LambdaLaTeX: Implementing the Lambda Calculus in LaTeX

1 Introduction

In this question, you will build up an implementation of the lambda calculus using only built-in LaTeX (macro) features. Keep in mind that a lot of the lambda expressions were given to you in the lecture, and it would be a good idea to refer to it periodically. Before you begin, here's a guide to how LaTeXmacros work. To create a new macro called "foobar", you write the following line of LaTeX code:

```
\newcommand{\foobar}[X]{WHAT_YOU_WANT_IT_TO_DO}
```

`\foobar` is the name of the command; `X` is the number of arguments it takes (this parameter is optional and you can omit the square brackets entirely); `WHAT_YOU_WANT_IT_TO_DO` is the body of the macro (the part where you actually tell it what to do. To reference arguments in your function use `#1`, `#2`, etc.
You will want to follow along in the tex source as well as the pdf, because you will implement macros in this tex file.
The first function we will define is the identity function **id**:

$$\textbf{id} \equiv \lambda x.x$$

id was defined here

Please put all definitions that you write in this file and expect to be graded between the `begintodo` and `finishtodo` macros. These will be used to slurp pieces of your file out for grading. Please don't delete them or you risk getting a zero on this problem.

tests for the function(s) **id** here
$10 = 10$
$1337 = 1337$

Most functions will have tests pre-written for you (like the above). They will start commented out so that the tex file compiles, but you should feel free to uncomment them to test your code.
The last piece of LaTeX syntax that we need to describe is macro (function) application. In the lambda calculus, if we write $\lambda x.\lambda y.(yx)$, we take the inner part to mean "give x to y as an argument" (in other words, apply x to y). An analogous LaTeX macro would be defined as follows:

```
\newcommand{\func}[2]{#2{#1}}
```

There are two important things to note about this implementation of the lambda expression. The first is that by saying the macro takes two arguments, we are mirroring that the lambda expression is abstracted twice (intuitively, there are two arguments). The second is that in LaTeX, `#2 #1` would **NOT** be function application. To apply a function in LaTeX, you **MUST** use curly braces. Try this out by defining the pairing function as given in lecture:

$$\textbf{pair} \equiv \lambda x.\lambda y.\lambda f.fxy$$

define the function **pair** here

2 Booleans

To really get started, we need a couple more functions which we define for you:

$$\textbf{true} \equiv \lambda x.\lambda y.x$$

true was defined here

$$\textbf{false} \equiv \lambda x.\lambda y.y$$

false was defined here

tests for the function(s) **booleans** here
$20 = 20$
$10 = 10$

If we're going to write some real functions, we're probably going to need the normal boolean operations: **and**, **or**, **not**. Define these functions here:

define the function **and** here

define the function **or** here

define the function **not** here

tests for the function(s) **boolean functions** here
$0 = 0$
$0 = 0$
$1 = 1$
$1 = 1$
$0 = 0$
$1 = 1$
$0 = 0$
$1 = 1$

One more thing we'll need is the "if-then-else" operator we covered in lecture.

define the function **ifthenelse** here

tests for the function(s) **ifthenelse** here
$1 = 1$
$2 = 2$

3 Lists

Now that we have basic booleans, let's continue making useful constructs to get to pairs and lists. If we have a pair, we'd like to be able to get the first and second elements out of the pair. We use **first** and **second** for this:

$$\textbf{first} \equiv \lambda p.p(\lambda x.\lambda y.x)$$

first was defined here

$$\textbf{second} \equiv \lambda p.p(\lambda x.\lambda y.y)$$

second was defined here

Now that we have pairs, we'd like to represent entire lists. We use cons cells like in the lecture. We represent the list $[1, 2, 3]$ as (1 (2 (3 `nil`))). We can construct lists by repeated applications of `cons`. To get the above list, we use

`cons`{1}{`cons`{2}{`cons`{3}{`nil`}}}

nil is easy enough to implement:

$$\textbf{nil} \equiv \textbf{false}$$

nil was defined here

$$\textbf{cons} \equiv \textbf{pair}$$

cons was defined here

To finish off our list definitions, we define **car**, which gets the first element of the list, and **cdr** which gets everything but the first element of the list. Here they are:

car was defined here

cdr was defined here

We would like to be able to tell if a list is empty; so, we define a macro **isempty** which returns **true** if the list is empty and **false** otherwise. Implement this function here (Hint: You'll want to use a helper function).

define the function **isempty** here

4 Recursion

As we discussed in lecture, we can use a fixed-point combinator to create recursive functions in the lambda calculus. We present two of them for you here:

fix was defined here

Y was defined here

Throughout the rest of this question, you'll probably want to use the Y-combinator, but you could exchange it for the other one if you wanted to!
Now let's use the Y-combinator to actually create a recursive function! **listId** takes in a list as an argument, recurses over every element in the list, and returns the list unchanged.

listId was defined here

To get a feel for how this type of recursion works, define the following function: **printlist** which prints out the list in a readable way. For example, if the given list is `cons{1}{cons{2}{cons{3}{nil}}}`, then **printlist** should output

(1 (2 (3 nil)))

define the function **printlist** here

tests for the function(s) **printlist** here

nil = nil
(1 nil) = (1 nil)
(1 (2 nil)) = (1 (2 nil))
(1 (2 (3 nil))) = (1 (2 (3 nil)))

5 Church Numerals

As in lecture, we'd like a way of representing numbers; so, we turn to Church Numerals. We'll define zero for you:

z was defined here

You define successor:

define the function **s** here

...and plus:

define the function **plus** here

Just like with lists, we need a way of testing for the base case. Please define **iszero**:

define the function **iszero** here

Here's predecessor (which is still really annoying):

p was defined here

We provide two ways of printing church numerals (via unary and via LaTeX's counters):

unary was defined here

printnum was defined here

tests for the function(s) **church numerals** here
$4 = 4$
$111 = 111$

6 Putting It All Together

Implement an **append** function which takes in two arguments x and L, and appends x to the list L.

define the function **append** here

Implement a **reverse** function which takes in a list L and returns a new list with the elements of L in reverse order.

define the function **reverse** here

Finally, implement a function **range** which takes in a church numeral which represents the number n and outputs the list `cons{1}{cons{2}{...cons{n}{nil}...}}`:

define the function **range** here

tests for the function(s) **range** here
(1 (2 (3 (4 nil)))) = (1 (2 (3 (4 nil))))