard UNIX tools, `lex` and `yacc`. The library provides a one-to-one feature correspondence with `lex` and `yacc`, and consequently, existing literature on the two tools was useful in guiding the process of programming the parser.

Additionally, the context-free grammar required for Funky can be handled by an LR(1) parser (provided that we perform the aforementioned disambiguation step for lexing, as described in Section 4.2.2). In the interest of keeping things as simple as possible, it makes sense to opt for a simpler parser.

PLY allows the programmer to associate a procedure with a particular non-terminal symbol in a context-free grammar, facilitating syntax-directed translation. This is instrumental in building the source tree.

Summary In this section, we have discussed the various different parsers available to us and justified the use of an LR(1) parser to process Funky’s grammar. We have seen how Funky uses PLY to parse the token stream from the lexing phase.

The `--dump-parsed` command line flag can be used to output a structured representation of the parse tree to `stdout`.

The result of the parsing phase is a *syntax tree* representing the structure of the given code.

4.4 Handling Imports

Most respectable programming languages ship with a built-in mechanism for importing code. This is absolutely crucial for reducing code duplication and creating modular, sustainable programs. It is also required if there is a desire for the language to have an extensive standard library.

Funky is no exception here. Using the `import` keyword, the programmer can import declarations from other Funky source files to use in their program. For example, if the user wants to access features provided by the standard library, they can preface their module with `import "stdlib.fky"` to import the standard library. Funky provides additional standard code files, such as `math.fky` and `logic.fky`, the declarations within which can be accessed using Funky's import mechanism. Funky's standard library is discussed further in Section 6.1.

Before we continue, let us see an example of Funky's import system. The code in Listing 6 shows the use of the import statement.

```
# Using imports from the standard library in Funky.
module test with

    import "stdlib.fky"

    double    = (*) 2
    increment = (+) 1

    # The 'compose' function is provided by stdlib.fky
    double_and_increment = increment ~compose~ double

    main = double_and_increment 100
```

Listing 6: Using imports in Funky.

Including External Declarations The mechanism for including external declarations is simple; upon encountering an import statement in the parsing phase of compilation, we search for the imported file. If the file exists, a new parser is instantiated and used to parse it, yielding a syntax tree. The declarations under this new syntax tree are merged with those from our existing syntax tree, giving a complete set of declarations. We also recursively process imports in the imported file, further expanding our list of declarations if necessary. Since Funky support out-of-order definitions (see Section 4.10.5), the method we use to merge the declarations is not important.

In simpler terms, the definitions from the imported file are directly copied to the required file. The advantage of this method is that the compiler does not need to make any special accommodations for the imports, since the syntax tree it is working with is as if the user had written all of the declarations in one big file. The disadvantage is that the syntax tree will be quite large if the imported file contains a significant number of declarations. This flaw is not too debilitating, thankfully, since after we have converted our syntax tree to the core tree via the process of desugaring (described in Section 4.7), we prune it, removing any unused definitions. This results in a small core tree representing only the practical code, which in turn means there is less work to do during type inference and the generated target code contains only what it needs.

Searching for an Import Any valid relative file path can be provided to an import statement. The Funky compiler reads the path and applies it relative to various search locations in an attempt to find the source file the programmer is referring to. At the moment, the search locations are (in order of priority):

- The directory containing the file currently being compiled.
- The funky `libs` directory, which is included with a standard install.

The search locations are ordered based on priority. Search paths listed first are tried first. This means that, if a source file is successfully found in an earlier search location, it takes precedence over any files that might be found in lower-priority search locations. To give a concrete example: suppose that the user is compiling a file `main.fky` in a directory with a file called `stdlib.fky`. If they use the import `import "stdlib.fky"` within `main.fky`, the neighbouring file `stdlib.fky` will be preferred over Funky's actual standard library.

As mentioned, the import statement takes a relative file path. That means that a statement such as `import "libs/math/logic.fky"` would be a perfectly valid import, given that the file it refers to exists. Funky would use this path relative to the search locations.

In the future, it would be possible to implement package support. This could be achieved by adding new search locations.

Preventing Cyclic Imports Since we recursively process imports, it is possible that source file A imports source file B, and source file B imports source file A. This is a cyclic import; both files import one another in a 'cycle'. This is problematic because, in the absence of any mechanism to recognise that a file has already been imported, file A would import B, which would import A, and so on.

The solution is simple; we maintain a set of file paths representing those files which have already been imported and do not require further processing. Upon re-encountering a file during the import process, we then know to ignore it.

As well as permitting cyclic imports, this mechanism means that importing is idempotent. Importing a file multiple times is the same as importing it once.

Summary In this section, we have discussed the mechanisms employed by Funky's import system to include functions and variables from different files. Funky's import system makes programming far more convenient, since programs can be broken up across multiple files, code duplication is reduced, and library modules can be implemented.

This phase modifies the syntax tree in-place by including the definitions from the imported files. The result, therefore, is simply a larger syntax tree. This sounds inefficient, but unused declarations are pruned later on in compilation, so there is no significant overhead to using import statements.

The `--dump-imports` command line flag can be used to output the parse tree with imports included to `stdout`.

4.5 Fixity Resolution

Functional programming languages differ in many ways from the pure/simply-typed lambda calculus. Syntactic sugar is often employed for convenience. Infix expressions are an example of this. They are commonplace in functional programming languages, but are ultimately just syntactic sugar for function application.

For simplicity, we want to represent all kinds of expressions in the same manner. The easiest way to do this is to convert infix expressions to a sequence of function applications. The conversion is from infix to prefix. For instance, the expression `x + y` is converted to `+ x y`, which is read as 'application of the `+` function to two arguments `x` and `y`'. By converting all infix expressions in this manner, the standard machinery for managing function applications can be reused.

However, this operation is not as trivial as it initially appears because of operator precedence and associativity. This must be accounted for in the conversion.

The process of converting an infix expression to a sequence of function applications is called *fixity resolution*. The parser runs fixity resolution on an as-needed basis when it encounters infix expressions. The fixity resolution code in Funky expects a simple Python list of tokens and produces the corresponding sequence of function applications. For instance, the flat infix expression list `1 + 2 * 3` would be converted to `+ 1 (* 2 3)`.

To ensure that the order of operations is preserved in an infix expression, it is crucial for the resolver to know the precedence and associativity of each operator. Each operator is assigned a precedence value (where a higher number means greater precedence) and an associativity (specifying right associativity, left associativity, or no associativity). By default, Funky's built-in operators have the same associativity and precedence as they do Haskell, which is fairly standard. They are tabulated in Table 2.

Precedence	**Left-Associative**	**Right-Associative**	**Non-Associative**
8		`**`, `^`	
7	`*`, `/`, `%`		
6	`+`, `-`		
5		`++`	`==`, `!=`, `<`, `<=`, `>`, `>=`
4		`and`	
3		`or`	

Table 2: Operator precedence and associativity in Funky.

4.5.1 Fixity Resolution Algorithm

A Haskell implementation of the algorithm for resolving fixities is given in Marlow et al. (2010), Section 10.6. It is described in imperative pseudocode below.

```
function RESOLVE_FIXITY($tokens$)
    return first element of $parse\_neg$('!!!', $tokens$)
end function
function PARSE_NEG($operator, tokens$)
    if first element of $tokens$ is '-' then
        $fix1 \leftarrow$ fixity of $operator$
        $prec1 \leftarrow$ precedence of $operator$
        if $prec1 \geq$ precedence of '-' then
            error: invalid negation
        end if
        $tokens \leftarrow$ TAIL($tokens$)
        $(r, rest) \leftarrow$ PARSE_NEG('-', $tokens$)
        return PARSE($operator$, $FunctionApplication$("$negate$", $r$), $rest$)
    else
        $tok \leftarrow$ HEAD($tokens$)
        $tokens \leftarrow$ TAIL($tokens$)
        return PARSE($operator, tok, tokens$)
    end if
end function
function PARSE($op1, exp, tokens$)
    if $tokens$ is empty then
        return $(exp, [\,])$
    end if
    $op2 \leftarrow$ HEAD($tokens$)
    $fix1 \leftarrow$ fixity of $op1$
    $prec1 \leftarrow$ precedence of $op1$
    $fix2 \leftarrow$ fixity of $op2$
    $prec2 \leftarrow$ precedence of $op2$
    if $prec1 == prec2$ and ($fix1 \neq fix2$ or $fix1 ==$ "nonassoc" ) then
        error: illegal expression
    end if
    if $prec1 > prec2$ or ($prec1 == prec2$ and $fix1 ==$ "leftassoc" ) then
        $tokens[0] \leftarrow op2$
        return $(exp, tokens)$
    end if
    $tokens' \leftarrow$ TAIL($tokens$)
    $(r, rest) \leftarrow$ PARSE_NEG($op2, tokens'$)
    return PARSE($op1$, $FunctionApplication(FunctionApplication(op2, exp), r), rest$)
end function
```

To resolve the fixity of a given infix expression, one would call the `resolve_fixity` function with a flat list of expression tokens. An example of such a list might be `[1, +, 6, *, 2]`. In general, input to the `resolve_fixity` takes the form of E_0 op_1 E_1 op_2 ... E_{n-1} op_n E_n.

The first significant point to note is that the string of tokens passed to the fixity resolution algorithm never contains bracketed sub-expressions. In other words, the fixity resolver never receives a list such as `[1, +, (, 2, *, 2, )]`. If a nested sub-expression is encountered by the parser, it will first resolve the fixity of that sub-expression, and the result (a tree of function applications) will be encoded into the enclosing expression list. For example, resolution of the expression `1 + (2 * 2)` would involve first resolving the sub-expression `2 * 2`, whose list of tokens is `[2, *, 2]`, to give `* 2 2`, then resolving the enclosing expression, whose list of tokens is now given as `[1, +, (* 2 2)]`.

This bottom-up process of resolving fixities follows quite naturally from the syntax-directed parsing process, since the nonterminal `EXP` in Funky's context-free grammar can ultimately reduce to a form that yields a nested `EXP`. Therefore, the solution is to perform fixity resolution whenever we reduce to an `EXP` nonterminal during parsing. This is embodied in the parser code as shown in Listing 7.

```
def p_EXP(self, p):
    """EXP : INFIX_EXP
    """
    p[0] = p[1]
    p[0] = fixity.resolve_fixity(p[0])
```

Listing 7: Expressions have their fixities resolved when they are encountered.

Notice that the `resolve_fixity` function immediately delegates to `parse_neg`, giving it an 'imaginary operator'. This 'imaginary' operator is not accessible in regular Funky programs, and its purpose is exclusively to 'kick off' the fixity resolution process. To achieve this, its precedence is set to be the lower than all other operators, forcing the `parse` function to consume the input in its entirety.

Before we discuss the purpose of `parse_neg`, it is beneficial to first familiarise ourselves with the `parse` function. The `parse_neg` function accounts for the special case where our tokens might contain a unary negation – indeed, it is true that the `parse_neg` function does exactly the same thing as `parse` as long as the first token in the list is not '`-`'. It is instructive to properly understand the general case, `parse`, before we explore the special case.

At each stage, we have a call to the `parse` function. At stage i, we will be looking at an expression E_i op_{i+1} E_{i+1} op_{i+2} ..., where the value E_i is held by the caller. The job of the `parse` function is to build the expression E_{i+1} op_{i+2} ..., returning it in terms of `FunctionApplication`s alongside the remaining infix expression. By inspecting the pseudocode, it should be clear that this is done recursively.

The `parse` function considers three cases:

1. If op_{i+1} and op_{i+2} have the same precedence, but have different associativities *or* are non-associative, the expression is illegal and we throw an error. This is because, if the operators have the same precedence but different associativities, the order of operations is undefined. Similarly, if the operators are non-associative, it means that their behaviour is undefined when used in sequence, which means the current context the `parse` function is in does not make sense. For example, the expression $x == y == z$ would trigger this condition because `==` is non-associative.
2. If op_{i+1} has greater precedence than op_{i+2} or op_{i+1} and op_{i+2} should associate to the left, we simply return the prefix expression we've built up so far to the caller alongside what remains to be processed; we know that the expression to the right of op_{i+1} is just E_{i+1}.
3. If neither of the above conditions hold, then it follows that both op_{i+1} and op_{i+2} associate to the right. This means that we want to build up the prefix representation for E_{i+1} op_{i+2} ... E_j. To compute this, we call `parseNeg`, passing it the arguments op_{i+2} and the tokens following on from op_{i+2}. The work that we have done so far essentially gives us E_i op_{i+1} (E_{i+1} op_{i+1} ... E_j) op_{j+1} .... This can be simplified to E_i op_{i+1} E_{i+1} op_{j+1} ... where the new E_{i+1} is (E_{i+1} op_{i+1} ... E_j), which is of the same form as described earlier. We can then recursively call `parse` again.

`parse_neg` handles unary negation, e.g. `-6 + 2`. The rule is simple: first, we check if the tokens actually begins with a '`-`'. If it does not, we can simply delegate the function call to `parse`. If it does, we compare the precedence of the unary negation operator with the precedence of the operator currently under consideration. If the operator currently under consideration has the same or a greater precedence, we raise an error – this arises in expressions such as `10 + -5` or `5 ** -2`, but not in `-5 + 10` or `-2 ** 5`. Explicit brackets are required around the '`-`' in these kinds of situations. Otherwise, if the expression is valid, we proceed in a similar manner to case 3 described above.

Worked Example Let us work through an example of fixity resolution for insight. Suppose that the parser has just encountered the expression `-x + 2`. The tokens, `[-, x, +, 2]`, are passed to the function `resolve_fixity`, which immediately delegates the call to `parse_neg`, passing in the 'imaginary operator' and the token list.

In parse_neg, we notice that the first operator in the token list is indeed '-'. We compare the precedence of '-' with the imaginary operator and find that our negation is valid. So, we recursively call parse_neg with all but the first token, which is this time delegated to just parse (since the tokens no longer begin with '-'), and find that case 2 is satisfied, since '-' and the next operator along, '+', have the same precedence and are both left associative. This invocation of parse returns (x, [+, 2]), which the callee sees and now knows that the '-' binds to x, with [+, 2] still to be resolved.

Finally, in the initial invocation of parse_neg, an additional recursive call is made to parse with the information that the -x part is already resolved. This time, case 3 is satisified, so a recursive call to parse_neg is performed. This loops right back around to parse again, to immediately return '2'. The callee now knows the way to construct the function application, so it does so, and recursively calls parse one last time. We hit the base case again, which returns our generated expression in its entirety.

In this case, the result is + (- x) 2. The algorithm is quite hard to follow in plain English and the reader would benefit from following through a trace of execution with pen and paper.

4.5.2 Fixity Declarations

It is possible to change the fixity of operators using the setfix directive. At parse-time, encountering a fixity declaration modifies the appropriate fixity rule. Any subsequent fixity resolution then applies the new rules. In other words, using this directive, the programmer can change the precedence and associativity of built-in operators.

This directive could be useful in certain niche applications. For example, the associativity of the exponentiation operator (**) tends to vary between computer systems and programming languages. C++, for instance, treats exponentiation as a left-associative operator, whereas in Python it is right-associative. In Funky, the programmer has the freedom to choose, as illustrated in Listing 8.

```
x = 2 ** 2 ** 3          # = 256 (Python-like behaviour)
setfix leftassoc 8 **
y = 2 ** 2 ** 3          # = 64 (C++-like behaviour)
```

Listing 8: Changing the associativity of exponentiation

Summary In this section, we have seen how infix expressions are just syntax sugar for prefix function application. The process of fixity resolution is converting these infix expressions to their prefix equivalent, accommodating for operator precedence and associativity. By doing this, we can re-use the mechanism for applying functions to evaluate infix expressions. In other words, we avoid having to write a translator for infix expressions *and* function applications by reducing all infix expressions to just instances of function applications.

Fixity resolution modifies the syntax tree in place.

4.6 Renaming

Renaming is carried out immediately after parsing, and involves renaming all user-named items in the code with machine generated names. It is a hygiene step; it is not strictly necessary, but it drastically simplifies stages in the latter parts of the compilation pipeline.

Renaming is carried out by, quite fittingly, the *renamer*, which traverses the entire source tree and renames all user-named items. In doing this, we also eliminate name shadowing. The renamer ensures that every distinct item in the code is labelled uniquely, and there is never ambiguity in what an identifier refers to.

This is invaluable for a number of reasons:

1. Ensuring that all names are unique in the source tree is significantly *safer*, because later on it is good to know that there is no risk of implicit name capture and that the program can be transformed without changing its meaning.
2. By assigning a machine generated name to each variable, the user's choice of variable names cannot possibly interfere with the generated code at the end of compilation. This issue only really arises in unique or contrived cases, but consider if the user had created a function in Funky with the label 'global' and wanted to compile to Python. If this was name was carried through to the code generator, it would end up trying to define a function called 'global', which is a Python keyword. By renaming the variables early, we avoid these sorts of conflicts.

4.6.1 Renaming the Syntax Tree

Renaming the code is simply a process of traversing the syntax tree and changing the identifiers of the relevant nodes. At the core of this is a scope object, which maintains a hierarchy of contextual information about the code. The traversal starts from the root node and progresses downwards.

The scope keeps a mapping from user-given names to their machine-generated counterparts. Upon encountering a user-given name in the syntax tree, a fresh name is generated, and an entry is added into the scope associating the two. Any subsequent encounters of a variable with the same name will be replaced with the same machine generated name, ensuring that the renamed code is consistent.

However, some additional consideration is required where there are nested lexical scopes. For example, if we apply the process described above to a line such as `f x = (lambda x -> x + 1) x`, the output of the renaming might resemble `v0 v0_1 = (lambda v0_1 -> v0_1 + 1) v0_1`, which does *not* eliminate name shadowing. We would hope that the output would resemble `v0 v0_1 = (lambda v1 -> v1 + 1) v0_1` instead, so that every variable is labelled uniquely. To solve this problem, our scope data structure must understand context. It must understand that certain constructs in the code create a nested scope and that variable names within this may override those declared in some outer context. This implies that our scope must have a hierarchical structure.

The idea is that, upon encountering a program construct that sets up a new lexical scope (e.g. a lambda or function definition), we add a new nested namespace to our existing scope. Each time we encounter a variable and need to rename it, we search our nested scopes upward toward the global scope until we reach a mapping that associates the user-named variable with its machine generated name. If no such mapping exists, we create one at the most-nested level. This means that the renamer will always rename a variable such that it refers to the most-local binding, which is the expected behaviour for lexically scoped names.

This ensures that every variable in the syntax tree has a unique name, as well as ensuring that the rules of lexical scoping are preserved.

The example below demonstrates how the renamer reassigns identifiers in the syntax tree. Notice that the name shadowing is eliminated, and all identifiers unambiguously refer to a single variable.

```
f x y = (lambda x -> x + y) x
        with y = 4
```

$\Longrightarrow$

```
v0 v0_0 v0_1 = (lambda v2 -> v2 + v1) v0_0
               with v1 = 4
```

4.6.2 Sanity Checks

In addition to re-labelling variables, the renamer also performs some first-level sanity checks on the code to verify that nothing is obviously wrong. The list below enumerates some of these checks:

- Checking that there are no references to a variables that don't exist.
- Checking that function definitions in implicit pattern matching always have the same number of parameters.
- Checking that parameters to a function have unique names.

- Checking that the code does not attempt to override a built-in name.
- Checking that there are no duplicate definitions of constructors for an algebraic data type.

Most of these are trivial checks. For instance, checking that function definitions always have the same number of parameters is just a matter of remembering the number of parameters used in a function definition and flagging an error if any later definitions under the same name have a different number of parameters.

What is slightly more involved is checking that there are no references to variables that do not exist. Since Funky supports out-of-order definitions (see Section 4.10.5), we cannot use the intuitive check of 'if a name is used that hasn't been previously defined, flag an error'; the name may be defined later. A bit more thought is required.

As it turns out, the solution is not all that complex. Upon encountering a variable that has not been previously defined, we add its name to a list of names that are pending a definition. When we find a definition for the name, we remove it from this list. If the list is non-empty by the end of the renaming process, we can be confident that the programmer has used a variable which has not been defined, so we can report the error. This has the nice side-effect that the names of any undefined variables are present in this list, so we can report exactly which variables are missing definitions to the user.

Summary In this section, we have provided a compelling case for wanting to rename all user-named items in the syntax tree with machine generated names. There are two main reasons: we avoid name shadowing, ensuring that all identifiers refer to *one* unique object, and we ensure that the user's choice of variable names is free of side-effects.

We have also shown how renaming allows us the opportunity to perform some basic sanity checks on the code. These sanity checks allow us to verify that nothing is obviously wrong – i.e. there are not references to variables that do not exist.

We have explained the mechanism for renaming: traversal of the syntax tree whilst building up a hierarchical mapping between user-given names and their machine-generated counterparts.

The renamer changes the names of nodes in the syntax tree in-place.

The `--dump-renamed` command line flag can be used to output the mapping from top-level user-defined names to their machine generated counterparts, and the renamed parse tree, to `stdout`.

4.7 Desugaring/Intermediate Language

Funky's user-facing syntax is relatively complex. As a result, the source tree generated by the parser is complex too. Compiling to the target language directly from the source tree, while technically possible, would be arduous; it would be necessary to account for a lot of different programming constructs and it is likely that there would be many edge cases to consider.

For this reason, Funky's source tree is compiled down into an intermediate language consisting of just a few basic constructs. By reducing the complex source tree into a basic *core* tree, we create a concise representation of the program that is conducive for further processing. This ultimately means:

- Type inference is vastly simplified. In fact, the structure of the core tree is such that type inference is a syntax-directed process.
- Optimisation is significantly easier by virtue of the fact that the core tree is a simpler data structure to work with.
- The core tree is very close to the basic typed lambda calculus. It is easier to convert this to a target language than to write a translator for the entire source language.
- Changes can be made to the syntax of the language without having to touch the code generator. For instance, if a new feature is added to the language, we only need to write code to desugar that feature to the intermediate language. Since the code generators accept input in the intermediate language, they do not need to make special adjustments.

The intermediate language has a total of ten different constructs, described in Table 3.

Core Construct	Description
`CoreTypeDefinition`	A binding between an algebraic data type and its name. I.e. `newtype Maybe = Just Integer | Nothing` binds the ADT `Just Integer | Nothing` to the identifier `Maybe`.
`CoreBind`	A binding between some object to a name. For instance, `x = 12` is represented by a `CoreBind` where `x` is the bind identifier and `12` is the bindee.
`CoreCons`	A construction in the context of pattern matching.
`CoreVariable`	A variable.
`CoreLiteral`	A literal.
`CoreApplication`	Application of a function to an argument.
`CoreLambda`	A lambda expression. Note that this is not exclusively for *anonymous* lambda expressions – all functions are ultimately compiled into an instance of a `CoreLambda`.
`CoreLet`	A series of local bindings made available in an expression. Formally, this is a recursive let binding, since bindings within the let body can reference themselves and each other.
`CoreMatch`	A match statement used for conditional execution. A match statement describes which path of execution to take by comparing a scrutinee with a series of alternative literals/-patterns, named alts.
`CoreAlt`	An alternative for a match statement. This describes the condition required for this alternative to match, and the corresponding expression to return if it does.

Table 3: The ten core constructs in Funky's intermediate language.

The process of desugaring is quite involved. It could be argued that this is the 'main stage' of compilation. There are many different transformations to consider, and the confluence them all gives rise to a rather complex set of tree rewrites. One thing in particular that is remarkably subtle is the process of desugaring pattern matching, which is discussed in detail in Section 4.8.

The sections below describe the most important rewrites the desugarer performs on the source tree to create the core tree.

4.7.1 Rewriting a Module

A module in Funky consists of a number of top-level declarations and a main function. Modules are reduced to a `CoreLet` statement, where the body of the let is the top-level bindings and the let expression is the body of the main function. The example below illustrates this.

```
module example with
    decl1 = ...
    decl2 = ...
    ...
    main = x + y + z
```

$\Longrightarrow$

```
let
    decl1 = ...;
    decl2 = ...;
    ...
in x + y + z
```

4.7.2 Rewriting If/Else

If-else statements are desugared to a `CoreMatch`. Specifically, they are reduced into a `CoreMatch` with two alternatives, `True` and `False`, where the condition in the if statement becomes the scrutinee.

An example translation is presented below.

```
x if cond else y
```

$\Longrightarrow$

```
match cond on {True -> x; False -> y}
```

4.7.3 Rewriting Function Definitions

Function definitions are desugared to a series of `CoreLambdas`. For each parameter, a new `CoreLambda` is produced. This allows for currying, which is a very popular feature in functional programming. An example translation is below.

```
f x y = ...
```

$\Longrightarrow$

```
f = lambda x -> lambda y -> ...
```

This transformation still applies even when implicit pattern matching is used as the function parameters. The details of how pattern matching is desugared are elaborated on in Section 4.8, but in short, any literals/patterns found in the function's parameter list are assigned fresh variable names, and are unpacked once again in the function body using a match statement. The example below shows this in concrete terms.

```
implies True False = False
implies _    _     = True
```

$\Longrightarrow$

```
implies = lambda x ->
          lambda y -> match x on ...
```

4.7.4 Rewriting Lambdas

Lambdas in Funky permit multiple parameters – for example, `lambda x y -> x + y` is perfectly valid. In the pure lambda calculus, and therefore in Funky's intermediate language, this is not allowed. If we encounter lambdas like this, we must rewrite them as a string of lambdas, each of which having only one parameter. This is a trivial rewrite – an example can be found below.

```
f = lambda a b -> ...
```

$\Longrightarrow$

```
f = lambda a -> lambda b -> ...
```

4.7.5 Rewriting Guarded Functions

In Funky, guards are syntactic sugar that allows different bodies of a function to execute depending on whether a condition is true or false. They are very similar to an if statement, which is a useful insight – guards can be desugared to an if statement, which are subsequently desugared to a `CoreMatch` with a boolean scrutinee. In the Funky compiler, we cut out the middleman – guards are compiled directly to a tree of `CoreMatch`es which determine which expression will be returned based on some conditionals.

```
signum n given n <  0 = -1
         given n == 0 = 0
         given n >  0 = 1
```

$\Longrightarrow$

```
f = lambda n -> match n < 0 on
            True  -> 0
            False -> match n == 0 on ...
```

4.7.6 Rewriting With

The `with` keyword in Funky has a direct mapping to to the `let` keyword. The translation is simple, demonstrated below.

```
area r = pi * r ** 2.0
         with pi = 3.14
```

$\Longrightarrow$

```
area r = let pi = 3.14
         in pi * r ** 2.0
```

Summary This section has introduced the idea of the core tree as an intermediate language. The core tree closely resembles the pure lambda calculus, making it a relatively easy translation candidate for the code generator. We have also given a high-level overview of some of the transformations the desugarer applies to the syntax tree to create the core tree.

The desugarer accepts the syntax tree as its input, and produces the core tree as an entirely new data structure.

We have not yet described the entire desugaring process. One of the most important parts of desugaring is compiling pattern matching. Section 4.8 elaborates on this.

4.8 Pattern Matching

As mentioned previously, desugaring pattern matching is remarkably subtle. Before discussing details, refer to Listing 9 to see an example of the expected output from pattern matching desugaring.

```
xor  True   False  =  True
xor  False  True   =  True
xor  _      _      =  False

# is equivalent to

xor a b = match a on
            True -> match b on
                      False -> True
                      _     -> False
            False -> match b on
                      True  -> True
                      _     -> False
```

Listing 9: XOR using pattern matching.

There are a lot of edge cases to consider when desugaring pattern matching. These include:

- Multiple arguments to a function – we need to consider the 'best' order to test the arguments to make as few comparisons as possible.
- Literal patterns.
- Constructors/algebraic data types. These are 'nested' patterns in a sense – we must think about how matching against such items should be implemented.
- Overlapping patterns. If two patterns overlap, it means that one is unreachable because its pattern is not matched by anything that isn't previously handled by an earlier match. We have to think about how to handle this situation.
- Detecting potential non-exhaustive patterns. If it is possible to supply a set of arguments to a function that doesn't match any of the patterns, how do we detect and handle this?

The first step to solving these problems lies in Maranget's algorithm, detailed in Section 4.8.2. However, we must first collect the different equations of functions together so that they are in a form suitable for Maranget's algorithm to run.

4.8.1 Condensing Function Definitions

Funky supports implicit pattern matching. When defining functions, you can define separate bodies for different patterns. Listing 10 defines a function that returns the requested argument, specified as an integer, using implicit pattern matching. If a number that is out of range is specified, the first argument is returned.

```
get x _ _ 0 = x # <- strictly, this is optional, but left for clarity
get _ y _ 1 = y
get _ _ z 2 = z
get x _ _ n = x
```

Listing 10: Funky's implicit pattern matching demonstrated with a basic function.

The earlier stages of desugaring would convert the code in Listing 10 into four `CoreBind`s, one for each of the separate definitions. This is not acceptable; there would then be four separate `CoreBind`s associating the identifier of `get` (as assigned by the renamer) with four separate `CoreLambda`s. This does not accurately represent what the programmer means when they declare a function in this manner.

To solve this, there is a second phase of desugaring that locates function binds with the same identifier and condenses them together, compiling them into a single function that is a decision tree (composed of `CoreMatch`es) representing the pattern matching. In other words, multiple definitions of the same function are condensed into a single `CoreLambda` representing the many different parameter configurations and function bodies as a decision tree.

However, there is one important edge case to consider with this second pass. Consider the functions `id x = x` and `id = lambda x -> x`. Functionally, they are equivalent, but it is worth noting that functions defined in the second notation can never use implicit pattern matching. However, when we desugar the code, the two functions are compiled down into the exact same `CoreLambda`, that is $\lambda x.x$. In this case, how are we to tell whether to attempt to condense the function binds if we cannot tell the notation that was originally used? The simple solution is to 'tag' `CoreLambda`s with the notation used to create them in the source code. Then, when condensing function definitions, we simply ignore anything declared with the `lambda` keyword.

The specific implementation is that `CoreLambda`s have a property `is_raw_lambda` which is set to `True` if and only if the function was defined using the keyword `lambda` explicitly.

4.8.2 Maranget's Algorithm

It is clear that pattern matching needs to be reduced into an efficient decision tree so that we can compile it into target languages with imperative conditional constructs. The algorithm used to generate decision trees from pattern matching is dubbed Maranget's algorithm, and is described in Maranget (2008).

Now consider the function in Listing 11. This function is given as an example in Maranget (2008). The semantics of pattern matching are that we evaluate the clauses one at a time, from top-to-bottom and left-to-right, returning the first one that matches. This sounds trivial, but we have a surprising amount of flexibility due to wildcards.

```
f  _      False  True   =  1
f  False  True   _      =  2
f  _      _      False  =  3
f  _      _      True   =  4
```

Listing 11: Example of pattern matching.

Maranget (2008) focuses more on the theoretical implementation of the algorithm. What follows is a slower-paced description of the algorithm with examples to show how Funky resolves the matching.

The first step in Maranget's algorithm is to create a matrix to represent the pattern matching. Each column represents a parameter to the function, and each row represents the pattern that returns a particular outcome.

$$\begin{array}{c c}
\begin{array}{ccc} x & y & z \end{array} & \\
\begin{pmatrix} \_ & False & True \\ False & True & \_ \\ \_ & \_ & False \\ \_ & \_ & True \end{pmatrix} & \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array}
\end{array}$$

Notice that for the first line to succeed we never even need to consider the first column. This is important; Maranget's algorithm produces optimal decision trees by finding the columns that must be tested and testing them first. To achieve this, we score the columns to determine what to test next.

The score of a column is defined as the vertical distance from the top until the nearest wildcard/variable. The column that maximises this score is the one that we should test next. In the above matrix, the columns have scores 0, 2, and 1 respectively – therefore, the second column should be tested first.

We now modify the pattern matrix by swapping the test column with the very first column. This gives us:

$$\begin{array}{c c}
\begin{array}{ccc} y & x & z \end{array} & \\
\begin{pmatrix} False & \_ & True \\ True & False & \_ \\ \_ & \_ & False \\ \_ & \_ & True \end{pmatrix} & \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array}
\end{array}$$

It is now time to generate the first split in the decision tree. We try to match the parameter for the first column with the value in the top-left element of the matrix. If the match is successful, we branch in one direction. If the branch is not successful, we branch in the other.

To determine what happens further along each of these branches, we compute two additional matrices: a specialised matrix, representing the matrix that remains if we've made a successful match, and a default matrix, which is the matrix we get if we don't find a match. We then recursively apply Maranget's algorithm to these matrices to generate nested decision tree for the 'match' and 'no match' outcomes.

To compute the specialised matrix, assume that we match on the top-left element of the modified pattern matrix. In this case, we assume we match on `False`. First, we remove all rows whose first element cannot be reconciled with this match. Specifically, any rows whose first value is not explicitly `False`, and whose first value is *not* a wildcard/variable, are to be removed. Next, we remove the first column from the pattern matrix since we're assuming we've already successfully matched for this column. This gives us us our final specialised matrix. In this case, we remove only the second row, since `True` is not compatible with `False`, and drop the first column. This leaves us with:

$$\begin{array}{c c}
\begin{array}{cc} x & z \end{array} & \\
\begin{pmatrix} \_ & True \\ \_ & False \\ \_ & True \end{pmatrix} & \begin{array}{c} 1 \\ 3 \\ 4 \end{array}
\end{array}$$

To compute the default matrix, assume that we did *not* match on the top-left element. This involves removing all rows from the pattern matrix whose first value is the equal to the top-left element. This makes intuitive sense; if we're assuming we didn't match on the top-left item, then we cannot match on these either.

In this case, we only remove the first row, leaving us with:

$$\begin{array}{c} \begin{array}{ccc} y & x & z \end{array} \\ \left(\begin{array}{ccc} True & False & \_ \\ \_ & \_ & False \\ \_ & \_ & True \end{array}\right) \end{array} \begin{array}{c} \\ 2 \\ 3 \\ 4 \end{array}$$

Once we have both the specialised and default matrices, we recursively apply Maranget's algorithm to both of them to determine the further branches in the decision tree. The base case is when the top row of our matrix consists entirely of wildcards/variables – in this case, we return the upper-most outcome, since if there are any other outcomes, they have no chance of being matched since wildcards greedily match anything.

An actual run of the algorithm is quite involved. However, for sake of further insight, let us examine an extra recursive calls for the 'match' outcome with the specialised matrix. We recursively call Maranget's algorithm on the specialised matrix:

$$\begin{array}{c} \begin{array}{cc} x & z \end{array} \\ \left(\begin{array}{cc} \_ & True \\ \_ & False \\ \_ & True \end{array}\right) \end{array} \begin{array}{c} \\ 1 \\ 3 \\ 4 \end{array}$$

Once again, we score the columns. It is immediately obvious that the second column has the maximum score, so we swap it with the first column, giving:

$$\begin{array}{c} \begin{array}{cc} z & x \end{array} \\ \left(\begin{array}{cc} True & \_ \\ False & \_ \\ True & \_ \end{array}\right) \end{array} \begin{array}{c} \\ 1 \\ 3 \\ 4 \end{array}$$

Now assume we match on `True`. We drop the second row, since `False` is incompatible with `True`, and remove the first column. The specialised matrix is then:

$$\begin{array}{c} x \\ \left(\begin{array}{c} \_ \\ \_ \end{array}\right) \end{array} \begin{array}{c} \\ 1 \\ 4 \end{array}$$

Note that the above specialised matrix satisfies the base case for Maranget's algorithm. This tells us that, if the parameter y is `False` and the parameter z is `True`, the output of the function is *always* 1, without even having to consider the parameter x.

Similarly, the default matrix is:

$$\begin{array}{c} \begin{array}{cc} z & x \end{array} \\ \left(\begin{array}{cc} False & \_ \end{array}\right) \end{array} \begin{array}{c} \\ 3 \end{array}$$

Which still requires further processing to reach the base case, but regardless, the progression through the algorithm should hopefully be clear by now.

4.8.3 Accounting for Constructors

Maranget's algorithm requires some modification if we want to be able to pattern match over algebraic data types. Indeed, pattern matching is arguably most valuable in facilitating the easy manipulation of data structures, so omitting this feature would be unwise. In a sense, matching over an algebraic data type is like nested pattern matching – we can use this fact to our advantage.

Suppose that we are pattern matching over the function in Listing 12.

```
newtype Pair = P Integer Integer
either_zero (P x 0) = True
either_zero (P 0 y) = True
either_zero _       = False
```

Listing 12: A function to check whether either element of a pair is zero.

Our pattern matrix is:

$$\begin{array}{cc} a & \\ \left(\begin{array}{c} \texttt{(P x 0)} \\ \texttt{(P 0 y)} \\ \texttt{\_} \end{array}\right) & \begin{array}{c} \texttt{True} \\ \texttt{True} \\ \texttt{False} \end{array} \end{array}$$

Following the usual algorithm, the first step is to sort the colums by score. Here, we do not need to do anything; there is only a single column. Next, we compute our specialised and default matrices. To compute the specialised matrix, we assume we match on the top-left element of the pattern matrix. In order to match the contents of the pair, we transform the matrix into a form where Maranget's algorithm can work. As it turns out, this is very easy – if the top-left element of the pattern matrix is ever a constructor, we 'expand' the constructor inside the pattern matrix and repeat Maranget's algorithm on the result.

In this case, the result of the expansion would be:

$$\begin{array}{cc} \begin{array}{cc} a_1 & a_2 \end{array} & \\ \left(\begin{array}{cc} \texttt{x} & 0 \\ 0 & \texttt{y} \\ \texttt{\_} & \texttt{\_} \end{array}\right) & \begin{array}{c} \texttt{True} \\ \texttt{True} \\ \texttt{False} \end{array} \end{array}$$

Maranget's algorithm can continue as normal. This extension generalises nicely; if there are deeper-nested constructions, the same technique works. We just run the algorithm as normal until we need to specialise/generalise on a construction, then we perform the expansion.

Summary At the start of Section 4.8.2, we listed five challenges that needed to be sovled to desugar pattern matching. These were: multiple arguments to a function, literal patterns, constructors and algebraic data types, overlapping patterns and non-exhaustive patterns. This section summarises how we have solved these problems using Maranget's algorithm.

Our first challenge, multiple arguments to a function, concerns the best order in which to consider the function's arguments to make pattern matching as efficient as possible. As we have seen, Maranget's algorithm uses a scoring criteria to determine the column to test first, which allows us to make a smaller number of branching decisions, since we only test what is necessary. This ultimately results in a more efficient decision tree.

Our second challenge is literal patterns. As we have seen, these are handled quite naturally by Maranget's algorithm. When specialising with a literal, we know to remove all rows which cannot be reconciled with our assumption that we have made a succesful match – this is a simple matter of removing any rows with a different literal value in their first column. Generalisation is the same, but this time we remove all rows with the same literal value in their first column. In any case, it is a simple matter of literal comparison.

Our third challenge is constructors and algebraic data types. We have discussed this in detail in Section 4.8.3.

Our fourth challenge is overlapping patterns. Recall that this is where one pattern will never be satisifed, because anything that would match it is already handled by an earlier match. Funky's implementation is to simply throw these away. If a pattern is unfulfillable, we assume that it was never even given in the first place. This has some interesting consequences, especially with regards to type inference. Specifically, it is possible to declare a function that looks like it should be obviously type-incorrect, but it is actually type-correct because the compiler throws away its overlapping patterns. Consider the code in Listing 13.

```
newtype List = Cons Integer List | Nil
concatenate (Cons x xs) ys  = Cons x (xs ~concatenate~ ys)
concatenate xs          Nil = xs
concatenate Nil         ys  = ys
concatenate Nil         Nil = "garbage"
```

Listing 13: Overlapping patterns in Funky.

On reading this code, it looks like it should be *obviously* type-incorrect. Similar code in Haskell would produce a type error. However, in Funky, the line `concatenate Nil Nil = "garbage"` is overlapped by the two patterns above it! The desugarer then strips this line away, leaving only the first three type-correct patterns. The Funky compiler will not complain if it encounters this code, despite how incorrect it may appear!

Finally, our fifth challenge is inexhaustive pattern matching. Funky will log 'potential' inexhaustive pattern matching when it detects that one or more outcomes in a decision tree lead to the `None` in Python. This is guaranteed to report inexhaustive pattern matches, but it is still only a heuristic; there are situations where it will report a false positive. Beyond this, there are no special provisions for detecting inexhaustive matches.

Inexhaustive pattern matches are instead accounted for at runtime. For example, the Python code generator defines a special exception `InexhaustivePatternMatchError` that is raised if there is no pattern matching a particular scrutinee at runtime.

The `--dump-desugared` command line flag can be used to output a prettified representation of the core tree to `stdout`.

4.9 Optimisation/Pruning the Core Tree

So far, we have done nothing special to the core tree to optimise it other than reducing pattern matching to efficient decision trees using Maranget's algorithm. One particular concern when writing a compiler is ensuring that the compiled code is as compact as possible. This is done for both time and space efficiency: the smaller the compiled code, the faster it will run (generally) because there is less code overhead that the interpreter/compiler must account for, and naturally, smaller code occupies less space on the computer's secondary storage.

It is therefore in our best interest to minimise the code produced. There are a number of techniques that we can use to achieve this. One of these techniques involves pruning the core tree of any unused declarations so that they are not included in the generated code. This is precisely what the Funky compiler does after desugaring.

The `main` variable defines what exactly is computed when a Funky program is run. Anything that `main` is dependent on *might* be used, so it must be kept in the core tree. However, if we can show that there is no dependency on a given declaration from `main`, it can obviously be removed; it is guaranteed to never be used. By removing unused declarations, we drastically reduce the size of the target code.

The idea is simple; we build up a dependency graph between all of the declarations in the code, including `main`. Any nodes in this dependency graph that are not reachable from `main` can be discarded, leaving us with only the required declarations. The process is simple, yet effective.

In fact, we can further generalise this process. Recall that the `main` variable in a Funky source file is desugared into a single global let – that is, `x = 2; main = x` is desugared to `let x = 2 in x`. With this insight, we realise that we can construct a dependency graph for every `CoreLet` in the code, thereby allowing us to prune unused declarations within let statements. This generalisation works in our favour; the dependency graphs generated here are augmented to account for the pruning (removing all pruned nodes), and permanently associated with the `CoreLet` object. They are later re-used to reorder the bindings (permitting out-of-order declarations) and in Tarjan's algoritm to find strongly connected components (necessary for type inference). This is discussed in more detail in Section 4.10.6.

Summary We have shown how the compiler removes unused bindings, resulting in smaller and therefore more efficient code. This involves dependency analysis – we create a dependency graph for the core tree and find which bindings the `main` method is dependent on. We can safely remove everything else.

4.10 Type Inference/Checking

The purpose of this stage of compilation is twofold:

- To infer the type of all entities in the code before code generation. This is necessary when compiling to strongly-typed target language, where the type of every variable must be declared at compile-time.
- To verify the type safety of the program at compile time – i.e. to ensure that the types of all entities in the code are consistent with one another and do not produce undefined behaviour. If a program passes the type checker, it is guaranteed to not crash[5] at runtime.

The observant reader may have noticed that type inference/checking occurs *after* desugaring in the Funky compiler. This is fairly typical for a compiler, but there is some contention in the functional programming community about whether type inference/checking should be done before or after reducing to an intermediate language. While one is not clearly better than the other, it is worth noting the following trade-offs:

- Type checking before desugaring means that the type checker must examine Funky's syntax tree directly, so the type checker has many cases to consider. By desugaring into an intermediate language before type checking, we significantly reduce the size and complexity of the type checker since there are far fewer cases to account for.
- However, desugaring before type-checking unfortuntely introduces the obligation that desugaring must preserve the set of which programs are type-correct. This is a problem because desugaring is inherently a lossy process.
- Desugaring is a one-way process. So, if we encounter a type error when checking a desugared program, it can be difficult to map this error back to the original source code to tell the programmer where they went wrong.

Given the limited time available for the project, the most appropriate choice for the Funky compiler was to have type-checking occur *after* desugaring. To summarise the above, this means Funky has reasonably concise type-checking code at the expensive of slightly worse error reporting.

Funky uses the Hindley-Milner type system for type-correctness. The inferencing algorithm used is commonly referred to as Algorithm W. This algorithm (or a derivative thereof) is used in both the ML family of languages (OCaml, F#, etc) and in Haskell.

[5]The term 'crash' here has a formal definition that includes hard crashes like segmentation faults. It does not include things like pattern-matching failure due to inexhaustive pattern matching, for example.

4.10.1 Hindley-Milner

The Hindley-Milner type system is a classical type system for the lambda calculus with parametric polymorphism. It was first described by J. Roger Hindley in Hindley (1969), and later revisited by Robin Milner in Milner (1978). The method was explored deeply through formal analysis by Luis Damas in Damas (1985), therefore the method is sometimes referred to as Damas-Hindley-Milner.

It is extremely common in functional programming languages and has been extended in various ways since its inception, most notably with type classes.

Informally speaking, Hindley-Milner is a formalisation of the intuition that a type can be deduced by the functionality it supports. It is not an 'algorithm', but rather a mathematical system allowing us to map types to arbitrary expressions in the lambda calculus. It defines:

- A formal way to discuss expressions and types. This allows us to make statements such as 'expression e has type t', written $e : t$.
- Some rules allowing us to derive the type of an expression given other type assumptions.
- A formal way to write expressions. This is just the lambda calculus.

First, let us formalise the notion of a type. Of course, the primitive types included in Funky, such as `Integer` and `Float`, are immediately valid. In addition, we define a rule for function types such that functions from one type to another are possible. We define this inductively: if t_1 and t_2 are types, then so is $t_1 \rightarrow t_2$. The grammar for valid types is therefore:

$$T ::= Integer \mid Float \mid ... \mid T \rightarrow T$$

Algebraic data types, which are defined by the user, are also available as types.

Now, we formalise the type inference rules. These rules define how we can go from some knowledge of expressions and their types to inferring types of more expressions. The standard Hindley-Milner type system defines six such rules, listed below:

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash_D x : \sigma} \quad [Var]$$

$$\frac{\Gamma \vdash_D e_0 : \tau \rightarrow \tau' \quad \Gamma \vdash_D e_1 : \tau}{\Gamma \vdash_D e_0 e_1 : \tau'} \quad [App]$$

$$\frac{\Gamma, x : \tau \vdash_D e : \tau'}{\Gamma \vdash_D \lambda x.e : \tau \rightarrow \tau'} \quad [Abs]$$

$$\frac{\Gamma \vdash_D e_0 : \sigma \quad \Gamma, x : \sigma \vdash_D e_1 : \tau}{\Gamma \vdash_D \text{let } x = e_0 \text{ in } e_1 : \tau} \quad [Let]$$

$$\frac{\Gamma \vdash_D e : \sigma' \quad \sigma' \sqsubseteq \sigma}{\Gamma \vdash_D e : \sigma} \quad [Inst]$$

$$\frac{\Gamma \vdash_D e : \sigma \quad \alpha \notin free(\Gamma)}{\Gamma \vdash_D e : \forall \alpha.\sigma} \quad [Gen]$$

The mathematical notation can be disconcerting. To paraphrase:

- The [Var] rule is simple and somewhat elementary. It states that if our type assumptions include 'x is of type σ', then we can make the inference that x is indeed of type σ.

- The [App] rule concerns application of a function to some argument. It states that if we can infer 'e_0 is of type $\tau \to \tau'$' from our existing set of type assumptions, and we can also infer that 'e_1 is of type τ', then the result of applying e_0 to e_1 is τ'. This makes intuitive sense if you consider the semantics of function application – it is almost saying 'a function applied to the correct input type gives the correct output type'.
- The [Abs] rule concerns lambda abstractions, or the definition of a function. It states: if it is true that given our existing type assumptions and the assumption that a variable x has type τ, we can infer that some variable e has type τ', we can also make the inference that a function that accepts x as an argument and returns e has type $\tau \to \tau'$.
- The [Let] rule concerns let bindings. It states: if, given our existing type assumptions, we can infer that e_0 has type σ, and also, given our existing type assumptions and the assumption that a variable x has type σ we can infer that a variable e_1 has type τ, then the result of the expression 'let $x = e_0$ in e_1' has the type τ. To summarise, the type of a let expression is the type of its body computed after adding the definitions to our type assumptions.
- The [Inst] rule concerns type variable instatiation, which we will discuss further in Section 4.10.3.
- The [Gen] rule concerns polymorphic generalisation, which we will discuss further in Section 4.10.3.

We must now find an algorithm to put these rules into action in order to infer types for the variables in our code. This is achieved by a process known as *unification.* In unification, the types for a well-structured program yield a set of constraints that, when solved, always have a unique principal type. In other words, if the code is well-typed, the constraints will yield an *unambiguous* type. If the expression is not well-typed, then one or more constraints are contradictory or unsatisfiable given the available types.

In Funky, Algorithm W is used to infer types, as explained in Section 4.10.2.

4.10.2 Algorithm W

Algorithm W is an efficient type inference algorithm that performs in almost linear time with respect to the size of the source. This makes it capable of to typing large programs in reasonable time. It is fairly standard in functional programming languages.

The implementation in Funky is as follows: the type-inferencer maintains a *context* or environment mapping variable identifiers in Funky's core tree to their type. To begin with, a 'default context' is used which defines the built-in types and type signatures of built-in functions, i.e. `+`. We recursively traverse the core tree from top-to-bottom, adding the types of variables in the code to the context as we go.

The actual inference code is guided by the Hindley-Milner inference rules detailed in Section 4.10.1. For instance, upon encountering a `CoreApplication` during traversal of the core tree, Funky's type inferencer performs the appropriate operations to reflect the [App] rule in Hindley-Milner.

The most basic 'unit' involved in the inference process is a *type operator.* These are simple n-ary type constructors which allow us to create things like algebraic data types. Built-in types in Funky are simply nullary type operators.

Type variables are used as convenient placeholders for when the type of something cannot be known at this immediate point in time (or for polymorphism, as we will see in Section 4.10.3). For example, when considering a function application, we must first compute the type of the function, then the type of the argument, and check that they coincide. A type variable will be used as a placeholder for the function type before its *actual* type is known. Later, when the type is discovered, we can instantiate the type variable to its actual type. Errors arise when a (non-generic) type-variable is instantiated to two unreconcilable types at two different points in the core tree – this means that some type rule has been violated and the code is not type-safe.

The core of the type-inferencer is a function called *unify.* This takes two types and tries to reconcile them with each other. In type-correct code, unification will simply equate types, instantiate type variables, and build up the context with more information about types. However, in badly-typed code, it is inevitable that we eventually try to unify two types that cannot be reconciled. This will raise an error.

The process of unification is relatively simple, and pseudocode is presented below.

```
function UNIFY($a, b$)
    if $a$ is a type variable then
        if $a \neq b$ then
            if $a$ occurs in $b$ then
                error: recursive unification
            end if
            instantiate $a$ to $b$
        end if
    else if $a$ is a type operator and $b$ is a type variable then
        UNIFY($b, a$)
    else if $a$ and $b$ are both type operators then
        if $a$ and $b$ have the same type name then
            for all $x, y$ in (a's types, b's types) do
                UNIFY($x, y$)
            end for
        else
            error: type mismatch
        end if
    end if
end function
```

In summary, the basic operation of Algorithm W is as follows: we recusively descend the core tree, inferring the type of each construct in the code as we go. We build up a set of type constraints from an initial set of defaults (representing the types of builtins) and use these to gradually build up a clearer picture of the types in the program. If a contradiction is found at any point (by unifying two unreconcilable types), the code is not type-safe.

4.10.3 Polymorphism

Certain functions are always of a specific type. For example, the `not` function which takes a boolean and returns its negation has the type $not : Bool \rightarrow Bool$. However, certain functions are generic and don't necessarily care about the type of their argument. The trivial example would be the identity function, `id`, which immediately returns whatever is given to it. We could introduce different functions for each different type we might want to perform `id` on, such as $id_{Integer} : Integer \rightarrow Integer$, $id_{Bool} : Bool \rightarrow Bool$, etc, or we could introduce universal quantification into our type inferencer so that we can work with types such as $id : \forall a \ . \ a \rightarrow a$.

This adds a great deal of flexibility to any functional programming language by sparing the programmer from having to write different functions for each possible type. Funky implements parametric polymorphism.

The implementation of parametric polymorphism in the Funky compiler is as follows: we maintain a set of all type variables known to be 'non-generic'. Type variables are added to this set when we have inferred a concrete type (i.e. $Integer$, $Bool \rightarrow Integer$) for them and they cannot be generalised. Any variable *not* inside this set thus *can* be generalised. This directly follows from Hindley-Milner's [Gen] rule.

If a type variable is generalised and is later used in the context of a concrete type, the generic variable is locally *instantiated*. For instance, if an `Integer` is given to the polymorphic `id` function, the generic type signature $id : \forall a \ . \ a \rightarrow a$ is instantiated to $id : Integer \rightarrow Integer$ in the most-local context. This instantiation may occur several times at different places in the code, allowing full flexibility with the generic functions. Similar to generalisation, this is directly follows from Hindley-Milner's [Inst] rule.

4.10.4 Type Classes

Type classes in Funky are employed to perform type inference where algebraic data types are concerned. This is achieved by adding constraints to type variables in parametrically polymorphic types. Simply put, these constraints say 'you can only instantiate this type variable to a list of pre-defined valid types'.

Type classes are *not* completely user facing – the user cannot create their own type classes and work with them like they might be able to in a language like Haskell. Their main purpose is to allow proper typing of algebraic data types. For example, consider a simple algebraic data type that is a binary tree of integers: `newtype Tree = Branch Tree Integer Tree | Leaf Integer`. A function that accepts a binary tree as a parameter, like `is_leaf (Leaf _) = True; is_leaf (Branch _ _ _) = False` has type-signature $is\_leaf : Tree \rightarrow Bool$. The user should be able to pass constructions of type `Branch` and `Leaf` to it, and nothing else. Type classes are used to enforce this in Funky.

Type classes are also used to implement operator overloading/ad-hoc polymorphism in Funky. There are a few built-in type classes in Funky that allow this. The best example of a built-in type class would be `Num`, which can be instantiated to either `Float` or `Integer`. The built-in operator `+` has type $Num \rightarrow Num \rightarrow Num$, allowing it to work with both integers and floats.

4.10.5 Dependency Analysis/Out-Of-Order Definitions

Funky supports out-of-order definitions, meaning a variable can be used before it is defined in the source code[6]. For clarity, the code in Listing 14 is perfectly valid in Funky, but illegal in a language like Python.

```
y = z + a
z = x + y
x = a + a
a = 2
```

Listing 14: Out-of-order definitions. This code is valid in Funky but invalid in Python.

This is useful for a number of reasons:

- It allows the programmer to think more about *what* their program does instead of being caught up in the details of which entities need to be declared before others.
- It is necessary for easy[7] recursion and mutual recursion. We discuss this more in Section 4.10.6.
- It makes the process of type inference more linear and thus simpler.

To allow for out-of-order bindings, we perform what is called *dependency analysis* and rearrange them so that they occur in the 'correct' order. The principle idea of binding dependency is simple: a binding x depends on a binding y in the *same list of definitions* if y is present anywhere in x's definition *or* if x depends on anything that depends on y.

With these rules in mind, it is trivial to traverse the core tree to find `CoreLets` with more than one binding and build up a graph of dependencies for each. The dependency graph for the bindings in Listing 14 is shown in Figure 3.

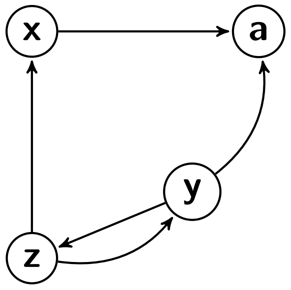


Figure 3: The dependency graph for the bindings in Listing 14.

[6]Note that ADT definitions using the `newtype` keyword do not support out-of-order definitions and must be defined in the correct logical order if they reference each other.

[7]Recursion *can* be achieved without support for out of order definitions using a fixpoint combinator, although it is desirable to allow the programmer to write recursive definitions naturally without requiring the use of such constructs.

Once we have the dependency graph for a `CoreLet`, we traverse it using a depth-first search to rearrange the bindings into reverse-dependency order.

The algorithm is simple – it is most precisely expressed with pseudocode, as below:

```
function REORDER_BINDINGS($bindings$)
    $dep\_graph \leftarrow$ CREATE_DEPENDENCY_GRAPH($bindings$)
    $reordered \leftarrow [\ ]$
    function DFS($at$)
        mark $at$ as visited
        for all neighbour $n$ of $at$ do
            if $n$ not visited then
                DFS($n$)
            end if
        end for
        APPEND($reordered, at$)
    end function
    for all node $n$ in $dep\_graph$ do
        if $n$ not visited then
            DFS($n$)
        end if
    end for
    return $reordered$
end function
```

This algorithm ensures that all bindings are preceded by their dependencies. However, there is a snag, explained in Section 4.10.6.

4.10.6 Mutually Dependent Bindings and Tarjan's Algorithm

The observant reader may have noticed that, in Section 4.10.5, the bindings in Listing 14, and by extension the dependency graph in Figure 3, have the interesting property that they contain *mutual dependencies*. That is, the variables z and y depend on each other, so there is no possible ordering that satisfies their dependencies.

Although the example in Listing 14 is somewhat contrived, this problem cannot be ignored. There are plenty of 'normal' scenarios where this snag can arise, because cyclic dependencies is inherent to mutual recursion. Mutual recursion is a special-case of mutually dependent bindings where two functions defined in the same listing call one another. Listing 15 is simple example of where mutual recursion might be useful.

```
# Mutual recursion in Funky.
module mutualrecursion with

    even 0 = True
    even n = odd  (n - 1)
    odd  0 = False
    odd  n = even  (n - 1)

    main = odd 15
```

Listing 15: Mutual recursion in Funky.

This problem must be dealt with. First of all, this situation requires special treatment in code generation to make sure that the target code is valid – it is no longer as simple as just outputting the definitions in the right order. One possible solution would be to declare the variables before they are defined, or to ensure that the variables are not evaluated until they are actually used. The implementation details are largely dependent on the features and behaviours of the target programming language and have to be accounted for on a case-by-case basis.

Second of all, mutually dependent bindings must be accounted for in type inference. For recursive and mutually recursive definitions, it is necesary to group the listing into *strongly connected components* based on the dependency graph. Then, type inference is performed separately for each strongly connected component to determine the most general type of each definition in that group. Failure to perform this extra step can result in overspecialisation of types, which could lead to a erroneous type errors being reported.

First, let us define 'strongly connected component'. In graph theory, a directed graph is strongly connected if there exists a path between all pairs of vertices. A strongly connected component of a directed graph is an extension of this, being a maximal strongly connected subgraph. The graph in Figure 4 hopefully illustrates this idea.

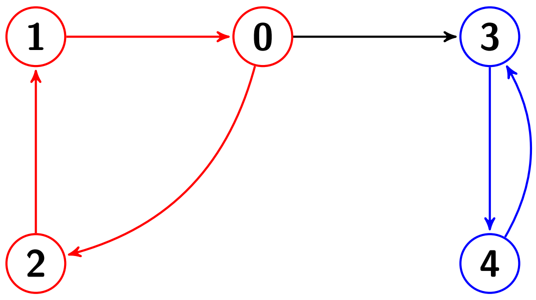


Figure 4: Colour-coded strongly connected components.

To find strongly connected components in a dependency graph, we use Tarjan's algorithm, described in Tarjan (1972). This algorithm takes a directed graph as input, and produces a partition of the graph's vertices into its strongly connected components. As we might expect, each vertex of the graph appears in exactly one of the strongly connected components. Any vertex not on a directed cycle forms a strongly connected component of just itself.

To understand Tarjan's algorithm, it is first necessary to understand the concept of a low-link value. Simply put, the low-link value of a node is the smallest node ID reachable from that node when doing a depth-first search, including itself. In order for that to make sense, we must label each of the nodes in our graph with a depth-first search. To do this, choose a node at random, and explore the graph using a depth-first search, labelling each node with how many nodes we have encountered before it (i.e. the 'time' at which we saw the node, so to speak).

With numerical identifiers assigned to nodes, it is trivial to compute their low-link values. For clarity, Figure 5 shows an example graph annotated with the low-link values of each node.

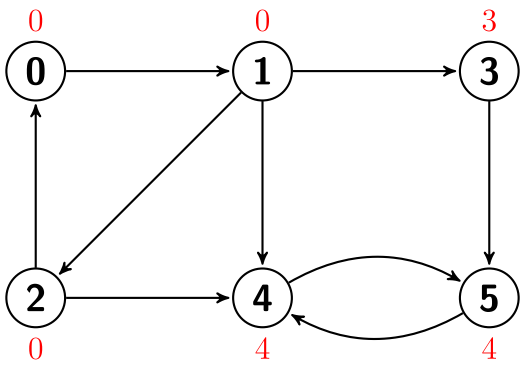


Figure 5: An example graph annotated with the low-link values, l, of each node.

Notice that, in Figure 5, the nodes with the same low-link values are strongly connected components! Tarjan's algorithm is based on this property. However, there is a catch; we cannot simply compute the low-link value of each node and use this to find strongly connected components, because this technique is highly sensitive to the order in which we perform our depth-first traversal when initially assigning IDs to nodes. For example, if we assign the ID 0 to a node that is reachable from *all other nodes*, every node in the graph will have a low-link value of 0, which does not convey useful information about the graph's strongly connected components.

Tarjan's algorithm circumvents this issue by introducing an *invariant* which prevents strongly connected components from interfering with each other's low-link values. Tarjan's algorithm maintains a set of 'valid' nodes from which to update low-link values from called the *update set*. Nodes are added to this set when they are encountered from the first time, and removed when a complete strongly connected component is found. Now, to update the low-link value of a node v_1 to that of node v_2's like before, we have the extra condition that v_2 must be in our update set.

Once again, Tarjan's algorithm is most precisely expressed programmatically. Below is Tarjan's algorithm in pseudocode. Running this code for a particular dependency graph labels the low-link values correctly. It is then trivial to extract the strongly connected components by simply examining which nodes have the same low-link values.

```
function GET_STRONGLY_CONNECTED_COMPONENTS($dependency\_graph$)
    $i \leftarrow 0$
    $scc\_count \leftarrow 0$
    $stack \leftarrow$ empty stack
    function DFS($at$)
        PUSH($stack, at$)
        $i \leftarrow i + 1$
        $at.id \leftarrow i$
        $at.low\_link\_value \leftarrow i$
        mark $at$ as visited
        for all neighbour $to$ of $at$ do
            if $n$ is unvisited then
                DFS($to$)
            end if
            if $to$ in $stack$ then
                $x \leftarrow at.low\_link\_value$
                $y \leftarrow to.low\_link\_value$
                $at.low\_link\_value \leftarrow$ MIN($x, y$)
            end if
        end for
        if $at.id == at.low\_link\_value$ then
            while $stack$ is nonempty do
                $node \leftarrow$ POP($stack$)
                $node.low\_link\_value \leftarrow at.id$
                if $node == at$ then
                    break
                end if
                $scc\_count \leftarrow scc\_count + 1$
            end while
        end if
    end function
    for all node $n$ in $dependency\_graph$ do
        if $node$ is unvisited then
            DFS($node$)
        end if
    end for
end function
```

Summary A lot has been covered in this section. In summary:

- The Funky compiler uses the Hindley-Milner type system.
- The algorithm used to infer types is Algorithm W, which is an efficient type inference algorithm that performs in near linear time.
- Algorithm W works by performing a top-down traversal of the tree inferring the type of each construct in the code. We build up a set of typing assumptions from an initial set of defaults and use these to gradually build up a clearer picture of the types in the program. If a contradiction is found, the code is not type safe.
- Principal types (i.e. most-general types) are inferred by maintaining a set of all variables known to be *not* generic, and generalising all others where possible.
- Ad-hoc polymorphism for built-in operators is achieved through the notion of type-classes. Type classes in Funky are a weaker notion than in languages like Haskell, and effectively mean 'a type variable with constraints such that it can only be instantiated to a subset of possible type operators'.
- Out-of-order definitions allow the user to specify definitions in any order they want. They are reordered by the compiler after performing dependency analysis.
- Tarjan's algorithm is used to find strongly connected components in let bindings, allowing us to type check recursive and mutually recursive functions without requiring things like primitive fixpoint operators.

The `--dump-types` command line flag can be used to output the types of all of the top-level objects in the core tree to `stdout`.

4.11 Code Generation

Code generation is the final stage of compilation, where the typed, optimised intermediate language is converted to a target language of choice. Traditionally, a compiler would generate assembly code, or raw binary that is readily executable by the computer. In the case of the Funky compiler, we instead compile the intermediate representation into a different source programming language. Since these different source programming languages are made up of 'just text', generating code for them is a matter of emitting strings in the source language such that the semantics of the program are preserved.

Funky offers a simple framework for generating code. It defines a `CodeGenerator` class which provides basic utility functions, such as inserting text into a buffer at a given indentation. Code generators should inherit from this class and implement their own `do_generate_code` method which accepts the type definitions and core tree as input and returns the source code for the target language as output.

Funky currently supports compilation to Python and Haskell. The code generator for Haskell was written 'just for fun' to demonstrate the modularity of the compiler, and was relatively straightforward since Funky's intermediate language maps over quite nicely. The list of currently implemented code generators is:

- A *strict* Python code generator, for generating Python code that uses a strict evaluation strategy.
- A *lazy* Python code generator, for generating Python code that uses a lazy evaluation strategy.
- A Haskell code generator.

4.11.1 Code Generation Basics

Like most of the phases of compilation, code generation involves a tree traversal. The basic idea is that we perform a top-down traversal of the tree, building up code snippets and 'emitting' them into a text buffer when appropriate. The process lends itself well to a recursive algorithm. Each of the constructs in the core tree is associated with a function that handles its code generation. These functions do two things:

- Emit code to a text buffer (if necessary).
- Return a string that represents 'this construct' in the target language.

Whether or not a function emits or not generally depends on whether or not it is part of a larger structure. For instance, the function associated with variables and function applications never emit code, since they are always found as part of some larger structure (i.e. the scrutinee of a match statement, the bindee of a bind). Therefore, they *return* a string to represent themselves in the target language. On the other hand, structures such as top-level definitions are not part of a larger structure so they do emit to the text buffer.

The process is somewhat involved and requires a lot of string formatting. There are also a lot of edge cases and nuances to account for. The process of code generation is quite mundane, so we only require a brief overview of the general process. What is far more interesting is the implementation of laziness, which is explained in Section 4.11.2.

4.11.2 Achieving Laziness with Generated Python Code

The Python programming language supports some lazy features (i.e. generators), but is generally not considered to be a lazy language. This presents a problem: one of the language's requirements is to support lazy evaluation, so how do we fulfil this if we are compiling to a language with limited lazy evaluation features?

The answer is to use Python's lambda mechanism. Lambdas in Python are not evaluated until they are called. This means if we wrap an expression with a lambda, it will not be evaluated until that lambda expression is called. If it is never called, the expression will never be evaluated. This is the basis of Funky's laziness implementation – we can exploit this mechanism by wrapping nearly all expressions in the code with a lambda, forming a complex tree of lambda expressions, and devising a helper 'trampoline' function that unravels the tree, evaluating only what is explicitly needed to produce a single result.

Implementation of Python Laziness We first create a data structure to represent a deferred evaluation, referred to in the literature as a *thunk*. Thunks in Funky are represented using the Python code shown in Listing 16.

```
class Thunk:

    def __init__(self, thunk):
        self.thunk = thunk

    def __call__(self):
        return self.thunk()

example_thunk = Thunk(lambda: 10 + 10)  #  <- declaring a thunk
print(example_thunk())                  #  <- evaluating a thunk
```

Listing 16: Python class to represent thunks used in the Funky code generator.

We also define a function, called a *trampoline function*, which unwraps nested thunks until a result is found. The Python code used to trampoline thunks is given in Listing 17.

```
def trampoline(bouncer):
    while callable(bouncer) and not inspect.isclass(bouncer) \\
        and len(inspect.signature(bouncer).parameters) == 0:
        bouncer = bouncer()
    return bouncer
```

Listing 17: Python code to trampoline thunks.

The code in Listings 16 and 17 is all that is required to evaluate expressions lazily. By carefully building up a tree of thunks based on the desugared code, and trampolining where an expression needs to be evaluated, we can evaluate any expression lazily.

However, to achieve *true* lazy evaluation, it is necessary that thunks are shared, or never recomputed. Without modification, the code in Listing 16 is prone to slowness because each time the result is required, it is recomputed. This can be avoided by adding memoization to the thunk class, as shown in Listing 18.

```
class Thunk:

    def __init__(self, thunk):
        self.thunk = thunk
        self.memo = None

    def __call__(self):
        if self.memo:
            return self.memo
        self.memo = trampoline(self.thunk)
        return self.memo
```

Listing 18: Thunks with memoization to allow true lazy evaluation.

Performance of Python Laziness Generating Python code that follows this structure adds significant power to the Funky language by allowing the programmer to define infinite data types (such as the infinite list of primes). However, the performance advantages of laziness are not always apparent. If the result of something is never used, then storing it lazily and never executing it is clearly more efficient than needlessly executing it. However, if you are definitely going to execute it, then storing it lazily and executing it later is just overhead, and is less efficient than just executing it strictly.

For large programs, the number of Python lambdas used to achieve laziness can be quite large. Creating and storing a thunk incurs some overhead. If we do not use laziness in our programs, they will be faster if we use strict evaluation rather than needlessly incurring overhead from creating thunks all of the time.

For this reason, the Funky compiler presents the choice of whether the user wants to compile with or without lazy evaluation. By default, Funky compiles *without* lazy evaluation, since the majority of new programmers will not need its features and will generally not be creating infinite data structures if they are new to functional programming.

4.11.3 Achieving Laziness with Generated Haskell Code

No special treatment is required. Haskell implements laziness natively, so the compiled Haskell code already enforces lazy evaluation.

Summary We have seen the basic ideas behind code generation and how laziness is implemented in Funky. The key insight in code generation is that the core tree is traversed top-down, converting each construct within to a code snippet and emitting them into a text buffer when appropriate.

We have also seen how laziness can be implemented in Python, a language that does not support lazy evaluation. The key insight here is that we use Python's lambda mechanism, alongside some memoization, to defer computations until they are actually needed. This allows us to define and work with infinite data structures, as well as achieve better performance in some contexts.

The `--dump-generated` command line flag can be used to output the generated code to `stdout` for manual inspection.

4.12 User Interface

4.12.1 Command-Line Compiler

Funky ships with a command-line utility for compiling Funky source code. It is accessible with the `funky` command, and has a number of command-line flags to aid with debugging, changing the output file, changing the code target, viewing detailed log messages, etc. The programmer can learn how to use the command-line utility by issuing the command `funky --help` – the output of this command is presented in Listing 19. The command-line interface is relatively self-explanatory, so no further explanation is required.

```
usage: funky [-h] [-V] [-v] [-q] [-u] [-c] [-e]
             (--output output_filename | --execute) [--dump-pretty]
             [--dump-lexed] [--dump-parsed] [--dump-imports] [--dump-renamed]
             [--dump-desugared] [--dump-types] [--dump-generated]
             [--target {haskell,python,python_lazy,intermediate}]
             input

positional arguments:
  input                 Input program (funky source).

optional arguments:
  -h, --help            show this help message and exit
  -V, --version         Output funky's version and quit.
  -v, --verbose         Be verbose. You can stack this flag, i.e. -vvv.
  -q, --quiet           Be quiet. You can stack this flag, i.e. -qqq.
  -u, --no-unicode      Do not use unicode characters in output (for old
                        terminals).
  -c, --no-colors       Do not use coloured output (for boring people).
  -e, --show-exception-traces
                        Show full exception traces.
  --output output_filename, -o output_filename
                        File to write compiled program to.
  --execute, -x         Do not create an output file, execute directly.
  --dump-pretty         Dump the prettified source code to stdout with syntax
                        highlighting.
  --dump-lexed          Dump the lexed (disambiguated) source code to stdout.
  --dump-parsed         Dump the parse tree to stdout.
  --dump-imports        Dump the parse tree (with imports included) to stdout.
  --dump-renamed        Dump the renamed parse tree to stdout (alongside
                        renamer mapping for top-level objects).
  --dump-desugared      Dump the core (desugared) funky code to stdout.
  --dump-types          Dump the types of all top-level objects in the core
                        tree to stdout.
  --dump-generated      Dump the generated code to stdout.
  --target {haskell,python,python_lazy,intermediate}
                        The target language for compilation (choose
                        'intermediate' if you want to output the intermediate
                        code as a serialised Python object).
```

Listing 19: Output of `funky --help`.

4.12.2 REPL

Many modern programming languages have a shell-like interface that allows the user to input commands and evaluate them interactively in their terminal. Such programs are referred to as read-evaluate-print loops, or REPLs for short.

Since one of the requirements of the project is to be welcoming to new users, a REPL is an absolute necessity. They are extremely valuable because they allow the programmer to experiment with the language. The user can issue commands and receive immediate feedback on whether they were syntactically and semantically correct. It is also common for programmers to use a REPL for testing quick code snippets within the language without having to create an entirely new source file.

Python's REPL can be accessed using the command `python`. Haskell's REPL can be accessed using the command `ghci`. In similar fashion, Funky's REPL can be accessed by issuing the command `funkyi` at the command-line. Again, `funkyi --help` shows `funkyi`'s command line options.

```
usage: funkyi [-h] [-V] [-v] [-q] [-u] [-c] [-e] [-z] [files [files ...]]

positional arguments:
  files                 Load these programs into the REPL.

optional arguments:
  -h, --help            show this help message and exit
  -V, --version         Output funkyi's version and quit.
  -v, --verbose         Be verbose. You can stack this flag, i.e. -vvv.
  -q, --quiet           Be quiet. You can stack this flag, i.e. -qqq.
  -u, --no-unicode      Do not use unicode characters in output (for old
                        terminals).
  -c, --no-colors       Do not use coloured output (for boring people).
  -e, --show-exception-traces
                        Show full exception traces.
  -z, --be-lazy         Generate lazy code where possible.
```

Listing 20: Output of `funkyi --help`.

Funky's REPL was developed after having already created the compiler. The modularity of the compiler code meant that developing the REPL was relatively straightforward, since it is able to piggyback off the code already in place to parse, analyse, and run Funky programs.

The REPL works by accepting user input in an infinite loop and iteratively building up a program in Funky intermediate code to represent the types and definitions the user has created. This is referred to as the *intermediate program*. When an expression, such as a function call or some numerical operation, is supplied as input, the REPL compiles the expression with respect to the intermediate program, then runs it. This all happens seamlessly behind the scenes.

The intermediate program is actually just a `CoreLet` statement. The bindings in this `CoreLet` are all of the definitions the user has created, and the body of the `CoreLet` is changed on-demand to reflect the latest expression the user has typed into the REPL for evaluation.

The REPL is able to distinguish between expressions and definitions by using two separate parsers – one for parsing expressions, and one for parsing definitions. These are the exact same parsers used by the compiler to parse Funky source code, but with different start symbols. Specifically, the parser for declarations has its start symbol set to `TOP_DECLARATIONS` in Funky's CFG, and the one for parsing expressions has its start symbol set to `INFIX_EXP`. The REPL first tries to parse the user input as a declaration – if this is successful, we know that we must add a new declaration to the intermediate program. Otherwise, the REPL tries to parse the user input as an expression – if this is successful, the expression is compiled and evaluated with respect to the intermediate program. If neither parsers understand the input, there is a syntax error.

In the case of a definition, the REPL takes the input string from the user, reduces it to a `CoreBind` in Funky intermediate code, type checks it, and adds it to the intermediate program.

In the case of an expression, the REPL takes the input string from the user, reduces it to Funky intermediate language, and sets it as the expression part of the intermediate program (global `CoreLet`). The entirety of the intermediate program is then type checked, and provided it is correct, the Python code generator generates its compiled code. This is executed within the REPL using the `exec` built-in function in Python.

The REPL is mostly self-documented. The user can issue commands to the REPL using the prefix '`:`'. For help, and a list of available commands, type `:help`.

5 Evaluation

5.1 Language Requirements

This section evaluates the developed system against the requirements defined for the language itself in the project proposal.

5.1.1 Syntax

Regarding the syntax of the language, the project proposal states:

> *"The language must have a concise, straightforward syntax that is welcoming to those new to functional programming. Haskell does a relatively good job in this respect. I plan to make the syntax as close to Python as possible, since Python is designed to be a highly readable language."*

This requirement is *fulfilled.* Despite being inherently subjective, it can be argued that Funky's syntax presents itself as a suitable blend between Python and Haskell code. Funky's syntax is both familiar to users that know functional programming, and welcoming to those who do not, but perhaps have some experience with Python programming instead.

5.1.2 Static Typing System

Regarding a static typing system, the project proposal states:

> *"The language must feature a static typing system, where the type of each variable is known and can be validated at compile-time. As well as having primitive types, the user should be able to declare and use their own types, and then use pattern matching against them."*

This requirement has been *fulfilled.* Funky is statically typed, as the type of each variable in the source code is inferred at compile time. This is done by the type checker, discussed extensively in Section 4.10. Additionally, Funky allows the user to define their own data types with the `newtype` keyword and perform pattern matching against them.

5.1.3 Language Features

Regarding language features, the project proposal states:

> *"The language must feature pattern matching, lambda expressions, and lazy evaluation."*

This requirement is *fulfilled.* The language features pattern matching, lambda expressions, and lazy evaluation.

5.1.4 Language Expressivity

Regarding the expressivity of the language, the project proposal states:

> *"In general, the language must be sufficiently expressive. This is hard to define and quantify, but generally I want the language to be expressive enough to define relatively sophisticated functions that can solve real world problems."*

This requirement is *fulfilled.* As per the Evaluation section of the project proposal, the intended strategy for demonstrating the expressivity of the language was to:

- Implement some common recursive algorithms, such as finding the length of a list or computing the nth element of the Fibonacci sequence.
- Implement algorithms for sorting a list of numbers. The specific algorithms mentioned were quick sort, to demonstrate list manipulation, and tree sort, to demonstrate recursion over a user-defined data structure.
- Implement an infinite data structure, such as the list of all prime numbers, to show that lazy evaluation is working.

Listings 21, 22, 23, and 24 contain sample functions for computing the nth element of the Fibonacci sequence, computing the length of a list, performing a quick sort, and performing a tree sort respectively.

```
# Computing terms of the fibonacci sequence in Funky.
module fibonacci with

    fib 0 = 0
    fib 1 = 1
    fib n = fib (n - 1) + fib (n - 2)

    main = fib 10
```

Listing 21: Function to find the nth element of the Fibonacci sequence in Funky.

```
# Finding the length of a list in Funky.
module listlength with

    newtype List = Cons Integer List | Nil

    length Nil          = 0
    length (Cons _ xs) = 1 + length xs

    main = length my_list
        with my_list = Cons 1 (Cons 2 (Cons 3 Nil))
```

Listing 22: Function to find the length of a list in Funky.

```
# The standard quicksort algorithm for integers in Funky.
module quicksort with

    import "stdlib.fky"
    import "intlist.fky"

    quicksort Nil          = Nil
    quicksort (Cons x xs) = (quicksort s) ~concatenate~ unit x ~concatenate~ (quicksort l)
                            with s = filter ((>)  x) xs
                                 l = filter ((<=) x) xs

    main = pprint (quicksort my_list)
        with my_list = Cons 5 (Cons 1 (Cons 7 (Cons 7 (Cons 3 Nil))))
```

Listing 23: Function implementing a quick sort for integers in funky.

```
# Treesort algorithm for integer in Funky.
module treesort with

    import "stdlib.fky"
    import "intlist.fky"

    newtype Tree = Branch Tree Integer Tree | Empty

    insert e Empty                                = Branch Empty e Empty
    insert e (Branch l v r) given e <= v          = Branch (insert e l) v r
                            given otherwise       = Branch l v (insert e r)

    inorder Empty = Nil
    inorder (Branch l v r) = inorder l ~concatenate~
                             unit v    ~concatenate~
                             inorder r

    treesort list = inorder (foldr insert Empty list)

    main = treesort my_list
           with my_list = Cons 5 (Cons 1 (Cons 7 (Cons 7 (Cons 3 Nil))))
```

Listing 24: Function implementing a tree sort for integers in funky.

Turing Completeness A Turing machine is a purely mathematical, idealised model of a computer. It is commonly defined as a theoretical machine containing three things: an infinitely long tape divided into cells which extends infinitely in both directions, a read/write head positioned over a single cell of the tape, and a control unit which decides on the next action. Since its inception in 1936, there have been many proposals for alternative, perhaps more 'pure', models of computation. In every case so far, these alternatives have been shown to be equivalent to a Turing machine, and thereby exhibit the same computational power. The notion of 'equivalence to a Turing machine' is called Turing completeness.

The fact that it appears as though we can't reach a more powerful notion of computation than what is offered by the Turing machine led the logician Alonzo Church to hypothesise that computation is fully captured by Turing's model. This is widely known as Chuch's Thesis – in a sentence, it states that the concept of an "effective procedure" is fully captured by the Turing machine model. Church's Thesis is widely believed to be true, but is currently unproven.

If we can go even further and prove that Funky is Turing complete, it would mean that it is able to express every computable function (contingent on Church's Thesis being true). To prove that a language is Turing complete, it suffices to implement a Turing machine within that language; if we can emulate one, we are clearly just as computationally expressive, which means that we can express any computable function.

Listing 25 shows one possible implementation of a Turing machine emulator in Funky. This stands as proof that Funky is Turing complete.

Lambda Calculus To further cement the proof that Funky is an expressive language, we can implement the pure lambda calculus within it. Since the pure lambda calculus is known to be equivalent to a Turing machine (i.e. it is Turing complete), it follows that if we are able to implement the lambda calculus in Funky, then it is also Turing complete. Thankfully, Funky lends itself quite naturally to implementing something like this.

An implementation of the lambda calculus in Funky is described in Listing 26, adapted from Fisher (2018).

```
# A sample implementation of a Turing machine in Funky.
module turing_machine with

    newtype Alphabet = One | Zero | Blank
    newtype Tape = TapeCons Alphabet Tape | TapeEnd
    newtype Machine = TM Tape Alphabet Tape

    newtype Program = ProgramCons (Machine -> Machine) Program | ProgramEnd

    read (TM _ h _) = h
    write s (TM ls _ rs) = TM ls s rs

    shift_left (TM (TapeCons l ls) h rs) = TM ls l (TapeCons h rs)
    shift_left (TM TapeEnd h rs)         = TM TapeEnd Blank (TapeCons h rs)

    shift_right (TM ls h (TapeCons r rs)) = TM (TapeCons h ls) r rs
    shift_right (TM ls h TapeEnd)         = TM (TapeCons h ls) Blank TapeEnd

    fresh_machine = TM TapeEnd Blank TapeEnd

    instruction_swap (TM ls One rs)  = (TM ls Zero rs)
    instruction_swap (TM ls Zero rs) = (TM ls One rs)
    instruction_swap (TM ls x rs)    = (TM ls x rs)

    run ProgramEnd machine        = machine
    run (ProgramCons i is) machine = run is (i machine)

    main = run instructions fresh_machine
           with instructions = ProgramCons (write One) (
                               ProgramCons shift_right (
                               ProgramCons (write One) (
                               ProgramCons shift_right (
                               ProgramCons (write One) (
                               ProgramCons instruction_swap (
                               ProgramCons shift_left (
                               ProgramCons instruction_swap (
                               ProgramCons shift_left ProgramEnd))))))))
```

Listing 25: Example implementation of a Turing machine emulator in Funky.

```
# A interpreter for the lambda calculus in Funky.
module lambdacalc with

    # Inspired by Jim Fisher's blog post code, available at:
    # https://jameshfisher.com/2018/03/15/a-lambda-calculus-interpreter
    # -in-haskell.html

    newtype LambdaExpression = App   LambdaExpression LambdaExpression
                             | Abs   LambdaExpression
                             | Var   Integer
                             | Const Integer

    eval (App fun arg) = match eval fun on
                             Abs body -> eval (sub 0 body)
                             x        -> App x arg
                         with sub n (App e1 e2) = App (sub n e1) (sub n e2)
                              sub n (Abs e)     = Abs (sub (n + 1) e)
                              sub n (Var y)     = arg if n == y else Var y
                              sub n x           = x
    eval x = x

    main = eval my_expr
        with my_expr = App (App (Abs (Abs (Var 1))) (Const 4)) (Const 5)
```

Listing 26: Example implementation of the pure lambda calculus in Funky.

5.2 Compiler Requirements

This section evaluates the developed system against the requirements defined for the compiler software in the project proposal.

5.2.1 Compilation

Regarding compilation, the project proposal states:

> *"The compiler must take the source code for a program written in my language and compile it down to raw C code. The primary reason for this is portability if our language compiles to C, we can leverage existing C compilers to prepare a binary for whichever architecture we want. This is not an uncommon procedure: the Glasgow Haskell Compiler (GHC) has a C backend that allows Haskell code to be compiled to C."*

This requirement has *changed*. Instead, the Funky compiler can compile to Python. Flexible machinery in the compiler exists to allow for translation to different target languages, so if compilation to a different target is of interest, it should be as easy as simply writing a new transformation module.

This requirement changed for two reasons:

1. Compilation to C is complex. C lacks lexically-scoped anonymous functions, and the closest we can get is function pointers. However, anonymous functions cannot be compiled directly to function pointers since functions must be declared at the top-level, preventing them from capture local environments. It very quickly became clear when writing a C target code generator that the runtime required to support lexically-scoped anonymous functons is very subtle.
2. Compilation to C, for our purposes, does not break the end-to-end product. As mentioned in the project proposal, the primary reason for compiling to C was to piggyback off existing technologies to allow our code

to run, making it *portable*. Python is already a well-supported language across many platforms, with a robust and feature-rich runtime. We can compile to Python and reap the same rewards as we would compiling to C, perhaps at the expensive of some efficiency. Ultimately, it is clear that Python is a more attractive compilation target.

It was anticipated in the project proposal that compilation to C might be difficult. The alternative compilation target suggested in the project proposal was LLVM intermediate code or raw machine code. In hindsight, these suggestions are even more difficult than compiling to C.

In conclusion, the compiler does *not* compile to C, but instead compiles to Python much to the same effect.

5.2.2 Command-Line Interface

With regards to a command line interface, the project proposal states:

> *"The compiler does not need to feature a GUI, but it must feature a CLI with command-line flags."*

This requirement is *fulfilled.* The Funky compiler is available through the `funky` command on the command-line. It provides a number of options useful for debugging, changing the output file, viewing detailed log messages, etc.

5.2.3 Interactive Mode

With regards to an interactive evaluation mode for the compiler, the project proposal states:

> *"The compiler must feature an interactive mode where the user can input statements line-by-line in their terminal."*

This requirement is *fulfilled.* Funky ships with a robust REPL which can be accessed using the command `funkyi` from the command line.

5.2.4 Modularity

With regards to modularity, the project proposal states:

> *"The compiler must be modular. Each stage of compilation should be as decoupled as possible. This makes it significantly easier to make changes in the future – for example, if you want to make a minor change to the syntax, you should only need to change the lexical analyser."*

This requirement is *fulfilled.* A lot of thought and care has gone into making the compiler as modular as possible. Each stage of compilation is performed by an independent Python module that is decoupled from the stages before and after it. There is a mini framework in place for code generation to make developing new target code generators as seamless as possible.

5.3 Performance

Performance is crucial in compiler engineering, especially so for state-of-the-art, industrial-strength compilers like Clang and G++. Two particularly important performance metrics are the compilation time of a program and the speed of the generated code.

Funky is a very small project compared to modern compilers used in industry. Such compilers have had countless hours of work put into them by dedicated teams of professional engineers. It is unreasonable to compare Funky to industry tools given its scope – rather, this section hopes to demonstrate that Funky's performance is practical for its purpose.

All tests were performed on a MacBook Pro (early 2015 model) with a 2.7 GHz Intel Core i5 processor.

5.3.1 Compilation Time

The Funky compiler is generally quite fast at compiling source files. Funky logs the time that it took to compile a source file, and these messages can be exposed by asking for verbose output using the `-vv` flag at the command line.

The compile times for each of the programs in the `sample_programs` subdirectory within the Funky package (see Appendix A) are recorded in Table 4. Each time is the average time across *three* separate trials. As you can see, the compilation time is not a cause for concern using any of the generators. Programs compile very quickly, usually in under 100 milliseconds. Using the lazy code generator seems to add some overhead (since it has to generate more complex code to achieve laziness), but this overhead is negligible.

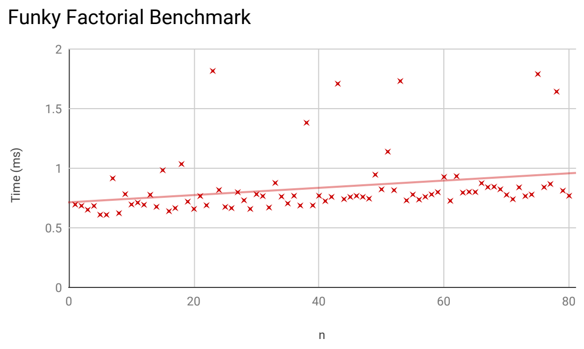
	Generator Time (milliseconds)			
Sample Program	Strict Python	Lazy Python	Haskell	**Row Average**
`sample_fix.fky`	18	17	30	**22**
`sample_listlength.fky`	19	21	34	**25**
`sample_mutualrecursion.fky`	29	36	21	**29**
`sample_fibonacci.fky`	19	68	27	**38**
`sample_randomfloat.fky`	29	49	37	**38**
`sample_peano.fky`	39	43	45	**42**
`sample_imports.fky`	35	46	60	**47**
`sample_randomsequence.fky`	37	65	42	**48**
`sample_fizzbuzz.fky`	58	45	47	**50**
`sample_expressiontree.fky`	59	56	44	**53**
`sample_lambdacalc.fky`	58	56	64	**59**
`sample_listindex.fky`	54	96	58	**69**
`sample_dictionary.fky`	71	84	61	**72**
`sample_fizzbuzz_list.fky`	60	119	62	**80**
`sample_set.fky`	98	102	90	**97**
`sample_turing.fky`	103	110	90	**98**
`sample_caesar.fky`	138	99	101	**113**
`sample_quicksort.fky`	173	168	199	**180**
`sample_treesort.fky`	211	201	218	**210**
`sample_parser.fky`	210	244	184	**213**
`sample_pascals.fky`	223	234	182	**216**
`sample_lazy_lists.fky`	271	451	194	**305**
Column Average	92	110	86	**97**

Table 4: Funky compilation times using the three different code generators across the included sample programs.

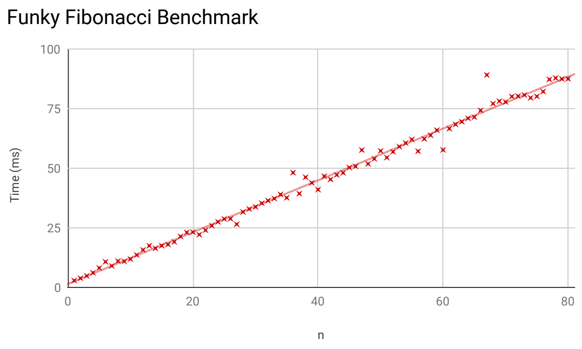
5.3.2 Runtime Benchmarks

A few algorithms written in Funky have been benchmarked to demonstrate that the Funky compiler produces reasonably fast code. All of the examples below were compiled using the *strict* Python code generator (with the exception of the fibonacci test, where the lazy generator was used to create a lazy list), and the results are taken after an average of *three trials*.

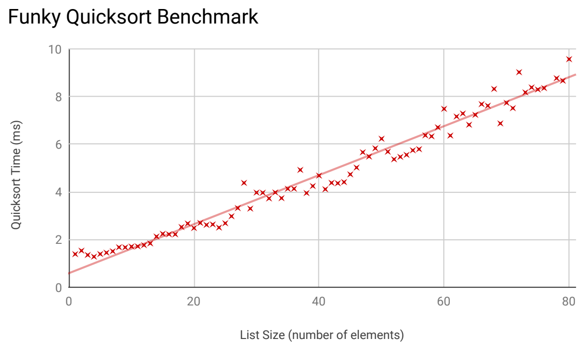
Note that the algorithms below written natively in Python will naturally run faster; it is unavoidable that the compilation process will introduce some inefficiencies, since the compiler is imperfect. However, the speed of the compiled programs is certainly acceptable for writing, running, and testing non-critical programs.



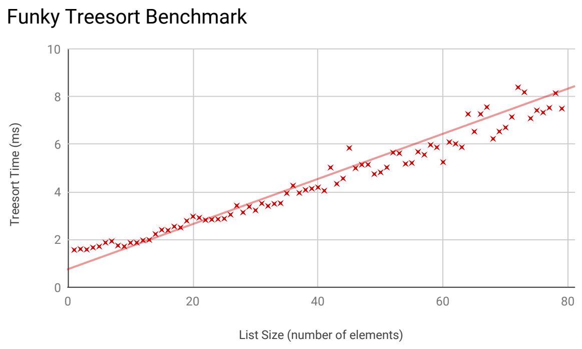
$factorial(n)$ for $1 \leq n \leq 80$.



$fibonacci(n)$ for $1 \leq n \leq 80$.



Quicksorting a random list of size n.



Treesorting a random list of size n.

Benchmark Comments These benchmarks show that the Funky compiler produces code that is reasonably fast. At the very least, this performance is tolerable when using `funkyi` to run Funky code interactively.

The basic numerical operation of computing the factorial of an integer executes quickly and scales well. For the range $1 \leq n \leq 80$ the average time taken to compute $factorial(n)$ was 0.84 milliseconds, with the maximum value being 1.82 milliseconds.

Computing the nth element of the Fibonacci sequence (using a lazy list) is noticeably slower. This is likely due in part to the fact that the lazy Python code generator is used in this instance rather than the strict generator. The implementation of laziness adds some non-negligible overhead. Nonetheless, the problem appears to scale linearly with the size of the input, and a time of 87.51 milliseconds to compute the 80th element of the Fibonacci sequence is certainly reasonable.

The quicksort and treesort algorithms also seem sensible, exhibiting similar performance characteristics. Both seem to scale linearly with the size of the input (at least, for this small sample size) and execute in less than 10 milliseconds for lists of size $n = 80$.

Hopefully the reader will agree that, for the scope of the project, the performance of Funky's generated Python code is acceptable.

6 Extensions and Extras

The project proposal details a short list of extensions that could be implemented if additional time is available. The project in its final form implements *two* of these extensions – a standard library of useful functions, and a basic notion of type classes. In addition to this, Funky provides a foreign function interface to Python, allowing the programmer to embed and run Funky programs within a Python script.

6.1 Standard Library

Funky ships with a series of modules which make up a standard library of commonly used functions. These modules reside in a special directory that is packaged with the compiler such that they can be accessed from any Funky source file. A complete description of the standard library is available in Appendix D. A listing of each of the standard library files, along with a brief description of the services they provide, is below:

- `stdlib.fky` provides functions and values which are considered essential, and likely to be used in most programs written in Funky. It also contains the `otherwise` boolean, typically used as a 'catch all' in guard statements.
- `math.fky` provides important mathematical constants (e.g. e, π) and functions (e.g. `sqrt`, `abs`).
- `logic.fky` provides some logical operators that are not primitives (e.g. logical implication, exclusive or).
- `string.fky` provides various functions essential for string manipulation.
- `intlist.fky` implements a list of integers and the common functions required for list manipulation.
- `lists.fky` implements various infinite lists.
- `random.fky` implements a linear congruential generator for generating pseudorandom numbers. It contains various helper functions to make generating pseudorandom numbers easy.
- `dict.fky` implements a key-to-value dictionary map as a first-class function.

6.2 Typeclasses

Basic typeclasses are implemented in Funky, but they are not user facing. This means that they exist, but the user is not able to modify or create them. Typeclasses in Funky are used to restrict parametric polymorphism. It might be desirable to have a function that is polymorphic, but not completely generic – for instance, its parameter accepts some restricted subset of types, but not absolutely everything.

They are also represent a much weaker notion of a typeclass than in other languages like Haskell. As a result, their behaviour is slightly different. This is, in part, due to a surprising lack of relevant literature on this topic and a lack of time. Nonetheless, Funky's implementation makes a valuable addition to the language.

Perhaps the best example of typeclasses in Funky is the `Num` typeclass. This typeclass, as its name suggests, represents a *numeric* type. A type variable constrained by this typeclass can be instantiated to either a primitive `Integer` or `Float`, but nothing else. By defining built-in functions such as the `+` function as accepting parameters of the `Num` typeclass, we can avoid having to define two functions for the addition of `Integer`s and `Float`s, and 'kill two birds with one stone'.

Using the REPL, examining the type of `+` shows that it is $Num \to Num \to Num$. The way that the `+` operator is defined is such that the two parameters must be of the same type – that means the same type variable is used for the first and second parameter[8]. The consequence of this is that when the type variable constrained by `Num` is instantiated, it is instantiated for the first and second parameters. Therefore, if we curry the function, e.g. `(+) 1`, the type will be $Integer \to Integer$ to reflect the fact that addition between a `Float` and a `Integer` is disallowed.

[8]this is not a deficiency of Funky, it simply makes type checking more strict, which makes compiling to different targets simpler because there are stronger guarantees (you are always guaranteed that two things are the same type when added).

We illustrate this in the REPL:

```
Lazy evalation disabled.
Startup completed (0.174s).

funkyi (0.9.0) repl
Ready!
For help, use the ':help' command.

funkyi> :type (+)
(+) :: Num -> Num -> Num
funkyi> :type (+) 1
(+) 1:: Integer -> Integer
```

`Num` is one of a few typeclasses in Funky. The others are:

- `Stringable` – this typeclass is made up of the types `Float`, `Integer`, and `Bool`. It represents 'data types which can be converted into a string'.
- `Intable` – this typeclass is made up of the types `Float`, `String`, and `Bool`. It represents 'data types which can be converted into an integer'.
- `Floatable` – this typeclass is made up of the types `Integer`, and `String`. It represents 'data types which can be converted into a float'.

6.3 Foreign Function Interface

It is possible to embed Funky code within a Python script and run it. This is facilitated by the fact that Funky compiles to Python code – it is therefore relatively straightforward to create an interface that allows the user to reconcile Funky and Python code.

Some programmers may find this useful. For example, some problems are better suited to be solved functionally rather than imperatively. By creating a synergy between Funky and Python, the programmer can get the 'best of both worlds'.

It is worth emphasising that these foreign function calls can be performed in an *entirely different Python project* as long as the compiler is installed. The project does not have to use Funky in any other way and can instead just make use of the foreign function interface.

Example With the Funky compiler installed, the programmer may use the following import in their Python code:

```
from funky.ffi import funky_prog
```

The programmer can then define a Funky program in their code:

```
example_prog = funky_prog("""
module example_prog with

    factorial 0 = 1
    factorial n = n * factorial (n - 1)

    main = factorial 5
""")
```

A Python callable is created under the identifier `example_prog`, which when called, yields the result of compiling and running the Funky code.

```
print(example_prog()) # prints '120'
```

It is worth noting that the data returned from foreign function interface calls are first-class Python values. In other words, if the embedded Funky program, say, returns an integer from its main method, the value returned from the foreign function interface will be an integer too, and can therefore be manipulated using all of the standard Python machinery[9].

This means code like the following is perfectly valid:

```
print(example_prog() + 5) # prints '125'
```

Use in Testing The Funky compiler and its tests are written in Python. The foreign function interface proved to be very useful for testing; it allows the testing code to directly run the Funky source being tested from within the test script, and check the result, without any special accommodations.

The author is deeply interested in this concept and wishes to expand it further. Unfortunately, due to time constraints, this has not been possible. The author therefore plans to expand Funky to support foreign function interfaces further after the project has been completed and formally marked.

7 Reflections

This section is a general reflection on the project now that it is complete.

7.1 Accomplishments

The accomplishments of the project are touched on throughout the dissertation. This section presents the parts of the project that the author considers its biggest achievements.

Original Requirements Fulfilled The original requirements set out in the project proposal have all been fulfilled, with the exception of compile-to-C – however, this has been redressed as discussed in Section 5.2.1. It was forewarned by the project supervisor at the very start of the year that the project appeared challenging, so the fact that all of the requirements have been met is a personal achievement.

Extras Implemented The language and compiler go beyond just the original requirements and implements some additional features. These are discussed in detail in Section 6.

Turing Complete The language is Turing complete, as shown in Section 5.1.4. It is possible to implement a Turing machine emulator and the lambda calculus within Funky – therefore, it is Turing complete.

Highly Polished The end product is highly polished. Significant time and energy has gone into making the compiler robust, bug-free, and easy to use.

Extensive Logging Logging is performed nearly everywhere in the code, making debugging easy.

[9]if a user-defined data structure is the output, the result from the foreign function interface will be the compiled program's internal representation of the data structure – a Python object.

Extensive Testing Funky has been tested thoroughly. Additionally, Funky has a suite of unit tests that are very useful for ensuring that changes to the compiler do not break existing functionality.

7.2 Problems and Future Work

Like any project, Funky is not perfect. An honest analysis of the problems is presented in this section in the hopes that readers undertaking a similar project can learn from these mistakes.

No Polymorphism for Algebraic Data Types The author considers this the single biggest weakness with the programming language. Unfortunately, polymorphic algebraic data types are not currently implemented. The practical implications of this means that parameters in the constructors for algebraic data types must be concrete types. It is not currently possible to declare a list of a generic type a – instead, if you wanted a list of integers, you would have to declare it as its own algebraic data type.

The following approaches were considered to solve this problem:

- Integrating polymorphic data types into the type inferencer. Unfortunately, this proved itself to be more challenging than anticipated, and the idea lost progression as a result of favouring the implementation of more pressing features.
- Adding syntax to support a 'special variable' in algebraic data type constructors that can accept any type. This idea was quickly dismissed as it would have a catastrophic effect for type safety; any function that used the data type would introduce 'type uncertainty' to itself.
- Creating a pre-processor that would allow the user to define multiple algebraic data types succinctly by exploiting macro expansions into the compiler. This is not unlike how C programmers might use the C pre-processor. For instance, the programmer would be able to use a macro such as `FORALL x in (Integer, Float); newtype List$x = Cons$x $x List$x | Nil$x; ENDFORALL` to define two data types `ListInt` and `ListFloat`. This would work, but it feels like a 'hack' and does not feel suitable for a high-level programming language.

In the end, none of the above approaches seemed satisfactory, so time was spent elsewhere improving the compiler.

No CPS Conversion All programs can be rewritten as tail calls using by converting them to continuation passing style (CPS). Continuation passing style is a programming style in which functions pass control onto a continuation, which specifies what happens next instead of returning values. Mechanical procedures exist for translating functional procedures to continuation passing style. The advantage of doing this is that converted functions are tail-recursive, meaning it is possible to apply tail-call optimisation.

If tail-call optimisation was in place, we could avoid allocating a new stack frame for each function call because the callee will simply return the value that it gets from the called function. Tail-call optimised recursive functions occupy constant stack space, so the problem of recursion overflow is entirely avoided.

Funky does not currently implement CPS conversion, and by extension, it has no tail-call optimisation. This is due to time constraints of the project.

The author hopes to extend the Funky programming language to support these features in the future.

8 Conclusion

The overall aim of the project was to develop a functional programming language that is welcoming to programmers new to the functional paradigm. The project delivers a programming language that matches this description, accompanied by a compiler and REPL, that achieves the goals set out in the project proposal.

Section 2 has documented the inspiration and motivation for the Funky programming language, justified its design choices, and explained the principles maintained to ensure that the language is user-friendly.

Section 4 has given a comprehensive overview of the Funky compiler, describing the process of compiling Funky code from end-to-end and elaborating on the details of each phase of compilation.

Section 5 has evaluated the project in its entirety against the requirements set out in the project proposal and has shown that these requirements have been met, as well as shown that both the Funky compiler and compiled Funky programs are reasonably performant through a series of benchmarks.

Section 6 has showcased some of the features built into Funky that were *not* necessitated by the project proposal's requirements.

Finally, Section 7 has reflected on the project as a whole with an analysis of the merits and shortcomings of the finished product.

Overall, working on the project and developing a compiler has been an informative and enjoyable experience. The project has been challenging, but also rewarding. The theory and implementation of compilers spans across many sub-disciplines of computer science – the unification of these topics into a single coherent product has been very fulfilling.

References

A. V. Aho, M. S. Lam, R. Sethi, and D. J. Ullman, *Compilers: Principles, Techniques, and Tools*, 2nd ed. Menlo Park, Massachusetts: Pearson, 1988.

T. Mogensen, *Basics of Compiler Design*, anniversary ed. Copenhagen, DK: University of Copenhagen, 2007.

K. D. Cooper and L. Torczon, *Engineering a Compiler*, 2nd ed. San Francisco, California: Morgan Kaufmann, 2004.

J. Hughes, "Why Functional Programming Matters," in *Research Topics in Functional Programming*, D. A. Turner, Ed. Boston, MA, USA: Addison-Wesley, 1990, pp. 17–42.

S. Marlow *et al.*, "Haskell Language Report," 2010, accessed 03/02/19. [Online]. Available: http://www.haskell.org/

H. P. Barendregt, *The Lambda Calculus: Its Syntax and Semantics*, 2nd ed. North Holland Publishing Co., 2014.

A. R. Meyer, "What is a model of the lambda calculus?" *Information and Control*, vol. 52, no. 1, pp. 87–122, 1982.

G. van Rossum, B. Warsaw, and N. Coghlan. (2001) Python Enhancement Proposal 8 – Style Guide for Python Code. Accessed 07/03/2019. [Online]. Available: https://www.python.org/dev/peps/pep-0008/

D. E. Knuth, "On the Translation of Languages from Left to Right," *Information and Control*, vol. 8, no. 6, pp. 607–639, 1965.

F. L. DeRemer, "Practical translators for LR(k) languages." Ph.D. dissertation, Massachusetts Institute of Technology, 1969.

B. Lang, "Deterministic Techniques for Efficient Non-Deterministic Parsers," vol. 14, 07 1974, pp. 255–269.

S. McPeak and G. C. Necula, "Elkhound: A fast, practical GLR parser generator," in *International Conference on Compiler Construction*. Springer, 2004, pp. 73–88.

L. Maranget, "Compiling Pattern Matching to Good Decision Trees," in *Proceedings of the 2008 ACM SIGPLAN Workshop on ML*, ser. ML '08. New York, NY, USA: ACM, 2008, pp. 35–46. [Online]. Available: http://doi.acm.org/10.1145/1411304.1411311

R. Hindley, "The Principal Type-Scheme of an Object in Combinatory Logic," *Transactions of the American Mathematical Society*, vol. 146, pp. 29–60, 1969.

R. Milner, "A theory of type polymorphism in programming," *Journal of Computer and System Sciences*, vol. 17, pp. 348–375, 1978.

L. Damas, "Type assignment in programming languages," Ph.D. dissertation, The University of Edinburgh, 1985, CST-33-85.

R. Tarjan, "Depth first search and linear graph algorithms," *SIAM Journal on Computing*, vol. 1, no. 2, 1972.

J. Fisher. (2018) A lambda calculus interpreter in Haskell. Accessed 01-03-2019. [Online]. Available: https://jameshfisher.com/2018/03/15/a-lambda-calculus-interpreter-in-haskell

Appendices

A Submission Structure

The project is given as a single `.zip` archive containing all of the relevant material. The directory structure of the archive is as follows:

```
ellis_1667900_submission.zip
├── README.txt
├── funky-1.0.0.tar.gz
├── funky
│   ├── funky
│   │   └── ...
│   ├── sample_programs
│   │   └── ...
│   ├── setup.py
│   └── ...
├── demonstration
│   └── ...
├── logos
│   └── ...
├── ellis_1667900_report.pdf
├── litreview.pdf
├── proposal.pdf
└── projectinspection.pdf
```

- `README.txt` directs the reader toward this appendix to understand the structure of the submission, and toward Appendix B for installation instructions, just in case it is unclear where to find the most important pieces of documentation.
- `funky-1.0.0.tar.gz` is a source distribution for the Funky compiler. This is essentially a 'ready made' package for the compiler, and is what most users will perform the installation with. Installation instructions are are provided in Appendix B.
- `funky` is a Python package for the compiler. The compiler is packaged using the standard Python utility `setuptools`.
- `funky/funky` is a directory containing the compiler source code.
- `funky/sample_programs` is a directory containing a plethora of example programs written in Funky, along with a basic script to compile them.
- `funky/setup.py` is the build script for the compiler.
- `demonstration` is a directory containing materials used to showcase Funky during the final presentation.
- `logos` contains various logos for Funky.
- `ellis_1667900_report.pdf` is this report.
- `litreview.pdf` is the literature review from the first semester.
- `proposal.pdf` is the project proposal from the first semester.
- `projectinspection.pdf` is the slides used for the project inspection in the first semester.

The Funky compiler's source code can be found under the directory `funky/funky`. The code is structured as follows:

```
funky/
├── _version.py ............ Houses a global version variable for Funky.
├── compiler.py ............ Integrates compiler machinery to compile end-to-end.
├── ds.py .................. Useful, non-specific data structures.
├── exitcode.py ............ Exit codes for Funky and what the correspond to.
├── ffi.py ................. Python foreign function interface implementation.
├── globals.py ............. Global variables used in Funky.
├── cli/
│   ├── compiler_cli.py .... Implements the funky command.
│   ├── repl.py ............ Implements the funkyi command
│   └── verbosity.py ....... Functions for managing logger verbosity.
├── corelang/
│   ├── builtins.py ........ Funky's built-in functions and types.
│   ├── coretree.py ........ Data structure representing the core tree.
│   ├── sourcetree.py ...... Data structure representing the syntax tree.
│   └── types.py ........... Data structures used for type inference.
├── desugar/
│   ├── desugar.py ......... Machinery for desugaring the syntax tree.
│   └── maranget.py ........ Maranget's algorithm for compiling pattern matching.
├── generate/
│   ├── gen.py ............. Framework for code generators.
│   ├── gen_haskell.py ..... Haskell code generator.
│   ├── gen_python_lazy.py . Lazy Python code generator.
│   ├── gen_python_strict.py Strict Python code generator.
│   └── runtime/
│       ├── haskell_runtime.py ....... Runtime for the Haskell code generator.
│       ├── python_runtime_lazy.py ... Runtime for the lazy Python code generator.
│       └── python_runtime_strict.py . Runtime for the strict Python code generator.
├── imports/
│   └── import_handler.py .. Handler for imports.
├── infer/
│   ├── infer.py ........... Type inference machinery.
│   └── tarjan.py .......... Tarjan's algorithm for strongly-connected components.
├── libs/
│   └── ... ................ Funky standard library.
├── parse/
│   ├── fixity.py .......... Fixity resolution machinery.
│   ├── funky_lexer.py ..... Funky lexer machinery.
│   └── funky_parser.py .... Funky parser machinery.
├── rename/
│   └── renamer.py ......... Renamer and sanity check machinery.
├── tests/
│   └── ... ................ Unit tests for the Funky compiler.
└── util/
    ├── color.py ........... Utility for coloured terminal output.
    ├── orderedset.py ...... Utility for ordered sets.
    └── specialchars.py .... Unicode characters.
```

(__init__.py files have been omitted for brevity.)

B Installation Instructions

The Funky compiler is written in Python 3 and packaged using `setuptools`. On a UNIX-like system, install `python3` and `python3-pip` if they are not already installed. This should be done using a package manager, where possible. Then, to install the compiler, run `pip3 install funky-1.0.0.tar.gz` (found in the toplevel directory of the submission) and restart your terminal.

If everything went well, the Funky compiler (and its dependency, PLY) should be installed to your local machine. You should now be able to run the command `funky` to use the compiler and `funkyi` to access the REPL.

You should *not* need root to perform the installation of the package itself, but if permission-related errors arise, try running the installation again with super-user priveleges.

The Funky compiler and REPL, along with this installation process, has been tested and shown to be working on macOS Mojave 10.14.3 and Ubuntu 18.04.

Typical Installation A typical installation of the compiler on a fairly standard UNIX-like system might resemble the following:

```
$ sudo $PACKAGE_MANAGER install python3 python3-pip
  ...
$ unzip ellis_1667900_submission.zip
  ...
$ cd ellis_1667900_submission/
$ pip3 install funky-1.0.0.tar.gz
  ...

(restart terminal)

$ funky -V
funky 1.0.0
```

C Project Log

Semester One	
Week	Plan
1	Read literature regarding compilation of functional programming languages and created notes. Read ahead in Compilers & Languages module to create a basic understanding of the process of compilation.
2	Finalised mental model of the project, created a draft project specification and begun to draft project proposal. Read numerous texts regarding compilation of function languages casually.
3	Drafted language specification. Defined language syntax, rules, and style. Written up various 'sample programs' to give a preliminary feel of the language before the compiler is developed. These will be used for test-driven development.
4	Finalised and submitted project proposal. Started looking at various different tools and libraries in Python that could aid the development of the compiler. Most notable discovery is PLY, which is a Python implementation of the UNIX-standard tools `lex` and `yacc`.
5	Created Python package for the compiler using `setuptools`. Created git repository on GitHub for version control.
6	Begun some basic prototyping/feasibility testing to see if it is simple enough to lex and parse Funky source code without the assistance of an external tool.
7	Made a start on the literature review. Collected all of my sources into a single BibTeX file and created a template for the literature review document. More reading.
8	Significant work on literature review. Reviewed all of the sources I plan to reference in the final report. Finalised and submitted literature review.
9	More prototyping and feasibility testing.
10	Preparation for project briefing. Created slides to demonstrate progress so far.
11	Took a short break from project work to focus on assignments for the end of first semester. Further reading.

Winter Break Started serious development of the compiler. Decided it would take too much time to write a custom parser library, so instead, the project will use PLY. Rewritten prototypes so far to use the PLY library. Implemented a framework for the general 'architecture' of the compiler, i.e. the flow of information from end-to-end. Completed a prototype implementation of the command line interface, as well as the lexing, parsing, fixity resolution, and renaming compilation phases, and made a start on the desugaring phase. Made a small start on Hindley-Milner type inference, to be completed in the second semester.

Semester Two	
Week	Plan
1	Finalised the desugaring phase of compilation and made a more significant start on the type-inference phase. The desugaring phase involved implementing Maranget's algorithm for compiling pattern matching to efficient decision trees. Implemented basic type inference using the Hindley-Milner type system (including polymorphism) and implemented Tarjan's algorithm for finding strongly connected components in a graph. Also fixed a few bugs and added command line flags for dumping the output of each phase to `stdout`.
2	Emailed Prof. Uday Reddy with a question about type inference and received a helpful response. Finalised type inference phase and tested it extensively. This week also involved significant refactoring of previous work, i.e. removing redundant code, modified the desugarer so that it separates type aliases/construction definitions from the toplevel let. Specific work on the type inferencer this week involved supporting type-inferring algebraic data types, which was a little more challenging than anticipated.
3	Diagnosed and fixed state-related error in implementation of Maranget's algorithm. Discussed with my supervisor how to best proceed with code generation. Researched C code generation, but resources seem limited. I am aware of a technique that can be employed to do this, but the implementation remains challenging. We agreed to create a Python code generator as a 'backup' in case compiling to C is too difficult. This ensures that there is still a working end-to-end product. Worked on a Python code generator and added support into the compiler for compiling to different targets, i.e. the user can specify at the command line which file they want to compile to. Now have a working end-to-end prototype, with some bugs.
4	Generally uneventful – small modifications and bugfixes to the compiler. Made a start on the final report.
5	More work on the report, created a rough first draft. Numerous bugfixes. Started developing a REPL for Funky. The user can launch 'funkyi' from the command line to spin up an interactive prompt where they can write 'temporary' programs in Funky.
6	Created a Haskell code generator extension to the language just for fun. Finalised the REPL as well as making it more efficient. Changed the syntax in certain places to be more 'Pythonic'. Examples of this include making if statements take the form `true_part if cond else false_part`, like in Python, rather than what it was previously, `if cond then true_part else false_part`, which is more like Haskell. I've also changed '&&' and '——' to 'and' and 'or' respectively, to be more like Python.
7	Numerous bug fixes. Added code to support infix function definitions. Tidied up the parser and made it more efficient. Made a start on an import system, as well as a standard library. Optimisations – pruning mechanism to remove any unused declarations from the Funky source code. Added `--target=intermediate` code generator which writes the intermediate code to a file.
8	General polishing of the compiler to prepare for the demonstration. More logging in the codebase, and various bugfixes. The compiler/REPL will now complain when the entire program (or the last statement in the REPL) evaluates to a function type, as these cannot be outputted. REPL now supports import statements from the prompt or from the command line. This is useful if you want to open a file and play around with the functions it defines. Also, you can now inspect typeclasses from the REPL. Added builtin type conversion functions.
9	More polishing of the compiler to prepare for the demonstration. Better state management in the REPL. Added `fail` builtin function for reporting errors at runtime, alongside a special variable `undefined`. Implemented lazy Python code generator – it is now possible to declare and use infinite data structures. Added `--dump-pretty` flag to pretty print source code. Added colourised and unicode output.
10	Preparing slides for demonstration. More polishing. Added laziness implementation for strings, which was previously missing.
11	Finishing touches on the compiler, as well as finalising the report.

D Funky Standard Library

The following appendix contains the source code of Funky's standard library for reference. The Funky standard library is discussed in Section 6.1.

This appendix should serve well as a reference to programmers wishing to make use of Funky's standard library in the future.

(appendix continued overleaf)

```
module stdlib with

    otherwise = True

    # Identity function
    id x = x

    # Constant function
    const x = lambda _ -> x

    # Boolean negation
    not True  = False
    not False = True

    # Flipping function application
    flip f x y = f y x

    # Function composition
    compose f g x = f (g x)

    # Repetition
    frepeat f 0 x = x
    frepeat f n x given n < 0     = fail "Cannot repeat for n < 0."
                  given otherwise = f (frepeat f (n - 1) x)

    # Fixpoint operator
    fix f = f (fix f)
```

Listing 27: Functions and values which are considered essential and likely to be used in most Funky programs.

```
# standard math functions in funky
module math with

    import "stdlib.fky"

    # mathematical constants
    e   = 2.718281828459045
    pi  = 3.141592653589793
    tau = 2.0 * pi

    abs n given n >= 0    = n
          given otherwise = -n

    signum n given n > 0     = 1
             given n < 0     = -1
             given otherwise = 0

    factorial 0              = 1
    factorial n given n >= 0 = n * factorial (n - 1)

    gcd a 0 = a
    gcd a b = gcd b (a % b)

    lcm a b = (a / g) * b
              with g = gcd a b

    coprime x y = gcd x y == 1

    recip x = 1.0 / x

    exp x = e ** x

    sqrt = flip (**) 0.5

    even n = n % 2 == 0
    odd    = not ~compose~ even

    max a b given a >= b    = a
            given otherwise = b

    min a b given a <= b    = a
            given otherwise = b
```

Listing 28: Important mathematical constants (e.g. *e* and *pi*) and functions (e.g. `sqrt` and `abs`)

```
# common logic operators in funky
module logic with

    # XOR operator
    True    ~xor~  False  =  True
    False   ~xor~  True   =  True
    _       ~xor~  _      =  False

    # logical implication
    True   ~implies~  False  =  False
    _      ~implies~  _      =  True
```

Listing 29: Common logical operators which are not primitives.

```
# common string operations in funky
module string with

    import "stdlib.fky"

    slice m n = (slice_from m) ~compose~ (slice_to n)

    str_head = slice_to 1
    str_tail = slice_from 1

    strlen "" = 0
    strlen st = 1 + strlen (slice_from 1 st)

    char_at n = slice n (n + 1)

    str_reverse "" = ""
    str_reverse st = str_reverse (str_tail st) ++ str_head st

    # 'comfort' function for Python users.
    print = id
```

Listing 30: Various functions essential from string manipulation.

```
# Useful lists in Funky.
module lists with

    import "intlist.fky"
    import "math.fky"

    ones   = Cons 1 ones
    nats   = iterate ((+) 1) 1
    znats = Cons 0 nats
    negs   = map negate nats

    evens = iterate ((+) 2) 0
    odds   = iterate ((+) 2) 1

    powers_of_two = iterate ((*) 2) 1

    fibs = Cons 0 (Cons 1 (zipwith (+) fibs (tail fibs)))

    primes = sieve (tail nats)
             with sieve (Cons p xs) = Cons p (sieve (filter (lambda x -> x % p > 0) xs))

    squares = zipwith (*) nats nats

    flip_flop = Cons 0 (Cons 1 flip_flop)
```

Listing 31: Various infinite lists.

```
# deterministic random number generator in funky
module random with

    # linear congruential generator -- see:
    # http://en.wikipedia.org/wiki/Linear_congruential_generator
    lcg a c m seed = (a * seed + c) % m

    # initialise our lcg -- these parameters are used by many ANSI C compilers
    # so are known to be reasonably good
    rand = lcg 1103515245 12345 (2 ^ 31)

    # helper functions
    randint seed lbound ubound = (rand seed) % (ubound - lbound) + lbound

    randfloat seed = to_float (randint seed 1 big_num) / to_float big_num
                     with big_num = 2 ^ 31
```

Listing 32: Linear congruential generator for generating pseudorandom numbers.

```
# module providing basic dictionary functionality.
# dictionaries are implemented as first-class functions.
module dict with

    import "stdlib.fky"

    # An empty dict is a function returning some default value
    empty_dict default = const default

    # When we add a pair to a dict, we wrap the existing dict function in a new
    # dict function that checks the corresponding key. If it matches, we return
    # `value`. Otherwise, we delegate to the nested dict.
    add_pair key value dict = lambda k -> value if k == key else dict k
```
Listing 33: A key-to-value dictionary map as a first-class function.

```
module intlist with

    import "stdlib.fky"

    newtype IntList = Cons Integer IntList | Nil

    unit x = Cons x Nil

    # basic functions
    Nil         ~concatenate~ ys = ys
    (Cons x xs) ~concatenate~ ys = Cons x (xs ~concatenate~ ys)

    head (Cons x _) = x
    head _          = fail "No head element of empty list!"

    tail (Cons _ xs) = xs
    tail _           = fail "No tail of empty list!"

    last (Cons x Nil) = x
    last (Cons _ xs)  = last xs
    last _            = fail "No last element of empty list!"

    init (Cons x Nil) = Nil
    init (Cons x xs)  = Cons x (init xs)

    null Nil = True
    null _   = False

    length Nil          = 0
    length (Cons _ xs) = 1 + length xs

    # list transformations
    map f Nil          = Nil
    map f (Cons x xs) = Cons (f x) (map f xs)

    reverse Nil          = Nil
    reverse (Cons x xs) = reverse xs ~concatenate~ (Cons x Nil)

    # list folds
    foldr f z Nil          = z
    foldr f z (Cons x xs) = f x (foldr f z xs)

    foldl f z Nil          = z
    foldl f z (Cons x xs) = foldl f (f z x) xs

    any p Nil                            = False
    any p (Cons x xs) given p x          = True
                      given otherwise = any p xs

    all p Nil                            = True
    all p (Cons x xs) given p x          = all p xs
                      given otherwise = False
```

```
sum     = foldr (+) 0
product = foldr (*) 1

maximum xs = foldr (lambda a b -> a if a >= b else b) (head xs) xs
minimum xs = foldr (lambda a b -> a if a <  b else b) (head xs) xs

repeat x = Cons x (repeat x)

replicate 0 x = Nil
replicate n x = Cons x (replicate (n - 1) x)

take 0 _           = Nil
take n Nil         = fail "Not enough elements in list."
take n (Cons x xs) = Cons x (take (n - 1) xs)

drop 0 xs          = xs
drop n (Cons x xs) = drop (n - 1) xs
drop n _           = Nil

takewhile p Nil         = Nil
takewhile p (Cons x xs) given p x         = Cons x (takewhile p xs)
                        given otherwise = Nil

dropwhile p xs = drop (length (takewhile p xs)) xs

x `elem`    ys = any ((==) x) ys
x `notelem` ys = not (x `elem` ys)

filter p Nil = Nil
filter p (Cons x xs) given p x         = Cons x rest
                     given otherwise = rest
                     with rest = filter p xs

nth 0 (Cons x xs)                     = x
nth n (Cons x xs) given n > 0         = nth (n - 1) xs
                  given otherwise = fail "Negative list index."
nth _ _                               = fail "List index out of bounds."

zipwith f (Cons x xs) (Cons y ys) = Cons (f x y) (zipwith f xs ys)
zipwith _ _ _                     = Nil

iterate f x = Cons x (iterate f (f x))

loop xs = xs `concatenate` (loop xs)

join Nil         _     = ""
join (Cons x Nil) delim = to_str x
join (Cons x xs)  delim = to_str x ++ delim ++ join xs delim

pprint list = "[" ++ join list ", " ++ "]"
```

Listing 34: A list of integers and common functions for list manipulation.

E Funky Sample Programs

The following appendix contains a plethora of code examples to perform various common tasks written in the Funky programming language. It is hoped that these examples will give credence to Funky's expressivity and demonstrate clearly that it is robust.

These examples should also serve quite faithfully as a guide for users who are familiarising themselves with the language.

(appendix continued overleaf)

```
module listlength with

    newtype List = Cons Integer List | Nil

    length Nil         = 0
    length (Cons _ xs) = 1 + length xs

    main = length my_list
           with my_list = Cons 1 (Cons 2 (Cons 3 Nil))
```

Listing 35: Finding the length of a list in Funky.

```
module quicksort with

    import "stdlib.fky"
    import "intlist.fky"

    quicksort Nil         = Nil
    quicksort (Cons x xs) = (quicksort s) ~concatenate~ unit x ~concatenate~ (quicksort l)
                            with s = filter ((>)  x) xs
                                 l = filter ((<=) x) xs

    main = pprint (quicksort my_list)
           with my_list = Cons 5 (Cons 1 (Cons 7 (Cons 7 (Cons 3 Nil))))
```

Listing 36: The standard quicksort algorithm for integers in Funky.

```
module treesort with

    import "stdlib.fky"
    import "intlist.fky"

    newtype Tree = Branch Tree Integer Tree | Empty

    insert e Empty                          = Branch Empty e Empty
    insert e (Branch l v r) given e <= v    = Branch (insert e l) v r
                            given otherwise = Branch l v (insert e r)

    inorder Empty = Nil
    inorder (Branch l v r) = inorder l ~concatenate~
                             unit v    ~concatenate~
                             inorder r

    treesort list = inorder (foldr insert Empty list)

    main = treesort my_list
           with my_list = Cons 5 (Cons 1 (Cons 7 (Cons 7 (Cons 3 Nil))))
```

Listing 37: Treesort algorithm for integer in Funky.

```
module peano with

    err_neg = "Cannot represent negative numbers with Nat."

    newtype Nat = Succ Nat | Zero
    one = Succ Zero

    int_to_nat 0 = Zero
    int_to_nat n = Succ (int_to_nat (n - 1))
    int_to_nat _ = fail err_neg

    nat_to_int Zero     = 0
    nat_to_int (Succ m) = 1 + nat_to_int m

    m ~add~ Zero     = m
    m ~add~ (Succ n) = add (Succ m) n

    m        ~sub~ Zero     = m
    Zero     ~sub~ n        = fail err_neg
    (Succ m) ~sub~ (Succ n) = sub m n

    m ~mul~ Zero        = Zero
    m ~mul~ (Succ Zero) = m
    m ~mul~ (Succ n)    = m ~add~ (m ~mul~ n)

    Zero ~pow~ Zero        = undefined
    m    ~pow~ Zero        = one
    m    ~pow~ (Succ Zero) = m
    m    ~pow~ (Succ n)    = m ~mul~ (m ~pow~ n)

    main = nat_to_int (pow (int_to_nat 2) (int_to_nat 5))
```

Listing 38: Implementation of Peano natural numbers in Funky.

```
module turing_machine with

    newtype Alphabet = One | Zero | Blank
    newtype Tape = TapeCons Alphabet Tape | TapeEnd
    newtype Machine = TM Tape Alphabet Tape

    newtype Program = ProgramCons (Machine -> Machine) Program | ProgramEnd

    read (TM _ h _) = h
    write s (TM ls _ rs) = TM ls s rs

    shift_left (TM (TapeCons l ls) h rs) = TM ls l (TapeCons h rs)
    shift_left (TM TapeEnd h rs)         = TM TapeEnd Blank (TapeCons h rs)

    shift_right (TM ls h (TapeCons r rs)) = TM (TapeCons h ls) r rs
    shift_right (TM ls h TapeEnd)         = TM (TapeCons h ls) Blank TapeEnd

    fresh_machine = TM TapeEnd Blank TapeEnd

    instruction_swap (TM ls One rs)  = (TM ls Zero rs)
    instruction_swap (TM ls Zero rs) = (TM ls One rs)
    instruction_swap (TM ls x rs)    = (TM ls x rs)

    run ProgramEnd machine          = machine
    run (ProgramCons i is) machine = run is (i machine)

    main = run instructions fresh_machine
           with instructions = ProgramCons (write One) (
                               ProgramCons shift_right (
                               ProgramCons (write One) (
                               ProgramCons shift_right (
                               ProgramCons (write One) (
                               ProgramCons instruction_swap (
                               ProgramCons shift_left (
                               ProgramCons instruction_swap (
                               ProgramCons shift_left ProgramEnd))))))))
```

Listing 39: A sample implementation of a Turing machine in Funky.

```
module lambdacalc with

    # Inspired by Jim Fisher's blog post code, available at:
    # https://jameshfisher.com/2018/03/15/a-lambda-calculus-interpreter
    # -in-haskell.html

    newtype LambdaExpression = App   LambdaExpression LambdaExpression
                             | Abs   LambdaExpression
                             | Var   Integer
                             | Const Integer

    eval (App fun arg) = match eval fun on
                             Abs body -> eval (sub 0 body)
                             x        -> App x arg
                         with sub n (App e1 e2) = App (sub n e1) (sub n e2)
                              sub n (Abs e)     = Abs (sub (n + 1) e)
                              sub n (Var y)     = arg if n == y else Var y
                              sub n x           = x
    eval x = x

    main = eval my_expr
           with my_expr = App (App (Abs (Abs (Var 1))) (Const 4)) (Const 5)
```

Listing 40: A interpreter for the lambda calculus in Funky.

```
module mutualrecursion with

    even 0 = True
    even n = odd  (n - 1)
    odd  0 = False
    odd  n = even  (n - 1)

    main = odd 15
```

Listing 41: Mutual recursion in Funky.

```
module test with

    import "stdlib.fky"

    double    = (*) 2
    increment = (+) 1

    # The 'compose' function is provided by stdlib.fky
    double_and_increment = increment ~compose~ double

    main = double_and_increment 100
```

Listing 42: Using imports from the standard library in Funky.

```
module fizzbuzz_list with

    import "stdlib.fky"

    newtype IList = ICons Integer IList | INil
    newtype SList = SCons String SList | SNil

    to_fizzbuzz n given n % 3 == 0 and n % 5 == 0 = "FizzBuzz"
                  given n % 3 == 0                = "Fizz"
                  given n % 5 == 0                = "Buzz"
                  given otherwise                 = to_str n

    range m n given m >  n     = INil
              given m == n     = ICons m INil
              given otherwise  = ICons m (range (m + 1) n)

    map f INil          = SNil
    map f (ICons i is)  = SCons (f i) (map f is)

    join SNil         delim = ""
    join (SCons x xs) delim = x ++ delim ++ join xs delim

    fizzbuzz n = map to_fizzbuzz (range 1 n)

    main = join (fizzbuzz 100) "\n"
```

Listing 43: Implementation of the classic interview question 'FizzBuzz' in Funky.

```
module randomfloat with

    import "random.fky"

    seed = 51780

    main = randfloat seed
```

Listing 44: Demonstrating the use of Funky's random library to generate a random float.

```
module caesar with

    import "stdlib.fky"
    import "random.fky"
    import "string.fky"

    newtype Maybe = Just Integer | Nothing

    alphabet = "abcdefghijklmnopqrstuvwxyz"

    index c ""                  = Nothing
    index c st given c == h     = Just 0
               given otherwise = match index c t on
                                    Nothing -> Nothing
                                    Just x  -> Just (x + 1)
               with
                h = str_head st
                t = str_tail st

    encode n "" = ""
    encode n st = encoded ++ encode n t
                  with h = str_head st
                       t = str_tail st
                       encoded = match index h alphabet on
                                   Nothing -> h
                                   Just i  -> char_at
                                              ((i + n) % (strlen alphabet))
                                              alphabet

    decode n = encode (-n)

    rot13 = encode 13

    main = encode key "this is a *secret message*"
           with key = rand 51773
```

Listing 45: Implementation of the Caesar cipher in Funky.

```
module dictionary with

    import "dict.fky"

    newtype Maybe = Just String | Nothing

    main = dict "Pete"
           with dict = add_pair    "John"     (Just "Smith")    (
                       add_pair    "George"   (Just "Michael")  (
                       add_pair    "Pete"     (Just "Wentz")    (
                       add_pair    "Michael"  (Just "Jackson")  (
                       empty_dict  Nothing
                       ))))
```

Listing 46: Using Funky's standard dictionary module to create a dictionary of names.

```
module lazy_lists with

    import "intlist.fky"
    import "math.fky"

    ones = Cons 1 ones
    nats = Cons 1 (map ((+) 1) nats)

    evens = Cons 2 (map ((+) 2) evens)
    odds  = Cons 1 (map ((+) 2) odds)

    main = take 20 evens
```

Listing 47: Defining lists lazily in Funky. Please compile with the –use-lazy flag.

```
module listindex with

    import "stdlib.fky"

    newtype List = Cons Integer List | Nil
    newtype Maybe = Just Integer | Nothing

    index e Nil = Nothing
    index e (Cons x xs) given e == x     = Just 0
                        given otherwise = match index e xs on
                                            Just x -> Just (x + 1)
                                            x      -> x

    main = index 7 my_list
           with my_list = Cons 5 (Cons 1 (Cons 7 (Cons 7 (Cons 3 Nil))))
```

Listing 48: Searching a list for the index of a particular item in Funky.

```
module fibonacci with

    fib 0 = 0
    fib 1 = 1
    fib n = fib (n - 1) + fib (n - 2)

    main = fib 10
```

Listing 49: Computing terms of the fibonacci sequence in Funky.

```
module pascals with

    import "stdlib.fky"
    import "intlist.fky"

    next_row vs = Cons 1 ((zipwith (+) vs (tail vs)) ~concatenate~ (Cons 1 Nil))
    nth_row n = frepeat next_row n (Cons 1 Nil)

    main = nth_row 30
```

Listing 50: Computes the nth row of Pascal's triangle in Funky.

```
module fix with

    fix f = f (fix f)

    main = fix (lambda x -> 0)
```

Listing 51: The fixpoint operator in Funky.

```
module sample_set with

    import "dict.fky"

    # we use a dictionary to represent a set of elements.
    empty_set = empty_dict False
    add elem set = add_pair elem True set

    newtype List = Cons Integer List | Nil

    has_duplicates list = aux list empty_set
                      with aux Nil s                    = False
                           aux (Cons x xs) s given s x  = True
                                             given otherwise =
                                                 aux xs (add x s)

    main = "has_duplicates my_list1 is " ++ to_str (has_duplicates my_list1)
           ++ "\n" ++
           "has_duplicates my_list2 is " ++ to_str (has_duplicates my_list2)
           ++ "\n" ++
           "has_duplicates my_list3 is " ++ to_str (has_duplicates my_list3)
           with my_list1 = Cons 5 (Cons 1 (Cons 7 (Cons 7 (Cons 3 Nil))))
                my_list2 = Cons 1 (Cons 2 (Cons 3 Nil))
                my_list3 = Cons 2 (Cons 1 (Cons 6 (Cons 1 Nil)))
```

Listing 52: Using Funky's standard dictionary library to implement a basic set.

```
module random_sequence with

    import "random.fky"

    seed = 51773

    rand_seq seed 0 = ""
    rand_seq seed n = to_str (randint seed 20 30)
                      ++ "\n" ++
                      rand_seq (seed + 1) (n - 1)

    main = rand_seq seed 10
```

Listing 53: Demonstrating the use of Funky's random library to generate a random sequence.

```
module expressiontree with

    newtype Expression = Const Float
                       | BinOp Expression
                               (Expression -> Expression -> Float)
                               Expression
                       | UnOp  (Expression -> Float)
                               Expression

    evaluate (Const x)     = x
    evaluate (BinOp a f b) = f a b
    evaluate (UnOp f x)    = f x

    add exp1 exp2 = (evaluate exp1) + (evaluate exp2)
    sub exp1 exp2 = (evaluate exp1) - (evaluate exp2)
    mul exp1 exp2 = (evaluate exp1) * (evaluate exp2)
    div exp1 exp2 = (evaluate exp1) / (evaluate exp2)

    test_exp = (BinOp (Const 5.0) add (BinOp (Const 3.0) div (Const 2.0)))

    main = evaluate test_exp
```

Listing 54: Representing and evaluating expression trees in Funky.

```
module fizzbuzz with

    import "stdlib.fky"

    fizzbuzz n = fizzbuzz_ 1 n

    fizzbuzz_ n m given n == m    = aux n
                  given otherwise = aux n ++ "\n" ++ fizzbuzz_ (n + 1) m
                  with aux x given x % 3 == 0 and x % 5 == 0 = "FizzBuzz"
                             given x % 3 == 0                = "Fizz"
                             given x % 5 == 0                = "Buzz"
                             given otherwise                 = to_str x

    main = fizzbuzz 100
```

Listing 55: An implementation of the classic interview question 'FizzBuzz' in Funky.

```
module parser with

    import "string.fky"

    newtype StringPair = P String String | Fail

    fst Fail = fail "Fail"
    fst (P s _) = s
    snd Fail = fail "Fail"
    snd (P _ s) = s

    return v = lambda inp -> P v inp
    failure  = lambda inp -> Fail

    item = lambda inp -> match inp on
                            "" -> Fail
                            _  -> P (str_head inp) (str_tail inp)

    parse p inp = p inp

    p ~sequence~ f = lambda inp -> match parse p inp on
                                       Fail     -> Fail
                                       P v out -> parse (f v) out

    p ~alt~ f = lambda inp -> match parse p inp on
                                  Fail     -> parse f inp
                                  P v out -> P v out

    sat p = item ~sequence~ (lambda x ->
            return x if p x else failure)

    digit = sat is_digit
            with is_digit "0" = True
                 is_digit "1" = True
                 is_digit "2" = True
                 is_digit "3" = True
                 is_digit "4" = True
                 is_digit "5" = True
                 is_digit "6" = True
                 is_digit "7" = True
                 is_digit "8" = True
                 is_digit "9" = True
                 is_digit _   = False

    char x = sat ((==) x)

    string "" = return ""
    string st = char h ~sequence~ (lambda v0 ->
                string t ~sequence~ (lambda v1 ->
                return (h ++ t)
                ))
                with h = str_head st
                     t = str_tail st
```

```
open_brace  = sat ((==) "[")
close_brace = sat ((==) "[")
comma       = sat ((==) ",")

many p = many1 p ~alt~ return ""

many1 p = p ~sequence~ (lambda v ->
          many p ~sequence~ (lambda vs ->
          return (v ++ vs)))

nat = many1 digit ~sequence~ (lambda xs ->
      return xs)

parse_intlist = char "[" ~sequence~ (lambda _ ->
                nat ~sequence~ (lambda _ ->
                many (
                    char "," ~sequence~ (lambda _ ->
                    nat))
                ) ~sequence~ (lambda y ->
                char "]" ~sequence~ (lambda z ->
                return "10"
                )))

main = parse_intlist "[1,2,3]"
```

Listing 56: A functional parser.

F Demonstration Code

The following appendix contains the code examples used in the project demonstration. They work well as a quick introduction to the language.

(appendix continued overleaf)

```
# Module are used to tell the compiler 'this is a Funky source file'
module basics_00 with

    # Comments are denoted with a '#' symbol -- this is a comment!

    # Variables are defined as you would expect from any high-level programming
    # language.
    name   =  "Bob"
    age    =  20
    third  =  0.33

    # Named function definitions can be declared using function application
    # notation, or equivalently, by creating a variable using a lambda
    # expression.
    double x    = x + x
    double_lam = lambda x -> x + x

    # For readability, named function definitions can use binary function
    # application syntax, as shown below
    x ~samesign~ y = x * y > 0

    # Functions can be defined in terms of other functions. Function
    # application is by juxtaposition. Like usual, function application is
    # right associative.
    quadruple x = double (double x)

    # Functions can be applied in binary notation for readability.
    some_variable = 10 ~samesign~ (-10)

    # Built-in operators are just infix functions! We can convert them to
    # postfix (i.e. a normal function) by wrapping them in parenthesis.
    example = 10 + 10 == (+) 10 10
```

Listing 57: The very basics of the Funky programming language.

```
# Slightly more advanced stuff in Funky.
module basics_01 with

    # Conditional expressions can be used with the if-else construct. The
    # syntax is the same as Python syntax.
    abs n = n if n >= 0 else -n

    # Guard statements are possible using the 'given' keyword.
    minor age given age <  18 = "You are a minor."
              given age >= 18 = "You are an adult."

    signum n given n <  0 = -1
             given n == 0 = 0
             given n >  0 = 1

    # Functions are first-class values -- they can be curried!
    increment = (+) 1

    # We can use local bindings for improved readability with the let and where
    # keywords.

    area r = let pi = 3.14
             in  pi * r ** 2.0

    energy wavelength = (h * c) / wavelength
                        with h = 6.62 * 10.0 ** (-34.0) # Planck's constant
                             c = 299792458.0            # Speed of light
```

Listing 58: Conditionals and local bindings.

```
# Pattern matching with literals.
module basics_02 with

    # Pattern matching can be performed in-expression with the match keyword.
    to_words n = match n on
                    0 -> "zero"
                    1 -> "one"
                    2 -> "two"
                    _ -> fail "I can't count that high!"

    # Or alternatively, implicit pattern matching can be used for control flow
    # in a more natural manner.
    is_zero 0 = "Yep, that's a zero alright."
    is_zero _ = "That's not a zero!"

    # Pattern matching is robust!
    True  ~xor~ False = True
    False ~xor~ True  = True
    _     ~xor~ _     = False

    True ~implies~ False = False
    _    ~implies~ _     = True

    # Lots of flexibility due to wildcards below -- the compiler handles it
    # just fine.
    f _     False True  = 1
    f False True  _     = 2
    f _     _     False = 3
    f _     _     True  = 4
```

Listing 59: Pattern matching over literals.

```
# Creating algebraic data types.
module basics_03 with

    # You can declare a new data type with the 'newtype' keyword.
    newtype Vector2D = V Float Float

    test_vector = V 5.2 6.1

    # Pattern matching can be performed against user-defined data types.
    fst (V x _) = x
    snd (V _ y) = y
    sum (V a b) (V x y) = V (a + x) (b + y)

    # Data types can be recursive (or even infinite, as we will see later!)
    newtype IntList = Cons Integer IntList | Nil

    test_list = Cons 5 (Cons 1 (Cons 7 (Cons 7 (Cons 3 Nil))))

    # Pretty print a list
    pprint list = "[" ++
                  aux list ++
                  "]"
                  with aux Nil = ""
                       aux (Cons x Nil) = to_str x
                       aux (Cons x xs)  = to_str x ++ ", " ++ aux xs

    # Given an IntList, find the total
    total Nil         = 0
    total (Cons x xs) = x + total xs

    # Given an IntList, find the maximum
    max Nil                         = fail "No maximum in empty list!"
    max (Cons x Nil)                = x
    max (Cons x xs) given x > m     = x
                    given x <= m = m
                    with m = max xs
```

Listing 60: Algebraic data types in Funky.

```
# Imports, etc
module basics_04 with

    # Funky has an import system -- you can use it to import definitions from
    # other files. There is a standard library available, which currently contains:
    #
    # dict.fky    -- a utility for creating key-value mappings
    # intlist.fky -- defines list of integers and useful functions for working
    #                with them
    # lists.fky   -- various infinite lists
    # logic.fky   -- logical operators, i.e. xor, logical implication
    # math.fky    -- math functions and constants
    # random.fky  -- pseudorandom number generator
    # stdlib.fky  -- standard library of basic utility functions
    # string.fky  -- string manipulation functions
    import "stdlib.fky"
    import "math.fky"

    # Imports can also be performed relative to the current file. For instance,
    # we can import one of the previous demo files:
    import "./basics_00.fky"

    # We can now use the definitions above in our code!
    import_test = gcd (double 15) (quadruple 7) #  == 2
    #             ^    ^           ^
    #             |    |           |
    #             |    +-----------+-- from basics_00.fky
    #             |
    #             +-- from math.fky
```

Listing 61: Importing external objects in Funky.

```
# Laziness!
module basics_05 with

    import "intlist.fky"

    # The list of ones is ONE followed by the list of ones.
    ones = Cons 1 ones

    # The head of the list of natural numbers is 1, and the tail
    # is the list of natural numbers, mapped with ((+) 1).
    nats  = Cons 1 (map ((+) 1) nats)

    # The list of natural numbers, but starting from zero.
    znats = Cons 0 nats

    # Even and odd numbers.
    evens = Cons 0 (map ((+) 2) evens)
    odds  = Cons 1 (map ((+) 2) odds)

    # Fibonacci sequence
    fibs = Cons 0 (Cons 1 (zipwith (+) fibs (tail fibs)))

    # List of prime numbers
    primes = sieve (tail nats)
             with sieve (Cons p xs) = Cons p (sieve (filter (lambda x -> x % p > 0) xs))

    squares = zipwith (*) nats nats
```

Listing 62: Creating infinite lists in Funky.