Homework 2: Lambda Calculus

Due date: October 20th (Thursday) at 11:59pm

Overview

This homework consists of two parts:

1. Implement a translator that compiles lambda calculus into SKI calculus.
2. Write more interesting programs in lambda calculus.

Some important notes are as follows:

- You will need to implement the two parts in order, as the second step depends on the first one.
- The homework was tested on the myth cluster (`myth.stanford.edu`), which has Python 3.8.10. If you have any issues with Python, you can use the cluster to do the assignment. You are free to develop your solution anywhere, but you should make sure that your solution works on the myth cluster, since that is where we will grade the assignment.

Part 1: Translation from lambda calculus to SKI calculus

In the first part of this assignment, you will write a translator that compiles an expression in lambda calculus to an expression in SKI calculus. Recall that an expression $\hat{e}$ in lambda calculus is defined as follows.

$$\hat{e} \ ::= \ \hat{x} \mid \lambda \hat{x}.\, \hat{e} \mid \hat{e}\ \hat{e}$$

Here we put hats over symbols to differentiate expressions in lambda calculus from those in SKI calculus (which is denoted by e without any hat). An expression $\hat{e}$ in lambda calculus is either a variable (denoted by $\hat{x}$), a lambda abstraction (denoted by $\lambda \hat{x}.\, \hat{e}$), or an application (denoted by $\hat{e}_1\ \hat{e}_2$).

The translation from lambda calculus to SKI calculus can be formalized with the translation function $\mathcal{T}$ which is defined recursively:

$$\begin{aligned} \mathcal{T}(\hat{x}) &= x, \\ \mathcal{T}(\lambda \hat{x}.\, \hat{e}) &= \mathcal{U}(x, \mathcal{T}(\hat{e})), \\ \mathcal{T}(\hat{e}_1\ \hat{e}_2) &= \mathcal{T}(\hat{e}_1)\ \ \mathcal{T}(\hat{e}_2). \end{aligned}$$

The translation function $\mathcal{T}$ receives an expression $\hat{e}$ in *lambda calculus*, and returns an expression $\mathcal{T}(\hat{e})$ in *SKI calculus* that is (functionally) equivalent to $\hat{e}$. The auxiliary function $\mathcal{U}$ is defined recursively as follows:

$$\begin{aligned} \mathcal{U}(x, S) &= K\ S, & \mathcal{U}(x, x) &= I, \\ \mathcal{U}(x, K) &= K\ K, & \mathcal{U}(x, y) &= K\ y \quad \text{if } y \text{ is a variable other than } x, \\ \mathcal{U}(x, I) &= K\ I, & \mathcal{U}(x, e_1\ e_2) &= S\ \ \mathcal{U}(x, e_1)\ \ \mathcal{U}(x, e_2). \end{aligned}$$

The function $\mathcal{U}$ takes a variable x and an expression e both in *SKI calculus*, and returns an expression $\mathcal{U}(x, e)$ in *SKI calculus* that is (functionally) equivalent to the lambda abstraction $\lambda x.\, e$.[1] You can check

[1] Technically, $\lambda x.\, e$ is not an expression in lambda calculus since e is an expression in SKI calculus (and not in lambda calculus), but we can intuitively understand $\lambda x.\, e$ as a lambda abstraction.

why the above definition of $\mathcal{U}(x,e)$ makes sense by confirming that $(\mathcal{U}(x,e)\ x)$ and $((\lambda x.\, e)\ x)$ evaluate to the same expression. For instance, for any variable y different from x, we have $(\mathcal{U}(x,y)\ x) = (K\ y\ x) \to y$ and $((\lambda x.\, y)\ x) \to (y[x{:=}x]) = y$; that is, $(\mathcal{U}(x,y)\ x)$ and $((\lambda x.\, y)\ x)$ evaluate to the same expression y.

To give you a better understanding of the translation function $\mathcal{T}$, we provide two examples:

$$\begin{aligned}&\mathcal{T}(\lambda\hat{x}.\,\lambda\hat{y}.\,\hat{y})\\ &=\mathcal{U}(x,\mathcal{T}(\lambda\hat{y}.\,\hat{y}))\\ &=\mathcal{U}(x,\mathcal{U}(y,\mathcal{T}(\hat{y})))\\ &=\mathcal{U}(x,\mathcal{U}(y,y))\\ &=\mathcal{U}(x,I)\\ &=K\ I,\end{aligned}\qquad\qquad \begin{aligned}&\mathcal{T}(\lambda\hat{x}.\,\lambda\hat{y}.\,\hat{x})\\ &=\mathcal{U}(x,\mathcal{T}(\lambda\hat{y}.\,\hat{x}))\\ &=\mathcal{U}(x,\mathcal{U}(y,\mathcal{T}(\hat{x})))\\ &=\mathcal{U}(x,\mathcal{U}(y,x))\\ &=\mathcal{U}(x,K\ x)\\ &=S\ \ \mathcal{U}(x,K)\ \ \mathcal{U}(x,x)\\ &=S\ (K\ K)\ I.\end{aligned}$$

Your task is to implement the translation function $\mathcal{T}$. Detailed instructions are as follows:

- **This assignment requires a working SKI interpreter**. Copy your implementation of the SKI interpreter from HW1 into `ski_eval.py`. See the last bullet point below if you weren't able to complete HW1.
- **Implement the translation function `tran` in `lam_tran.py`.** Given an expression $\hat{e}$ in lambda calculus, the function `tran` returns the expression $\mathcal{T}(\hat{e})$ in SKI calculus. For your reference, the CAs' implementation has fewer than 30 lines of code.
- To implement the translator, you may need to refer to `src/lam.py` which gives the definition of lambda expressions $\hat{e}$. In our Python framework, a variable $\hat{x}$ is represented as `Var("x")`, a lambda abstraction $\lambda\hat{x}.\,\hat{e}$ as `Lam("x",e)`, and an application $\hat{e}_1\ \hat{e}_2$ as `App(e1,e2)`.
- You can test your translator by running `python3 src/main_lam.py test/test.lam`, where `test.lam` stores lambda expressions to be translated by your translator. To see how to write lambda expressions in `test.lam`, please refer to Part 2.
  - If you weren't able to complete the SKI interpreter from HW1, you can run this function with the flag `--soln_ski_eval`, which will run with a working interpreter. For example: `python3 src/main_lam.py test/test.lam --soln_ski_eval`. **NOTE: This will only work on the Myth machines.**

Part 2: Programming in lambda calculus

In the next part of this assignment, you will write several programs in lambda calculus. Expressions in lambda calculus can be written in `*.lam` files in the exactly same way you wrote `*.ski` files, except the following: a lambda abstraction $\lambda\hat{x}.\hat{e}$ can now be written in `*.lam` as `\x.e`, while `S`, `K`, and `I` are no longer special symbols in `*.lam`; and any name defined in the definition block should not be used as a variable of any lambda abstraction (i.e., $\hat{x}$ in $\lambda\hat{x}.\hat{e}$ should be different from any defined names). Please refer to `test/test.lam` for more examples of how to write lambda expressions in `*.lam`.

You can evaluate your `lam` file, say `test/test.lam`, by running the command `python3 src/main_lam.py test/test.lam`. This command translates all expressions in `test.lam` into SKI expressions using your translator (implemented in Part 1), evaluates those SKI expressions using your interpreter (implemented in HW1), and finally print the results (which are SKI expressions).

Your task is to write lambda expressions that implement several functions. For this task, we assume as before that all booleans (`tt`, `ff`), numerals (`_0`, `inc`), pairs (`pair`, `fst`, `snd`), and lists (`nil`, `cons`) are encoded as lambda expressions in the manner described in lecture. We provide the definitions of `tt`, $\cdots$, `cons` at the top of `problem.lam`. Detailed instructions are as follows:

- **Write the implementation of the following six functions in `problem.lam`.** In the following specification of these functions, `n` and `m` denote arbitrary numerals encoding the integers n and m, and `l` denotes an arbitrary list that encodes the list of $k\,(\geq 0)$ elements, `a1`, $\cdots$, `ak`.

- Finding an appropriate way of applying primitive recursion (of numerals or of lists) is the key to implementing most of the following functions. To implement `fib`, `dec`, and `half` in particular, making use of pairs could be helpful.
- It may take a few seconds to evaluate some expressions in the test part of `problem.lam`. This is mainly because the translation from lambda calculus to SKI calculus (implemented in Part 1) can substantially increase the size of an expression.. The CAs' solution takes about 20 seconds to run `python3 src/main_lam.py problem.lam`.
- `is_zero`: `(is_zero n)` evaluates to true (i.e., a lambda expression that encodes true) if $n = 0$, and to false otherwise.
- `len`: `(len l)` evaluates to k.
- `num_zero`: Assume that `a1`, $\cdots$, `ak` encode the integers $a_1, \cdots, a_k$. Then, `(num_zero l)` evaluates to the number of zeros in $a_1, \cdots, a_k$ if $k \geq 1$, and to 0 otherwise.
- `fib`: `(fib n)` evaluates to F_n, where F_n is defined by $F_0 = 1$, $F_1 = 2$, and $F_{i+2} = F_i + F_{i+1}$ for all $i \geq 0$.
- `dec`: `(dec n)` evaluates to $n - 1$ if $n \geq 1$, and to 0 otherwise.
- `half`: `(half n)` evaluates to $\lfloor n/2 \rfloor$, where $\lfloor x \rfloor$ denotes the largest integer that is less than or equal to x.

Submitting your work

- Please make sure you have edited `README.txt` to include your student ID number (the last 8-digit number).
- **Generate a tarball file `solution.tar.gz` by running `python3 src/submit.py` and upload the tarball file to Canvas.** Please make sure that the tarball file contains `lam_tran.py`, `problem.lam`, and `README.txt`, and that the script gives you no errors or warnings.