1 Introduction

This project seeks to implement the lambda-calculus in the Haskell language.

1.1 How to Run

As this is implemented in Haskell, the language will have to be installed using Chocolatey. Of course, it would be nice to have an easier way to install Haskell but the instructions are clear and concise, and can be found at this link: `https://www.haskell.org/platform/`.

Along with the required packages for the language to run, you will also receive GHCi, which is the compiler. To run a Haskell file, navigate to the directory that the file is in using command prompt. When you arrive at this directory, simply type the following command: `ghci filename.hs`. The functions that are available to use will be described in later sections.

There is also an option of running the code on WinGHCi, a GUI that makes it easier to do so than on the command line. The link for instructions can be found here: `https://ryanjohnson.website/install-guide-for-winghci-2522/`. Note that WinGHCi is only compatible with Windows and may require Visual Studio to be installed.

1.2 Prerequisites

The prerequisites for this project are:

1. Lambda-terms consist of three constructors: variable, lambda and apply. These match the three cases of the definition of a lambda-term: M ::= x | λ.x | M M
2. A function, known as "pretty", that converts terms to lambda-terms by representing λ as a \
3. An example lambda term, λa.λx.(λy.a) x b, is provided for testing. Bear in mind that on the interpreter, it will show as: `\a. \x. (\y. a) x b`, as touched on in prerequisite 2.

2 Church numerals

Denote the following Church numerals:

- N_0 = λf.λx.x
- N_1 = λf.λx. f x
- N_2 = λf.λx. f (f x)

- $N_3 = \lambda f.\lambda x.\ f\ (f\ (f\ x))$

This function is recursive and returns some church numeral for any given number $i \geq 0$. The following example shows an input and output of this function:

```
*ghci> numeral 2
\f. \x. f (f x)
```

3 Variables

3.1 Computing an infinite list of variables

This function computes an infinite list of alphabets, with repetitions suffixed by a number (for example, the first occurrence of the letter f would simply show as f, the second would show as f1, the third as f2, and so on). To find specific variables in the list, simply enter its index (any positive integer).

The following example shows how to use this function and the output returned:

```
*ghci> [variables !! i | i <- [0,1,25,26,27,100,3039]]
["a","b","z","a1","b1","w3","x116"]
```

3.2 Filtering a list of variables

This function takes two lists as input. The first list is the list that is being filtered. The second list defines what is to be removed from the first list (if present, of course!). The return result is the first list with all variables from the second list removed from it. Any variables that are in the second list but not the first are simply ignored.

```
*ghci> filterVariables ["y","z","a1","a2"] ["y","a1","a3"]
["z","a2"]
```

3.3 Generate a new variable

Given a list of variables, the `fresh` function finds the first variable alphabetically that doesn't occur in the list. Using the infinite list created in variables, this list is then filtered with the list given and the first variable in the newly filtered infinite list is returned.

```
*ghci> fresh ["a","b","x"]
["c"]
```

3.4 Ordering variables in a term

Given a term, the function `used` takes every variable name and places them in an alphabetically-ordered list. These variable names can come from either free variables or those or bound to a lambda in an abstraction.

```
*ghci> used example
["a","b","x","y"]
```

It is even possible to generate a new variable from the returned list, using the `fresh` function.

```
*ghci> used example
["a","b","x","y"]
*ghci> fresh it
["c"]
```

4 Capture-avoiding substitution

Now we dive into the key operations present in the lambda-calculus. We start with capture-avoiding substitution. This is a method that (obviously) changes the names of variables in a term. However, there's a catch; before substitution, a new variable name is picked and checked whether it already exists in a term. If the name already exists, there is a good chance that a variable which was originally free (i.e., not bound to a lambda) could be captured, or bound. Therefore, it is checked again. There are two functions that are used to implement this.

4.1 Renaming a variable

The first function, unimaginatively named `rename`, takes three arguments: `x`, `y` and `m`. `x` is the variable name you want to replace, `y` is the variable name that you want to insert, and `m` is the term that you want the renaming to occur in. Hence, the structure of the function is as follows: `rename x y m`. For example:

```
*ghci> rename "b" "z" example
\a. \x. (\y. a) x z
```

4.2 Avoiding variable capture

Now that renaming is established, it's pivotal to avoid variable capture for reasons explained in the introduction to Section 4. This is implemented in a function `substitute`, with the same arguments as that of `rename` (i.e., the format is `substitute x y m`). This function also calls `fresh`, as both `y` and `m` have to be checked for the existence of `x`, to make sure that:

1. `x` already doesn't exist somewhere
2. `y` and/or `m` aren't just simply the variable `x`

The following example shows a substitution in the example term, where Church numeral 1 comes in place of the variable name `b`.

```
*ghci> substitute "b" (numeral 1) example
\c. \a. (\a. c) a (\f. \x. f x)
```

5 Beta reduction

Now that one key operation, capture-avoiding substitution, has been established, it's now time to implement beta reduction. Beta-reduction is where given a term of the form $(\lambda x.M)N$, the following step is carried out an arbitrary number of times: $(\lambda x.M)N \rightarrow_\beta M[N/x]$. M and N are lambda-terms, and the end result $M[N/x]$ is simply the term M, with N replacing the variable name x. Beta reduction is the most important operation in the lambda calculus. The reduction step is carried out until the term is in normal form (i.e., not of the format $M[N/x]$).

5.1 Beta stepping

The function `beta` implements one reduction step at a time. This is done by looking for a redex (also known as a reducible expression). This is simply a term of the form $(\lambda x.M)N$. Consider a term $(\lambda x.xxx)(\lambda f.x.fx)$. This term has exactly one redex, and thus can be reduced. This is the same term but with the redex underlined: $\underline{(\lambda x.xxx)(\lambda f.x.fx)}$. Applying a beta reduction step gives the following term: $(\lambda f.x.fx)(\lambda f.x.fx)(\lambda f.x.fx)$. There are two redexes in this term, which can be noted the following ways:

1. $\underline{(\lambda f.x.fx)(\lambda f.x.fx)}(\lambda f.x.fx)$
2. $(\lambda f.x.fx)\underline{(\lambda f.x.fx)(\lambda f.x.fx)}$

The `beta` function will reduce the first redex. However, there also needs to be consideration in the case of nested redexes. This is taken into account and will be explained in due course.

```
*ghci> Apply example (numeral 1)
(\a. \x. (\y. a) x b)(\f. \x. f x)
*ghci> beta it
[\c. (\b. \f. \x. f x) c b, (\a. \x. a b) (\f. \x. f x)]
*ghci> it !! 1
(\a. \x. a b)(\f. \x. f x)
*ghci> beta it
[\c. (\b. \f. \x. f x) b]
*ghci> beta (head it)
[\c. \a. b a]
*ghci> beta (head it)
[]
```

Note a: the example code listing contains the operator !!, this simply makes the previous term into a list so that the first half of the term is hidden since it's in normal form

Note b: the term `head` *appears in the last two lines of the listing.* `head` *simply refers to the first item in the list.*

5.2 Reducing a term to normal form

An all-at-once version of the `beta` function, the function `normalize` performs all reduction steps in one call. There are some interesting cases, as some terms can cause infinite reduction (i.e., never reach normal form). There are four steps that occur in `normalize` in the following order:

1. Output the current term
2. Apply a reduction step (in this case, a call of `beta`)
3. Return an empty output if there are no feasible reduction steps
4. If there is a feasible reduction step, take the result of the first step and recursively call `beta` until normal form is reached

The `head` command is not needed for any cases in `normalize`, despite the function being mathematically similar to `beta`.

```
*ghci> normalize (Apply (numeral 2)(numeral 2))
(\f. \x. f (f x))(\f. \x. f (f x))
\a. (\f. \x. f (f x)) ((\f. \x. f (f x)) a)
\a. \b. (\f. \x. f (f x)) a ((\f. \x. f (f x)) a b)
\a. \b. (\b. a (a b)) ((\f. \x. f (f x)) a b)
\a. \b. a(a ((\f. \x. f (f x)) a b))
\a. \b. a(a ((\b. a (a b)) b))
\a. \b. a (a (a (a b)))
```

5.3 Applicative-order reduction

There are two reduction orders: normal order and applicative order. While both are intended to achieve normal form, there are two key differences as well as advantages and disadvantages to each.

- What are the alternative names for these reduction orders?
  - Normal order reduction is known as leftmost outermost reduction
  - Applicative order reduction is known as leftmost innermost reduction
- How do these orders interpret a redex $(\lambda x.M)N$?
  - Normal order reduction reduces a redex before any reductions in M and N
  - Applicative order reduction reduces a redex after any reductions in M and N
- What are the advantages and disadvantages of these reduction orders?
  - Normal order reduction guarantees to reach normal form but this comes at the cost of duplicating work as much as possible
  - Applicative order reduction doesn't duplicate work but it doesn't always reach normal form

5.3.1 Example

Here is an example to aid the understanding of this. Consider the following term: $(\lambda x.xxx)(\lambda f.\lambda x.fx)$. The first reduction step is straightforward, and gives:

$$\underline{(\lambda f.\lambda x.fx)}(\lambda f.\lambda x.fx)(\lambda f.\lambda x.fx)$$

The redex has been underlined in black. However, there is actually a second feasible redex which has been highlighted in blue. However, since there are no nested redexes, normal order reduction is followed. Hence, this redex doesn't need to be reduced yet. Now, reducing this term (with the redex underlined) gives the following term:

$$\underline{(\lambda x.(\lambda f.\lambda x.fx)x)(\lambda f.\lambda x.fx)}$$

Now, there are two routes that we can take to reduce this as shown. The redex underlined will follow normal order reduction, as shown in the following steps with redexes underlined.

$$\begin{aligned}&\underline{(\lambda x.(\lambda f.\lambda x.fx)x)(\lambda f.\lambda x.fx)}\\ \rightarrow_\beta\ &\underline{(\lambda f.\lambda x.fx)(\lambda f.\lambda x.fx)}\\ \rightarrow_\beta\ &\lambda x.\underline{(\lambda f.\lambda x.fx)x}\\ \rightarrow_\beta\ &\lambda x.\lambda y.xy\end{aligned}$$

The second reduction route follows from the redex highlighted in blue, which is of course nested. This will follow applicative order reduction, as shown in the following steps with redexes highlighted.

$$\begin{aligned}&(\lambda x.(\lambda f.\lambda x.fx)x)(\lambda f.\lambda x.fx)\\ \rightarrow_\beta\ &(\lambda x.\lambda y.xy)(\lambda f.\lambda x.fx)\\ \rightarrow_\beta\ &\lambda y.(\lambda f.\lambda x.fx)y\\ \rightarrow_\beta\ &\lambda y.\lambda x.yx\end{aligned}$$

Of course, both reduction orders take the same number of steps to reach normal form. However, the difference in the first steps should be clear enough to show how these orders differ.

5.3.2 Implementations

Now that these differences have been established, it is time to touch on two new functions. The first, `a_beta`, is the applicative order equivalent to the function `beta`, which of course performs beta steps using normal order reduction. The second, `a_normalize`, is the applicative order equivalent to the function `normalize`. `a_normalize` still performs all steps in a single call, but using applicative order reduction.

```
*ghci> a_normalize (Apply (numeral 2)(numeral 2))
(\f. \x. f (f x))(\f. \x. f (f x))
\a. (\f. \x. f (f x)) ((\f. \x. f (f x)) a)
\a. (\f. \x. f (f x)) (\b. a (a b))
\a. \c. (\b. a (a b)) ((\b. a (a b)) c)
\a. \c. (\b. a (a b)) (a (a c))
\a. \c. a (a (a (a c)))
```

Note that in the example provided of `a_normalize`, the reduction takes less steps than `normalize` with the same term. This is because in the second step, the leftmost innermost redex is reduced in `a_normalize`. However, this doesn't mean that applicative order reduction is always quicker. Two example terms have been provided to show this.

The first, `example1`, is the application of a Church numeral 2 to a Church numeral 2, which is actually the term shown in the examples for the two normalization functions. The second, `example2`, is the application of a Church numeral 2 to a Church numeral 0. For the term `example2`, normal order reduction actually takes less steps to reduce to normal form.

```
*ghci> normalize example2
(\f. \x. f (f x)) (\f. \x. x)
\a. (\f. \x. x) ((\f. \x. x) a)
\a. \b. b
```

6 Outro

This document has provided the instructions to the Lambda Calculus interpreter, implemented in Haskell. If there are any enquiries, then please highlight them on Github. There is also another Haskell implementation, this time of the Krivine Abstract Machine (KAM), a separate project that does make use of some of the functions present in this interpreter. The link to KAM can be found here: `https://github.com/manvirdx/Abstract-Machine`