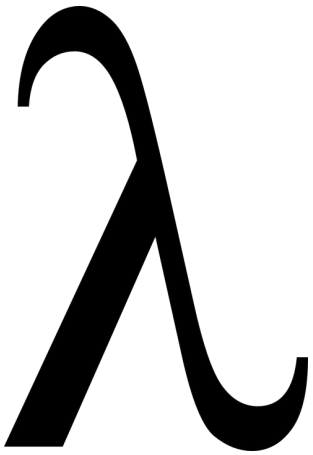
# Lambda Calculus



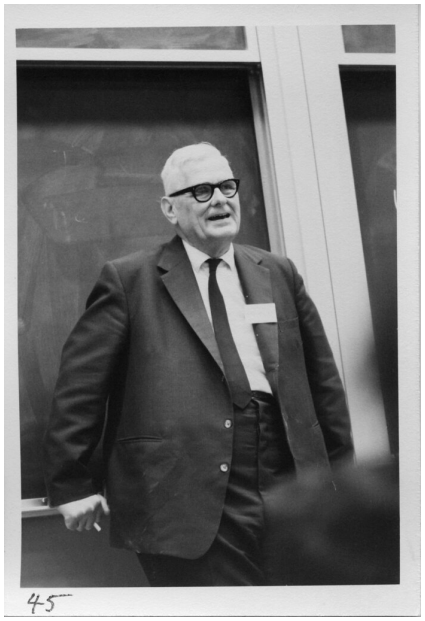
# CONTENTS

# 1 ORIENTATION

## 1.1 PROBLEM

Lambda calculus was developed to solve a centuries old problem, namely the Entscheidungsproblem that began with Leibniz. This problem asks if there exists an algorithm that given a mathematical statement, it can answer whether it is true or false. In 1936, Church and Turing proved that it is impossible to solve the Entscheidungsproblem (so the answer to this problem is undecidable), using their theorem which is now known as the Church-Turing thesis. This theorem states that a function defined on the natural numbers can be calculated with an effective method (mechanical procedure), if and only if it is computable on a Turing machine. To prove this theorem, Church developed lambda calculus; a model of computation that can simulate a Turing machine. Lambda calculus formalizes the concept of 'effective computability': what functions can computers compute using algorithms? In this way, any computable function can be evaluated and expressed using lambda calculus. It expresses computation based on function abstraction and application. So in lambda calculus, everything such as numbers, boolean values, if and else statements, etc. can be expressed as functions and their applications and abstractions. Later on we will see that these two concepts, function abstraction and function application, will serve as the core of lambda calculus. Lambda calculus lies at the foundation of many popular functional programming languages such as Haskell.

**Figure 1:** Alonzo Church




Its main drawback is that it only uses anonymous functions (so functions without names) for building everything such as numbers, Boolean's etc. In this regard, object oriented programming languages such as Python provide much cleaner and simpler code to manipulate objects and storing data to objects to use them again later. Moreover, lambda calculus can get messy really quickly, so object oriented programming languages like Python make it easier to understand your code, which is why they have more use cases.

## 1.2 APPLICATIONS

Lambda calculus seems abstract. What can it be used for? Well, there are several applications of lambda calculus in mathematics and computer science.

### 1.2.1 Fundamental understanding of computation

Lambda calculus provides a fundamental understanding of what it means for something to be computable. In other words, it formalized the study of computation using functions, which we are very familiar with. Since lambda calculus is a formal system, many popular functional programming languages have lambda calculus as its foundation such as Haskell and Lisp. Because of its usefulness, object oriented programming languages such as Python also implemented function definition using lambda calculus. Therefore, it is also possible to create your own programming language using the language of lambda calculus. Another application is in its use in research. Lambda calculus as a formal system is useful for modeling the behavior of different kinds of code before implementing them. Therefore, they are frequently used in test phases of research.

### 1.2.2 Applications in pure mathematics

Lambda calculus also has applications in (pure) mathematics such as category theory, topology, and logic. For example, they are used to understand certain types of categories, called cartesian closed categories, they are used to understand partial combinatory algebras (PCAs) which is a subfield of mathematical logic, and they are used in topology to understand homotopy type theory, the study of certain maps between topological spaces which allows for a (partial) classification of topological spaces, and abstract stone duality, which develops lambda calculus to be applied to topology. Homotopy type theory is one of the most exciting new developments in modern mathematics, providing a bridge between topology and mathematical logic.

## 1.3 MAIN CHALLENGES

The main challenges of this project would be to understand (untyped and typed) lambda calculus, implement lambda calculus in Python, create our own programming language with lambda calculus, and write efficient and clean code such that anyone reading our code would understand them. The first because lambda calculus is a completely new language for computation and a system within mathematical logic with its own rules and syntax. The second because lambda calculus is a formal system with its own rules, so creating code for this system from scratch will be challenging, but at the same time interesting and fun. The third, because we need to create an extensive library of methods and add functionality into our program (which may in addition lead to many errors and messy code) and the last one, because

lambda calculus is pretty sophisticated so writing clean code can become hard sometimes.

# 2 HOW DOES LAMBDA CALCULUS WORK?

## 2.1 FUNCTIONAL PROGRAMMING

Nearly every programming language can be sorted into two categories. The first, and most common group is called 'object oriented programming'. This is a category for every programming language that uses labels to store groups of data. Python is part of this group. Lambda calculus is part of the other smaller group: 'functional programming'. This group contains the languages that exclusively use functions to generate and transform data. Lambda calculus is one of the first instances of functional programming. Although in this modern era we mostly use languages such as Haskell.

## 2.2 LAMBDA CALCULUS

Lambda calculus is a simple language that only consists of so-called **lambda terms** and the manipulation of them. One of the defining properties of lambda calculus is that no names are attached to functions, which are called *anonymous functions*. These anonymous functions are called *lambda abstractions* in the language of lambda calculus. Lambda terms are recursively defined as seen below.

- <lambda term>:= <variable >| <abstraction >| <application >
- <abstraction>:= λ<variable >.<lambda term >
- <application>:= <lambda term > <lambda term >

The |-symbol is used to denote that any one of variable, abstraction, or application is a valid lambda term.
A 'variable is an identifier/placeholder/name/label for which we can use any letter $a, b, c$, etc. It is a dummy variable, meaning that it exists for the sole purpose of being a placeholder (to be substituted in the future).

Roughly speaking, an abstraction of the form $\lambda x.M$ maps $x$ to $M$ which may be dependent on $x$. Note that it doesn't evaluate anything; it just creates a function which can be evaluated using an application. This is because an application of the form $x\ y$ applies the function $x$ on the input $y$. An application can also be written using parentheses. It is a convention to omit parentheses if we evaluate multiple applications, in which case we write it in a way such that it associates from the left. In other words, if $E_1, E_2, \ldots, E_n$ are lambda terms, then we write $E_1\ E_2\ \cdots\ E_n$ for $(\cdots((E_1\ E_2)\ E3)\cdots\ E_n)$.

An example of an abstraction is $\lambda x.x$, the identity function, and an example of an application is $(\lambda x.x\ y)$ which evaluates to $y$. Another example of an abstraction is $\lambda x.(\lambda y.(x\ y))$ which is a valid expression by the above definition.

## 2.3 FREE AND BOUND VARIABLES

Consider the abstraction $\lambda x.M$. The lambda symbol binds the variable $x$ appearing in $M$ to the abstraction. The $x$-terms that appear in $M$ are bound and $x$ is then called a *bound variable*. We therefore call $\lambda x$ a binder, to hint that the variable $x$ is getting bound. Any other variables that appear in the abstraction are called *free variables* since they don't influence the way that the abstraction is evaluated when we apply an application to it.

## 2.4 BASIC OPERATIONS

As we have said, lambda terms can also be manipulated using simple operations. The main operations are called **substitution, $\alpha$-conversion, and $\beta$-reduction**.

Substitution is the process of replacing all free variables in an expression with another expression. This is written as $M[x := N]$ which means that we replace all terms $x$ in $M$ by $N$. These substitutions are defined for lambda terms below.

- $x[x := N] = N$
- $y[x := N] = y$, if $x \neq y$
- $(M_1\ M_2)[x := N] = M_1[x := N]M_2[x := N]$
- $(\lambda x.M)[x := N] = \lambda x.M$ (since the $x$-terms in $M$ are bound variables)
- $(\lambda y.M)[x := N] = \lambda y.(M[x := N])$, if $x \neq y$ and $y$ is not a free variable in $N$

$\alpha$-conversion allows for the renaming of bound variables in lambda abstractions, such that the structure of your abstraction doesn't change (so bound variables stay bounded and free variables stay free, keeping your function the same). For example, since variable names are placeholders, we would like $\lambda x.x$ to be equivalent to $\lambda y.y$. We then say that they are $\alpha$-equivalent.

$\beta$-reduction is what we would normally call function application. It therefore describes how we should apply an expression to a function. This is defined in terms of substitution: $(\lambda x.M\ N) := M[x := N]$.

## 2.5 TYPED LAMBDA CALCULUS

When talking about lambda calculus, we have considered a special form of lambda calculus called the untyped lambda calculus. But there exists another form of lambda calculus called the typed lambda calculus which

incorporates *Type Theory* into the syntax. This means that we now attach objects called types to lambda terms to make our language a bit more nuanced, such as integers, boolean values, etc. The Curry-Howard correspondence provides a direct relationship between typed lambda calculus, and more generally computer programs, and mathematical proofs. One of the applications of this correspondence are proof assistants such as Coq and Lean, which are widely used in current pure mathematical research (see Liquid Tensor Experiment). Typed lambda calculus has more applications than untyped lambda calculus due to this nuance (such as the previously mentioned Homotopy Type Theory (HoTT)).

There are many benefits to introducing types in lambda calculus. For example, it makes it easy to name and organize concepts, it gives useful information to the programmer about what kind of data got manipulated by the program, and it ensures that the program runs the way you want it to run (we don't want to add integers to booleans for example). There are many different types of typed lambda calculus.

In increasing order of sophistication:

- Simply typed lambda calculus ($\lambda_{\rightarrow}$)
- System T
- System F ($\lambda 2$)
- System F$\underline{\omega}$ ($\lambda\underline{\omega}$)
- Lambda-P ($\lambda P$)
- System F$\omega$ ($\lambda\omega$)
- Calculus of constructions ($\lambda C$)

where the symbol next to the names correspond to the symbols used in the **λ-cube**; a framework which schematically describes the connection between all typed lambda calculi, introduced by Henk Barendregt. We will not go in depth about all typed lambda calculi, but each typed lambda calculus below another increases nuance and sophistication. Due to time restrictions, we will only implement the simply typed lambda calculus in our Python program. So let us now explore the simply typed lambda calculus.

## 2.6 SIMPLY TYPED LAMBDA CALCULUS

Simply typed lambda calculus consists of a simple type system which we will define first. We denote $V$ for the set of variables.

*Definition* 2.6.1 (The set $\mathcal{T}$ of simple types). We define two standard types.

- Variable type. If $\alpha \in V$, then $\alpha \in \mathcal{T}$.
- Arrow type. If $\sigma, \tau \in \mathcal{T}$, then $(\sigma \to \tau) \in \mathcal{T}$.

These are the basic rules from which we can build the simply typed lambda terms and the set of types of simply typed lambda terms.

*Definition* 2.6.2 (The set $\Lambda^{\to}$ of simply typed ($\lambda_{\to}$) lambda terms). We define:

- Variable. If $x \in V$, then $x \in \Lambda^{\to}$
- Application. If $M, N \in \Lambda^{\to}$, then $(M\ N) \in \Lambda^{\to}$
- Abstraction. If $x \in V$, $\sigma \in \mathcal{T}$ and $M \in \Lambda^{\to}$, then $(\lambda x : \sigma.M) \in \Lambda^{\to}$

Note that this definition of $\lambda_{\to}$-terms is identical to the recursive definition we provided for the untyped lamba terms with only one exception: namely the abstraction definition. In this case, we not only need to specify the bound variable in the abstraction, but also its type. We now provide some examples.

- $(\lambda x : \alpha.x) : \alpha \to \alpha$
- $(\lambda x : \alpha.x)\ y : \alpha$

The first example says that the identity function, where $x$ is of type $\alpha$ is of type $\alpha \to \alpha$. The second says that the identity function of type $\alpha \to \alpha$ applied to $y$ is of type $\alpha$.
Note that this is not defined in Definition 2.6.2. However we would like to define it since functions are only defined for elements from its domain. Therefore, we need another separate set of axioms from which we can derive all valid (more conventionally: *legal*) lambda terms.
In order for us to state them, we use new notation which is conventional in the field of mathematical logic. Generally we write our rules in the form

$$\frac{P_1\ P_2\ \cdots P_n}{\mathtt{conclusion}}$$

where $P_1\ P_2\ \cdots P_n$ are our premises (assumptions). You may recognize this as $(P_1 \wedge P_2\ \wedge \cdots \wedge P_n) \Rightarrow \mathtt{conclusion}$. Next we define a few definitions.

*Definition* 2.6.3. We define:

- Let $M \in \Lambda^{\to}$ and $\sigma \in \mathcal{T}$. A **statement** is of the form $M : \sigma$. Here, $M$ is called the **subject** and $\sigma$ is called the **type**.
- Let $x \in V$ and $\sigma \in \mathcal{T}$. A **declaration** is a statement $x : \sigma$.
- A **context** $\Gamma$ is a set of declarations with different subjects.
- Let $M \in \Lambda^{\to}$, $\Gamma$ a context and $\sigma \in \mathcal{T}$. A **judgement** is of the form $\Gamma \vdash M : \sigma$.
- A term $M \in \Lambda^{\to}$ is called **legal** is there exist a context $\Gamma$ and $\sigma \in \mathcal{T}$ such that $\Gamma \vdash M : \sigma$.

To put it more simply, a context is a set of declarations that we need, in order to derive the truth of a judgement. We will leave out the parentheses for contexts, inline with the usual convention.
Here are some examples of judgements with legal terms.

- $\emptyset \vdash (\lambda x : \alpha.x) : \alpha \to \alpha$.
- $y : \alpha \vdash (\lambda x : \alpha.x)\ y : \alpha$.
- $y : \alpha \to \sigma \vdash (\lambda x : \alpha.(y\ x)) : \alpha \to \sigma$.

We can finally define the additional axioms, called derivation rules, that we need to define the simply typed lambda calculus. From top to bottom, we have the rules for variables, abstractions and applications.

*Definition* 2.6.4 (Derivation rules for simply typed lambda calculus).

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma} \tag{2.6.1}$$

$$\frac{\Gamma, x : \sigma \vdash e : \tau}{\Gamma \vdash (\lambda x : \sigma.e) : \sigma \to \tau} \tag{2.6.2}$$

$$\frac{\Gamma \vdash M : \sigma \to \tau \quad \Gamma \vdash N : \sigma}{\Gamma \vdash (M\ N) : \tau} \tag{2.6.3}$$

With these rules defined, we can check if a given simply typed lambda term is legal/valid.
We now provide some examples taken from the examples above to get a feel for it. We apply the derivation rules from bottom to top and see if the top condition is legal.

$$\frac{x : \sigma \vdash x : \sigma}{\emptyset \vdash (\lambda x : \alpha.x) : \alpha \to \alpha}$$

$$\frac{\dfrac{y{:}\alpha, x{:}\alpha \vdash x{:}\alpha \qquad y{:}\alpha \vdash y{:}\alpha}{y{:}\alpha \vdash (\lambda x{:}\alpha.x){:}\alpha \to \alpha \qquad y{:}\alpha \vdash y{:}\alpha}}{y : \alpha \vdash (\lambda x : \alpha.x)\ y : \alpha}$$

$$\frac{\dfrac{y{:}\alpha \to \sigma, x{:}\alpha \vdash y{:}\alpha \to \sigma \qquad x{:}\alpha}{y{:}\alpha \to \sigma, x{:}\alpha \vdash (y\ x){:}\sigma}}{y : \alpha \to \sigma \vdash (\lambda x : \alpha.(y\ x)) : \alpha \to \sigma}$$

$$\frac{\dfrac{\dfrac{y{:}\alpha \to \beta, z{:}\alpha \vdash y{:}\alpha \to \beta \qquad y{:}\alpha \to \beta, z{:}\alpha \vdash z{:}\alpha}{y{:}\alpha \to \beta, z{:}\alpha \vdash (y\ z){:}\beta}}{y{:}\alpha \to \beta \vdash \lambda z{:}\alpha.(y\ z){:}\alpha \to \beta}}{\emptyset \vdash \lambda y : \alpha \to \beta.(\lambda z : \alpha.(y\ z)) : (\alpha \to \beta) \to (\alpha \to \beta)}$$

They are all legal terms since the declarations at the top are all contained in their contexts. This immediately shows that $\emptyset \vdash (\lambda x : \alpha.(y\ x)) : \alpha \to \sigma$ is an illegal term. Another example of an illegal term is:

$$\frac{\dfrac{\dfrac{x{:}\alpha, y{:}\beta \vdash x{:}\sigma \to \tau \qquad x{:}\alpha, y{:}\beta \vdash y{:}\sigma}{x{:}\alpha, y{:}\beta \vdash (x\ y){:}\tau}}{x{:}\alpha \vdash \lambda y{:}\beta.(x\ y){:}\tau}}{\emptyset \vdash \lambda x : \alpha.(\lambda y : \beta.(x\ y)) : \alpha \to \tau}$$

We derive from the top right premise that $\sigma = \beta$ in order for $y$ to be legal. However, the top left premise gives a contradiction since $x : \alpha, y : \beta \vdash x : \beta \to \tau$ but $x : \beta \to \tau \notin x : \alpha, y : \beta$. Therefore, $\emptyset \vdash \lambda x : \alpha.(\lambda y : \beta.(x\ y)) : \alpha \to \tau$ is not a legal term.

# 3 OUR CODE

In this chapter, we will explain all the methods and code that we have in our Python file.

## 3.1 ASPECT 1: LAMBDA TERMS

In this section we will explain the methods in the class LambdaTerm. Our subsections consist of each method inside this class.

### 3.1.1 fromString

We will provide two methods that we came up with, the first of which was less general and less 'clean' but a little bit faster than the second.

```
@staticmethod
def fromString(string):
    """Construct a lambda term from a string."""

    new_string_list = string.split(' ')

    for i in range(len(new_string_list)):
        if new_string_list[i][0] == 'λ':
            variable = Variable(new_string_list[i][1])
            body = LambdaTerm.fromString(new_string_list[i][3:])
            new_string_list[i] = Abstraction(variable, body)
        else:
            new_string_list[i] = Variable(new_string_list[i])

    for i in range(len(new_string_list)-1):
        new_string_list[0] = Application(
            new_string_list[0], new_string_list[1+i])

    output = new_string_list[0]
    return output
```

Our initial idea was to get an input, which can only be left associative (so evaluated from the left) without parentheses and separated by a space to indicate that it is an application, so an example would be λx.x a. Then, this input would be split by a space and then we would run a for-loop through the list and check if the first letter is a lambda. If it is, then it would be an abstraction and otherwise a variable. In the first case, we would then recursively call the function again on the body of the abstraction. We would then replace each list item by its lambda term and finally reconstruct a lambda term from the list by continually building an application by going through the list.

First of all, this approach is not general. This is because we cannot choose how we want to build our lambda term; the only possible way lambda terms we can construct are left associative lambda terms. So we could not build a lambda term from λx.(λy.(y x)). All in all, this code is pretty bad.

**Time complexity**

The time complexity of this algorithm is $O(n)$, since in the worst case scenario (so when we have a string with only spaces between variables), the first for-loop has a time complexity of $O(n/2) = O(n)$ ($O(n/2)$ since the spaces in-between will be gone) and the second one too $O(n/2-1) = O(n)$. Therefore: $O(n) + O(n) = O(n)$.
Note that the recursive statement in the first for-loop doesn't change our time complexity, since the recursion tree will only split once each time, and the depth of the recursion tree scales linearly with the length of our original input $n$. Moreover, once the method has been evaluated in the deeper nodes, the length of the input will be reduced to one for that particular string, so the time complexity scales linearly with $n$ and the time complexity would still be $O(n)$.

**Second code**

We then tried to take advantage of the __repr__ method in Python. This is essentially a string which can give detailed information about the object: for example the lambda term written in our Python code. So we could define __repr__ in such a way that we would get: repr(Abstraction(Variable(x), Variable(x))) = 'Abstraction(Variable(x), Variable(x))'. This method gave us an idea to first convert the string into a repr, and then convert the repr into a lambda term using the eval() method: a method that converts a string into a Python expression if it is possible. We chose this method, because it would be the most clean, general and simple code, such that anyone reading this code would understand it. Our Python implementation is as follows, which uses a nested method.

```
@staticmethod
def fromString(string):
    """Construct a lambda term from a string."""
    def fromStringtoRepr(string2):
        for i, j in enumerate(string2):
            if j == 'λ':
                return f"Abstraction(Variable('{string2[i+1]}'), {fromStringtoRepr(string2[i+3:])})"
            elif j == '(':
                tracker = 0
                for k, l in enumerate(string2[i+1:]):
                    if l == '(':
                        tracker += 1
                    elif l == ')':
                        tracker -= 1
                    elif l == ' ':
                        if tracker == 0:
                            return f"Application({fromStringtoRepr(string2[i+1:k+1])}, {fromStringtoRepr(string2[k+2:-1])})"

            else:
                return f"Variable('{j}')"
    return eval(fromStringtoRepr(string))
```

Our input can contain parentheses around applications, and this time, we can choose which application will be evaluated first, second, etc., so it doesn't have to be left associative. The basic idea is as follows.

- Enumerate over our given string.

- If we encounter a lambda, then return the abstraction of the letter next to it (the variable) and the body, where we call the function on the body.
- If we encounter a left parenthesis, then that means that it will be an application, but we have to be careful because there could be other applications inside our current application. In other for us to ignore them, we must keep track of how many parentheses we have seen while looping inside the initial parentheses. This is why we implemented a 'tracker'. This tracker adds one if it encounters another left parenthesis and deletes one if it encounters a right parenthesis. Now, if it encounters a space and the tracker is equal to zero, then that means that the right hand side of the space is exactly the N in the expression (M N), while the left hand side is M. Thus we build an application out of it and call the method on the sub-strings M and N.
- Else, it is a variable so return the variable with its symbol.
- Finally, we return the eval method called on the string that got turned into a repr by our above method.

**Time complexity**
The time complexity of this algorithm is a bit worse than the first one due to the nested for-loop. After an investigation, we can see that the number of executions (in the worst case) is equal to: $(n-1)+(n-2)+(n-3)+\cdots+1$, since in the first loop we would have to go through the entire list minus one element, the second time (when the recursive statement is called in the second for-loop) we would have to go through the entire list minus two elements, and so forth. From mathematics we know that $1+2+3+\cdots+n = (n^2+n)/2$, so $(n-1)+(n-2)+(n-3)+\cdots+1 = (n^2+n)/2-n = O(n^2)$.

In conclusion, the second method is more general and much more readable at the expense of a quadratic time complexity. We believe that the positives outweigh the negative. We made this method a static method, because we wanted this method to be independent of the class or instance.

### 3.1.2 alphaConversion

This method is a manifestation of the concept of $\alpha$-conversion from lambda calculus (see section 2.4). Our Python code is as follows.

```
@staticmethod
def alphaConversion(**kwargs):
    """Change the symbols in your lambda term with new ones.\n
    \n
    The first argument has to be a lambda term and the following arguments should be the symbol you want to replace and the symbol replacing that symbol.\n
    \n
    Example: LambdaTerm.alphaConversion(LambdaTerm=F, symbol='a', replacesymbol='e', symbol2='b', replacesymbol2='f')"""

    keys = list(kwargs)
    values = list(kwargs.values())
    for i in range(1, len(keys), 2):
        LambdaTerm_repr = repr(values[0])
        # We want to replace the letters in quotation marks: 'a', 'x' etc.
        LambdaTerm_repr = LambdaTerm_repr.replace(
            f"'{values[i]}'", f"'{values[i+1]}'")
        values[0] = eval(LambdaTerm_repr)
    return eval(LambdaTerm_repr)
```

As can be read from the docstring, this method can take in any number of arguments (**kwargs) and creates a list of keys and their values based on the input. Since the arguments demand an input of the form: LambdaTerm=F,

symbol='a', replacesymbol='e', symbol2='b', replacesymbol2='f', we want to get the second argument and then every second argument after that in order to get the symbols. Therefore we created a for-loop that selects exactly those elements. Then, what we do is that we replace every symbol given with the 'replacesymbol' by making use of the replace method in Python. More specifically, we search for the given symbols *in quotation marks* in the repr of the lambda term, since we don't want to replace the ordinary letters in the repr. This is because every variable is of the form: Variable('a') in repr form.

A restriction of this method is that it cannot switch two already existing symbols due to our for-loop. This is because our for-loop replaces the first element and if we would replace the second element, then we would change all elements. So a possible extension to our code would be, to implement a separate switch method for switching two elements.

**Time complexity**
The time complexity of this algorithm is $O(n)$ because our first for-loop has constant time complexity (the number of symbols we want to replace is constant) while the replace function has a time complexity of $O(n)$, because the replace function needs to go through the entire string in the worst possible case. Here, $n$ is the length of the lambda term inserted, in repr form.

### 3.1.3 reduce

```
def reduce(self):
    """Beta-reduce."""
    # loop/recursion while applying the substitute function. If the previous substitute gives the
    same one as the current, then return (i.e. if it is nonreducible).
    lijst = [self.substitute()]
    while True:
        lijst.append(lijst[-1].substitute())
        if repr(lijst[-2]) == repr(lijst[-1]):
            break
    return lijst[-1]
```

The basic idea behind our reduce method is very simple. We create a list and recursively apply the substitute method on itself (we will later see the substitute method for every class). We append the substituted term in the list and check if the previous and the new term are equal. If they are, then that means that the term could not be reduced any further and so we break out of the while loop. Lastly, we return the last element of the list to the user.

The time complexity is difficult to say, since it is dependent on the lambda term itself (which can have many different manifestations; there is no one way to write a lambda term).

### 3.1.4 __eq__

Sometimes two lambda functions can be written differently, but still represent the same function. This is exactly when two lambda terms are $\alpha$-equivalent. To check whether two different looking functions are equivalent, we used a trick. When we first simplify both functions to their most primal

forms, equivalent functions now are very similar. The only remaining difference is the fact that different symbols can be used for the variables. $\lambda x.x$ is the same as $\lambda y.y$ for example. So with the use of a list we can check if every letter corresponds to an equivalent letter. We wrote two different codes to follow out those two steps. Although the first code ran a bit faster we still chose to go for the second code because it was more insightful and lambda functions are usually not that long.

The first code is the long one.

```
def __eq__(self, other):
    """Alpha-equivalence"""

    self = str(self.reduce())
    other = str(other.reduce())

    letters_self = ''
    letters_other = ''

    database = string.ascii_letters
    used_letters_self = []
    used_letters_other = []

    if len(self) != len(other):
        return False

    for i in range(len(self)):
        if self[i] not in used_letters_self:
            if self[i] in database:
                letters_self += self[i]
                used_letters_self.append(self[i])

    for i in range(len(other)):
        if other[i] not in used_letters_other:
            if other[i] in database:
                letters_other += other[i]
                used_letters_other.append(other[i])

    for i in range(len(used_letters_self)):
        self = self.replace(letters_self[i], database[i])
        other = other.replace(letters_other[i], database[i])

    if self == other:
        return True
    else:
        return False
```

In this method, the main idea was to turn the lambda terms into strings, and loop over the strings. If the string element in self is a letter (which we check by checking if that string element is in string.ascii_letters), then we add that letter to letters_self and used_letters_self. Likewise with the string 'other'. Lastly, we replace every letter from letters_self and letters_other in self and

other, to a common letter from string.ascii_letters. We finally check if self is equal to other.

**Time complexity**
The time complexity of this algorithm is the length $n$ of the string representation of the lambda term (time complexity of the replace function) times the number of different variables in the lambda term (the last for-loop) $m$: so $O(n \cdot m)$. This is due to the last for-loop which scales with this respect and this for-loop has the biggest time complexity out of all the other for-loops. The other for-loops have operations inside of them of $O(1)$ (and the other for-loops themselves scale the same as the replace function in the last for-loop).

Now for the second code.

```
def __eq__(self, other):
    self = str(self.reduce())
    other = str(other.reduce())
    if len(self) != len(other):
        return False
    lijst = []
    for i in range(len(self)):
        lijst.append([self[i], other[i]])
        if i > 0:
            for j in range(i-1):
                if lijst[j][0] == self[i] and lijst[j][1] != other[i]:
                    return False
                elif lijst[j][0] != self[i] and lijst[j][1] == other[i]:
                    return False
    return True
```

The general idea behind the second one is to change the lambda terms to strings and create one list. Then, if their lengths are different, then they are not equal to each other. Else, we loop over one string and append each pair of letters at that index from both strings into the list. We then check with another for loop if a letter was already used before. If it was used before and the pair of that letter at the current index is not equal to the previous one, then we return false. If we lived through the for-loops, then we return true.

**Time complexity**
The time complexity of this algorithm is a bit worse. Since the number of computations in the for loop is equal to $1+1+2+\cdots+(n-2) = (1+2+\cdots+n)+1-(n+(n-1)) = \frac{n^2+n}{2}+1-2n+1 = \frac{n^2}{2}-\frac{3n}{2}+2$, where $n$ is the length of the string, we get the result that the time complexity of this method is $O(n^2)$. This is a bit worse than the previous one, since $m < n$.

## 3.2 ASPECT 2: VARIABLES

```python
class Variable(LambdaTerm):
    """Represents a variable."""

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

    def __repr__(self):
        return f"Variable('{self.symbol}')"

    def __str__(self):
        return self.symbol

    def substitute(self):
        return self
```

As we can see from the picture above, our Variable class is pretty simple. When creating a variable, we only pass in one argument, namely a symbol of the variable as a string. If we print a variable or call the str method, then it will return the symbol and the substitute method called on the Variable will return itself.

## 3.3 ASPECT 3: ABSTRACTIONS

```python
class Abstraction(LambdaTerm):
    """Represents a lambda term of the form λx.M."""

    def __init__(self, variable, body):
        self.variable = variable
        self.body = body

    def __repr__(self):
        return f'Abstraction({repr(self.variable)}, {repr(self.body)})'

    def __str__(self):
        return f'λ{str(self.variable)}.{str(self.body)}'

    def __call__(self, argument):
        return Application(self, argument).reduce()

    def substitute(self, argument='0'):
        # λx.M: Check if an argument was given. If an argument was given, then replace the body of the abstraction by the argument
        and return only that body. Else return itself with its argument reduced (if possible).

        if str(argument) != '0':
            self_repr = repr(self.body).replace(
                repr(self.variable), repr(argument))
            return eval(self_repr)
        else:
            return Abstraction(self.variable, self.body.substitute())
```

An Abstraction λx.M requires a variable x and a body M. The string and repr method are recursive, and the call method (which gets called if we use an abstraction as a function) takes in an argument and returns an application of itself and its argument, analogous to for example (λx.x a) which we can write in our code as Abstraction(Variable('x'), Variable('x'))(Variable('a')).

**substitute**
We designed this method in such a way that when the substitute method is called on an abstraction, without passing any argument in the substitute method (see section 3.4) then we will return the abstraction itself with its body further substituted/reduced.

Else, if we get an argument from the substitute method from the Application class, then we apply the same method as the one from alphaConversion: namely replace the self.variable in the body with the given argument (using the repr method, because that's the most efficient) and return the eval of that new body (we don't need the self.variable anymore).

**Time complexity**
The time complexity of this algorithm is $O(n)$ due to the replace method, where $n$ is the length of the lambda term (in repr form).

## 3.4 ASPECT 4: APPLICATIONS

```python
class Application(LambdaTerm):
    """Represents a lambda term of the form (M N)."""

    def __init__(self, function, argument):
        self.function = function
        self.argument = argument

    def __repr__(self):
        return f'Application({repr(self.function)}, {repr(self.argument)})'

    def __str__(self):
        return f'({str(self.function)} {str(self.argument)})'

    def substitute(self):
        # (M N): Check type compatibility first: argument is only substitutable if M is an
        abstraction. If M is an abstraction, substitute the argument. Else return itself with its
        arguments reduced (if possible).

        if isinstance(self.function, Abstraction):
            return self.function.substitute(self.argument)
        else:
            return Application(self.function.substitute(), self.argument.substitute())
```

An Application (M N) requires two arguments, namely the function M and the argument N.

The substitute method checks if M is an abstraction using the 'isinstance' method. If M is an abstraction, then we call the substitute method in the Abstraction class with N as argument. Else, we return the application itself, while further reducing M and N.

## 3.5 (PART OF) OUR OWN PROGRAMMING LANGUAGE

Lambda calculus provides a formal system in which we can define anything we want, as long as it abides the rules of the things that we define. This gives us ample freedom to build our own programming language. We will build our programming language based on *Church encoding*: a means of representing data and operators in lambda calculus.

### 3.5.1 Arithmetic operations

In this subsection, we will explain the arithmetic operations that we have added.

**NATURAL NUMBERS** Before defining the arithmetic operations, we first have to define the natural numbers. The most easiest way to define numbers is by first defining zero as $0 := \lambda s.(\lambda z.z)$ from which we can define the other natural numbers by

$$\begin{aligned} 1 &:= \lambda s.(\lambda z.(s\ z)) \\ 2 &:= \lambda s.(\lambda z.(s\ (s\ z))) \\ 3 &:= \lambda s.(\lambda z.(s\ (s\ (s\ z)))) \\ &\vdots \end{aligned}$$

We can see from this definition that the $s$ inside the parentheses acts as a 'successor'. So $(s\ z)$ is the successor of $z$, and so forth.
With the natural numbers defined, we can proceed with other methods for doing operations on these numbers.

**SUCCESSOR** The successor function takes a number as an argument and adds one to it. We can define this function as $\lambda w.\lambda y.\lambda x.(y((wy)x))$. Let's work out an example. Suppose that we want to find the successor of 2. This is exactly equal to:

$$\begin{aligned} (\lambda w.\lambda y.\lambda x.(y\ ((w\ y)\ x))\ \lambda s.(\lambda z.(s\ (s\ z)))) \\ \lambda y.\lambda x.(y\ ((\lambda s.(\lambda z.(s\ (s\ z)))\ y)\ x)) \\ \lambda y.\lambda x.(y\ (\lambda z.(y\ (y\ z)))\ x)) \\ \lambda y.\lambda x.(y\ (y\ (y\ x))) = 3 \end{aligned}$$

This derivation process is enough to show that every natural number defined in the previous paragraph, will be converted to its successor. The Python code is shown below. We distinguish between two scenario's: the first is when the number is already written in its lambda representation. The second scenario is when that number is given as an integer. We would then need to convert it first, before applying the function.

```
@staticmethod
def successor(number):
    '''Adds one to the given number and returns its lambda term'''

    if isinstance(number, Abstraction):
        return Abstraction(Variable('w'), Abstraction(Variable('y'), Abstraction(Variable('x'), Application(Variable('y'), Application(Application(Variable('w'), Variable('y')),
        Variable('x'))))))(number).reduce()
    else:
        return Abstraction(Variable('w'), Abstraction(Variable('y'), Abstraction(Variable('x'), Application(Variable('y'), Application(Application(Variable('w'), Variable('y')),
        Variable('x'))))))(LambdaTerm.fromNumber(number)).reduce()
```

Another way to apply the successor function is to define it in Python like an object:

```
successor = LambdaTerm.fromString('λw.λy.λx.(y ((w y) x))')
```

and then use this definition as a function in Python, which then calls the __call__ method from the class Abstraction.

**ADDITION** For this method, we provide two methods, the first of which is three times as fast as the second one. However, the second one has shorter code.

```
def __add__(self, other):
    '''Adds two numbers represented as lambda terms'''
    # NOTE: We don't check if self and other are valid numerical lambda terms as defined above.

    # Apply the successor function 'self' times. We chose this method instead of the naive method, due to lower computation time.
    self_number = LambdaTerm.toNumber(self)
    other_number = LambdaTerm.toNumber(other)

    if self_number < other_number:
        for i in range(self_number):
            other = LambdaTerm.successor(other)
        return other
    else:
        for i in range(other_number):
            self = LambdaTerm.successor(self)
        return self
```

Our first method converts the lambda numbers into integers, and then checks which one is greater than the other. If for example self is less than other, then we apply the successor function self times to other. Otherwise, we do it the other way around.

Our second method looks like the one below.

```
    self_number = LambdaTerm.toNumber(self)
    other_number = LambdaTerm.toNumber(other)

    number = self_number + other_number
    lambda_number = LambdaTerm.fromNumber(number)
    return lambda_number
```

This method is the more naive version. We convert the lambda numbers to integers, add then up, and then convert them back into lambda terms. This method is way slower than the first one.

**MULTIPLICATION** We can define multiplication as

$$\lambda x.\lambda w.\lambda y.(x\ (w\ y)), \tag{3.5.1}$$

and to multiply two numbers, we apply the lambda term to both numbers one after the other.

For example, if we want to multiply 1 with 2, then we would get

$$\begin{aligned}
&((\lambda x.\lambda w.\lambda y.(x\ (w\ y))\ \lambda s.(\lambda z.(s\ z)))\ \lambda s.(\lambda z.(s\ (s\ z))))\\
&(\lambda w.\lambda y.(\lambda s.(\lambda z.(s\ z))\ (w\ y))\ \lambda s.(\lambda z.(s\ (s\ z))))\\
&\lambda y.(\lambda s.(\lambda z.(s\ z))\ (\lambda s.(\lambda z.(s\ (s\ z)))\ y))\\
&\lambda y.(\lambda s.(\lambda z.(s\ z))\ \lambda z.(y\ (y\ z)))\\
&\lambda y.\lambda z.(\lambda z.(y\ (y\ z))\ z)\\
&\lambda y.\lambda z.(y\ (y\ z)) = 2
\end{aligned}$$

Our Python code is shown below.

```
def __mul__(self, other):
    # multiplication of two numbers x and y

    return (Abstraction(Variable('x'), Abstraction(Variable('w'), Abstraction(Variable('y'),
    Application(Variable('x'), Application(Variable('w'), Variable('y'))))))(self).reduce())
    (other).reduce()
```

We first apply equation 3.5.1 to self and reduce it, and then we apply that to other and reduce it.

**FROMNUMBER** This method applies the successor function 'number' times to zero.

```
    @staticmethod
    def fromNumber(number):
        '''Converts a number to its lambda term representation'''

        if number == 0:
            return zero
        else:
            output = LambdaTerm.successor(zero).reduce()
            for i in range(number-1):
                output = LambdaTerm.successor(output).reduce()
            return output
```

**TONUMBER** We thought of two methods for this function. We first noticed that due to our definition of the successor function, we could count the number of 'y'-terms in the string representation of the lambda number to get the integer back. However, we noticed that there was a more general way to do this, independent of the choice of the variable y, namely: count the number of times the word 'Variable' appears in the repr of the lambda term and subtract by three (since we want to for example count the number of times s appears in the application part of the number: $\lambda s.(\lambda z.(s\ z))$ here, s appears once inside the parentheses). Moreover, the second code was 1.13 times faster than the first.

The first image was our initial code.

```
@staticmethod
def toNumber(LambdaTerm):
    '''Converts a lambda term to its base 10 number representation'''
    # NOTE: We don't check if LambdaTerm is a valid number defined above.

    if LambdaTerm == zero:
        return 0
    else:
        count = 0
        for i in str(LambdaTerm):
            # Our choice of the variable name 'y' is arbitrary. This is due to how we defined the successor function.
            if i == 'y':
                count += 1
        return count-1
```

The second image is our final code.

```python
@staticmethod
def toNumber(LambdaTerm):
    '''Converts a lambda term to its base 10 number representation'''
    # NOTE: We don't check if LambdaTerm is a valid number defined above.

    if LambdaTerm == zero:
        return 0
    else:
        count = repr(LambdaTerm).count('Variable')
        return count-3
```

### 3.5.2 Conditionals

We lay the ground work for implementing conditionals into our Python program. However, we could not implement this part completely in Python due to time constraints. However, we provide a suggestion for further development in the discussion.

**TRUE** We can define true as $T := \lambda x(\lambda y.x)$. So it takes two inputs and returns the first one.

**FALSE** We can define false as $F := \lambda x(\lambda y.y)$. So it takes two inputs and returns the second one.

We can now define other logical operators using this definition for true and false.

**AND** The AND operator which we can define as $AND := \lambda x.(\lambda y.((x\ y)\ F)) = \lambda x.(\lambda y.((x\ y)\ \lambda a.(\lambda b.b)))$ functions the same way as multiplication and addition, in the sense that it applies this operator to two arguments, one after the other.

From here on out, we will abbreviate the definitions using previously defined terms (such as F in the definition of AND). But watch out: we have to use different symbols for these terms compared to their original definitions, such that there is no confusion in the beta reduction process when we for example apply these terms to the new functions in which these terms already appear in. This is exactly the problem that we ran into. More on it in the discussion chapter.

**OR** We can define the OR operator as $OR := \lambda x.(\lambda y.((x\ T)\ y))$. It functions the same way as the AND, addition and multiplication.

**NEGATION** We can define the negation ($\neg$) operator as $\neg \coloneqq \lambda x.((x\ F)\ T)$. Example: if we apply this operator to T, we should get back F. So

$$\begin{aligned}(\lambda x.((x\ \lambda a(\lambda b.b))\ \lambda c(\lambda d.c))\ \lambda e(\lambda f.e))\\ (((\lambda e(\lambda f.e)\ \lambda a(\lambda b.b))\ \lambda c(\lambda d.c))\\ (\lambda f.(\lambda a(\lambda b.b))\ \lambda c(\lambda d.c))\\ \lambda a(\lambda b.b)\\ F\end{aligned}$$

**CONDITIONAL TEST** It appears to be beneficial to have a function that gives true when a number is zero and false otherwise. This 'conditional test' can be defined as $Z \coloneqq \lambda x.(((x\ F)\ \neg)\ F)$. If we apply the number $0$ to this function, we get:

$$\begin{aligned}(\lambda x.(((x\ F)\ \neg)\ F)\ 0)\\ (((0\ F)\ \neg)\ F)\\ ((I\ \neg)\ F)\\ (\neg\ F)\\ T\end{aligned}$$

where I is the identity function $\lambda x.x$. Now, $(0\ F) = I$, because

$$\begin{aligned}(\lambda s.(\lambda z.z)\ F)\\ \lambda z.z\\ I\end{aligned}$$

It is easy to see that if we had a number $n$ instead of $0$ in the above equations, then we would get back $(N\ F) = F$, and any function applied to $F = \lambda x.(\lambda y.y)$ gives us back the identity function, so $(F\ \neg) = I$, such that any natural number $n \neq 0$ gives us false. We leave it as an exercise for the reader to confirm the above reasoning.

**CONDITIONAL STATEMENTS** We can write simple conditional statements such as "if P then A, else B" in lambda calculus, where P, A, and B are statements, to make it function a bit more like a programming language. This is easily done by $((P\ A)\ B)$. This is because if P is true, then $P = T = \lambda x(\lambda y.x)$, and since T returns the first input, we get back A. If P is false, then $P = F = \lambda x(\lambda y.y)$, and since F returns the second input, we get back B.

**RECURSION** Recursion can be defined by first defining a new function $Y \coloneqq \lambda g.(\lambda h.(g\ (h\ h))\ \lambda h.(g\ (h\ h)))$. Then any function R applied to Y gives us

$$(Y\ R) = (R\ (Y\ R)) \tag{3.5.2}$$

(confirm this!). We have hereby shown how to use recursion. However, there lies a hidden problem here. Namely: the recursion will never terminate! If we try to reduce a recursive function in our code, then we reach a recursion depth due to equation 3.5.2. This is pretty unhandy if we want to make

recursion useful. There is (fortunately) a fix for this. It can be solved by extending the untyped lambda calculus to simply typed lambda calculus; so by introducing types. More specifically through something called *normalization* in simply typed lambda calculus.

We can apply recursion to many concepts and build new tools with it to expand our programming language, such as recursive sums and factorials among others.

**RECURSIVE SUM** Consider the functions:

$$\phi = \lambda p.(\lambda z.((z\ (\mathrm{successor}\ (p\ T)))\ (p\ T)))$$
$$P = (\lambda n.((n\ \phi)\ \lambda e.((e\ 0)\ 0))\ F)$$

where $0 = \lambda s.(\lambda z.z)$.
We can then define the recursive sum as $\lambda r.\lambda n.(((Z\ n)\ 0)\ ((n\ \mathrm{successor})\ (r\ (P\ n))))$.

# 4 CONCLUSION

Lambda calculus is an extensive formal language and a pioneer within the functional programming languages with many uses within computer science and mathematics. It provides us a way to formalize computation using functions, with which we are all familiar with. Although it merely consists of three different operators, it can do a bundle of various tasks (and is even Turing-complete in simply typed lambda calculus).
Its main disadvantages compared to object oriented programming languages like Python is that it gets messy really quickly (so it can become difficult to reread and understand your code) and it uses anonymous functions which makes it difficult to re-use your code in other places (which adds to the messiness).

To gain a better understanding we tried to recreate lambda calculus in Python. This was a nice way to combine our existing knowledge about Python and learn a new way of programming with it.
With a bit of effort we managed to define all three different lambda terms: the variables, abstractions and applications. With a firm base to build upon, we slowly expanded our code to implement

- $\beta$-reduction to reduce an expression to its irreducible form.
- $\alpha$-equivalence to check whether two lambda terms are equal.
- fromString method to convert a string into a lambda term.
- alphaConversion to convert a variable into another variable (which doesn't change the purpose of the lambda term; only the symbols).

Later we would go on to implement other functions and applications to make a beginning at creating our own programming language.

- We added the natural numbers (including zero) and arithmetic operations on them such as addition, multiplication, the successor method, fromNumber method (create a lambda term from a number), and toNumber method (convert a lambda term to a number).
- And we implemented basic conditionals such as True, False, AND operator, OR operator, negation, and the conditional test.
- Lastly we made a beginning with recursion and the recursive sum. We could not however finish it (see the Discussion).

# 5 DISCUSSION

In this section, we will briefly cover some suggestions for improvements and extensions.

- Improve alphaConversion method.
- Implement simply typed lambda calculus.
- Add new methods and functionality to our own programming language.

## 5.1 IMPROVE ALPHACONVERSION

We made our own alphaConversion method which we introduced in chapter 3. However, this method can look really messy when written in Python as seen below.

```
conditional_test = Abstraction(Variable('i'), Application(Application(Application(Variable('i'), LambdaTerm.alphaConversion(LambdaTerm=F, symbol='a',
replacesymbol='j', symbol2='b', replacesymbol2='k')),
                                LambdaTerm.alphaConversion(LambdaTerm=negation, symbol='x', replacesymbol='l')), LambdaTerm.alphaConversion(LambdaTerm=F,
                                symbol='a', replacesymbol='m', symbol2='b', replacesymbol2='o')))
```

Moreover, there may be an easy way to automatically assign new (temporary) variable symbols when calling methods or defining new functions using other functions. Example: consider the recursive sum introduced in chapter 3. We can write this easily in Python as:

```
recursive_sum = LambdaTerm.fromString(
    f'λr.λn.((({conditional_test} n) {zero}) ((n {successor}) (r ({P} n))))')
```

However, we quickly encounter an obstacle, since the functions inside the definition of the recursive sum are dependent on the exact symbols already used, and every other method inside our Python code is dependent on the exact symbols used. So, β-reduction can go wrong in our code if we use the specific symbol x for the methods zero and conditional_test.
This also addresses a bigger problem when extending our programming language: we eventually run out of alphabet letters that we can use, or we have to keep track of which symbols we already used and which ones we can still use, which is very cumbersome.

Since there is an $\alpha$-equivalence of terms (so chosen variable symbols are arbitrary), we believe that it is wise to make an improved alphaConversion method that can automatically assign new variable symbols when defining new functions or incorporate it in already existing methods (so temporarily assign new variable symbols during a method procedure and at the end assign new variable symbols), such that there would be no errors in for

example bèta reduction and other methods that will be implemented in the future.

We believe that this can be fixed in a reasonable amount of time.

## 5.2 IMPLEMENT $\lambda_{\rightarrow}$ LAMBDA CALCULUS

An extension of our code could be the implementation of the formal system of simply typed lambda calculus, which we have explained in section 2.6. We think that this can be done if we add a new variable *type* in the class definition of a lambda term, and incorporating the rules from section 2.6 in existing (and future) methods such as $\beta$-reduction.

Implementing simply typed lambda calculus resolves the issue of never-ending computations such as recursion, so it is recommended to implement simply typed lambda calculus first, before adding new things to our programming language.

This may require more time, but we think that the implementation is fairly straightforward if you follow the theory from section 2.6.

## 5.3 ADD NEW FUNCTIONALITY

We suggest a few new methods, which may improve the programming language. First, try to add:

- Factorial function using recursion.
- Extend arithmetic by defining $\mathbf{Z}$, subtraction, $>$, $<$, $\geqslant$, $\leqslant$, etc.
- While loop.
- For loop.

Then, extend it by first adding new methods and functionality to match the versatility of Python, and then add more 'niche' methods which may be used for the specific use-case of the programming language.

For example, one may go into the direction of creating a functional programming language like Haskell, a programming language like that of a proof-assistant (like Coq, Lean, etc.), a mix of both worlds, or something entirely different! We leave it up to the creativity of the reader.

One can make our programming language in Python a bit more like a new programming language by using a Python package called PLY (see this documentation). Before one does this, one should first learn about the basics of how programming languages work. There is a popular book about this if you're interested which is called *Compilers: Principles, Techniques, and Tools* by Alfred Aho, which is better known as "the Dragon Book".