Lambda Calculus and Types

Sergey Ichtchenko

Mini-project
Hilary Term 2025

Contents

Abstract

In this mini-project, we will explore different theoretical and practical results in Lambda Calculus.
In Question 1, we will aim to show that, although leftmost reductions can always find a β-normal form for a λ-term, this reduction may not be efficient: through a few subparts, we build up to a result which states that, with respect to the size of the λ-term, leftmost reduction pathways can grow to be quadratic in length while the optimal pathway will remain at most linear in length. This is an interesting result, as it shows that the length of the leftmost reduction path can be used to show computability results but may not be applicable to reason about complexity, as its length can be longer than optimal.
In Question 2, we will build various arithmetic and mathematical operations within λ-calculus and show how they are defined. While we know from the lectures that all computable functions are λ-definable, these constructions give insight on how exactly such operations are implemented recursively, as opposed to the standard imperative manner many programmers are familiar with.
Finally, in Question 3, we will define various data structures and implement a larger project, to ultimately show that DFAs and their related operations can be implemented in λ-calculus. This construction also serves as a proof that λ-calculus is strictly more expressive than DFAs and regular languages, as the latter can be implemented in the former.
Throughout this mini-project, we will formally prove why each λ-term that we define indeed reduces to the right term. In addition, in Questions 2 and 3, we will provide examples of terms with their inputs and what they reduce to. This is interesting to see in addition to the formal reasoning provided in the theorems and proofs. To evaluate the λ-terms, we will use the `lcalc` tool available at https://github.com/ryspark/lcalc. The λ-terms that we will be evaluating for questions 2 and 3 can be found in Appendices A and B, respectively. Note that these appendices have the directive `#import "common"` at the beginning. This simply allows us to use common terms that we have seen in lectures, such as `ISZERO`, when defining our other terms. However, any terms that have not been defined during the course are re-defined in the appendices.

Question 1

In this question, we will be discussing the difference in length between the optimal and leftmost reduction paths for β-normalisable terms. Note that we will be indicating leftmost reductions with the standard arrow $\xrightarrow{l}$ and optimal reductions with the new notation $\xrightarrow{op}$.

Part (a)

For every natural number k, find a β-normalisable term t such that the leftmost reduction sequence starting at t is exactly k steps longer than the optimal reduction sequence.

To define such a term, we will define a term that repeatedly duplicates a term $(\mathbf{i}\,\mathbf{i})$. The idea is that, if the duplication is done first in the leftmost reduction, then it will take longer to reduce all of the terms than if the term $(\mathbf{i}\,\mathbf{i})$ is reduced first and then the duplication is done.

Definition 1.1: Term with an additively longer leftmost reduction sequence

For any $k \in \mathbb{N}$, let $t_k = (\lambda x.\underbrace{x\,x\,\ldots\,x}_{k+1 \text{ times}})(\mathbf{i}\,\mathbf{i})$

Theorem 1.2: Correctness of t_k

For any $k \in \mathbb{N}$, the leftmost reduction sequence from t_k is exactly k steps longer than its optimal reduction sequence.

Proof:

Consider the case when $k = 0$.
$t_0 = (\lambda x.x)(\mathbf{i}\,\mathbf{i}) = \mathbf{i}\,(\mathbf{i}\,\mathbf{i}) \to \mathbf{i}\,\mathbf{i} \to \mathbf{i}$. While there are two ways to reduce the original expression $\mathbf{i}\,(\mathbf{i}\,\mathbf{i})$, they both result in the same result. Thus all reduction pathways have the same length, as required.

Now let $k > 0$.

$$t_k = (\lambda x.\underbrace{x\,x\,\ldots\,x}_{k+1 \text{ times}})(\mathbf{i}\,\mathbf{i})$$

$$t_k \xrightarrow{op} (\lambda x.\underbrace{x\,x\,\ldots\,x}_{k+1 \text{ times}})\,\mathbf{i} \xrightarrow{op} \underbrace{\mathbf{i}\ldots\mathbf{i}}_{k+1 \text{ times}} \xrightarrow{op} \ldots \xrightarrow{op} \mathbf{i}$$

$$t_k \xrightarrow{l} \underbrace{(\mathbf{i}\,\mathbf{i})\ldots(\mathbf{i}\,\mathbf{i})}_{k+1 \text{ times}} \xrightarrow{l} \mathbf{i}\,\underbrace{(\mathbf{i}\,\mathbf{i})\ldots(\mathbf{i}\,\mathbf{i})}_{k \text{ times}} \xrightarrow{l} \underbrace{(\mathbf{i}\,\mathbf{i})\ldots(\mathbf{i}\,\mathbf{i})}_{k \text{ times}} \xrightarrow{l} \ldots \xrightarrow{l} \mathbf{i}$$

In the optimal reduction case, we use one step to reduce $(\mathbf{i}\,\mathbf{i})$ to $\mathbf{i}$, one step to apply the duplication function to $\mathbf{i}$, and then k steps to reduce all of the applications of $\mathbf{i}$ to just one for a total of $k + 2$ reduction steps.
In the leftmost reduction, we use one step to apply the duplication function to $(\mathbf{i}\,\mathbf{i})$, and then $2k + 1$ steps to reduce the remaining pairs of identity terms down to one remaining one for a total of $2k + 2$ reduction steps.

The difference between these two is $2k + 2 - (k + 2) = k$, as required. □

Part (b)

For every natural number k, find a β-normalisable term t such that the leftmost reduction sequence starting at t is at least k times longer than the optimal reduction sequence.

We can expand on the idea of part (a). In Definition 1.1 we replicated the term $(\mathbf{i}\,\mathbf{i})$ a number of times. Here we will do the same, but also replicate the replicated sequence to obtain a multiplicative increase in the number of reduction steps of the leftmost reduction path.

Definition 1.3: Term with a multiplicatively longer leftmost reduction sequence

For all natural numbers k, let $t_k = (\lambda x.\underbrace{x\ldots x}_{k\text{ times}})[(\lambda y.\ \underbrace{y\ldots y}_{k+1\text{ times}}\ )\,(\mathbf{i}\,\mathbf{i})]$

Theorem 1.4: Correctness of t_k

For any $k \in \mathbb{N}$, the leftmost reduction sequence from t_k is at least a factor of k longer than its optimal reduction sequence.

Proof:

When $k = 0$, the theorem holds trivially as any reduction sequence is at least 0 times as long as the optimal reduction sequence.
When $k > 0$, observe that

$$t_k = (\lambda x.\underbrace{x\ldots x}_{k\text{ times}})[(\lambda y.\ \underbrace{y\ldots y}_{k+1\text{ times}}\ )\,(\mathbf{i}\,\mathbf{i})]$$

$$t_k \xrightarrow{op} (\lambda x.\underbrace{x\ldots x}_{k\text{ times}})[(\lambda y.\ \underbrace{y\ldots y}_{k+1\text{ times}}\ )\,\mathbf{i}] \xrightarrow{op} (\lambda x.\underbrace{x\ldots x}_{k\text{ times}})(\ \underbrace{\mathbf{i}\ldots\mathbf{i}}_{k+1\text{ times}}\ ) \xrightarrow{op} \ldots \xrightarrow{op}$$

$$\xrightarrow{op} (\lambda x.\underbrace{x\ldots x}_{k\text{ times}})\,\mathbf{i} \xrightarrow{op} \underbrace{\mathbf{i}\ldots\mathbf{i}}_{k\text{ times}} \xrightarrow{op} \ldots \xrightarrow{op} \mathbf{i}$$

$$t_k \xrightarrow{l} \underbrace{((\lambda y.\ \underbrace{y\ldots y}_{k+1\text{ times}}\ )\,(\mathbf{i}\,\mathbf{i}))\ldots((\lambda y.\ \underbrace{y\ldots y}_{k+1\text{ times}}\ )\,(\mathbf{i}\,\mathbf{i}))}_{k\text{ times}} \xrightarrow{l} \ldots \xrightarrow{l}$$

$$\xrightarrow{l} \underbrace{((\lambda y.\ \underbrace{y\ldots y}_{k+1\text{ times}}\ )\,(\mathbf{i}\,\mathbf{i}))\ldots((\lambda y.\ \underbrace{y\ldots y}_{k+1\text{ times}}\ )\,(\mathbf{i}\,\mathbf{i}))}_{k-1\text{ times}} \xrightarrow{l} \ldots \xrightarrow{l} \mathbf{i}$$

For the optimal reduction pathway, we require one step to reduce $(\mathbf{i}\,\mathbf{i})$ to $\mathbf{i}$, one step to apply the replication function with parameter y, k steps to reduce the $\mathbf{i}$ terms, one more step to apply the replication function with parameter x and finally $k-1$ steps to reduce all of the remaining $\mathbf{i}$ terms. This makes a total of $1+1+k+1+k-1 = 2k+2$ reduction steps.

For the leftmost reduction path, we start by reducing the outermost duplication expression. Then, each time we reduce an expression involving λy we require $2k+2$ reduction steps to reduce it to an identity, followed by one more reduction to get rid of the identity. This is repeated $k-1$ times, after which the last remaining expression only requires $2k+2$ steps as the last $\mathbf{i}$ is not reduced. This makes for a total of $1+(2k+2+1)(k-1)+(2k+2) = 2k^2+3k$ steps.

Thus, since $2k^2+3k = k(2k+2)+k$, the leftmost reduction pathway is always at least a factor of k longer than the optimal reduction pathway. □

Part (c)

Let $|t|$ be the number of nodes in the construction tree of t. Find a family T of β-normalisable terms such that:

- For every $n \in \mathbb{N}$, there is a term $t \in T$ with $|t| \geq n$;
- There is a constant $c \in \mathbb{N}$, such that for every term $t \in T$, an optimal reduction sequence from t has a length of at most $c \cdot |t|$;
- There is a constant $d \in \mathbb{N}$, such that for every term $t \in T$, the leftmost reduction sequence from t has a length of at least $d \cdot |t|^2$.

We will approach this in a similar way as before, using multiplying terms to generate a sequence of terms $t \in T$ for which the optimal reduction pathway grows linearly in length and the leftmost pathway grows quadratically in length with respect to the number of nodes in the tree of t.

Definition 1.5: The family T

For every integer k, let

$$t_k = (\lambda x. \underbrace{x \ldots x}_{k \text{ times}})((\lambda y. \underbrace{y \ldots y}_{k \text{ times}})\,((\lambda y. \underbrace{z \ldots z}_{k \text{ times}})\,\mathbf{i}))$$

Then, we define the family T as

$$T = \{t_k\}_{k=37}^{\infty}$$

Theorem 1.6: Correctness of the family T

The family T satisfies the following conditions:

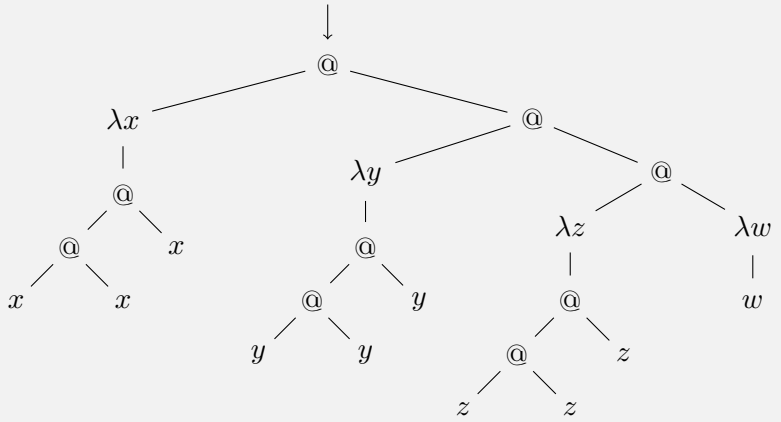
(i) For every $n \in \mathbb{N}$, there is a term $t \in T$ with $|t| \geq n$;

(ii) There is a constant $c \in \mathbb{N}$, such that for every term $t \in T$, an optimal reduction sequence from t has a length of at most $c \cdot |t|$;

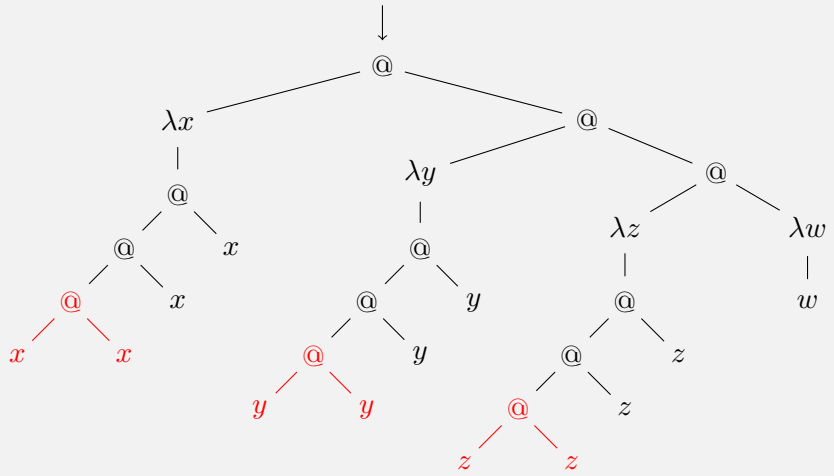
(iii) There is a constant $d \in \mathbb{N}$, such that for every term $t \in T$, the leftmost reduction sequence from t has a length of at least $d \cdot |t|^2$.

Proof:

We begin the proof by noting that the construction tree of t_k will always have $6k + 5$ nodes for any integer $k \geq 3$. To prove this, let us draw the construction tree of t_3:



For $k = 3$, we can see from the diagram that $|t_3| = 23 = 6 \cdot 3 + 5$. To get from the construction tree of t_k to t_{k+1}, we replace the leftmost leaf with the label x by the application of x to x, and similarly for y and z, an example of which is highlighted in red in the below diagram:



Thus, to get from t_k to t_{k+1}, we remove three nodes and add in nine nodes, resulting in an increase of 6 nodes. Hence by induction, each construction tree has $|t_k| = 6k + 5$ nodes. Since $37 \leq k < \infty$ and k becomes arbitrarily large, condition (i) holds.

The optimal reduction path will always have $3k$ reduction steps: one application of the term $\lambda z.z \ldots z$ to the identity, $k - 1$ reductions of identities, one application of the term $\lambda y.y \ldots y$ to the identity, $k - 1$ reductions of identities, one application of the term $\lambda x.x \ldots x$ to the identity, and $k - 1$ reductions of identities again for a total of $3k$ reductions. Thus we can let $c = 1$, and since $3k < 6k+5 = 1(6k+5) = c|t_k|$ for all $37 \leq k < \infty$, condition (ii) holds.

The leftmost reduction path will always have $k^3 + k^2 + k$ reduction steps. The reduction starts by applying $\lambda x.x \ldots x$ to its argument, which duplicates it k times. Then, for each of these k duplications, $\lambda y.y \ldots y$ duplicates its argument k times, and within each of these, $\lambda z.z \ldots z$ duplicates the identity k times. There are then k identity reductions in each case. So, there is a total of k^3 identity reductions, k^2 applications of $\lambda z.z \ldots z$, and k applications of $\lambda y.y \ldots y$. Furthermore, the final group to be reduced has one fewer applications as the final identity is not reduced but stays as the result of the computation. Hence the leftmost reduction path has $1 + k^3 + k^2 + k - 1 = k^3 + k^2 + k$ reduction steps.
We now let $d = 1$. Observe that the inequality $k^3+k^2+k > 1(6k+5)^2 = d|t_k|^2$ holds if and only if $k \geq 37$, which is why we impose this condition on the family T. Hence, condition (iii) also holds.

Therefore all of the conditions of Theorem 1.6 hold and we are done. □

Question 2

In each of the below parts (a)-(f), we are required to prove that the provided function is definable and find a term that defines it.

Part (a)

$$\phi_1 \colon \mathbb{N}^2 \to \mathbb{N} \quad \phi_1(n_1, n_2) = \gcd(n_1, n_2)$$

We will define a term for ϕ_1 by implementing the Euclidean algorithm in λ-calculus. One well-known recursive definition of the Euclidean algorithm is the following:

```
def euclidean_algorithm(a, b):
    if b == 0:
        return a
    else:
        return euclidean_algorithm(b, a % b)
```

Listing 1: The Euclidean algorithm in Python

To implement this, we will also need to implement the modulo operator for positive integers. To do this, it suffices to write a λ-term that takes in two integers and repeatedly subtracts one from the other until a small enough value is reached.

Definition 2.1: The modulo operator

We define the term *mod* as follows:

$$mod = \mathbf{y}(\lambda f n d.(lt\, n\, d)\, n\, (f\, (d\, pred\, n)\, d))$$

Where $\mathbf{y}$ is the fixed point combinator and lt is the less-than operator seen in lectures.

Lemma 2.2: Correctness of mod

Given two Church numerals n and d, $mod\, n\, d$ reduces to the encoding $\ulcorner n \pmod d \urcorner$

Proof:

The expression $n \pmod d$ is defined as the unique number x such that $0 \leq x < d$ and $x + kd = n$ for some integer k.
So, to obtain x, it suffices to subtract d from n a total of k times, which we know will be achieved when the inequality $x < d$ holds for the first time.
The λ-term *mod* takes parameters n and d, and compares them using the less-than operator. If $n < d$, then we have reached our result and it is returned. Otherwise, the function is called recursively on $d\, pred\, n$ and d, the former of which reduces to $\underbrace{pred(pred(...(pred}_{d \text{ times}}\ n))$ which reduces to $\ulcorner n - d \urcorner$. □

We can now define the term ϕ_1. The implementation below replicates the logic outlined in Listing 1.

Definition 2.3: The term ϕ_1

Define the term ϕ_1 as follows:

$$\phi_1 = \mathbf{y}(\lambda f a b.(zero?\, b)\, a\, (f\, b\, (mod\, a\, b)))$$

Theorem 2.4: Correctness of ϕ_1

Given two Church numerals a and b, $\phi_1\, a\, b$ reduces to $\ulcorner \gcd(a, b) \urcorner$

Proof:

The logic is implemented based on the algorithm in Listing 1. If the second argument is zero, the first one is returned. If the second argument is nonzero, then the function recurses with the second argument in the first position and the first argument modulo the second in the second position. □

Let us test this term by using a few examples. For this and all future examples in this section, we will use the `lcalc` tool along with the terms defined in Appendix A. Note that `lcalc` outputs λ-terms that it recognises, such as boolean values and Church numerals, in their readable form instead of in their pure λ-term form.

Example 2.5: Testing the term ϕ_1

Given the following input

```
PHI1 2 3
PHI1 3 2
PHI1 9 3
```

We get the following output

```
1
2
3
```

Thus the term works as expected on these examples.

Part (b)

$$\phi_2 \colon \mathbb{N}^5 \to \mathbb{N} \quad \phi_2(n_1, n_2, n_3, n_4, n_5) = \gcd(n_1, n_2, n_3, n_4, n_5)$$

By number theory, we know that gcd is associative. That is, $\gcd(\gcd(x, y), z) = \gcd(x, \gcd(y, z)) = \gcd(x, y, z)$ for all integers x, y, z. Thus also $\gcd(n_1, n_2, n_3, n_4, n_5) = \gcd(n_1, \gcd(n_2, \gcd(n_3, \gcd(n_4, n_5))))$. We therefore implement the term ϕ_2 as follows:

Definition 2.6: The term ϕ_2

The term ϕ_2 is defined as

$$\phi_2 = \lambda abcde.\phi_1\, a\, (\phi_1\, b\, (\phi_1\, c\, (\phi_1\, d\, e)))$$

Theorem 2.7: Correctness of ϕ_2

Given five Church numerals a, b, c, d, e, the term ϕ_2 reduces to $\ulcorner \gcd(a, b, c, d, e) \urcorner$

Proof:

The proof is immediate, since the λ-term uses the associativity of gcd by applying the function ϕ_1 defined in Definition 2.3 to the input terms. □

Let us evaluate this term on a few examples. It should be noted that evaluating this term is already incredibly slow, even though it has a β-normal form and eventually reduces to it.

Example 2.8: Testing the term ϕ_2

Given the following input

```
PHI2 2 2 2 2 2
PHI2 3 6 9 3 9
```

We get the following output

```
2
3
```

Thus the term works as expected on these examples.

Part (c)

$$\phi_3 : \mathbb{N} \to \mathbb{N} \quad \phi_3(n) = \begin{cases} 1 & n \text{ is prime} \\ 0 & \text{otherwise} \end{cases}$$

Recall that an integer n is prime if and only if its only divisors are 1 and n itself. Furthermore, an integer k is a divisor of n if and only if $n \pmod k = 0$. Thus it suffices to check the value of $n \pmod k$ for all integers $1 < k < n$, returning 0 if any of these are divisors and returning 1 otherwise.

Definition 2.9: The terms *prime* and ϕ_3

We define ϕ_3 using an auxiliary term *prime* as follows:

$$\phi_3 = \lambda n.(zero?\, n)\ \ulcorner 0 \urcorner\ ((zero?\, (pred\, n))\ \ulcorner 0 \urcorner\ (prime\, (pred\, n)\, n))$$
$$prime = \mathbf{y}(\lambda f k n.(zero?\, (pred\, k)\ \ulcorner 1 \urcorner\ ((zero?\, (mod\, n\, k))\ \ulcorner 0 \urcorner\ (f\, (pred\, k)\, n)))$$

Theorem 2.10: Correctness of ϕ_3

Given a Church numeral n, the term $\phi_3\, n$ reduces to $\ulcorner 1 \urcorner$ if n is prime and $\ulcorner 0 \urcorner$ otherwise.

Proof:

The term ϕ_3 starts by checking the two edge cases, 0 and 1, which are not prime by definition. If n is neither of these, then it begins the algorithm by passing *prime* the largest number smaller than n and n itself.
The term *prime* first checks if k is equal to one by checking if its predecessor is zero. If it is, we have checked all the numbers $1 < k < n$ without returning, so n is prime and we return $\ulcorner 1 \urcorner$.
Otherwise, the term checks whether k is a factor of n using the term *mod* defined in Definition 2.1. If it is, then it reduces to $\ulcorner 0 \urcorner$ since n is not prime. Otherwise, it recurses on the predecessor of k to test the next number smaller than k.
Thus ϕ_3 and *prime* implement the naïve prime checking algorithm. □

We will use this term to check whether the natural numbers up to 10 are prime.

Example 2.11: Testing the term ϕ_3

Given the following input

```
PHI3 0
PHI3 1
PHI3 2
PHI3 3
PHI3 4
PHI3 5
PHI3 6
PHI3 7
PHI3 8
PHI3 9
PHI3 10
```

We get the following output

```
0
0
1
1
0
1
0
1
0
0
0
```

Thus the term works as expected on these examples.

Part (d)

$$\phi_4 \colon \mathbb{N}^2 \to \mathbb{N} \quad \phi_4(n_1, n_2) = \text{ number of common prime factors of } n_1 \text{ and } n_2$$

Recall from number theory that any common divisor of two integers a and b will also divide $\gcd(a, b)$. We will use this fact to implement a λ-term that will iterate through all integers less than the gcd of the two inputs, and check whether they are in fact prime factors.

Definition 2.12: The terms *common* and ϕ_4

The term ϕ_4 is defined using an auxiliary term *common* as follows:

$$\begin{aligned} \phi_4 &= \lambda xy.\mathit{common}\,(\phi_1\,x\,y)\,(\phi_1\,x\,y) \\ \mathit{common} &= \mathbf{Y}(\lambda f k n.(\mathit{zero?}\,k)\,\ulcorner 0\urcorner\,((\mathit{zero?}\,(\mathit{mod}\,n\,k))\,(\mathit{add}\,(\phi_3\,k)\,(f\,(\mathit{pred}\,k)\,n))\,(f\,(\mathit{pred}\,k)\,n))) \end{aligned}$$

Where *add* is the term implementing addition of two integers from the lectures.

Theorem 2.13: Correctness of ϕ_4

Given two Church numerals a and b, the term $\phi_4\,a\,b$ reduces to the Church numeral representing the number of prime factors of a and b.

Proof:

The term ϕ_4 begins by calculating the GCD of the two input numbers using the term ϕ_1 defined in 2.3. Since the GCD is, by definition, the greatest common factor of the two integers, we know that there will be no larger common prime factors. In addition, since every common factor of two integers divides the GCD, it suffices to check all the numbers less than or equal to the GCD for primeness and divisibility.

The term *common* checks whether its first input k is zero. If it is, it can reduce to $\ulcorner 0\urcorner$, since there can be no smaller prime factors and 0 is not itself prime.
Otherwise, it checks whether k is a factor of the second argument n using the term *mod*. If it is not, then it simply recurses on the previous value of k. However, if it is, then in addition to recursing it calculates whether k is prime and adds the result to the value of the recursive call, thereby adding 1 if the factor is prime as required and 0 otherwise.
Therefore, this term implements counting how many common prime factors two numbers have, as required in the theorem. □

Again, we will test this on a few examples, which take a long time to evaluate.

Example 2.14: Testing the term ϕ_4

Given the following input

```
PHI4 2 3
PHI4 2 4
PHI4 6 6
```

We get the following output

```
0
1
2
```

Thus the term works as expected on these examples.

Part (e)

$$\phi_5 \colon \mathbb{N} \to \mathbb{N} \quad \phi_5(n) = \begin{cases} \text{Length of Collatz sequence starting at n} & n < 10^{20} \\ 0 & n \geq 10^{20} \end{cases}$$

We will first implement the term *collatz*, which returns the next step in the Collatz sequence, after which we will implement the main length counting function ϕ_5.

Definition 2.15: The term *collatz*

We will define the term *collatz* using the auxiliary functions *equals*?, *even*?, and *half*:

$$\begin{aligned} \mathit{equals?} &= \mathbf{y}(\lambda f a b.(\mathit{zero?}\, a)(\mathit{zero?}\, b)((\mathit{zero?}\, b)\, \mathbf{f}\, (f\, (\mathit{pred}\, a)\, (\mathit{pred}\, b)))) \\ \mathit{even?} &= \mathbf{y}(\lambda f n.(\mathit{zero?}\, n)\, \mathbf{t}\, (\mathbf{not}\, (f\, (\mathit{pred}\, n)))) \\ \mathit{half} &= \mathbf{y}(\lambda f k n.((\mathit{equals?}(\mathit{add}\, k\, k)\, n)\, k\, (f\, (\mathit{pred}\, k)\, n))) \\ \mathit{collatz} &= \lambda n.(\mathit{even?}\, n)\, (\mathit{half}\, n\, n)\, (\mathit{succ}(\mathit{add}\, n\, (\mathit{add}\, n\, n))) \end{aligned}$$

Lemma 2.16: Correctness of *collatz*

The term *equals*? checks whether two Church-encoded numbers a and b are equal.
The term *even*? evaluates to $\mathbf{t}$ if its parameter n is even and $\mathbf{f}$ otherwise.
The term *half* takes in two values, k and n, where n is even, and evaluates to $\ulcorner \frac{n}{2} \urcorner$. The value of k must be any value larger than $\frac{n}{2}$, so it suffices to use $k = n$ and evaluate $\mathit{half}\, n\, n$.
Finally, given a Church numeral n, the term $\mathit{collatz}\, n$ reduces to $\ulcorner \frac{n}{2} \urcorner$ if n is even and $3n + 1$ if n is odd.

Proof:

The term *equals*? returns the correct result: if a is zero, then it returns whether b is zero or not; if a is not zero but b is zero, then it returns false; otherwise, it repeats the check for the predecessors of the numerals.

The term *even*? evaluates to $\mathbf{t}$ if its input is $\ulcorner 0 \urcorner$. For all subsequent numbers, it evaluates to the negation of the result for the predecessor, which by induction is $\mathbf{t}$ for even numbers and $\mathbf{f}$ for odd numbers.

The term *half* takes in two values, k and n. If $k + k = n$, it reduces to k as it has found the value $k = \frac{n}{2}$. Otherwise, it will repeat the check for the predecessor of k. The term $\mathit{half}\, k\, n$ is β-normalisable if and only if $k \geq \frac{n}{2}$ and n is even. Thus we shall only evaluate $\mathit{half}\, n\, n$ for even values of n in the term below.

Finally, the term *collatz* takes in a term n as a parameter. It checks whether the encoded integer is even using the term *even*?, and evaluates $\mathit{half}\, n\, n$ if it is, since this term is β-normalisable in this case. Otherwise, it evaluates to $\ulcorner 3n + 1 \urcorner$ using the successor term and the addition of the term n three times. □

We can now implement the required term in a straightforward way. Observe that the number 10^{20} can be encoded as the Church numeral $\ulcorner 10^{20} \urcorner$.

Definition 2.17: The term ϕ_5

The term ϕ_5 is defined as follows:

$$\mathbf{y}(\lambda fn.(lt\, n\, \ulcorner 10^{20} \urcorner)\, ((zero?\, (pred\, n))\, \ulcorner 0 \urcorner\, (succ\, (f\, (collatz\, n))))\, (\ulcorner 0 \urcorner))$$

Theorem 2.18: Correctness of ϕ_5

Given a Church numeral n, the term $\phi_5\, n$ reduces to the Church numeral corresponding to the length of the Collatz sequence starting at n if $n < 10^{20}$, and $\ulcorner 0 \urcorner$ otherwise.

Proof:

If the input term n for the term ϕ_5 is more than 10^{20}, it immediately returns 0 as required. Otherwise, it checks whether the Collatz process has reached the end by checking if $n = 1$ by checking if its predecessor is zero: since zero never appears in any Collatz sequence, the fact that $pred \ulcorner 0 \urcorner = \ulcorner 0 \urcorner$ is of no importance. If it is, then the term reduces to $\ulcorner 0 \urcorner$ as required. On the other hand, if the argument is not one, then the term will reduce to the successor of the value returned by the recursive call with the argument $collatz\, n$, which results in the term counting the number of steps in the Collatz sequence starting at n.

In addition, since the Collatz conjecture has been verified for all values of $n < 10^{20}$, the defined term is β-normalisable. □

We will now test the term on the integers from 1 to 5. Note that including the Church numeral for 10^{20} would not be practical for these tests, as it would take up too much memory and lead to extremely large runtimes for the reduction simulation. As we know that the Collatz sequences for these values never exceed 16, we have replaced 10^{20} with 17 in this instance.

Example 2.19: Testing the term ϕ_5

Given the following input

```
PHI5 1
PHI5 2
PHI5 3
PHI5 4
PHI5 5
```

We get the following output

```
0
1
7
2
5
```

Note that the Collatz sequences in these test cases are the following:

```
1
2 1
3 10 5 16 8 4 2 1
4 2 1
5 16 8 4 2 1
```

Thus the term works as expected on these examples.

Part (f)

$$\phi_6 \colon \mathbb{N}^2 \to \mathbb{N} \quad \phi_6(n_1, n_2) = |\{n \in \mathbb{N} \colon 0 < n \leq n_1 \wedge \phi_5(n) = n_2\}|$$

We will again rely on a brute-force approach, enumerating all values of n which are at most n_1 and checking whether the previously defined term $\phi_5\, n$ evaluates to n_2.

Definition 2.20: The term ϕ_6

The term ϕ_6 is defined as follows:

$$\phi_6 = \mathbf{y}(\lambda f n t.(zero?\, n)\ \ulcorner 0 \urcorner\ ((equal?\, (\phi_5\, n)\, t)\, (succ\, (f\, (pred\, n)\, t))\, (f\, (pred\, n)\, t)))$$

Theorem 2.21: Correctness of ϕ_6

Given two input Church numerals n and t, the term $\phi_6\, n\, t$ evaluates to

$$\ulcorner |\{k \in \mathbb{N} \colon 0 < k \leq n \wedge \phi_5(k) = t\}| \urcorner$$

Proof:

If the input term n is zero, then the term evaluates to $\ulcorner 0 \urcorner$.
Otherwise, it checks whether the term $\phi_5\, n$ and t are equal using the *equal*? term. If they are not, then the term recurses on the previous value of n. If they are, then the term recurses in the same way but also evaluates to the successor of the result of the recursion, thereby counting the number of times when the condition holds. □

Again, we can test this term on a few small examples.

Example 2.22: Testing the term ϕ_6

Given the following input

```
PHI6 2 1
PHI6 3 1
PHI6 3 2
```

We get the following output

```
1
1
0
```

Thus the term works as expected on these examples.

Question 3

In this question, we will be encoding Deterministic Finite Automata (DFAs) in Lambda Calculus.

Part (a)

Suppose $Q \subseteq \mathbb{N}$, i.e. every state is identified by a natural number.
Devise an encoding for strings and DFAs in λ-calculus.

We will begin constructing the encoding by defining some auxiliary data structures that will come in useful.
Recall from lectures that a pair $\langle s, t\rangle$ was defined as $\lambda f.f\, s\, t$. For convenience, we will also define the pairing function $pair = \lambda xyf.f\, x\, y$, which when given two arguments s, t will return the pair $\langle s, t\rangle$ as defined above. We will extend this to a list of items by pairing an element of a list with the tail of the list as follows:

Definition 3.1: Lists and list operations

- A list is defined recursively as either the end indicator, defined as the identity term $\mathbf{i} = \lambda x.x$, or a pair $\langle a, l\rangle$ where a is any term and l is another list.
- The term *head* is defined as $head = \lambda x.x\, \mathbf{t}$
- The term *tail* is defined as $tail = \lambda x.x\, \mathbf{f}$
- The term *empty*? is defined as $empty? = \lambda l.l\, (\lambda xy.\mathbf{t})\, \mathbf{f}\, \mathbf{f}$
- The term *index* is defined as $index = \lambda li.head\ \ [i\ \ tail\ \ l]$

It is easy to see that the *head* and *tail* terms do what is expected: from lectures, we know that $\langle s, t\rangle\, \mathbf{t} = s$ and $\langle s, t\rangle\, \mathbf{f} = t$. By definition, the first element is always the head of the list and the second element is its tail. Proving that the last two terms work as intended is slightly more complicated, so we will state those as lemmas.

Lemma 3.2: The term *empty*?

When applied to a list, the term *empty*? checks whether the list is empty or not and returns true or false accordingly. That is, it satisfies the following rules:

$$empty?\ \ \mathbf{i} = \mathbf{t}$$
$$empty?\ \ \langle s, t\rangle = \mathbf{f}$$

Proof:

We will prove each case separately. For the former case,

$$\begin{aligned} empty?\ \ \mathbf{i} &= (\lambda l.l(\lambda xy.\mathbf{t})\, \mathbf{f}\, \mathbf{f})\, \mathbf{i} \\ &= \mathbf{i}\, (\lambda xy.\mathbf{t})\, \mathbf{f}\, \mathbf{f} \\ &= (\lambda xy.\mathbf{t})\, \mathbf{f}\, \mathbf{f} \\ &= (\lambda y.\mathbf{t})\, \mathbf{f} \\ &= \mathbf{t} \end{aligned}$$

On the other hand, for any pair $\langle s, t\rangle$,

$$\begin{aligned} \mathit{empty?}\ \langle s,t\rangle &= (\lambda l.l(\lambda xy.\mathbf{t})\,\mathbf{f}\,\mathbf{f})\,(\lambda f.f\,s\,t) \\ &= (\lambda f.f\,s\,t)\,(\lambda xy.\mathbf{t})\,\mathbf{f}\,\mathbf{f} \\ &= ((\lambda xy.\mathbf{t})\,s\,t)\,\mathbf{f}\,\mathbf{f} \\ &= \mathbf{t}\,\mathbf{f}\,\mathbf{f} \\ &= (\lambda xy.x)\,\mathbf{f}\,\mathbf{f} \\ &= \mathbf{f} \end{aligned}$$

□

Lemma 3.3: The term *index*

The term *index* takes in a list and the Church encoding of an integer, and returns the element at that position in the list. In this case, the list is zero-indexed, i.e. the first item in the list has index 0 and so on. For indices longer than the length of the list, the behaviour of this term is undefined.
Formally, the term satisfies the following property:

$$\mathit{index}\ \langle a_1, \langle a_2, \langle ..., \langle a_n, \mathbf{i}\rangle\rangle\rangle\rangle\ \ulcorner i \urcorner = a_i$$

where $0 \leq i < n$ is an integer.

Proof:

The term works since the Church encoding of an integer i applied to a function will evaluate that function i times on its parameters.

$$\begin{aligned} \mathit{index}\ \langle a_1, \langle a_2, \langle ..., \langle a_n, \mathbf{i}\rangle\rangle\rangle\rangle\ \ulcorner i \urcorner &= (\lambda li.\mathit{head}\ [i\ \ \mathit{tail}\ \ l])\,\langle a_1, \langle a_2, \langle ..., \langle a_n, \mathbf{i}\rangle\rangle\rangle\rangle\ \ulcorner i \urcorner \\ &= \mathit{head}\ \ (\ulcorner i \urcorner\ \ \mathit{tail}\ \ \langle a_1, \langle a_2, \langle ..., \langle a_n, \mathbf{i}\rangle\rangle\rangle\rangle) \\ &= \mathit{head}\left(\underbrace{\mathit{tail}(\mathit{tail}(...(\mathit{tail}}_{i \text{ times}}\,\langle a_1, \langle a_2, \langle ..., \langle a_n, \mathbf{i}\rangle\rangle\rangle\rangle)))\right) \\ &= \mathit{head}\,\langle a_i, \langle a_{i+1}, \langle ..., \langle a_n, \mathbf{i}\rangle\rangle\rangle\rangle) \\ &= a_i \end{aligned}$$

□

We can now move on to defining the encoding of strings and DFAs in λ-calculus.
We will begin with strings. As the alphabet $\Sigma = \{0, 1\}$, it suffices to define two elements that will represent these numbers and create a list of them in order.

Definition 3.4: Encoding of a string

Let $s = a_0 \ldots a_n$ be a string, where $a_i \in \{0, 1\}$ for all i. Define the encoding of each element a_i as

$$\ulcorner a_i \urcorner = \begin{cases} \mathbf{f}, & \text{if } a_i = 0 \\ \mathbf{t}, & \text{if } a_i = 1 \end{cases}$$

Then, the encoding of the string s is defined as the list

$$\ulcorner s \urcorner = \langle \ulcorner a_0 \urcorner, \langle ..., \langle \ulcorner a_n \urcorner, \mathbf{i}\rangle\rangle\rangle$$

Now we can encode DFAs using the above definition as a basis. First, observe that the question states

that the state set $Q \subseteq \mathbb{N}$. Without loss of generality, we may assume that, in fact, $Q = \{0, ..., n\}$ for some integer n. This is because any finite state set can be enumerated and relabelled to consecutive integers, with the transition function δ, initial state q_0, and accepting state set F relabelled accordingly with no effect on the structure of the DFA.

Definition 3.5: Encoding of a DFA

A deterministic finite automaton $A = (Q, \delta, q_0, F)$ can be encoded as a list

$$\ulcorner A \urcorner = \langle \ulcorner q_0 \urcorner, \langle \ulcorner \delta \urcorner, \langle \ulcorner F \urcorner, \mathbf{i} \rangle \rangle \rangle$$

where the components $\ulcorner q_0 \urcorner$, $\ulcorner \delta \urcorner$, and $\ulcorner F \urcorner$ are defined below.

For $\ulcorner q_0 \urcorner$, we simply consider q_0 as an integer and take its Church encoding.

For $\ulcorner \delta \urcorner$, we will consider each state separately and combine the encodings into a list. Recall that $\delta : Q \times \{0, 1\} \to Q$ where $Q = \{0, \ldots, n\}$. Let $a \in Q$ be a state. Then, define a λ-term

$$\delta_a = \langle \ulcorner \delta(a, 1) \urcorner, \ulcorner \delta(a, 0) \urcorner \rangle$$

where the expressions $\ulcorner \delta(a, 1) \urcorner$ and $\ulcorner \delta(a, 0) \urcorner$ refer to the Church encodings of the integers returned by the function δ. We can now define the list $\ulcorner \delta \urcorner$ as

$$\ulcorner \delta \urcorner = \langle \delta_0, \langle \delta_1, \langle ..., \langle \delta_n, \mathbf{i} \rangle \rangle \rangle \rangle$$

For $\ulcorner F \urcorner$, we will construct a list containing a boolean λ-term for each state in order, where $\mathbf{t}$ indicates that the state is accepting and $\mathbf{f}$ indicates that it is not. First, for all integers $i \in Q$, let

$$F_i = \begin{cases} \mathbf{t}, & \text{if } i \in F \\ \mathbf{f}, & \text{if } i \notin F \end{cases}$$

Then, we can define $\ulcorner F \urcorner$ as a list of these elements:

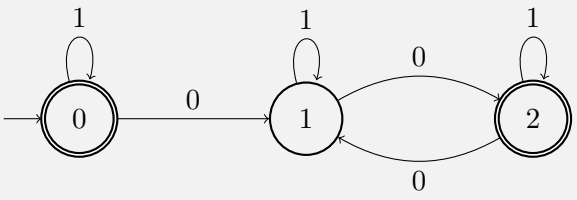
$$\ulcorner F \urcorner = \langle F_0, \langle F_1, \langle ..., \langle F_n, \mathbf{i} \rangle \rangle \rangle \rangle$$

This concludes the definition of the encoding of a DFA A.

Let us conclude this exercise by giving an example of encoding a string and a DFA using this definition.

Example 3.6: Encoding a string and a DFA

Consider the string $s = 001001$ and the following DFA A:



Using our encoding, we get the following:

$$\begin{aligned}
\ulcorner s \urcorner &= \langle \mathbf{f}, \langle \mathbf{f}, \langle \mathbf{t}, \langle \mathbf{f}, \langle \mathbf{f}, \langle \mathbf{t}, \mathbf{i} \rangle \rangle \rangle \rangle \rangle \rangle \\
\ulcorner q_0 \urcorner &= \ulcorner 0 \urcorner = \lambda xy.y \\
\ulcorner \delta \urcorner &= \langle \langle \ulcorner 0 \urcorner, \ulcorner 1 \urcorner \rangle, \langle \langle \ulcorner 1 \urcorner, \ulcorner 2 \urcorner \rangle, \langle \langle \ulcorner 2 \urcorner, \ulcorner 1 \urcorner \rangle, \mathbf{i} \rangle \rangle \rangle \\
\ulcorner F \urcorner &= \langle \mathbf{t}, \langle \mathbf{f}, \langle \mathbf{t}, \mathbf{i} \rangle \rangle \rangle \\
\ulcorner A \urcorner &= \langle \ulcorner q_0 \urcorner, \langle \ulcorner \delta \urcorner, \langle \ulcorner F \urcorner, \mathbf{i} \rangle \rangle \rangle \\
&= \langle \ulcorner 0 \urcorner, \langle \langle \langle \ulcorner 0 \urcorner, \ulcorner 1 \urcorner \rangle, \langle \langle \ulcorner 1 \urcorner, \ulcorner 2 \urcorner \rangle, \langle \langle \ulcorner 2 \urcorner, \ulcorner 1 \urcorner \rangle, \mathbf{i} \rangle \rangle \rangle, \langle \langle \mathbf{t}, \langle \mathbf{f}, \langle \mathbf{t}, \mathbf{i} \rangle \rangle \rangle, \mathbf{i} \rangle \rangle \rangle
\end{aligned}$$

This is equivalent to the follwing λ-calculus code interpreted by `lcalc`, where the terms `PAIR` and `I` are defined in Appendix B.

```
str     := PAIR F (PAIR F (PAIR T (PAIR F (PAIR F (PAIR T I)))))
q0      := 0
delta   := PAIR (PAIR 0 1) (PAIR (PAIR 1 2) (PAIR (PAIR 2 1) I))
final   := PAIR T (PAIR F (PAIR T I))
Autom   := PAIR q0 (PAIR delta (PAIR final I))
```

Part (b)

Find a term t such that

$$\lambda\beta \vdash t \ \ulcorner A \urcorner \ \ulcorner s \urcorner = \textbf{true} \quad \text{if } A \text{ accepts } s$$
$$\lambda\beta \vdash t \ \ulcorner A \urcorner \ \ulcorner s \urcorner = \textbf{false} \quad \text{if } A \text{ does not accept } s$$

We are required to find a term t that simulates executing the DFA and returning its result. We will do this in several steps, building up to the full term t.
First, we will define a term that will allow us to simulate one step of the DFA:

Definition 3.7: The *step* function

The term *step* is defined as

$$step = \lambda q \delta x.(index\, \delta\, q)x$$

Lemma 3.8: Correctness of *step*

Given a state q, the encoded transition function δ, and a symbol x which is either $\mathbf{t}$ or $\mathbf{f}$, the term *step* return a Church-encoded integer representing the next state.

Proof:

First, the term applies *index* to δ and q to get the q^{th} element of the list δ, i.e. the pair $\langle \ulcorner \delta(q,1) \urcorner, \ulcorner \delta(q,0) \urcorner \rangle$ of transitions out from the state q. It then applies this pair to x.
If $x = \mathbf{t}$, then we know that this is an encoding of the symbol 1 and $\langle \ulcorner \delta(q,1) \urcorner, \ulcorner \delta(q,0) \urcorner \rangle \mathbf{t} = \ulcorner \delta(q,1) \urcorner$ by properties of pairs.
Similarly, if $x = \mathbf{f}$, we know that it is an encoding of the symbol 0 and $\langle \ulcorner \delta(q,1) \urcorner, \ulcorner \delta(q,0) \urcorner \rangle \mathbf{f} = \ulcorner \delta(q,0) \urcorner$. □

Next, we want to define a term that will allow us to simulate arbitrarily many steps of the DFA by repeatedly applying *step*.

Definition 3.9: The *run* function

The *run* function is defined as follows, and uses an auxiliary run' function to execute the actual recursive algorithm on the decoded parameters:

$$run = \lambda A s.run'\,(head\, A)\,(head\,(tail\, A))\, s$$
$$run' = \mathbf{y}\,(\lambda f q \delta s.(empty?\, s)\, q\,(f\,(step\, q\, \delta\,(head\, s))\, \delta\,(tail\, s)))$$

Where $\mathbf{y} = \lambda f.(\lambda x.f\,(x\,x))\,(\lambda x.f\,(x\,x))$ is the fixed point combinator seen in lectures.

Lemma 3.10: Correctness of *run*

Given the encoding of a DFA A and an encoded string s, the term *run* return a Church-encoded integer representing the final state of execution after the DFA has processed all of the symbols from s starting at state q_0.

Proof:

The *run* function first decodes the parameters q_0 and δ from the DFA A by taking the first and second elements of the list encoding. It then passes these parameters to run'.
The function *run* uses the Y-combinator to define itself recursively. We can consider the following

syntactically invalid but semantically equivalent function

$$run' = \lambda q \delta s.(empty?\, s)\, q\, (run'\, (step\, q\, \delta\, (head\, s))\, \delta\, (tail\, s))$$

Thus, run' takes in a state q, a transition function δ, and a string s.

If we have exhausted all of the elements in the string, then $s = \mathbf{i}$, and $empty?\, s = \mathbf{t}$. In this case $(empty?\, s)\, q\, (run'\, (step\, q\, \delta\, (head\, s))\, \delta\, (tail\, s)) = \mathbf{t}\, q\, (run'\, (step\, q\, \delta\, (head\, s))\, \delta\, (tail\, s)) = q$. Thus run' will terminate and return the current state q, which is now the final state.

If s still has symbols remaining, then $s = \langle x, s' \rangle$ for some string s' and thus $empty?\, s = \mathbf{f}$. So $(empty?\, s)\, q\, (run'\, (step\, q\, \delta\, (head\, s))\, \delta\, (tail\, s)) = \mathbf{f}\, q\, (run'\, (step\, q\, \delta\, (head\, s))\, \delta\, (tail\, s)) = run'\, (step\, q\, \delta\, (head\, s))\, \delta\, (tail\, s)$.
This will execute *step* defined in Definition 3.7 to get the next available state, and recursively call run' on the next state and the tail of s to advance through the whole string.

In every recursive execution of run', we are reducing the length of the list s and handle the case when s is empty. Hence the algorithm will terminate and thus the terms $run'\, q\, \delta\, s$ and $run\ \ulcorner A \urcorner\ \ulcorner s \urcorner$ are β-normalisable when given valid encodings. □

Finally, we are ready to define the term t demanded by the question. This will work by running the run function and indexing the encoding of the accepting set F with the result.

Definition 3.11: The term t

The term t is defined as follows:

$$t = \lambda A s.index\, (head(tail(tail\, A)))\, (run\, A\, s)$$

Theorem 3.12: Correctness of t

Given a valid encoding of a DFA A and a string s, the term t will return $\mathbf{t}$ if A accepts s and $\mathbf{f}$ otherwise.

Proof:

First, observe that by Lemma 3.10 the term $run\, A\, s$ will reduce to the Church encoding of the final state q after running the DFA over all of the symbols in s. Furthermore, $head(tail(tail\, A)))$ will reduce to the encoding $\ulcorner F \urcorner$ of the set of accepting states.

Now recall that by Definition 3.5 we defined $\ulcorner F \urcorner$ as a list, where the i^{th} element is $\mathbf{t}$ if i is an accepting state, and $\mathbf{f}$ otherwise. Thus it suffices to find the element at the index of the final state q, which is done using the previously defined term *index*, and we are done. □

We may now test our encoding on a few examples of a DFA and input string. Note that the source code is again executed by the `lcalc` λ-calculus interpreter and is available in Appendix B.

Example 3.13: Testing the term t

We will first apply the term t to the automaton encoded in Example 3.6. With the following input

```
t Autom str
```

We get the following output:

```
TRUE
```

Since `lcalc` displays the term $\lambda xy.x$ as `TRUE`, the term t works correctly in this case.

We may also modify the terms slightly. Let us evaluate the above term, but change the definition of `str` to the following:

```
str     := PAIR F (PAIR F (PAIR T (PAIR F (PAIR F (PAIR T (PAIR F I))))))
```

The output is then

```
0
```

Where `0` is the way `lcalc` displays the term $\lambda xy.y$.

Finally, let us change some more properties of the automaton: we evaluate the original string, but we now start at state 1, and only state 1 is an accepting state.

```
str     := PAIR F (PAIR F (PAIR T (PAIR F (PAIR F (PAIR T I)))))
q0      := 1
delta   := PAIR (PAIR 0 1) (PAIR (PAIR 1 2) (PAIR (PAIR 2 1) I))
final   := PAIR F (PAIR T (PAIR F I))
```

By following the DFA diagram in the previous example, we expect the automaton to terminate in state 1 and thus accept the string. Indeed, when evaluating this term we get the following output:

```
TRUE
```

Thus the term works as expected on these examples.

Part (c)

Find a term t_{empty} such that

$$\lambda\beta \vdash t_{empty}\ \ulcorner A \urcorner = \mathbf{true} \quad \text{if } A \text{ does not accept any string}$$
$$\lambda\beta \vdash t_{empty}\ \ulcorner A \urcorner = \mathbf{false} \quad \text{if there is at least one string accepted by} A$$

By automata theory, we know that the language of a DFA A is empty, $L(A) = \varnothing$, if and only if there does not exist a path from the initial state q_0 to any state $q \in F$ when considering the states and transitions δ as a graph.
We will implement a depth-first-search (DFS) algorithm in λ-calculus to traverse the structure of the DFA and return the appropriate result as outlined in the question.
First, to implement a DFS algorithm, we must design a data structure to store which states we have visited and which we have not, and to be able to read and write to this data structure. The simplest way to do this is using a list containing the Church-encoded numerals of the visited states in any order. Then, to append a new state to this list, we can add it to the front of the list. To check whether a state has been visited, we traverse the list and see whether any of the numerals match the one we want to check. This idea is made more formal in the following definition:

Definition 3.14: List of visited states

A list of visited states is any list containing Church numerals corresponding to states in any order.
The terms *add* and *visited*? are defined as follows:

$$\begin{aligned} \mathit{add} &= \lambda qv.\mathit{pair}\, q\, v \\ \mathit{visited}? &= \mathbf{y}(\lambda f qv.(\mathit{empty}?\, v)\, \mathbf{f}\, ((\mathit{equals}?\, q\, (\mathit{head}\, v))\, \mathbf{t}\, (f\, q\, (\mathit{tail}\, v)))) \end{aligned}$$

Where *equals*? was defined in Definition 2.15, $\mathbf{y}$ is the Y-combinator, and *pred* is the predecessor function from the lectures.

Lemma 3.15: Correctness of *add* and *visited*?

When given a state q and list of states v, the *add* function adds the state to the front of the list. When given a state q and list of states v, the *visited*? function checks whether the value q is an element of the list v, returning $\mathbf{t}$ if so and $\mathbf{f}$ otherwise.

Proof:

The term *add* is correct, as prepending an element to a list is equivalent to returning a pair of an element and a list by Definition 3.1.

The term *visited*? works by recursively iterating over a list v. If the list is empty, it returns $\mathbf{f}$, as we know that q was then not a member of the list. If the list is nonempty, it checks whether the first element of the list is equal to q using the term *equal*?, returning $\mathbf{t}$ if so and recursing otherwise. Thus, *visited*? also returns the correct result. □

We can now define the tern t_{empty} using an auxiliary term dfs that will recurse through the states of the DFA using the transition rules and check whether any of the states are reachable.

Definition 3.16: The term t_{empty}

The term t_{empty} and auxiliary term dfs are defined as follows:

$$\begin{aligned} t_{empty} &= \lambda A.dfs\,(head\,A)\,(head(tail\,A))\,(head(tail(tailA))\,\mathbf{i} \\ dfs &= \mathbf{y}(\lambda f q \delta F v.(visited?\,q\,v)\,\mathbf{t}\,((index\,F\,q)\,\mathbf{f}\,(\mathbf{and} \\ &\qquad (f\,(step\,q\,\delta\,\mathbf{t})\,\delta\,F\,(add\,q\,v)) \\ &\qquad (f\,(step\,q\,\delta\,\mathbf{f})\,\delta\,F\,(add\,q\,v)) \\ &\quad))) \end{aligned}$$

where the term $\mathbf{and} = \lambda ab.a\,b\,\mathbf{f}$ is the encoding of the logical AND function in λ-calculus.

Theorem 3.17: Correctness of t_{empty}

Given the encoding of a DFA A, the term t_{empty} will reduce to $\mathbf{t}$ if the language recognised by the DFA is empty and $\mathbf{f}$ otherwise.

Proof:

The term t_{empty} simply splits the encoding of the DFA A into q_0, δ, and F, and passes them to the term dfs along with an empty list $\mathbf{i}$. Thus it suffices to show that dfs is implemented correctly.

The term dfs is a recursive term which implements the following logic: iterate through all possible paths in the DFA, and return true for all paths which loop to an already existing state without passing through an accepting state, and false for the paths that at some point reach an accepting state. We know that if at least one of these paths returns false, then there exists a path to an accepting state and the language of the DFA is nonempty and we return false. Otherwise, if all paths return true, we can return true as no path has reached an accepting state and thus the language is empty.

During each iteration the term checks whether it has already visited the state that it has been passed using the $visited?$ function defined in Definition 3.14. If it has, then it can return $\mathbf{t}$ as the path it has taken did not result in any accepting states before looping back. If it has not, then it checks whether the state is in the set of accepting states using the same method as in Definition 3.11 and returns $\mathbf{f}$ if it is. Otherwise, it recursively calls itself with the next state along both possible paths, adding the current state to the list of visited states for each call, with the results combined together using the **and** operator.

Thus the algorithm implements the logic as desired and t_{empty} works correctly. □

Let us test this term on a few examples.

Example 3.18: Testing the term t_{empty}

Let us again use the example DFA defined in Example 3.6.
Since the language of the DFA is nonempty as shown in Example 3.13, we expect that the below term evaluates to $\lambda xy.y$:

```
tempty Autom
```

And indeed, evaluating this term yields

```
0
```

As expected.

We can now modify the automaton to test the negative case. First, let us modify the set of accepting states to be empty:

```
q0      := 0
delta   := PAIR (PAIR 0 1) (PAIR (PAIR 1 2) (PAIR (PAIR 2 1) I))
final   := PAIR F (PAIR F (PAIR F I))
Autom   := PAIR q0 (PAIR delta (PAIR final I))
```

Evaluating the above term now yields

```
TRUE
```

As expected.

Finally, let us change the term such that there do exist accepting states but they are inaccessible from the starting state:

```
q0      := 0
delta   := PAIR (PAIR 0 1) (PAIR (PAIR 1 0) (PAIR (PAIR 2 1) I))
final   := PAIR F (PAIR F (PAIR T I))
Autom   := PAIR q0 (PAIR delta (PAIR final I))
```

The term `tempty Autom` now also reduces to

```
TRUE
```

Thus the term t_{empty} works correctly on these examples.

Part (d)

Find a term t_{prefix} such that

$$\lambda\beta \vdash t_{prefix}\ \ulcorner A \urcorner\ \ulcorner s \urcorner = \textbf{true} \quad \text{if there is a string } s' \text{ for which } A \text{ accepts } ss'$$
$$\lambda\beta \vdash t_{prefix}\ \ulcorner A \urcorner\ \ulcorner s \urcorner = \textbf{false} \quad \text{if there is no string } s' \text{ for which } A \text{ accepts } ss'$$

By automata theory, we know that for any DFA A and string s, there exists a string s' such that A accepts ss' if and only if the language of the automaton $A' = (Q, \delta, q_1, F)$ is nonempty, where q_1 is the state reached after running s on automaton A.
Thus it suffices to simulate the execution of our original automaton using the term run and then use the term t_{empty} to decide the problem.

Definition 3.19: The term t_{prefix}

The term t_{prefix} is defined as

$$t_{prefix} = \lambda A s.\textbf{not}(t_{empty}\,(pair\,(run\,A\,s)\,(tail\,A)))$$

where **not** is the negation operator from the lectures defined as $\textbf{not} = \lambda x.x\,\textbf{f}\,\textbf{t}$

Theorem 3.20: Correctness of t_{prefix}

Given an encoding of a DFA A and a string s, the term t_{prefix} will reduce to **t** if there is a string s' such that A accepts ss' and **f** otherwise.

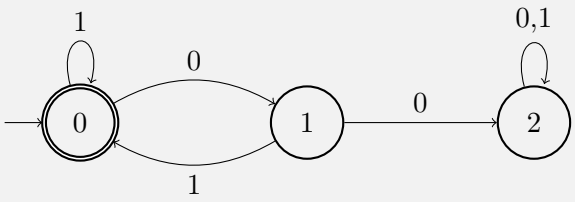
Proof:

The term $run\,A\,s$ reduces to the state q_1 obtained after running the automaton A on s. The pairing function then constructs a new automaton, which is the same as A apart from the initial state q_0, which is now set to q_1. This is then given to the term t_{empty}, which will check whether the resulting automaton has an empty language, and the negation of this result is returned as if the language is empty, there does not exist a string s' and vice versa.

Let us test this term on a slightly different example DFA to before.

Example 3.21: Testing the term t_{prefix}

For this test, we will use the following DFA:



That is, this is a DFA that has a single trap state 2, from where no further accepting states can be reached. This DFA is encoded as follows:

```
q0      := 0
delta   := PAIR (PAIR 0 1) (PAIR (PAIR 0 2) (PAIR (PAIR 2 2) I))
final   := PAIR T (PAIR F (PAIR F I))
Autom   := PAIR q0 (PAIR delta (PAIR final I))
```

We will test this by evaluating the following terms, only the last of which should end up in the trap state:

```
tprefix Autom I
tprefix Autom (PAIR F I)
tprefix Autom (PAIR F (PAIR F I))
```

And indeed, when we evaluate these terms, we get the following β-normal forms:

```
TRUE
TRUE
0
```

Thus the term works as expected on these examples.

Part (e)

Find two terms p and q such that for all DFAs A and strings s, we have the equalities

$$\begin{aligned}\textbf{not}\ (t_{empty}\ \ulcorner A\urcorner) &= t_{prefix}\ \ulcorner A\urcorner\ p\\ \textbf{not}\ (t_{prefix}\ \ulcorner A\urcorner\ \ulcorner s\urcorner) &= t_{empty}(q\ \ulcorner A\urcorner\ \ulcorner s\urcorner)\end{aligned}$$

We can consider the two terms p and q separately, as they do not appear in the same term.
For p, we observe that the language accepted by A is nonempty if and only if there exists a prefix s' for the empty string ε such that A accepts s'. Thus, the encoding $\mathbf{i}$ of the empty string should work for p.
For q, we can use the same observation as we made in part (d): there exists a suffix for a string s which is accepted by A if and only if the language recognised by the automaton $A' = (Q, \delta, q_1, F)$ is nonempty, where q_1 is the state obtained after running A on the input s. Equivalently, there does not exist such a suffix if and only if $L(A) = \varnothing$. Thus the term q should run A on s and make a new automaton with a different starting state, just as we did in Definition 3.19.
Let us prove these relationships formally.

Definition 3.22: The terms p and q

The terms p and q are defined as follows:

$$\begin{aligned}p &= \mathbf{i}\\ q &= \lambda As.pair\ (run\ A\ s)\ (tail\ A)\end{aligned}$$

Theorem 3.23: Correctness of p and q

The terms p and q satisfy the equalities

$$\begin{aligned}\textbf{not}\ (t_{empty}\ \ulcorner A\urcorner) &= t_{prefix}\ \ulcorner A\urcorner\ p\\ \textbf{not}\ (t_{prefix}\ \ulcorner A\urcorner\ \ulcorner s\urcorner) &= t_{empty}(q\ \ulcorner A\urcorner\ \ulcorner s\urcorner)\end{aligned}$$

Proof:

For p, consider the following sequence of β-equalities:

$$\begin{aligned}t_{prefix}\ \ulcorner A\urcorner\ p &= t_{prefix}\ \ulcorner A\urcorner\ \mathbf{i}\\ &= \textbf{not}(t_{empty}\ (pair\ (run\ \ulcorner A\urcorner\ \mathbf{i})\ (tail\ \ulcorner A\urcorner)))\\ &= \textbf{not}(t_{empty}\ (pair\ \ulcorner q_0\urcorner\ (tail\ \ulcorner A\urcorner)))\\ &= \textbf{not}(t_{empty}\ \ulcorner A\urcorner)\end{aligned}$$

For q, simply expand the definition of t_{prefix} to obtain the result:

$$\begin{aligned}&\textbf{not}\ (t_{prefix}\ \ulcorner A\urcorner\ \ulcorner s\urcorner) =\\ &= \textbf{not}\ ((\lambda As.\textbf{not}(t_{empty}\ (pair\ (run\ A\ s)\ (tail\ A))))\ \ulcorner A\urcorner\ \ulcorner s\urcorner)\\ &= \textbf{not}\ (\textbf{not}(t_{empty}\ (pair\ (run\ \ulcorner A\urcorner\ \ulcorner s\urcorner)\ (tail\ \ulcorner A\urcorner)))) =\\ &= t_{empty}\ (pair\ (run\ \ulcorner A\urcorner\ \ulcorner s\urcorner)\ (tail\ \ulcorner A\urcorner)) =\\ &= t_{empty}\ (\lambda As.pair\ (run\ A\ s)\ (tail\ A)\ \ulcorner A\urcorner\ \ulcorner s\urcorner)\\ &= t_{empty}\ (q\ \ulcorner A\urcorner\ \ulcorner s\urcorner)\end{aligned}$$

As a small note, the equality $\mathbf{not}(\mathbf{not}\, x) = x$ holds for all boolean λ-terms x since

$$\mathbf{not}(\mathbf{not}\, \mathbf{t}) = (\lambda x.x\, \mathbf{f}\, \mathbf{t})\, ((\lambda y.y\, \mathbf{f}\, \mathbf{t})\, \mathbf{t}) = (\lambda x.x\, \mathbf{f}\, \mathbf{t})\, \mathbf{f} = \mathbf{t}$$
$$\mathbf{not}(\mathbf{not}\, \mathbf{f}) = (\lambda x.x\, \mathbf{f}\, \mathbf{t})\, ((\lambda y.y\, \mathbf{f}\, \mathbf{t})\, \mathbf{f}) = (\lambda x.x\, \mathbf{f}\, \mathbf{t})\, \mathbf{t} = \mathbf{f}$$

□

Conclusion

In this mini-project, we have proven theoretical results related to λ-calculus, as well as defining functional $\lambda\beta$-terms in practice. We have tested all of the terms that we defined, and formally proved that they work as intended.

One limitation of the testing was that evaluating λ-terms is slow. In practice, evaluation only works on small examples with small integers, as anything larger is too slow to evaluate in a reasonable timeframe. For example, Church numerals are defined in a very inefficient way that makes it easy to define λ-terms, but storing a number such as 10^{20} as required in Question 2 would be impractical in the real world. This meant that tests had to be limited to small cases and were by no means exhaustive. Nevertheless, one benefit of λ-calculus is that it yields itself well to mathematical analysis, and we were able to construct formal proofs of correctness without needing to resort to tests.

Overall, by establishing theoretical results, defining useful terms, and embedding a whole different model of computation (DFAs) in λ-calculus, we have shown its power as a Turing-complete model of computation and the ways in which it allows us to reason differently about programming and algorithms.

Appendix A: Lambda terms for Question 2

```
#import "common"

MOD     := Y (λf.λn.λd.(< n d) n (f (d PRED n) d))
PHI1    := Y (λf.λa.λb.(ISZERO b) a (f b (MOD a b)))

PHI2    := λa.λb.λc.λd.λe.PHI1 a (PHI1 b (PHI1 c (PHI1 d e)))

PRIME   := Y (λf.λk.λn.(ISZERO (PRED k) 1 ((ISZERO (MOD n k)) 0 (f (PRED k) n))))
PHI3    := λn.(ISZERO n) 0 ((ISZERO (PRED n)) 0 (PRIME (PRED n) n))

COMMON  := Y (λf.λk.λn.(ISZERO k) 0 ((ISZERO (MOD n k)) (+ (PHI3 k) (f (PRED k) n))
             (f (PRED k) n)))
PHI4    := λx.λy.COMMON (PHI1 x y) (PHI1 x y)

T       := λx.λy.x
F       := λx.λy.y
NOT     := λx.x F T
EQUALS  := Y (λf.λa.λb.(ISZERO a)(ISZERO b)((ISZERO b) F (f (PRED a) (PRED b))))
EVEN    := Y (λf.λn.(ISZERO n) T (NOT (f (PRED n))))
HALF    := Y (λf.λk.λn.((EQUALS (+ k k) n) k (f (PRED k) n)))
COLLATZ := λn.(EVEN n)(HALF n n)(SUCC (+ n (+ n n)))
PHI5    := Y (λf.λn.(< n 17) ((ISZERO (PRED n)) 0 (SUCC (f (COLLATZ n)))) 0)

PHI6    := Y (λf.λn.λt.(ISZERO n) 0 ((EQUALS (PHI5 n) t)(SUCC (f (PRED n) t))(f (PRED n) t)))
```

Appendix B: Lambda terms for Question 3

```
#import "common"

I       := λx.x
T       := λx.λy.x
F       := λx.λy.y

PAIR    := λx.λy.λf.(f x y)
HEAD    := λx.x T
TAIL    := λx.x F
EMPTY   := λl.l (λx.λy.T) F F
INDEX   := λl.λi.HEAD (i TAIL l)

step    := λq.λd.λx.(INDEX d q) x
runp    := Y (λf.λq.λd.λs.(EMPTY s) q (f (step q d (HEAD s)) d (TAIL s)))
run     := λA.λs.runp (HEAD A) (HEAD (TAIL A)) s
t       := λA.λs.INDEX (HEAD (TAIL (TAIL A))) (run A s)

EQUALS  := Y (λf.λa.λb.(ISZERO a)(ISZERO b)((ISZERO b) F (f (PRED a) (PRED b))))
ADD     := λq.λv.PAIR q v
VISITED := Y (λf.λq.λv.(EMPTY v) F ((EQUALS q (HEAD v)) T (f q (TAIL v))))
AND     := λa.λb.a b F

dfs     := Y (λf.λq.λd.λx.λv.(VISITED q v) T ((INDEX x q) F (AND (f (step q d T) d x
           (ADD q v)) (f (step q d F) d x (ADD q v)))))
tempty  := λA.dfs (HEAD A) (HEAD (TAIL A)) (HEAD (TAIL (TAIL A))) I

NOT     := λx.x F T
tprefix := λA.λs.NOT (tempty (PAIR (run A s) (TAIL A)))
```