Homework 1: SKI Calculus

Due date: October 13th (Thursday) at 11:59pm

Overview

This homework consists of three parts:

1. Implement an interpreter that evaluates expressions in an extended SKI calculus.
2. Write simple programs in SKI calculus.
3. Write simple programs in NumPy, a real-world array programming language.

Some important notes are as follows:

- You will need to implement the first two parts in order, as the second step depends on the first one. For this reason, you should also start the homework early.
- The homework was tested on the myth cluster (`myth.stanford.edu`), which has Python 3.8.10 and NumPy 1.19.5. 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: Interpreter for SKI calculus

In the first part of this assignment, you will implement an interpreter for an extended SKI calculus. An extended SKI expression (or term) e is defined as follows.

$$e \ ::= \ S \mid K \mid I \mid x \mid e\, e$$

This definition states that an SKI expression e is either one of the S, K, and I combinators, a variable (denoted by x), or an application (denoted by $e_1\ e_2$). For example, $S\ (K\ x)\ I\ (y\ I)$ is a valid extended SKI expression, where x and y are two different variables. Next, recall that the rewrite rules for an SKI expression e are:

$$I\ e \to e, \qquad K\ e_1\ e_2 \to e_1, \qquad S\ e_1\ e_2\ e_3 \to (e_1\ e_3)\ (e_2\ e_3).$$

The three rules do not say anything about variables, which indicates that variables cannot be rewritten in any way. For example, we have the following rewrite sequence where the last expression cannot be rewritten further:

$$S\ (K\ x)\ I\ (y\ I) \to (K\ x\ (y\ I))\ (I\ (y\ I)) \to x\ (I\ (y\ I)) \to x\ (y\ I).$$

Your first task is to implement an interpreter for the extended SKI calculus which, given an SKI expression e, applies the rewrite rules presented above until no further rules are applicable, and then returns the final SKI expression e'. Formally, your interpreter should take an SKI expression e and return an SKI expression e' that satisfies the following:

$$e \to^* e' \qquad \text{and} \qquad e' \text{ cannot be rewritten further.} \tag{1}$$

Detailed instructions are as follows:

- **Implement the interpreter function `eval` in `ski_eval.py`.** Given an SKI expression e, the function `eval` should return an SKI expression e' that satisfies Property (1). The CAs' implementation has fewer than 30 lines of code. All of the SKI programs you will write or generate in subsequent parts of this homework can be written using primitive recursion, so you should not be overly concerned about whether your interpreter uses a reduction strategy that guarantees termination (if termination is possible).
- To implement the interpreter, you may need to refer to `src/ski.py` which gives the definition of SKI expressions e. In our Python framework, the combinators S, K, and I are represented as `S()`, `K()`, and `I()`, a variable x as `Var("x")`, and an application $e_1\ e_2$ as `App(e1,e2)`.
- To check whether an expression `e` is, for example, the K combinator, you can use `isinstance(e, K)` which returns `True` if `e` is indeed the K combinator, and `False` otherwise.
- You can test your interpreter by running `python3 src/main_ski.py test/test.ski`, where `test.ski` stores SKI expressions to be evaluated by your interpreter. To see how to write SKI expressions in `test.ski`, please refer to Part 2.

Part 2: Programming in SKI calculus

In the second part of this assignment, you will write a few simple programs in (extended) SKI calculus. To see how you can write SKI expressions in `*.ski` files, let's take a look at `test/test.ski`:

```
def inc = S (S (K S) K);
def _0  = S K;
def _1  = inc _0;
_0 f x;
_1 f x;
(inc _1) f x;
```

Line 1 declares that every occurrence of the name `inc` in the following lines (e.g., lines 3 and 6) is replaced by `(S (S (K S) K))`. Lines 2–3 similarly define the names `_0` and `_1`. Line 4 specifies that your interpreter evaluates the expression `_0 f x` and prints the result. Since `f` and `x` were not defined in the previous lines, they are considered variables in the expression `_0 f x`. Lines 5–6 give two more expressions to be evaluated and printed out. Assuming that you have successfully completed Part 1, running `python3 src/main_ski.py test/test.ski` should print the following:

```
x
(f x)
(f (f x))
```

Here are a few rules concerning the syntax of `*.ski` files:

- The definition block (which lists the definition of names) should come before the expression block (which lists expressions to be evaluated).
- If a name is defined, the definition must come before any uses. (This condition also implies there are no recursive definitions.)
- Any name and variable should consist solely of lower-case letters, digits, and underscore, and cannot start with a digit.

Your second task is to write SKI expressions that implement some simple functions. For this task, we assume that booleans (`tt`, `ff`) and numerals (`_0`, `inc`) are encoded in SKI as described in lecture. We provide the definitions of `tt`, ..., `inc` at the top of `problem.ski`. Detailed instructions are as follows:

Write the implementation of the following four functions in `problem.ski`.

- `or`, `and`, `not`: These functions are the prefix-version of the standard boolean functions. For example, for booleans `b1` and `b2`, `(and b1 b2)` should evaluate to true (i.e., an SKI expression that encodes true) if `b1` and `b2` both encode true, and to false otherwise. Note that the resulting expression doesn't need to be exactly equal to the given definition of `tt` – evaluating to true just means `(and b1 b2) x y` $\rightarrow^*$ `x`.
- `is_odd`: For any numeral `n`, `(is_odd n)` should evaluate to true if `n` encodes an odd integer, and to false otherwise.

Hints:

- For each function, you may find it easiest to first define it with function arguments (e.g., `swap x y = y x`) and then using the abstraction algorithm from lecture 2 to convert the function into a pure combinator.
- There are many possible ways to write each function. Each function can be implemented in fewer than 8 combinators, but any correct solution will be accepted.

Part 3: Array Programming with NumPy

As discussed in lecture, combinator calculi such as SKI have influenced the design of languages that emphasize using primitive recursive combinators in "whole data type" operations. The most prominent examples today are array programming languages, particularly NumPy.

In this section of the assignment, you will solve some short programming exercises that utilize NumPy's array combinators. Usage of these combinators enables concise programming and efficient implementations of the combinators yield efficient complete programs.[1] The NumPy reference, available at `https://numpy.org/doc/1.19/reference/index.html`, documents its very large number of array operations.

Restrictions. The goal of these exercises is to use the NumPy combinators instead of writing Python code that loops over NumPy arrays. Thus, your solutions for the following functions must only:

- Call NumPy array functions (i.e., functions imported from the `numpy` module or member functions of NumPy arrays).
- Slice, create, and modify NumPy arrays.
- Declare and reference variables.

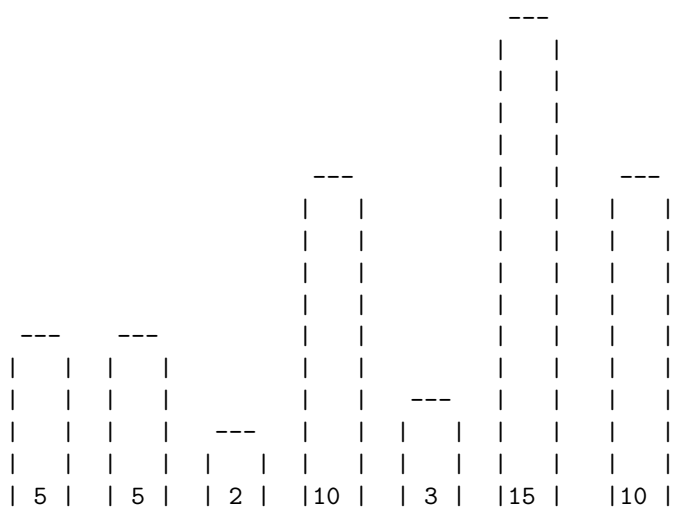
In particular, your solutions must *not* use control flow constructs such as `while` and `for` loops, nor the built-in Python `map()` and `filter()` functions.

Write the implementation of the following three functions in `numpy_problems.py`. We have provided some test cases in `test/test_numpy.py`, which you can use by running `python3 test/test_numpy.py`. Feel free to add more to the same file.

Problem 3.1 (Find-Missing): Given a positive integer n and a sorted array S containing a subset of the range $[0, n)$, return a sorted array containing the missing integers from the range $[0, n)$. For example, given $n = 6, S = [0, 2, 5]$, `find_missing`$(n, S) = [1, 3, 4]$. You may assume that S contains no duplicate entries. The CAs' solution is 1 line of code, using 3 combinators.

[1] Accelerated (CuPy `https://github.com/cupy/cupy/`) and distributed (cuNumeric `https://github.com/nv-legate/cunumeric`, Dask `https://github.com/dask/dask`) implementations of the NumPy library deliver large speedups for unmodified NumPy programs.

Problem 3.2 (Skyline): Given an array S that encodes the heights of buildings in a city skyline, return the total number of unique buildings that are visible when standing to the left of the skyline. A building is visible if it is taller than all buildings to its left. In the example below, a person could see 3 unique buildings, as the first building with height 5 is visible, then 10, then 15. Specifically, `skyline([5, 5, 2, 10, 3, 15, 10]) = 3`. The CAs' solution is 1 line of code, using 3 combinators.



Problem 3.3 (Parentheses-Matching): Given an array S containing `'('` and `')'`, return whether or not S is a balanced set of parentheses. Concretely, this means that each opening parenthesis has a closing parenthesis, and for each pair of opening and closing parentheses, the opening parenthesis exists to the left of the closing parenthesis. For example, `matched(['(', ')']) = True` and `matched(['(', ')', ')']) = False`. The CAs' solution is 4 lines of code, using 4 combinators and some indexing operations.

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 `ski_eval.py`, `problem.ski`, `numpy_problems.py`, and `README.txt`, and that the script gives you no errors or warnings.