A World of Functions - Programming in λ Calculus

Will Boyd

Year 4 Project
School of Mathematics
University of Edinburgh
March 2023

Contents

"One of the most fruitful techniques of language analysis is explication through elimination. The basic idea is that one explains a linguistic feature by showing how one could do without it" - James Hiram Morris, Jr, [1].

Introduction

λ Calculus is a technique of function notation originally developed by Alonzo Church in the 1920s, to facilitate exploration of the fundamentals of mathematics and logic through the view of "functions" rather than sets; verbs rather than nouns.

Chapter 1

Lambda Calculus

1.1 The Syntax

At its heart, λ-calculus is simply a notation for defining functions [2]. The following λ expression

$$\lambda x.\ M$$

defines a function which takes an input x, and produces an output M. Perhaps the main feature of λ notation is its *anonymity*. Although one can name a λ function for convenience, for example

$$\text{LAMBDA} := \lambda x.\ M,$$

it is by no means required by the syntax. The polynomial $p(x) = x^2 + 2x$ can be expressed in λ notation as

$$p(x) \quad \to \quad \lambda x.\ x^2 + 2x,$$

and thus

$$\implies p(2) \quad \to \quad \left(\lambda x.\ x^2 + 2x\right)(2) = 8.$$

1.1.1 Application

Making use of parentheses, we can apply our λ function to some N,

$$(\lambda x.\ M)(N),$$

in which case, we would substitute N for all instances of x in M. Introducing square brackets now, we can formalise this notion [2]:

$$(\lambda x.\ M)(N) = M[x := N] \quad .$$

λ notation is advantageous when defining functions which output other functions [4]. For instance, consider the λ function

$$\text{TWICE} := \lambda f.\ \lambda x.\ f(f(x)).$$

TWICE takes a function, f, as input and returns $f \circ f$. To illustrate this, we apply TWICE to p,

$$\text{TWICE}\,(p) = (\lambda f.\ \lambda x.\ f(f(x)))\,(p) = p\,(p(x)) = (x^2 + 2x)^2 + 2(x^2 + 2x)$$

$$\implies \text{TWICE}\,(p)\,(2) = \big((2)^2 + 2(2)\big)^2 + 2\,\big((2)^2 + 2(2)\big) = 80.$$

1.1.2 Bound and Free variables

Consider once again our general λ function

$$\lambda x.\ M.$$

The $\lambda x.$ part of the syntax does two things. Firstly, and pretty crucially, it lets us know there's a function coming - every function is introduced by a λ. It's second purpose is to specify the *bound variable*, x. The bound variable represents the role of the potential input in the action of the function. *Free variables* represent any variables in the M not representing the role of potential input. To illustrate, in the expression

$$\lambda x.\ xy,$$

x is a bound variable and y is a free variable. Every variable in a λ -expression is either bound or free [3].

1.2 Reduction and Normal Form

1.2.1 β-reduction

We can simplify (or *reduce*) λ expressions by 3 main rules: β-reduction, α-conversion and η-reduction. We have already mentioned β-reduction - it is the action

$$(\lambda x.\ M)(N) \quad \to_\beta \quad M[x := N],$$

where we replace the equality sign from above with $\to_\beta$ simply to emphasise that this equality comes from a pre-defined mechanical axiom; it is a symptom of the syntax.

1.2.2 α-conversion

The second rule, α-conversion says that we can always rename bound variables in a λ -expression. For example

$$\lambda x.\ x \quad \to_\alpha \quad \lambda y.\ y \quad \to_\alpha \quad \lambda z.\ z,$$

where again we are resisting the urge to use an equality sign simply to emphasise that this equality is due to how the syntax is defined. Often it is necessary to rename bound variables in this way. Consider the example

$$(\lambda x.\ (\lambda y.\ xy))(y).$$

We might be forgiven for attempting to reduce this expression thus;

$$(\lambda x.\ (\lambda y.\ xy))(y) \quad \rightarrow_\beta \quad \lambda y.\ yy;$$

however, we would be mistaken to do so. Here, the first instance of y (in the nested λ function) is a bound variable; whereas the second instance of y is a free variable. When we sub in, we conclude erroneously that we have been left with the function $\lambda y.yy \rightarrow_\alpha \lambda t.\ tt$; however, the first occurrence of y in this expression is the same free variable, whereas the second one is still bound. In fact, the correct reduction would be to first use α-conversion, then β-reduction.

$$(\lambda x.\ (\lambda y.\ xy))(y) \quad \rightarrow_\alpha \quad (\lambda x.\ (\lambda t.\ xt))(y) \quad \rightarrow_\beta \quad \lambda t.\ yt.$$

1.2.3 η-reduction

The axiom behind η-reduction has a philosophical element. It conveys the idea of *extensional equivalence* of functions [4]. This is the notion that if two functions produce the same output when given the same input, for all possible inputs, they can regarded as the same function. In other words, a function is defined by what it does, not how exactly it does it. η-reduction says that, for all x, for any F within which x does not freely occur,

$$\lambda x.\ Fx \quad \rightarrow_\eta \quad F$$

where F is a function which takes a single variable. The η-reduction axiom simply recognises the intuitive fact that, if a function **A** takes its input and applies it to another function **B**, then **A** and **B** will always produce the same output for a given input, and thus can be interchanged.

1.2.4 Normal Form

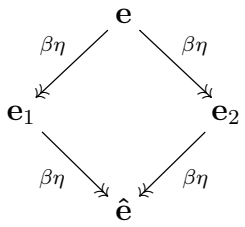
By the above axioms, λ expressions can be *reduced*. We define a reduction of a λ term by either the β or η axioms (involving or not involving an α-conversion) as a $\beta\eta$-reduction. When an expression has no more opportunities for *$\beta\eta$-reduction*, we say it is in *$\beta\eta$-normal form*, or just *normal form* [5].

1.2.5 Confluence

Using the axioms above, it is possible to reduce a λ expression by many strategies. Often one could make valid reduction steps in a multitude of different orders, leading to a branching effect between possible reduction routes. This leads us to a *very* important property of the $\beta\eta$-reduction of λ functions, called *confluence*. The notion of confluence is formally defined by the Church-Rosser Theorem [5]. First we define $\twoheadrightarrow_{\beta\eta}$ to be a (perhaps empty) sequence of $\beta\eta$-reductions and reverse $\beta\eta$-reductions.

Theorem 1 (Church-Rosser)**.** *For any λ expression $\boldsymbol{e}$, if $\boldsymbol{e} \twoheadrightarrow_{\beta\eta} \boldsymbol{e}_1$, and $\boldsymbol{e} \twoheadrightarrow_{\beta\eta} \boldsymbol{e}_2$, then there must exist some $\hat{\boldsymbol{e}}$ such that $\boldsymbol{e}_1 \twoheadrightarrow_{\beta\eta} \hat{\boldsymbol{e}}$ and $\boldsymbol{e}_2 \twoheadrightarrow_{\beta\eta} \hat{\boldsymbol{e}}$. In other*

words, $\beta\eta$-reduction is confluent.



Confluence is illustrated in the diagram above. Proof of Church-Rosser can be found in Appendix 2A of [5], and in [7].

1.2.6 Uniqueness and Equivalence

The following is an incredibly powerful result which follows from the Church-Rosser theorem.

Theorem 2 ($\beta\eta$-Uniqueness)**.** *If a λ expression can be reduced to $\beta\eta$-normal form, then this normal form is unique.*

See the proof in [5] (Collorary 7.13.1). We can conclude then, that if one manages to reduce a given λ expression to normal form, this normal form will be the same regardless of reduction strategy [8]. Further, all lambda terms must have at most one normal form.

A definition which follows naturally from the notion of confluence is that of $\beta\eta$-equivalence.

Definition 1 ($\beta\eta$-equivalence)**.** *For any two λ functions $\boldsymbol{P}$, $\boldsymbol{Q}$, $\boldsymbol{P}$ is $\beta\eta$-equivalent to $\boldsymbol{Q}$ iff $\boldsymbol{P} \twoheadrightarrow_{\beta\eta} \boldsymbol{Q}$.*

Thus, a set of λ functions which all reduce to the same normal form, are all $\beta\eta$-equivalent to each other, as well as $\beta\eta$-equivalent to the normal form itself. If a λ expression is $\beta\eta$-equivalent to a normal form, we say that the expression *has* a normal form.

1.2.7 Leftmost reduction and Divergence

If a λ function has a normal form, there exists a reduction strategy which guarantees arrival at this normal form in finite $\beta\eta$-reduction steps. Known as the *leftmost-reduction strategy* [5], it simply involves always prioritising the leftmost opportunity for $\beta\eta$-reduction. This result was first proven by Curry in 1958, we state it as the Leftmost-reduction Theorem.

Theorem 3 (Leftmost-reduction Theorem)**.** *If a λ expression $\boldsymbol{X}$ has a normal form $\boldsymbol{X^*}$, then a leftmost reduction strategy has a finite number of steps, and terminates at $\boldsymbol{X^*}$.*

So if a λ expression *can* be reduced to normal form, then a leftmost reduction strategy will reach it eventually. A proof of this theorem is found in Section 13.2 of [6]. The theorem seems to suggest that there exist λ expressions which *have* a normal form, but require a leftmost reduction strategy to reach it.

$$(\lambda x.\ 1)(\lambda x.\ xx)(\lambda x.\ xx),$$

is an example of such an expression [3]. It has normal form 1, however a *rightmost* reduction strategy (where the rightmost opportunity for $\beta\eta$-reduction is prioritised) would never reach it.

Functions which do not have a normal form are called *divergent*. These expressions will reduce forever, regardless of strategy.

Collorary 1 (Divergence)**.** *If a λ expression does not reach a normal form in finite leftmost reductions, then the expression must be divergent.*

An example of a divergent function is

$$(\lambda x.\ xx)\,(\lambda x.\ xx)$$

$$\rightarrow_\beta \quad xx[x := (\lambda x.\ xx)] = (\lambda x.\ xx)\,(\lambda x.\ xx)$$

which, as motivated above, will $\beta\eta$-reduce to itself an infinite number of times.

1.3 Currying and a Function Universe

Thus far we have omitted a very important fact from our explanations. In pure λ Calculus everything is a function, all variables and terms; and every juxtaposition of consecutive terms represents function composition. It is a syntax designed for expression purely by functions, and in this report we will use it to reassemble two fundamental mathematical structures.

A system composed entirely of functions naturally requires that each function takes functional input. It also requires that we can evaluate all functional compositions. This means all functions must take just one function as input. We achieve this feat using a technique called Currying.

Currying is the namesake of pioneering combinatory logician Haskell Curry - who spent his whole career attributing the concept to Moses Schönfinkel. Funnily enough it was first developed by Gottlob Frege; who published the idea in 1893 [2]. It says that we can transform functions which take multiple arguments into a series of functions which take one input function applied in series. Some examples may help to illustrate this idea. To aid our explanation, we will momentarily break our "everything in λ is a function" rule; and use some good old-fashioned numbers. The function ADD-X-TO-Y takes two numbers as input and outputs their sum.

$$\text{(ADD-X-TO-Y)}(1, 2) = \text{(ADD-[X := 1]-TO-[Y := 2])} = 3.$$

We can curry the above expression like so:

$$\text{(ADD-X-TO-Y)}(1,2) \quad \rightarrow \quad \text{(ADD-X-TO-Y)}(1)(2) = \text{(ADD-[X := 1]-TO-Y)}(2)$$

$$= \text{(ADD-[X := 1]-TO-[Y := 2])} = 3.$$

Here's a more general example written in λ notation. The following λ function, expressed both curried and uncurried, takes four arguments and multiplies them together.

$$\lambda w, x, y, z.\ w \cdot x \cdot y \cdot z \quad \rightarrow \quad \lambda w.\ \lambda x.\ \lambda y.\ \lambda z.\ w \cdot x \cdot y \cdot z.$$

Applying it to the integers $1, \ldots, 4$ we get

$$(\lambda w.\ \lambda x.\ \lambda y.\ \lambda z.\ w \cdot x \cdot y \cdot z)(1)(2)(3)(4)$$

$$= (\lambda x.\ \lambda y.\ \lambda z.\ 1 \cdot x \cdot y \cdot z)(2)(3)(4)$$

$$= (\lambda y.\ \lambda z.\ 1 \cdot 2 \cdot y \cdot z)(3)(4)$$

$$= (\lambda z.\ 1 \cdot 2 \cdot 3 \cdot z)(4) = 1 \cdot 2 \cdot 3 \cdot 4 = 24.$$

You might say that the λ functions above became more and more *specialized* as their predecessors absorbed arguments. Defining functions in a curried fashion - and thus giving ourselves the ability to inter compose and evaluate any and all of them - allows us to build more complex, specialized functions out of more basic, fundamental ones. Truly functions become building blocks; and with this newfound masonry we can build pretty much anything.

Chapter 2

Booleans and The Integers

In this chapter we will programme the boolean and integer systems we know and love in λ Calculus. All the functions I define in this section, unless explicitly attributed to someone else, were devised by Alonzo Church.

2.1 Boolean Logic

Presented in Code Cell 1 is our boolean logic in λ functions.

Code Cell 1

```
## definitions
TRUE = lambda x: lambda y: x
FALSE = lambda x: lambda y: y

NOT = lambda f: lambda x: lambda y: f(y)(x)
AND = lambda f: lambda g: lambda x: lambda y: f(g(x)(y))(y)
OR = lambda f: lambda g: lambda x: lambda y: f(x)(g(x)(y))
```

We define `TRUE` as the function which takes a function x as input, outputting a function which given any input y, outputs x. `FALSE` acts somewhat oppositely, taking any function x and outputting the *identity function*, which, given any y as input, gives that y as output. Put more simply, `TRUE` takes two curried inputs and returns the first; `FALSE` takes two curried inputs and returns the second.

The function `NOT` takes a function f (designed to be either `TRUE` or `FALSE`) and two more curried inputs: x then y. It then applies these x and y in reverse order to f, such that inputted `TRUE` will act like `FALSE` and vice versa. The function `AND` has been constructed to take two boolean functions, and produce an output function which behaves like a third boolean - which boolean that is depends on the inputted functions, according to the laws of the AND gate of boolean logic. Similarly `OR` is designed and constructed to mimic exactly a boolean OR gate.

We could continue in this way, constructing more specialized boolean operators, however, for our purposes what we have is sufficient. Code Cell 2 presents the function `bool_SHOW`, which lets us know whether its inputted function behaves

like TRUE, or FALSE. If it happens to behave like neither then it lets us know that too.

Code Cell 2

```
def bool_SHOW(f):
    try:
        if f("honey")("badger") == "badger":
            return "FALSE"
        elif f("honey")("badger") == "honey":
            return "TRUE"
    except:
        return "not boolean"
```

In Code Cell 3 we use bool_SHOW to test our newly defined AND, OR and NOT.

Code Cell 3

```
print("OR(TRUE)(FALSE) = ", bool_SHOW(OR(TRUE)(FALSE))
print("OR(FALSE)(FALSE) = ", bool_SHOW(AND(TRUE)(TRUE)))
print("AND(TRUE)(TRUE) = ", bool_SHOW(AND(TRUE)(FALSE)))
print("AND(FALSE)(TRUE) = ", bool_SHOW(AND(FALSE)(TRUE)))
print("NOT(FALSE) = ", bool_SHOW(NOT(FALSE)))

Output:
# OR(TRUE)(FALSE) = TRUE
# OR(FALSE)(FALSE) = FALSE
# AND(TRUE)(TRUE) = FALSE
# AND(FALSE)(TRUE) = FALSE
# NOT(FALSE) = TRUE
```

2.1.1 A note on crystalline structures

It's important at this stage that the reader does not attempt to intuit these functional representations of Boolean objects. They are not intuitively defined. Neither are they counter intuitive - TRUE and FALSE are opposite choices between two things, a very Boolean-like notion - however taken out of their current context these two functions will give no indication of being boolean. The same is true for the functions NOT, AND and OR. The only time that these abstract λ functions behave like (and therefore arguably *are*) the boolean system, is in the presence of one another. This is the notion of a *crystalline structure* in mathematics, a topic we will revisit in depth. A direct comparison can be made to the algebraic representation of boolean logic. In boolean algebra: $\{1, 0\}$ represent True and False; the operators $\{+, \cdot\}$ represent the Or and And; and the conjugate operation $\overline{}$ represents Not. Apart from the $\overline{}$ operator, the physical definition of which is to invert a boolean object (i.e. turn a 1 to a 0 and vice versa), these objects and operators do not have any intrinsic relation to their boolean roles. Only when used together do they become the boolean logic system. Both this system and our new functional one are crystalline constructions of boolean logic in different mathematical media.

2.2 The Integers

2.2.1 The Church Numerals and Iteration

Alonzo Church came up with a new way of thinking about the positive integers. He proposed that they could be represented as function, which take any function f and compose it with itself. So `ONE` would compose f with itself once, `TWO` would compose it with itself twice, `THREE` thrice, and so on. These are known as the Church Numerals [*find source*]. We define a few in Code Cell 4.

Code Cell 4

```
ZERO = lambda f: lambda x: x # == FALSE
ONE = lambda f: lambda x: f(x)
TWO = lambda f: lambda x: f(f(x))
THREE = lambda f: lambda x: f(f(f(x)))
FOUR = lambda f: lambda x: f(f(f(f(x))))
FIVE = lambda f: lambda x: f(f(f(f(f(x)))))
SIX = lambda f: lambda x: f(f(f(f(f(f(x))))))
```

By defining numbers in this way, we have developed functional *iteration*. When applying a function to one of our church numerals we are effectively iterating its action a finite number of times - precisely what a `for` loop does. We can make use of this newfound functionality as we set out to construct integer arithmetic.

We would like to be able to add our new integers together. To do that we first must define a function which takes a church numeral and outputs next one. Then adding is just finding the *next number* as many times as we are wishing to add. We define `NEXT` in Code Cell 5.

Code Cell 5

```
NEXT = lambda A: lambda f: lambda x: f(A(f)(x))
```

This function finds the next number after A, one of our new positive integers. To make sense of its construction, remember what A is and what it does. It's a function which takes a function as input and applies it to itself A times. So we are applying f to itself A times, then once more. So `NEXT`(A) is the church numeral $(A + 1)$.

We can now add B to C. We simply apply `B(NEXT)` to `C`. The function `B(NEXT)`, B iterations of `NEXT`, returns the church numeral 'B after' that inputted; it is an example of a more specialised function made up of the composition of more basic ones, the construction of which is made possible by currying. It is in this fashion that we define `ADD` in Code Cell 6, in terms of `NEXT` as well as in its pure λ form.

Code Cell 6

```
ADD = lambda B: lambda A: B(NEXT)(A)
ADD = lambda B: lambda C: B(lambda A: lambda f: lambda x:
    f(A(f)(x)))(C) # Written anonymously
```

```
ADD = lambda a: lambda b: a(lambda a: lambda b: lambda c:
    b(a(b)(c)))(b) # ...and completely anonymously
```

Now that we can add, we can multiply. Multiplying B by A is just adding B to zero A times - thus we apply `A(ADD(B))` to `ZERO`. We define `MUL` in Code Cell 7 thus.

Code Cell 7

```
MUL = lambda B: lambda A: A(ADD(B))(ZERO)
MUL = lambda a: lambda b: a((lambda a: lambda b: a(lambda a: lambda
    b: lambda c: b(a(b)(c)))(b))(b))(lambda a: lambda b: b)

POW = lambda A: lambda B: A(B)
```

Also defined in Code Cell 7 is `POW`. Its simple construction, A composed with B, is quite easy to imagine. A compositions of B compositions of f is B^A compositions of f.

2.2.2 Memory and Subtraction

We would like to be able to subtract, and thus we require a function `PRIOR`, which returns the numeral *before*. Counting backwards is remembering what came before, and thus to construct `PRIOR` we first need a mechanism of *memory*.

For this task need a function which simultaneously holds two pieces of information. In Code Cell 8 we define the functions `PAIR`, `FIRST`, `SECOND` and `NEXT_PAIR`.

Code Cell 8

```
PAIR  = lambda x: lambda y: lambda f: f(x)(y)

FIRST = lambda P: P(TRUE)
SECOND = lambda P: P(FALSE)

NEXT_PAIR = lambda x: PAIR(SECOND(x))(NEXT(SECOND(x)))
```

`PAIR` takes two functions x and y and stores them together: it makes a pair. `FIRST` takes the function `PAIR(x)(y)` and applies it to `TRUE`, returning the first function of the pair; `SECOND` applies it to `FALSE`, returning the second function. These functions allow us to select an element from a pair of elements. These three functions, along with `NEXT`, allow the construction of `NEXT_PAIR`, which takes a pair of numerals as input, selects the second numeral of the pair and creates a new pair containing that numeral and the one proceeding it. If we start at $(0, 0)$ and iterate `NEXT_PAIR` N times, we end up with $(N-1, N)$; returning the first element gives us the numeral $(N-1)$. This is the mechanism of `PRIOR`, which is defined in Code Cell 9 along with the subtraction function `SUB`.
In a pattern that will be familiar by now, `SUB` is simply the iteration of `PRIOR`.

Code Cell 9

```
PRIOR = lambda N: FIRST(N(NEXT_PAIR)(PAIR(ZERO)(ZERO)))
```

```
SUB  = lambda N: lambda M: M(PRIOR)(N)
```

2.2.3 Negative Numbers

A vital ingredient needed to build the full integer system is what we will refer to as an integer's *direction*: positive or negative. We will consider an integer to be a function comprised of two features: a direction and a count, and we will define it as a pair containing either `TRUE` or `FALSE` (corresponding to the positive and negative directions) and a Church numeral (see Code Cell 10 below).

Code Cell 10

```
posSIX = PAIR(TRUE)(SIX)
posFIVE = PAIR(TRUE)(FIVE)
posFOUR = PAIR(TRUE)(FOUR)
posTHREE = PAIR(TRUE)(THREE)
posTWO = PAIR(TRUE)(TWO)
posONE = PAIR(TRUE)(ONE)
zero = PAIR(TRUE)(ZERO)
negONE = PAIR(FALSE)(ONE)
negTWO = PAIR(FALSE)(TWO)
negTHREE = PAIR(FALSE)(THREE)
negFOUR = PAIR(FALSE)(FOUR)
negFIVE = PAIR(FALSE)(FIVE)
negSIX = PAIR(FALSE)(SIX)
```

Given that the integer zero has no (or perhaps arbitrary) direction, the direction with which its defined is purely up to convention. Standard mathematical convention counts zero among the positive integers, and so we have done the same. Zero is a very important function. It is the barrier, the limbo between the positive and the negative - whenever we change sign we must go through zero. It is therefore essential that we are able to identity a `zero` input; something we can do so with `ISZERO`, as defined in Code Cell 11.

Code Cell 11

```
ISZERO = lambda N: N(lambda x: FALSE)(TRUE)
ISZERO = lambda N: N(lambda x: (lambda x: lambda y: y))(lambda x:
    lambda y: x)

bool_SHOW(ISZERO(EIGHT))
bool_SHOW(ISZERO(ZERO))

Output:
# 'FALSE'
# 'TRUE'
```

`ISZERO` takes a Church numeral as input, and to it applies the function which takes any input and returns `FALSE`. To the resulting function is applied `TRUE`. It

is not difficult to imagine that only when ZERO is the inputted numeral is TRUE returned.

2.2.4 Conditional Branching in λ

ISZERO is an example of a function with a *conditional* mechanism. To make this clearer, consider that it may also be defined: ISZERO = lambda N: TRUE if N is ZERO else FALSE. The capability of a function to make decisions on output based on conditions on the input is known as *conditional branching* [1]. When defining functions in Python, this capability is given by the if syntax. All we will require are our Boolean logic constructions.

We would like to be able to change an integer's direction. Depicted in Code Cell 12, NEG switches a function's direction from TRUE to FALSE, and vice versa - except if that function is zero. It uses conditional branching within its mechanism, using the function OR to make a decision on between whether to switch the direction of its input or keep it the same (if it's zero).

Code Cell 12

```
NEG = lambda f: PAIR(OR(NOT(FIRST(f))(ISZERO(SECOND(f)))(SECOND(f))

NEG = lambda f: PAIR(NOT(FIRST(f)))(SECOND(f)) if f is not zero
    else PAIR(ISZERO(f))(SECOND(f))
```

We require conditional branching when doing integer arithmetic due their directional element. In Code Cell 13, we define NEXTint and PRIORint. Boolean logic functions AND and OR are used to decide whether we are crossing zero, and from what direction, and therefore if the output integer should have a different direction. In turn, we use the integers direction to decide whether it is appropriate the increase or decreased the count (using NEXT and PRIOR).

Code Cell 13

```
def NEXTint(N):
    count = SECOND(N)
    direction = FIRST(N)
    new_count = direction(NEXT)(PRIOR)(count)
    new_direction = OR(direction)(ISZERO(new_count))
    return PAIR(new_direction)(new_count)

def PRIORint(N):
    count = SECOND(N)
    direction = FIRST(N)
    new_direction = AND(NOT(ISZERO(count)))(direction)
    new_count = new_direction(PRIOR)(NEXT)(count)
    return PAIR(new_direction)(new_count)

negSEVEN = PRIORint(negSIX)
negEIGHT = PRIORint(negSEVEN)
```

```
negNINE = PRIORint(negEIGHT)
negTEN = PRIORint(negNINE)
negELEVEN = PRIORint(negTEN)
negTWELVE= PRIORint(negELEVEN)
posSEVEN = NEXTint(posSIX)
posEIGHT = NEXTint(posSEVEN)
posNINE = NEXTint(posEIGHT)
posTEN = NEXTint(posNINE)
posELEVEN = NEXTint(posTEN)
posTWELVE = NEXTint(posELEVEN)
```

Using `NEXTint` and `PRIORint`, we can define `ADDint` by iteration. Conditional branching is also utilised once again, as the direction of the integer being added determines whether we are going backwards or forwards. Subtraction is simply in the Addition is the opposite direction. See `ADDint` and `SUBint` in Code Cell 14.

Code Cell 14

```
ADDint = lambda B: lambda A:
    SECOND(B)(FIRST(B)(NEXTint)(PRIORint))(A)

SUBint = lambda B: lambda A: ADDint(NEG(A))(B)

MULint = lambda B: lambda A:
    FIRST(B)(SECOND(B)((ADDint)(A)))(SECOND(B)((SUBint)(A)))(zero)
```

Also in Code Cell 14 we define `MULint`. Multiplication between integers is just is repeated addition, or subtraction, onto, or from, zero. Whether we subtract from zero or add to it is decided the direction of the integer we multiply by.

Code Cell 15

```
showint(ADDint(posSEVEN)(negELEVEN))
showint(SUBint(negFOUR)(posSIX))
showint(MULint(negFOUR)(negFIVE))

Ouput:
# -4
# 10
# 20
```

2.2.5 Fixed Points and Recursion

A very important and useful result in λ Calculus is the Fixed Point Theorem.

Theorem 4 (Fixed Point)**.** *For all functions F there exists a function X such that $F(X) = X$. This X is known as the* fixed point *of F.*

This theorem is directly proven by the existence of functions known as *fixed-point*

combinators, such as Curry's Fixed Point Y-Combinator:

$$Y := \lambda f.\ ((\lambda x.\ f\,(xx))\,(\lambda x.\ f\,(xx)))\,.$$

Subbing in $Y(F)$ for X,

$$X = Y(F) = ((\lambda x.\ F\,(xx))\,(\lambda x.\ F\,(xx)))$$

$$= F\,((\lambda x.\ F\,(xx))\,(\lambda x.\ F\,(xx))) = F(Y(F)) = F(X). \quad \square$$

Curry's Y-combinator is just one of infinitely many fixed-point combinators that exist. Other famous examples include the Turing Combinator Y [5], and the Z Combinator.

It is perhaps clear at this point why this function Y might be useful. It gives us a mechanism of *recursion* - a vital tool for computation. For example consider the recursive function

```
FACT = lambda x: 1 if x==0 else x * FACT(x - 1)
```

which returns the factorial of its input. We can re-express `FACT` in pure λ functions, re-designed for church numeral input and output.

```
FACT = lambda x: ISZERO(x)(ONE)(MUL(x)(FACT(PRIOR(X))))
```

`FACT` cannot appear in its own definition, so we define a second function

```
G = lambda f: lambda x: ISZERO(x)(ONE)(MUL(x)(f(PRIOR(X))))
```

`G` is designed specifically such that `FACT` = `G(FACT)`. Recall that this makes `FACT` the fixed point of `G`. Thus

```
F = Y(G)
```

where

```
Y = lambda f: (lambda x: f(x(x)))(lambda x: f(x(x)))
```

is Curry's fixed point combinator. Recursion achieved by use of a fixed-point combinator we will hence refer to as *fixed-point recursion*. A pitfall of fixed-point recursive functions when using them for computation is that they do not have a normal form [5].

2.2.6 The Formulae behind Integer Arithmetic

I hope its clear at this stage that we could go on. The functions we have defined thus far seem complicated at first glance, but are in fact just combinators, designed and arranged so that they follow the same very basic rules of interaction that govern the integers - the ones we memorised, and may well have intuited, in primary school. Re-crafting the system in a functional medium gives us a greater understanding of *why* these rules are logical. Take the famously confusing but

incredibly fundamental mathematical fact: $-1 \times -1 = +1$. Lets see if our new lambda system agrees...

```
showint(MULint(negONE)(negONE))

Output:
# 1
```

Having created these functions from scratch - we know why too. The function '$-1\times$' takes an integer as input, and subtracts this integer from zero for 1 iteration. So, -1×-1 returns the result of subtracting -1 from 0 once. Subtraction of an integer is simply addition of the integer with the same count, but opposite direction - thus, we are adding 1 to 0 once. So $-1 \times -1 = 1$.

2.3 Turing Completeness

We have learned in this section that, as a computing language, λ Calculus is capable of iteration, memory, conditional branching and recursion. In fact, although the above examples serve as excellent motivation for the programming potential of λ, they barely scratch the surface of its capability. Its possible for instance, to code a computational cost model in λ Calculus, allowing for the development of complexity theory [9]. If truth be told, λ Calculus can express any *computable* function.

What is the definition of a computability? Well, for years mathematicians did not have a formal definition, just a general but imprecise notion widely referred to as *effective calculability*. Then, in the 1930s, three definitions for computable functions arose [4]:

1. *General Recursive* functions: a class of functions developed by Herbrand and Gödel.
2. λ-*definable* functions: those expressible in λ Calculus. This definition was proposed by Church himself.
3. *Turing Computable* functions: those computable by Alan Turing's Turing Machine.

In the very same decade, the mathematicians sorted it out amongst themselves. Church and Turing proved the equivalence of these three definitions - showing that all three sets of functions named above were in truth the same set (see [10], [11]). They developed the Church-Turing Thesis, which proposed equivalence between this set of functions (widely referred to now as the Turing Computable functions) and the notion of effective calculability. Nearly one hundred years later, Turing Computability is the generally accepted formal definition of computability. A programming language with the same computing capability as a Turning Machine is known as Turing Complete: needless to say, λ Calculus is such an example. So I guess Alan Turing won that one.

Chapter 3

SKI Calculus

3.1 Intro to Combinators

Combinators are λ functions containing only bound variables. They introduce no new information or free variables; simply returning some composition of the functions inputted. Examples of combinators are our previously defined `TRUE` and `FALSE` functions and (as you might have guessed) Curry's fixed-point combinator, `Y`.

We define now, in Code Cell 16, three new combinators of great signifance: `S`, `K` and `I`.

Code Cell 16

```
S = lambda x: lambda y: lambda z: x(z)(y(z))
K = lambda x: lambda y: x # == TRUE
I = lambda x: x.
```

Together, they make up the programming language known as SKI Calculus[1].

3.2 SKI Calculus Reduction Properties

3.2.1 Weak Reduction and Weak Normal Form

We define SKI functions as any composition involving only S, K and I. Just like reduction of λ expressions, SKI functions can be reduced: by the combinatory actions that are their literal definitions. This is a form of β-reduction. We define this reduction of SKI combinator expressions as *weak reduction* [5].

Definition 2 (Weak Reduction)**.** *The act of replacing, in an SKI expression, either an occurrence of* I(x) *with* x*, an occurrence of* K(x)(y) *with* x*, or an occurrence of* S(x)(y)(z) *with* x(z)(y(z))*.*

We have expressed S, K and I in λ form, and thus, in this context, weak reduction is equivalent to the action of β-reduction. However, to emphasise that SKI Cal-

[1]Note to the reader: in this article we have adopted the pronunciation "skee calculus".

culus is an entirely different programming language, we define things seperately.

We will represent a single weak reduction step with the symbol $\rightarrow_w$ and a (perhaps empty) sequence of weak reduction steps and reverse weak reduction steps with the symbol "$\twoheadrightarrow_w$".

Definition 3 (Weak Normal Form)**.** *If an SKI expression contains no opportunity for weak reduction, we say it is in* weak normal form*. For an SKI expression* $\boldsymbol{U}$*, if* $\boldsymbol{U}$ *weak reduces to a weak normal form* $\boldsymbol{X}$*, we call* $\boldsymbol{X}$ *the weak normal form of* $\boldsymbol{U}$*.*

Definition 4 (Weak Equivalence of SKI Functions)**.** *Two SKI functions* $\boldsymbol{U}$ *and* $\boldsymbol{V}$ *are said to be* weakly equivalent *iff* $\boldsymbol{U} \twoheadrightarrow_w \boldsymbol{V}$*.*

3.2.2 Confluence and Uniqueness in SKI

Given its similarity to β-reduction, the fact weak reduction is confluent may fail to surprise. We state this result formally as a face of the Church-Rosser Theorem.

Theorem 5 (Church-Rosser)**.** *For any SKI expression* $\boldsymbol{U}$*, if there exist* $\boldsymbol{U_1}$ *and* $\boldsymbol{U_1}$ *such that* $\boldsymbol{U} \twoheadrightarrow_w \boldsymbol{U_1}$ *and* $\boldsymbol{U} \twoheadrightarrow_w \boldsymbol{U_2}$*, then there must must exist* $\hat{\boldsymbol{U}}$ *such that* $\boldsymbol{U_1} \twoheadrightarrow_w \hat{\boldsymbol{U}}$ *and* $\boldsymbol{U_2} \twoheadrightarrow_w \hat{\boldsymbol{U}}$*. In other words,* weak reduction is confluent*.*

A proof is found in Appendix 2 of [5]. Like before, from confluence follows uniqueness of the weak normal form.

Theorem 6 (SKI Uniqueness)**.** *If a SKI function* $\boldsymbol{U}$ *has a normal form* $\boldsymbol{U^*}$*, then* $\boldsymbol{U^*}$ *is unique.*

3.2.3 Leftmost Reduction and Divergent SKI Functions

The Leftmost Reduction Theorem also applies to SKI Calculus. We restate it here in this context.

Theorem 7 (Leftmost Reduction Theorem)**.** *If a SKI function* $\boldsymbol{U}$ *has a normal form* $\boldsymbol{U^*}$*, then a leftmost reduction strategy will terminate at* $\boldsymbol{U^*}$ *after finite steps.*

Where the definition of a leftmost reduction strategy remains unchanged.

We also define divergent SKI functions identically to before.

Definition 5 (SKI Divergence)**.** *If a SKI function does not have a normal form, we call it* divergent*.*

Collorary 2. *If a SKI function does not reach a normal form in finite leftmost reductions, then said function must be divergent.*

For an example of a divergent SKI function, consider

$$((\mathrm{S}(\mathrm{I}))(\mathrm{I}))((\mathrm{S}(\mathrm{I}))(\mathrm{I})).$$

This expression will reduce by strictly *leftmost* reductions thus

$$\begin{aligned}((\mathrm{S}(\mathrm{I}))(\mathrm{I}))((\mathrm{S}(\mathrm{I}))(\mathrm{I})) &\to_w \mathrm{I}((\mathrm{S}(\mathrm{I}))(\mathrm{I}))(\mathrm{I}((\mathrm{S}(\mathrm{I}))(\mathrm{I})))\\ &\to_w ((\mathrm{S}(\mathrm{I}))(\mathrm{I}))(\mathrm{I}((\mathrm{S}(\mathrm{I}))(\mathrm{I})))\\ &\to_w \mathrm{I}(\mathrm{I}((\mathrm{S}(\mathrm{I}))(\mathrm{I})))(\mathrm{I}(\mathrm{I}((\mathrm{S}(\mathrm{I}))(\mathrm{I}))))\\ &\twoheadrightarrow_w ((\mathrm{S}(\mathrm{I}))(\mathrm{I}))(\mathrm{I}(\mathrm{I}((\mathrm{S}(\mathrm{I}))(\mathrm{I}))))\end{aligned}$$

and so on. Clearly by the leftmost reduction strategy this function will never reach a normal form, and therefore must be divergent. This situation also a great example of the shortfalls of strict leftmost reduction. Consider the following reduction strategy of the same expression.

$$\begin{aligned}((\mathrm{S}(\mathrm{I}))(\mathrm{I}))((\mathrm{S}(\mathrm{I}))(\mathrm{I})) &\to_w \mathrm{I}((\mathrm{S}(\mathrm{I}))(\mathrm{I}))(\mathrm{I}((\mathrm{S}(\mathrm{I}))(\mathrm{I})))\\ &\to_w ((\mathrm{S}(\mathrm{I}))(\mathrm{I}))(\mathrm{I}((\mathrm{S}(\mathrm{I}))(\mathrm{I})))\\ &\to_w ((\mathrm{S}(\mathrm{I}))(\mathrm{I}))((\mathrm{S}(\mathrm{I}))(\mathrm{I}))\\ &\to_w \mathrm{I}((\mathrm{S}(\mathrm{I}))(\mathrm{I})))(\mathrm{I}((\mathrm{S}(\mathrm{I}))(\mathrm{I})))\\ &\twoheadrightarrow_w ((\mathrm{S}(\mathrm{I}))(\mathrm{I}))((\mathrm{S}(\mathrm{I}))(\mathrm{I}))\end{aligned}$$

By straying from strict leftmost reductions, the true property of the combinator quickly emerges.

3.3 Universality of SKI Calculus

With those properties and definitions out of the way, we move our attention to what makes SKI calculus truly significant: its a Turing Complete Programming language.

The property of SKI Calculus that we are about to show is that, *any* λ Calculus function can be expressed as a SKI function. First we define a notation called *bracket abstraction*, from [5].

Definition 6 (SKI Bracket Abstraction)**.** *For any terms* $\boldsymbol{M}$*,* $\boldsymbol{N}$ *and* x *in SKI Calculus, we define the term* $[x].\boldsymbol{M}$ *by induction on* $\boldsymbol{M}$*, thus:*

1. *For any SKI term* $\boldsymbol{N}$*,* $([x].\boldsymbol{M})(\boldsymbol{N}) \to_w \boldsymbol{M}[x := \boldsymbol{N}]$*,*
2. $K\boldsymbol{M} = [x].\boldsymbol{M}$*,*
3. $I = [x].x$*,*
4. $\boldsymbol{M} = ([x].\boldsymbol{M}x)$ *(if* x *is not a free variable in* $\boldsymbol{M}$*, this axiom is strongly linked to* η*-reduction),*

5. *S([x].**M**)([x].**N**) = [x].**MN** (if rules 2. and 3. do not apply).*

Now we define SKI abstraction of λ functions, from [4].

Definition 7 (SKI Abstraction of λ Functions)**.** *The SKI abstraction of a λ function Q, a mapping written as T(Q), is defined recursively thus:*

1. *T(x) = x, for any variable x,*
2. *T(MN) = T(M)T(N), for any expressions M and N,*
3. *T(λx. M) = [x].T(M)*

By these axioms, any λ expression can be translated to some functional composition of S's K's and I's: the map T as defined above is surjective. Thus, by Definition 7, amongst every possible SKI expression lie every possible λ expression. This has great consequences. Given that λ Calculus is a Turing Complete programming language, capable of computing anything computable, SKI Calculus must be too. Amongst every SKI function is every Turing Computable function. This fact is known as the Universality of SKI Calculus.

In the next section, we look to motivate this universality, by using Python to look for some of the λ functions we have already defined in Section 2, this time made up only of `S`, `K` and `I`. All Python functions mentioned will included in Appendices.

Chapter 4

SKI Combinatorics in Python - An Alternative Strategy

4.1 Intro and Star Form

The strategy we will employ to find these functions you might say is one of brute force. The chronically limited vocabulary of the SKI language results in a language both very simple and impossibly cryptic. We know very well what the combinators themselves do, however figuring out what a large composition of them does is a difficult and time-consuming task. Therefore, we are going to attempt to generate the set of every possible composition of S's, K's and I's; apply them all to specific inputs and observe the result for all expressions. This way we hope to identify a few SKI functions with familiar properties.

We wish to find every possible SKI function *in Python*. This stipulation is important, because when we generate our Pythonic SKI functions we wish to be clear on how Python will evaluate them. Evaluation by Python in this context will ultimately amount to weak reduction of the expression: by a reduction strategy of Python's choosing.

There are two evaluation rules Python follows, the knowledge of which will allow us to understand the reduction strategy it will emply. Firstly, Python is known as *eager* evaluator: meaning that any function application is immediately computed when Python comes across it. Secondly, Python will evaluates expressions left to right just as you or I would. We can conclude therefore that Python will adopt a strictly leftmost reduction strategy. In order then to avoid confusion, we adopt a new notation for expressing function composition. A notation we will hence refer to as *star form*. Star form emphasises and clarifies how our functions will be evaluated once given to Python. Some examples of star form should be sufficient as its explanation.

$$\text{S(K)(K)} \xrightarrow{\text{star form}} \text{((S * K) * K)}$$

$$\text{((S(K))(K))} \xrightarrow{\text{star form}} \text{((S * K) * K)}$$

$$\text{S(K(K))} \xrightarrow{\text{star form}} \text{(S * (K * K))}$$

$$\text{(S(I)(I))(S(I)(I))} \xrightarrow{\text{star form}} \text{((S * I) * I) * ((S * I) * I))}$$

$$\text{((S(I))(I))((S(I))(I))} \xrightarrow{\text{star form}} \text{((S * I) * I) * ((S * I) * I))}$$

$$\text{(S(I(I)))(S(I(I)))} \xrightarrow{\text{star form}} \text{((S * (I * I)) * ((S * (I * I)))}$$

We will generate our SKI functions as Python strings in star form, before defining a class that tells Python to treat "*" as function application.

4.2 Finding Compositions by Function Trees

We can't find all the compositions of S, K and I infinitely long. A more reasonable goal might be to find all the possible SKI functions n long, for some modest n.

Our first task is to generate *all possible compositions*. Hence, for now, we define an arbitrary selection of n functions $0, 1, ..., i, ..., n$, where each $i \in S, K, I$. You may think of these functions as *placeholders*, for which S, K or I can be substituted later. So how do we find every possible composition of these n functions? It is straightforward for very small n. Ff we take $n = 1$ for example, trivially we get only one composition:

$$0.$$

If we take $n = 2$ we also only get one:

$$0(1).$$

We get two compositions for $n = 3$:

$$0(1)(2) \quad \text{and} \quad 0(1(2)),$$

and for $n = 4$ we have five:

$$0(1)(2)(3), \quad 0(1(2))(3), \quad 0(1)(2(3)), \quad 0((1(2))(3)) \quad \text{and} \quad 0(1(2(3))).$$

Quickly, however, as n increases finding them on paper stops being viable. We need a method to find all possible compositions general n. One answer lies in observing how these compositions can be represented by *function trees*. See Figure 4.1

Thinking about our expressions as triangular tree structures now, like those depicted in Figure 4.1, we will hence refer to our placeholder functions $0, \ldots, n$ as *leaves*. Reforming our problem in this new context then: We would like to generate the set of all possible n-leaf trees. For now, we don't mind about duplicates: as long as our set contains every possible tree.

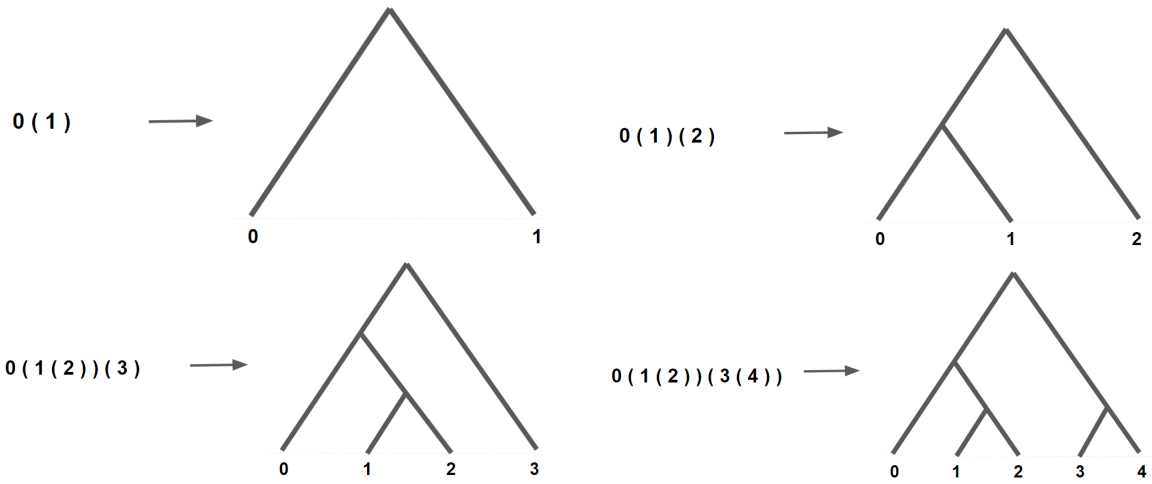


Figure 4.1: *Examples of the representation of functional composition by "triangular" tree structures. Each branch meeting point represents the application of one function to another.*

4.2.1 Drawing Codes

One way to solve this problem is to notice that every tree is drawn the same way. We will always have leaves $0, \ldots, n$ ordered left; the left and right edges of the tree will always represent the first and last functions in the expression. There is an over-arching "triangle", the apex of which represents the final function application Python will compute. In some order, we draw a straight line, or branch, from each of the inner functions, either left or right, until it meets another branch. This idea is illustrated in Figure 4.2. Importantly, this drawing strategy we can easily *encode*. For example, referring again to Figure 4.2, we can draw the tree depicted - which happens to represent the composition (((0 * 1) * (2 * 3)) * 4) - by the code

```
[0, 2, 1]
["L", "R", "L"]
```

It's important to note that this is not the only way to draw the same tree - far from it. There are certainly multiple drawing codes like this one that recreate the same result. The key point here, is that all trees are draw in this way, and thus, if we take the list of all permutations of this encoding for n-leaf trees, and generate the tree described by each item in this list, the resulting list of trees will contain every possible n-leaf tree.

Each such encoding of an n-leaf tree is a permutation of the positive integers from 0 to $n-2$, and an $(n-2)$-item-long list of "R"s and "L"s. Therefore, the resulting list of generated trees will have $(n-2)! * 2^{n-2}$ elements. Even for modest n this will be a huge number, and in reality we expect the number of *distinct* trees in this set to be far lower.

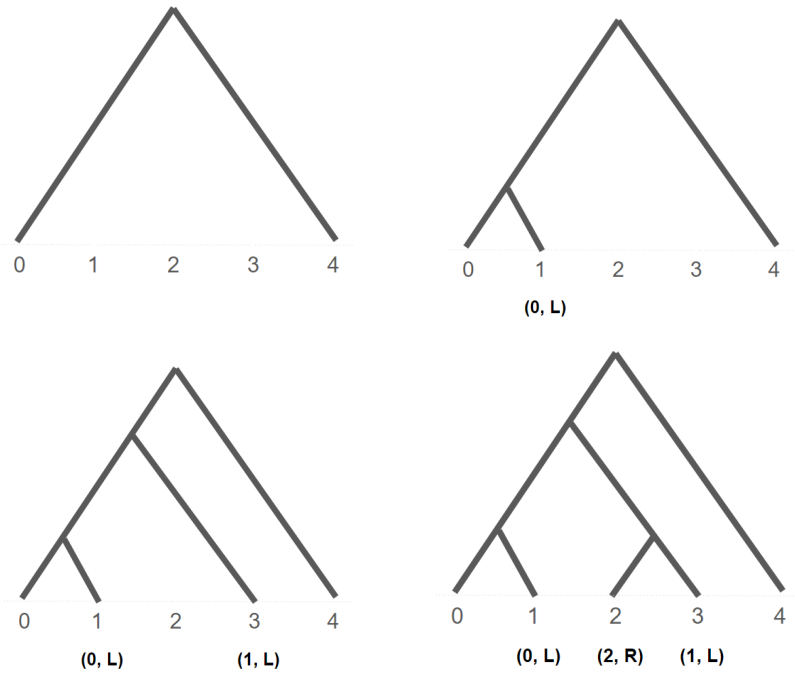


Figure 4.2: *Left to right, top to bottom, encoding the stages of drawing a triangular function tree. "R" or "L" refers to a branch to be drawn right or left; the integer coupled with the direction refers to the order branches are drawn.*

Having generated the set of all drawing codes, we define `tree_builder`, which takes a drawing code as input and identifies the locations of all branch meeting points: encoding them as pairs of consecutive branch numbers in sublists, where each sublist corresponds to a vertical level of the corresponding tree. Where two branches meet at level i, they are treated as a single branch at level $i + 1$. This idea is best illustrated in Figure 4.3.

The *tree form* by which `tree_builder` would represent the tree depicted in Figure 4.3 is

```
[[[1, 2], [4, 5]], [[1, 2]], [[0, 1]], [[0, 1]]]
```

A second function, `tree_to_exp`, takes a tree form as input and returns the corresponding function composition in star form and as a Python string. To give a simple example, consider the tree in Figure 4.4. This tree would be represented in Pythonic tree form as

```
[[[0, 1]], [[0, 1]], [[0, 1]], [[0, 1]]],
```
(4.1)

whereas the corresponding function composition would be expressed in star form as

$$((((0 * 1) * 2) * 3) * 4). \tag{4.2}$$

We use `tree_builder` and `tree_to_exp` to translate our whole list of drawing codes to their corresponding compositions as star form strings. From the result-

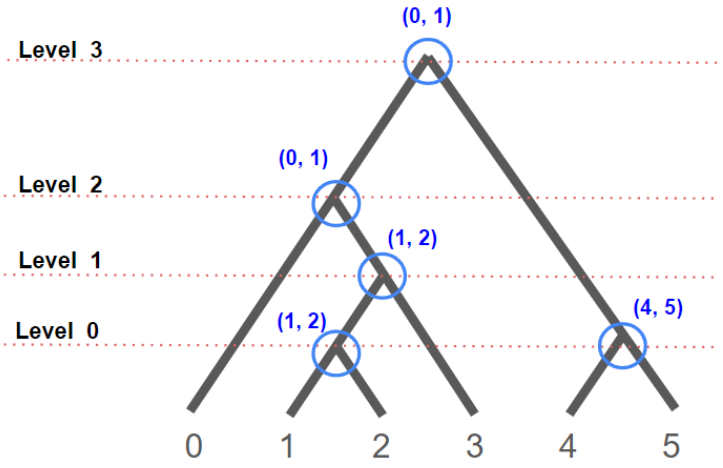


Figure 4.3: *How* `tree_builder` *represents a tree in its output.*

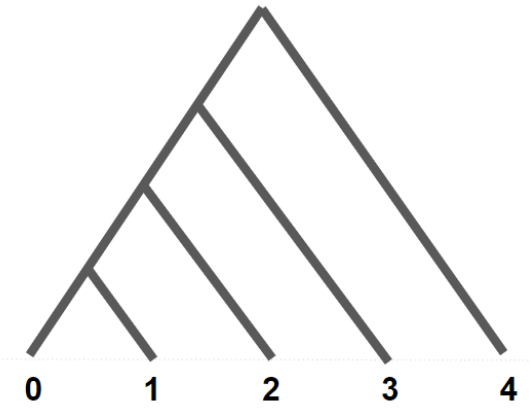


Figure 4.4: *The tree corresponding to (4.1) and (4.2).*

ing list we remove duplicate compositions.

When we count the number of distinct compositions for each $n = 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, \ldots,$ we obtain the sequence $1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, \ldots$. These are known as the Catalan numbers. The number of function compositions of length n is the $(n-1)^{\text{th}}$ Catalan number,

$$C_{n-1} = \frac{1}{n}\binom{n-1}{2n-2}.$$

This is a well-known result surrounding the Catalan numbers [29], and we will discuss this relationship a bit later on in section [**input section**].

Considering for $n = 2, 3, 4, 5, 6, 7, 8, 9, 10, \ldots$ we are processing

$$(n-2)!2^{n-2} = 1, 2, 8, 48, 384, 3840, 46080, 645120, 10321920, \ldots$$

drawing code combinations, to produce just

$$C_{n-1} = \frac{1}{n}\binom{n-1}{2n-2} = 1, 2, 5, 14, 42, 132, 429, 1430, 4862, \ldots$$

distinct compositions, we are creating a *huge* number of duplicates for relatively small n. Clearly our current method is a highly inefficient one. In our pursuit of certainty that our set contained every possible composition combination, we have done a lot of unnecessary computation. In the next section 4.2.2, we attempt an improvement.

4.2.2 Branch Counting

The main benefit of our drawing codes method in 4.2.1 was also it's main drawback. Its great simplicity allowed us to create a very large list of trees, within which we could be confident resided every possibility. However, it created a huge number of duplicates, and so took a long time for relatively small n. We look now to develop a more streamlined method. To do this we look for a more *restrictive* property shared by all trees.

Consider Figure 4.5, where we introduce the idea of expressing a function

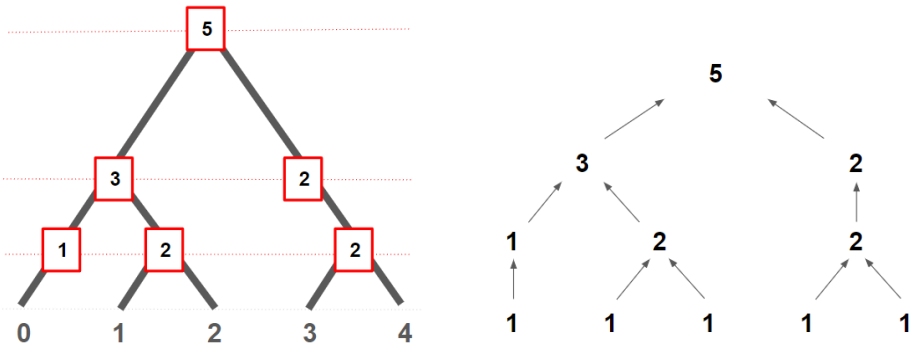


Figure 4.5: *Technique for tree creation by branch-count pyramids. On the left, the red box at each branch meeting point signifies the number of leaves connected to it. On the right, a picture representation of a branch-count pyramid.*

tree as its *branch-count pyramid*. Notice two things: firstly, at each level of a branch-count pyramid, all branch counts must be equal to the total number of leaves in the tree. Secondly, observe that, for a branch-count pyramid to represent a valid tree, each branch count at a certain level must either be equal to the corresponding branch count below, or be equal to the sum of the corresponding two below it. We express these two axioms formally. Consider a branch count pyramid as a set of *levels*, where the i^{th} level (levels are illustrated by the red horizontal lines depicted in Figure 4.5) is represented as a list of natural numbers $L_i = [x_1^i, \ldots, x_k^i]$ corresponding to branch counts.

Definition 8 (Axioms of Validity for Branch-count Pyramids)**.** *For a branch count pyramid to represent a valid n-leaf function tree, the following axioms must hold.*

1. *L_0 must be a list of n 1s.*
2. *The top level, L_{top} must be $[n]$.*
3. *For all $L_i = [x_1^i, \ldots, x_j^i, \ldots, x_k^i]$ between the top and bottom levels, for all x_j^i we must either have that*
   (a) *$x_j^i = x_j^{i-1}$,*
   (b) ***or*** *$x_j^i = x_j^{i-1} + x_{j+1}^{i-1}$.*
4. *For all $L_i = [x_1^i, \ldots, x_j^i, \ldots, x_k^i]$ between the top and bottom levels, $x_1^i + \cdots + x_j^i + \cdots + x_k^i = n$.*

Using this set of axioms we hope to generate a list of *pyramid forms* (each corresponding to a function composition) which contains far fewer duplicates than our list of drawing codes.

Our task then, is to assemble all valid pyramid combinations as efficiently as possible. We start by defining a recursive function `get_combos`, which takes a natural number n as input and outputs the set of all combinations of natural numbers which sum to n. We make use of `get_combos` in defining the function at the heart of our method: `next_levels`, which takes a pyramid level L_i as input, and outputs the set of all possible next levels, i.e. all L_{i+1} which satisfy the axioms in Definition 8.

Next, we define `get_pyramids`, another recursive function which takes a pyramid base level L_0 of size n, and uses `next_levels` to produce a list of all produce all possible pyramids of base length n in *pyramid form*:

$$[L_0, \ldots, L_i, \ldots, L_{\text{top}}]$$

Finally, we define `tree_formation`, which takes a pyramid form and generates the corresponding tree form, as produced by our earlier function `tree_builder`. To better picture these representations, consider the tree depicted in Figure 4.5. In Python, the *pyramid form* of this particular tree we would be

$$[[1,1,1,1,1],[1,2,2],[3,2],[5]],$$

with the corresponding *tree form* being

$$[[[1,2],[3,4]],[[0,1]],[[0,1]].$$

For $n = 2, 3, 4, 5, 6, 7, 8, 9, 10, \ldots$, we find the number of valid pyramid forms to be

$$1, 2, 7, 34, 214, 1652, 15121, 160110, 1925442, \ldots.$$

When we transform to a list of n-leaf tree forms, and remove duplicates, we are left again with C_{n-1} distinct trees for each n.

4.2.3 Dyck Paths - An Aside

We feel it pertinent to discuss the relationship between our composition structures and the Catalan numbers. It can be rigorously proven that C_{n-1} gives the number of possible n-leaf function trees. For a proof see Chapter 5 of [12][1]. We will not give a proof here, however we will discuss a highly intuitive observation in an effort motivate its existence.

It can also be proven that the n^{th} Catalan Number C_n is the number of *Dyck Paths* of length $2n$ (see Chapters 2 and 4 of [12]). A Dyck Path of length $2n$ is a connected path in 2-D space which starts at $(0, 0)$ and ends at $(n, 0)$; is made up of n discrete diagonally upwards movements and n discrete diagonally downwards movements; and never crosses the x-axis. For example, Figure 4.6 shows the $C_2 = 2$ Dyck Paths of length 4. Now, remember that the number of

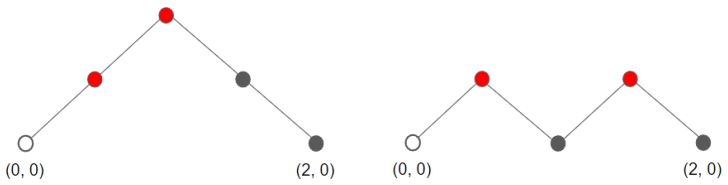


Figure 4.6: *The two Dyck Paths of length 4. Red dots represent the end of an upward diagonal movement, grey dots the end of a downward diagonal movement.*

n-leaf function compositions is equal to C_{n-1}, and so the number of Dyck Paths of length 4 is equal to the number of 3-leaf compositions (i.e. two). They are

$$0(1)(2), \quad 0(1(2)).$$

Consider now Figure 4.7, which illustrates a bijection between the two Dyck Paths of length 4 and the two 3-leaf function compositions. This image is by no means proof of anything, but motivates a a very intuitive bijection between the set of length n-leaf compositions and and the set of Dyck Paths of length $2n-2$. Credit for the observation of this bijection is owed in part to Steven Wolfram, who hints at it in [13].

In fact this bijection holds true for all n. We wrote a class `dyckpaths` which generates the Dyck Paths of size $2n-2$ which in turn are easily translated, by the bijection shown in Figure 4.7, one by one to form the complete set of n-leaf function compositions. It does so very efficiently, without creating duplicates and

[1] In this paper the author refers to our function trees as "ordered trees", and depicts them slightly differently, however they clearly represent the same concept.

with computational cost linear with n. Due to time pressures we did not manage to adapt this method to produce expressions in star form, and so we simply share it as an aside.

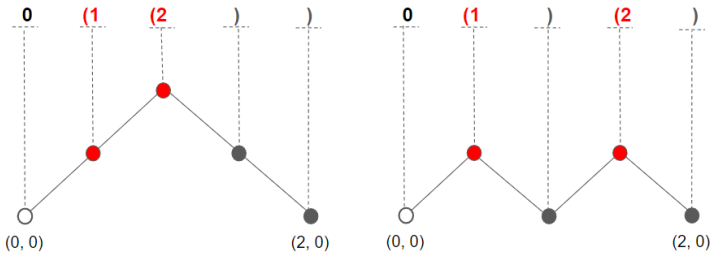


Figure 4.7: *A bijection illustrated between the Dyck Paths of length 4 and the 3-leaf function compositions.*

4.3 SKI Combinatorics

4.3.1 SKIsimplify

Now that we have all compositions of n functions, it is a simple task to create the set of all possible SKI function compositions of size n by subbing in each of the 3^n permutations with repetition of n `S`s `K`s and `I`s. We end up with a bank of $3^n C_{n-1}$ SKI functions. This is huge number for relatively small n, and so, limited by computing power, we only generate expressions up to size $n = 9$.

One problem we now face is that a number of this resulting set of functions are divergent. The evaluation of these functions will lead to a stack overflow[2] in Python. Another problem is the effective anonymity of the vast, vast majority of our bank of SKI functions: aside from a handful of known quantities, such as `S(K)(K)`, it is virtually impossible to know what each function does without first evaluating and testing it. To solve both of these problems we define another function: `SKIsimplify`.

`SKIsimplify` takes one of our SKI compositions as a string, in star form, and uses a string parsing sub-function called `find_inputs` to identify opportunities for weak reduction before physically enacting them and updating the string expression in accordance. We remind you now of the axioms of weak reduction, this time expressed in star form.

$$((K * A) * B) \quad \rightarrow_w \quad A$$

$$(((S * A) * B) * C) \quad \rightarrow_w \quad ((A * C) * (B * C))$$

[2]A stack overflow is complete depletion of memory space, often caused by an infinitely recursive function.

$$(I * A) \quad \rightarrow_w \quad A.$$

We reduce in the same manner as Python would, by a strict leftmost strategy. After each weak reduction step we reiterate with the updated string. Iterations will continue until either the expression reaches a weak-normal form, or until a specified maximum iteration count is reached. If a normal form is reached, it is labelled as a normal form and returned. If the maximum iteration count is reached, the expression is labelled *volatile* and saved separately from the set of normal forms.

Once we either convert to normal form, or label as volatile and discard, each of our SKI expressions, we remove duplicates once again. We therefore end up very large bank of SKI functions, none of which are weakly equivalent. The table below gives an idea of the proportions of SKI functions that are volatile for each n, and the proportions weakly equivalent for each n. Its perhaps not surprising but definitely noteworthy that we only see volatile (and hence probably divergent) forms for $n > 5$. Another important but perhaps foreseeable result is that the proportion of distinct normal forms compared with original bank size decreases significantly with n.

n	Original Number	After Volatile Removal	Distinct Normal Forms
2	9	9	9
3	54	54	30
4	405	405	108
5	3402	3402	438
6	30,618	30,613	1920
7	288,683	288,469	8690
8	2,814,669	2,808,424	40,584
9	28,146,690	28,010,829	192,725

4.3.2 Volatile SKI Functions

Divergent expressions will all naturally be labelled volatile. However not all volatile expressions will be divergent. We set our maximum iteration count as 50, inspired by this [13] frankly amazing article by Steven Wolfram, in which he thoroughly examines the behaviour of a few highly volatile but ultimately convergent combinator expressions. He looks at one extraordinary case of a combinator that takes $137,356,329$ reduction steps and reaches a maximum size of $105,723$ before arriving at its normal form. For the sake of simplicity and computing time, we proceed without the few convergent functions which take more then 50 steps to find their normal.

4.4 Discovering functions in SKI

It is now time to test our functions: to iterate over them all, applying each to specific input, and observe the output. We still cannot use Python function evaluation at this stage, as the moment we compose each of our normal forms with

other functions we will create divergent expressions. Thus, we will use our trusty `SKIsimplify` to physically reduce expressions. Another benefit of this method is that we can more easily observe the functions we are creating.

Note that we already know about a few useful SKI functions. For instance,

$$\texttt{I} = \lambda x.\ x \to_\eta \lambda f.\ \lambda x.\ f(x) = \texttt{ONE},$$

and so `I` and `ONE` are interchangeable. We also known that `K(I)` is the function which takes any function and gives the identity. Thus

$$\texttt{K(I)} = \texttt{ZERO} = \texttt{FALSE}.$$

And finally, `K` = `TRUE`. With these observations in hand we can start function testing.

We test for the function `OR` by running Code Cell 17.

Code Cell 17

```
possible_ORs = []
for SKI_func in SKI_data:

      TT = SKIsimplify("(("+ SKI_exp + " * K) * K)")[0]
      FF = SKIsimplify("(("+ SKI_exp + " * (K * I)) * (K * I))")[0]
      TF = SKIsimplify("(("+ SKI_exp + " * K) * (K * I))")[0]
      FT = SKIsimplify("(("+ SKI_exp + " * (K * I)) * K)")[0]

      if TT == "K" and FF == "(K * I)" and TF == "K" and FT == "K":
            possible_ORs.append(SKI_func)
```

We get many possibilities for `OR`, one of which is S(I)(K(K)). Sure enough, when we evaluate like normal in Python

```
OR = ((S * I) * (K * I))

print(bool_SHOW(eval(OR)(TRUE)(TRUE)),
    bool_SHOW(eval(OR)(FALSE)(TRUE)),
      bool_SHOW(eval(OR)(TRUE)(FALSE)),
          bool_SHOW(eval(OR)(FALSE)(FALSE)))

Output:
# TRUE, TRUE, TRUE, FALSE
```

In a very similar fashion we find `NOT` = S(S(I)(K(K(I))))(K(K)) and `AND` = S(S)(K(K(K(I)))).

To find `NEXT` we have to work a bit harder. We run the code

Code Cell 18

```
possible_NEXTs = []
for SKI_func in SKI_data:

    perhaps = SKIsimplify('(((' + SKI_exp + ' * S) * K) * I)')[0]
    if perhaps == '(K * ((S * K) * I))':
        possible_NEXTs.append(SKI_func)
```

Where we are looking for outputs in the form `'(K * ((S * K)* I))'` after inputting `S`, `K` and `I` consecutively based on the form of the λ expression for `NEXT` as defined in Code Cell 5. As we might have expected, multiple functions give this output, however, applying them all repeatedly to `ONE`, we are able to identify `NEXT` = S((S(K(S)))(K)). Thus we can define `TWO` = S((S(K(S)))(K))(I); and `THREE` = S((S(K(S)))(K))((S((S(K(S)))(K)))(I)); and so on.

We find `ADD` = S(I)(K(S((S(K(S)))(K)))) very similarly to `NEXT`.

Due to time constraints and, more importantly, constraints on computing power, we were not able to find further familiar functions in SKI form. However what we have found already give us the capability of iteration and conditional branching as discussed in Section 2. Already we have a small sense of the universality, the Turing Completeness, of SKI Calculus.

Conclusion

λ Calculus is, at its heart, is simply a notation for defining functions. However it is also perhaps the most effective tool the mathematics community has for making clear the power of purely function mathematical languages.

Appendices

Appendix A

```
    import copy
from itertools import product
from itertools import permutations
import numpy as np

def tree_builder(di, order):

    di1 = di.copy()
    order1 = order.copy()

    tree = []

    n = len(di1)

    fs = list(np.linspace(0, n-1, n, dtype=int))

    while len(di1) > 2:

        lvl = []
        branch = []

        for i, d in enumerate(di1):

            if d == "L" and di1[i-1] == "R":
                lvl.append([i-1, i])
                branch.append([fs[i-1], fs[i]])

        to_remove = []

        for i, l in enumerate(lvl):

            k = l[0]

            if order1[k] < order1[k+1]:
                to_remove.append(k+1)

            elif order1[k] > order1[k+1]:
```

```
            to_remove.append(k)

        j = 0
        for i in range(len(di1)):
            if i in to_remove:
                del di1[i-j]
                del order1[i-j]
                del fs[i-j]
                j += 1

        tree.append(branch)

        fs = list(np.linspace(0, n-1-len(to_remove), n-len(to_remove),
            dtype=int))

    tree.append([[0, 1]])
    return tree
```

Appendix B

```
def tree_to_exp(tree, n):

f = list(range(0, n))
f = [str(i) for i in f]

f2 = [i for i in list(range(0, n))]

exps = [f, ]
nums = [f2, ]

for i in range(len(tree)):

    exp = []
    num = []

    j = 0
    while j < len(f):

        if [j, j+1] in tree[i]:
            term = f"({f[j]} * {f[j+1]})"
            exp.append(term)

            no = [f2[j], f2[j+1]]
            num.append(no)
            j += 2

        else:
```

```
                term = f"{f[j]}"
                exp.append(term)

                no = f2[j]
                num.append(no)
                j += 1


        exps.append(exp)
        nums.append(num)
        f = exp
        f2 = num

    return exps[-1][0]
```

Appendix C

```
    def find_combos_recursive(arr, index, num, reducedNum, arrs):

    if (reducedNum < 0):
        return

    if (reducedNum == 0):
        arrs.append(arr[:index])
        return

    prev = 1 if(index == 0) else arr[index - 1]

    for k in range(prev, num + 1):

        arr[index] = k

        find_combos_recursive(arr, index + 1, num, reducedNum - k, arrs)


def get_combos(n, arrs):

    arr = [0] * n

    find_combos_recursive(arr, 0, n, n, arrs)
```

Appendix D

```
    import math
from itertools import permutations
```

```
def next_levels(canopy):

    n = sum(canopy)
    width = len(canopy)

    if width < 3:
        return [n]

    arrss = []
    get_combos(n, arrss)

    arrs = [arr for arr in arrss if width > len(arr) >=
        math.ceil(width/2)]
    arrs = [list(permutations(a)) for a in arrs]

    list_sum = [a for arr in arrs for a in arr]
    arrs = list_sum
    arrs = list(set(arrs))

    valid_arrs = []
    for arr in arrs:

        j = 0
        i = 0
        valid = True

        while j < width:
            j += 1
            i += 1

            if canopy[j-1] != arr[i-1] and j == width:
                valid = False
                break

            if canopy[j-1] != arr[i-1]:
                if canopy[j-1] + canopy[j] == arr[i-1]:
                    j += 1
                else:
                    valid = False
                    break

        if valid == True:
            valid_arrs.append(arr)

    return valid_arrs
```

Appendix E

```
def get_pyramids(Z, tree_list):

lvls = next_levels(Z[-1])

array = [Z.copy() for i in range(len(lvls))]

for i in range(len(lvls)):
    array[i].append(lvls[i])

for arr in array:
    if type(arr[-1]) == int:
        tree_list.append(arr)
    else:
        get_pyramids(arr, tree_list)
```

Appendix F

```
def tree_formation(tree):

big_tree = []

for i in range(len(tree)-2):
    sub_tree = []
    j = -1
    k = -1
    while j < len(tree[i])-2:
        j += 1
        k += 1
        if tree[i][j] != tree[i + 1][k]:
            if tree[i][j] + tree[i][j + 1] == tree[i + 1][k]:
                sub_tree.append([j, j+1])
                j += 1

    big_tree.append(sub_tree)

big_tree.append([[0, 1]])
return big_tree
```

Appendix G

```
def find_input(multiplier, exp):
```

```
    right = 0
    left= 0
    inputs = False
    number = False

    leng = len(multiplier)

    inp = ""
    num = ""

    for i, char in enumerate(exp):

        if exp[i-4-leng:i] == "(" + multiplier + " * ":
            inputs = True
            right = 0
            left = 0

        if inputs and left == 0 and char.isnumeric():
            number = True
            num += char

        if number and left == 0 and not char.isnumeric():
            inp = num
            return inp

        if char == "(":
            left += 1
        if char == ")":
            right += 1

        if inputs and left >= right:
            inp += char

        if inputs and not number and left == right:
            left = 0
            right = 0
            return inp
```

Appendix H

```
    def get_structure(exp):
    '''
    Splits an expression into its functions (in order) and its
        composition structure.
    '''
    new_exp = ""
    SKI = []
```

```
    j = 0
    for char in exp:
        if char == "S" or char == "K" or char == "I":
            new_exp += f"{j}"
            SKI.append(char)
            j += 1
        else:
            new_exp += char

    return new_exp, SKI

    import re

    def subin_to_exp(exp, SKI):

        l = len(SKI) - 1
        for i, s in enumerate(SKI):
            exp = re.sub(fr"{l - i}", fr"{SKI[l - i]}", exp)

        return exp

    ### defining SKIsimplify()
import re

def SKIsimplify(SKI_exp):

    exp, SKI = get_structure(SKI_exp)

    only_S = False

    shortest_exp = (len(exp), exp, SKI)

    it = 0
    j = -1
    while j < len(SKI)-1:
        it += 1
        j +=1


        mult = f"{j}"
        # identifying the functions curried inputs
        inputs = []
        while len(inputs) <= 3:
            inp = find_input(mult, exp)
            if inp == None:
                break

            inputs.append(f"{inp}")
            mult = f"({mult} * {inp})"
```

```
## Create inputss, its just inputs but has a \ before every
    symbol so re.sub can read it
inputss = []
for inp in inputs:
    inps = ""
    for i in inp:
        if i == "(" or i == ")" or i == "*":
            i = re.sub(fr"\{i}", fr"\\{i}", i)
        inps += i
    inputss.append(inps)



### Performing the action of the combinators in the expression

if SKI[int(j)] == "K" and len(inputs) >= 2:

    exp = re.sub(fr"\(\({j} \* {inputss[0]}\) \*
        {inputss[1]}\)", fr"{inputs[0]}", exp)
    j = -1
    only_S = False

elif SKI[int(j)] == "I" and len(inputs) >= 1:

    exp = re.sub(fr"\({j} \* {inputss[0]}\)", fr"{inputs[0]}",
        exp)
    j = -1
    only_S = False

elif SKI[int(j)] == "S" and len(inputs) >= 3 and only_S:

    exp = re.sub(fr"\(\(\({j} \* {inputss[0]}\) \*
        {inputss[1]}\) \* {inputss[2]}\)",
                fr"(({inputs[0]} * {inputs[2]}) * ({inputs[1]} *
                    {inputs[2]}))", exp)
    j = -1
    only_S = False

if j == len(SKI) - 1 and not only_S:
    j = -1
    only_S = True


## Renumbering our new expression, and finding the new order of
    S, K, Is
nums = []
new_exp = ""
count = 0
num = ""
number = False
```

```
        for i, char in enumerate(exp):
            if char.isnumeric() and len(exp) == 1:
                number = True
                new_exp += "0"
                nums.append(char)
            if char.isnumeric() and len(exp) != 1:
                number = True
                num += char
            if number and not char.isnumeric():
                number = False
                new_exp += f"{count}"
                nums.append(num)
                num = ""
                count += 1
            if not number:
                new_exp += char

        ## Reording our SKI list according to the new combination after
            the action of the combinators
        new_SKI = []
        for no in nums:
            new_SKI.append(SKI[int(no)])

        ## Updating our expression and SKI list
        SKI = new_SKI
        exp = new_exp

        ## update the shortest new_exp so far - will be returned if the
            combo is volatile
        if len(exp) < shortest_exp[0]:
            shortest_exp = (len(exp), exp, SKI)
        if it > 100:
            return subin_to_exp(shortest_exp[1], shortest_exp[2]), "vol"

    return subin_to_exp(exp, SKI), "con"
```

Appendix I

```
class FNCT():

def __init__(self, string):
    self.str = string
    self.f = eval(string)

def __mul__(self, other, *args):
    new_str = "(" + self.str + other.str + ")"
    return FNCT(new_str)
```

```
    def __call__(self, other):
        return self.f(other)
```

Appendix J

```
    class dyckpaths:

    def __init__(self, n, one_count, zero_count):

        self.left = None
        self.right = None

        if one_count < n:
            self.left = dyckpaths(n, one_count + 1, zero_count)
        if zero_count < one_count:
            self.right = dyckpaths(n, one_count, zero_count + 1)


    def get_dyckpaths(self, root, paths):

        if self.left:
            self.left.get_dyckpaths(root + [1], paths)

        if self.right:
            self.right.get_dyckpaths(root + [0], paths)

        if self.left == None and self.right == None:
            paths.append(root)

paths = []
dyckpaths(4, 0, 0).get_dyckpaths([], paths)

exps = []

for path in paths:
    exp = ''
    num = -1
    for p in path:
        if p == 1:
            num += 1
            exp += f'({num}'
        if p == 0:
            exp += ')'

    exps.append(exp)
```

```
print(exps)
```

Bibliography

[1] Morris, Jr, J.H. (1966) Lambda-Calculus Models of Programming Languages. dissertation.

[2] Alama, J. and Korbmacher, J. (2018) The lambda calculus, Stanford Encyclopedia of Philosophy. Stanford University. Available at: https://plato.stanford.edu/entries/lambda-calculus/ (Accessed: March 31, 2023).

[3] COMS W3261 CS Theory Lecture 23: The lambda calculus I (no date) L23-Lambda-Calculus-I. Available at: $http://www.cs.columbia.edu/aho/cs3261/Lectures/L23-Lambda_Calculus_I.html$ (Accessed: March 31, 2023).

[4] Garcia, R. Category Theory and Lambda Calculus (2018). Universidad de Granada.

[5] Hindley, R.J. and Seldin, J.P. (2008) Lambda Calculus and Combinators an Introduction. Cambridge University Press.

[6] Barendregt, Hendrik Pieter (1984). The Lambda Calculus: Its Syntax and Semantics. Elsevier.

[7] Nisnevich, A. (2013) The Consistency of Lambda Calculus. dissertation.

[8] 1 lambdacalculusevaluation - cornell university (no date). Cornell University. Available at: https://www.cs.cornell.edu/courses/cs4110/2012fa/lectures/lecture14.pdf (Accessed: March 31, 2023).

[9] Ugo Dal Lago, Simone Martini, The weak lambda calculus as a reasonable machine, Theoretical Computer Science, Volume 398, Issues 1–3, 2008, Pages 32-50, ISSN 0304-3975, https://doi.org/10.1016/j.tcs.2008.01.044. (https://www.sciencedirect.com/science/article/pii/S0304397508000583)

[10] Alonzo Church. An unsolvable problem of elementary number theory. American Journal of Mathematics, 58(2):345–363, 1936

[11] A. M. Turing. Computability and ń-definability. The Journal of Symbolic Logic, 2(4):153–163, 1937

[12] Roman, S. (2019) An introduction to Catalan numbers / Katalan Shu Ru Men / (mei) Shidiwen Luoma Zhu. Haerbin Shi: Haerbin gong ye da xue chu ban she.

[13] Max, H. (2020) Combinators: A centennial view-stephen wolfram writings, Stephen Wolfram Writings RSS. Available at: https://writings.stephenwolfram.com/2020/12/combinators-a-centennial-view/ (Accessed: March 31, 2023).

[14] "The church-turing 'thesis' as a special corollary of Gödel's completeness theorem" (2013) Computability [Preprint]. Available at: https://doi.org/10.7551/mitpress/8009.003.0005.

[15] Tall, D. (2011) CRYSTALLINE CONCEPTS IN LONG-TERM MATHEMATICAL INVENTION AND DISCOVERY. FLM Publishing Association, Edmonton, Alberta, Canada.

[16] Wolfram, S. (1970) Outline of the principle: A new kind of science: Online by Stephen Wolfram [page 717], Wolfram Science and Stephen Wolfram's 'A New Kind of Science'. Wolfram Media, Inc. Available at: https://www.wolframscience.com/nks/p717–outline-of-the-principle/ (Accessed: March 31, 2023).

[17] The on-line encyclopedia of integer sequences® (OEIS®) (no date) The On-Line Encyclopedia of Integer Sequences® (OEIS®). Available at: https://oeis.org/ (Accessed: March 31, 2023).