COMP2022/2922: Programming Languages, Logic and Models
Assignment 1

Due: 23:59pm Sunday 15th September (end week 6)

Submission details

Due **23:59pm Sunday 15th September 2019**. Canvas treats 23:59:01 as late. I recommend submitting earlier, as submission sites often respond more slowly close to deadlines.

Late submission

The late submission policy is detailed in the administrivia lecture slides from week 1. Please notify me if you intend to make a late submission, or if you believe you will not be able to submit, to make it easier for me to support you.

Submission format

- Exercises 1, 2, 4, 5:

  You must submit a repor ritten parts of the report must p ages, of course.

  Please make sure the exp
  see the scope of the parentheses. Use at least one of:

  - Lisp-style indentation (as demonstrated in this docume
  - Variable sized parentheses, and/or
  - Colour

  If you use fixed-width fonts for the expressions, please make sure that your prose descriptions are still in a normal font.

- Exercise 3:

  Please submit your code to the second submission point, also on Canvas. Your code should be in a plain text file, so that we can execute it directly.

A note on Academic Integrity

I would very much prefer that you invented your own encodings based what you've learned in the past 4 weeks. However, if your submission does rely on any examples / work found outside the the course, then:

1. Cite your sources properly;
2. Take care to distinguish clearly between your own work, and the cited work;
3. Explain the cited work in your own words, to demonstrate you fully understand it.

Appropriately cited work will be awarded *partial* marks in based on the marker's evaluation of how much the student has contributed to the answer. Using other's work *without* proper citations is plagiarism, which can be subject to severe penalties (and it makes me sad every time I have to report a student for it.)

Information: Shortcuts!

Church Encodings

In this assignment, you may use the following Church encodings:

- Booleans: `IFELSE`, `TRUE`, `FALSE`, `NOT`, `AND`, `OR`
- Numerals: `ZERO`, `SUCCESSOR`, `PLUS`, `MULT`, `EXP`, `ISZERO`
- Pairs: `FIRST`, `SECOND`, `PAIR`, and the `[a, b]` notation
- Lists: `CONS`, `ISEMPTY`, `HEAD`, `TAIL`, and the `{a, b, c, ...}` notation

... without bothering to expand them to the actual formulas. e.g. you are permitted to write things like `(MULT 3 4) = 12`. Furthermore, you may assume:

- `(MINUS x y)` is $x - y$ where $x \geq y$
- `(PREDECESSOR x)` is $x - 1$ where $x \geq 1$

However, if you want to do anything else with those two expressions then you are required to prove that they behave the way you say they do.

Reductions

When performing reductions you may:

- Perform implicit α-conversions at the same time as expanding a named encoding.
- Bring multiple argume

- Perform obvious and si

  However, please do *not* combine different *types* of st acros, using the Y-combinator, etc.) on the same line, or where the steps depend directly o unsure, write the steps in detail. For example, this would be fine:


```
  (\xyz.yxz) (NOT FALSE) (PRED 5) (AND FALSE TRUE)
= (PRED 5) (NOT FALSE) (AND FALSE TRUE)
= 4 TRUE FALSE
```

Exercise 1: Comparators [20 marks]

A comparator is a function used to compare two expressions. In this exercise you will make some comparators for Church Numerals.

(a) Write a combinator `LT` which takes two arguments, and outputs `TRUE` if $x < y$, or `FALSE` otherwise. Briefly justify the correctness of your solution, then reduce the following three examples

```
(LT 2 4)
(LT 2 2)
(LT 4 2)
```

(b) Write a combinator `GT` which outputs `TRUE` if $x > y$, or `FALSE` otherwise. Briefly justify the correctness of your solution. You do not need to give a worked example.

(c) Write a combinator `EQ` which outputs `TRUE` if $x = y$, or `FALSE` otherwise. Briefly justify the correctness of your solution. You do not need to give a worked example.

(d) Write a recursive expression `F` that, when given a list of Church Numerals, returns a list where any number not strictly greater than the previous one is omitted. e.g.

```
F {1, 3, 2, 4} = {1, 3, 4}
F {1, 3, 2, 3, 4} = {1, 3, 4}
```

Briefly justify the correctness of your solution. You do not need to give a worked example.

Information: Binary Search Tree

Consider the following simple e


```
NIL = PAIR TRUE TRUE
TREE = \xlr.{x, l, r}
LEAF = \x.TREE x NIL NIL        ; leaf nodes have empt
ROOT = HEAD                     ; the value is the first e
LEFT = \t.HEAD (TAIL t)         ; the left subtree is th
RIGHT = \t.HEAD (TAIL (TAIL t)) ; the right subtree is the third element
```


These expressions use it to implement a binary search tree for Church Numerals:

```
;; return the tree with x inserted
INSERT-H = (\ftx.IFELSE (ISNIL t)
                        (LEAF x)
                        (IFELSE (EQ x (ROOT t))
                                t
                                (IFELSE (LT x (ROOT t))
                                        (TREE (ROOT t) (f (LEFT t) x) (RIGHT t))
                                        (TREE (ROOT t) (LEFT t) (f (RIGHT t) x)))))
INSERT = Y INSERT-H

;; return TRUE if it is in the tree, FALSE otherwise
SEARCH-H = (\ftx.IFELSE (ISNIL t)
                        FALSE
                        (IFELSE (EQ x (ROOT t))
                                TRUE
                                (IFELSE (LT x (ROOT t))
                                        (f (LEFT t) x)
                                        (f (RIGHT t) x))))
SEARCH = Y SEARCH-H
```

Exercise 2: TreeMap [25 marks]

The **Map** abstract data type describes an interface to store [key, value] pairs, with operations **put** and **get** to store and retrieve the values, indexed using their keys. A **TreeMap** is a concrete datatype implementing **Map** where a search tree indexed using the keys is used to store the data. In this exercise you will invent a simple encoding of **TreeMap** that stores Church Numerals strictly greater than 0 as the keys and values:

(a) `PUT`: a combinator which, given a **TreeMap**, key, and value, returns an equivalent **TreeMap**, except the given key maps to the new value. Briefly justify the correctness of your solution. Hint: modify `INSERT`

(b) `GET`: a combinator which, given a **TreeMap** and a key, returns the corresponding value. If the key is not in the **TreeMap**, it should return 0. Briefly justify the correctness of your solution. Hint: modify `SEARCH`

(c) Demonstrate your `PUT` encoding, by reducing the following expression until it is a composition of `TREE`, `LEAF`, `NIL`, and `PAIR` expressions (with no other λ's remaining). You may find it helpful to show part of your working, but it is not required.

```
(PUT (PUT (PUT (PUT (PUT NIL 4 6) 2 3) 3 7) 2 5) 6 7)
```

(d) Simplify it further, by rewriting the resulting expression using the bracketed list `{a, b, c, ...}` and pair `[a, b]` notation instead of `TREE`, `LEAF` and `PAIR`.

(e) Suppose you needed to store arbitrary expressions, not just Church Numerals ≥ 1, meaning we couldn't use 0 to signify 'not found'. What changes would you make to the encoding to deal with this problem? You don't need to write the a

Exercise 3: Lisp [20 ma

A code skeleton is provided which provides function definitions for some o earlier exercises. Some of the functions have been implemented for you al

You are required to:

- Finish implementing the incomplete functions.
- Comment your code.
- Add some more example computations, to test that your solution works.

You do not need to implement any functions that are not already named in the skeleton code (although you might find it useful to add some). Some of the function names are a bit longer/more descriptive than the ones in the theory exercises, to avoid conflicts with Lisp keywords. The required functions are:

- Exercise 1:
  - exercise-1d (exercise 1d)
- Exercise 2:
  - insert-bst (already partly implemented)
  - put-map (exercise 2a)
  - get-map (exercise 2b)

You are required to write your code in a functional style:

- The body of each function should be a single expression

- Do not put `print` statements inside function bodies
- Forbidden keywords:
  - variables: `sq`, `setq`, `let`
  - loops: `for`, `when`
- Useful keywords:
  - `defun`, `lambda`
  - lists:
    - construction: `cons` (one step), `list` (all at once)
    - head: `car`, `first`
    - tail: `cdr`, `rest`
    - nth: `first`, `second`, `third`, ...
  - `t` means 'true'
  - `nil` encodes 'false' or 'empty list'
  - `(null x)` is `t` if `x` is empty/false, otherwise it is `nil`
  - `if`. This one is particularly important – directly implementing the lambda expression for `IFELSE` will lead to problems, because the Lisp implementations we are using do not reduce in normal order or use lazy evaluation).

Information: Arithmetic Expression Trees

Arithmetic expressions can be thought of as tree structures.

```
NUM = \x.(PAIR 0 x)
VAR = \x.(PAIR 1 x)
BOP = \x.(PAIR 2 x)
TYPE = FIRST
VALUE = SECOND
ISNUM = \x.(= 0 (TYPE x))
ISVAR = \x.(= 1 (TYPE x))
ISBOP = \x.(= 2 (TYPE x))
```

Consider the following tree:

```
(TREE (BOP +)
      (TREE (BOP -)
            (LEAF (NUM 3))
            (LEAF (NUM 1)))
      (LEAF (NUM 4)))
```

This represents the expression `(+ (- 3 1) 4)`. In infix notation, that is $((3-1)+4)$
Variables are enumerated as Church Numerals, starting from 1. e.g.

```
(TREE (BOP +)
      (TREE (BOP EXP)
            (LEAF (VAR 1))
            (LEAF (NUM 2))
      (TREE (BOP *)
            (LEAF (VAR 1))
            (LEAF (VAR 2)))))
```

... represents the expression `(+ (EXP x 2) (* x y))`, where x is variable 1 and y is variable 2. In infix notation, that is $(x^2 + xy)$

Exercise 4: Arithmetic Expression Trees [15 marks]

Write combinators for the following computations:

(a) `EVALUATE` computes the the answer to an arithmetic expression that does not include variables. e.g.

```
EVALUATE (TREE (BOP +)
               (TREE (BOP -)
                     (LEAF (NUM 3))
                     (LEAF (NUM 1)))
               (LEAF (NUM 4)))
= 6
```

(b) `SUB-ONE` should take an arithmetic expression, a variable id, and value, and return the expression where all occurences of that variable have been replaced with its value. e.g. if

```
A = (TREE (BOP +)
          (TREE (BOP EXP)
                (LEAF (VAR 1))
                (LEAF (NUM 2)))
          (TREE (BOP *)
                (LEAF (VAR 1))
                (LEAF (VAR 2))))
```


then


```
SUB-ONE A 1 6 = (TREE (BOP +)
                      (TREE (BOP EXP)
                            (LEAF (NUM 6))
                            (LEAF (NUM 2)))
                      (TREE (BOP *)
                            (LEAF (NUM 6))
                            (LEAF (VAR 2))))
```


(c) `SUB-ALL` should rewrite an arithmetic expression to replace all occurences of the variables specified as keys in a treemap, with the corresponding values.

```
B = (PUT (PUT NIL 1 8) 2 3)
EVALUATE (SUB-ALL A B) = 88
```

Briefly justify the correctness of each of your solutions.

You do not need to do any error handling with regard to types or edge cases. For example, you may assume that any expression applied to `EVALUATE` does not contain any variables, so your encoding doesn't need to check that.

Exercise 5: Lambda calculus trees [20 marks]

The techniques you used in exercise 4 can be used to evaluate other types of expressions. For example, Lambda calculus itself!

(a) Outline a simple data structure encoding lambda calculus expressions in a tree-like structure. Briefly explain any differences between this and the encoding we used for the arithetic tree.

(b) Write a combinator `ISREDEX`, which, when an encoding of a lambda expression is applied to it, is equivalent to `TRUE` if it is of the form $((\lambda x.M) \cdot N)$, or `FALSE` otherwise. Briefly justify the correctness of your solution.

(c) Write a combinator `ISNORMAL`, which, when an encoding of a lambda expression is applied to it, is equivalent to `TRUE` if it is in β-normal form, or `FALSE` otherwise. Briefly justify the correctness of your solution.

(d) Describe how you would design an encoding `NORMALISE` which would (try to) reduce an expression to its normal form. You do *not* need to actually write the encoding, just outline the main ideas.