CSCI3390-Assignment 4.

due October 31

A whole bunch of questions about the little programming languages. You don't have to do all of them. A 'perfect' paper is 100 points.

Some of the problems below (labeled P) are pencil-and-paper questions, others (labeled C) require that you write and run code, which you will submit as separate files. For the tiny Python fragment, submit just a single code file: this should be the file `recursive_function_land.py` posted on the website, with the functions that you write appended. Make sure that you 'cheat-check' your code—I will! For the λ-calculus problems, you should likewise append your answers to the file `lambdaland.py`, which I provide.

Problems 6,7,20 and 21 are all worth 20 points. Problems 4,5 and 13 are worth 15 points. The rest are all worth 10 points.

1 Tiny Python (*aka* Rubber Boa)

1. *(P)* Show how to implement the standard Python `if-else` construction in tiny Python. To be precise, suppose `f,g,h` are functions of one variable that have already been defined. Write a function `H(x,y)` that returns `f(y)` if `x=0`, `g(y)` if `x=1` and `h(y)` otherwise. This is required in the proof that Turing machines can be simulated in Tiny Python. That is, write the equivalent of the standard Python

```
def H(x,y):
    if x==0:
        z=f(y)
    elif x==1:
        z=g(y)
```

```
else:
    z=h(y)
return z
```

2. *(C)* Write a function `power(x,y)` that returns x^y. (If $x = y = 0$, the function should return 1.)

3. *(C)* Write a function `equals(x,y)` that returns 1 if x and y are equal, and 0 otherwise. This should show you how to encode `while` loops like `while u!=1`, which we required in the proof that Turing machines can be simulated in the tiny Python language.

4. *(C)* Write a function `prime(x)` that returns 1 if `x` is prime, and 0 otherwise. Do this without using `while`.

5. *(C)* Write a function `log(x,y)` that returns the largest integer z such that $y^z \leq x$. Do this without using `while`

6. *(P)* This problem is more theoretical. We want a careful proof of the assertion made in the notes that functions computable in the tiny Python language are all partial recursive functions (i.e., obtainable from 0 and increment through application of composition, primitive recursion, and μ-recursion). The converse assertion, that every partial recursive function is computable in the programming language, was justified in the notes by the fragments of code implementing `for` and `while`.

   To do this, consider a sequence σ of lines of the programming language. Let $x_1, \ldots, x_k$ be variables that include all the variables in the sequence σ. Then executing σ updates the values of these k variables, essentially computing k different partial functions

   $$x_i \leftarrow f_i(x_1, \ldots, x_k).$$

   Show by induction on the length of the sequence σ that each of these functions is partial recursive.

7. *(P)* Another 'theory' problem: Suppose I revise the apparatus of the programming language so that it has no while or for loops. However, I will allow a proper decrement operation (which in Python is `v=max(0,v-1)`), an `if` statement that has the form `if v==0: STATEMENT` and *recursive* function calls: That is, a function is allowed to call itself. Show that this

programming language is also computationally universal. To do this, show that every partial recursive function can be implemented this way: Essentially this boils down to showing that primitive recursion and μ-recursion can both be implemented through recursive function calls.

2 Counter Machines

I did not include any sort of subroutine ability in the counter machine simulator, so each solution to the problems below is a complete program. Nonetheless, by means of some clever copying and pasting, you can save yourself a lot of typing. For instance, multiplication has adding somewhere inside, and powering has multiplication.

8. (C) Write a counter machine program that starts with a value x in counter a (with all other counters assumed to be zero) and finishes with the value x in counters a and b. This trick for 'copying' values will be useful in solving some of the other problems.

9. *(C)* Implement 'proper subtraction' as a counter machine program. Your program should start with its arguments x and y in counters a and b, and finish with $\max(x - y, 0)$ in in counter a. .

10. *(C)* Implement multiplication as a counter machine program. Your program should start with its arguments in counters a and b and finish with its result in counter a.

11. *(C)* Implement powering as a counter machine program. Your program should start with its arguments in counters a (base) and b (exponent) and finish with its result in counter a.

12. *(C)* Implement integer division as a counter machine program. Your program should start with its arguments in counters a and b (divisor) and finish with its results in counter a (quotient) and b (remainder).

13. *(C)* Implement primality testing as a counter machine program. Your program should start with its argument in counter a and finish with 1 (prime) or 0 (composite) in counter a. (The result should be 0 for input 0 and 1.)

3 FRACTRAN

14. The FRACTRAN program

$$[2]$$

when applied to the initial value $1 = 2^0$ produces the infinite sequence

$$2^1, 2^2, 2^3, \ldots.$$

That is, in terms of our convention on the encodings of inputs and outputs, it is generating the sequence

$$1, 2, 3, \ldots$$

Write a FRACTRAN program with an infinite loop that generates the sequence

$$1, 0, 1, 0 \ldots.$$

15. Write a FRACTRAN program that computes the maximum of its two arguments. The answer should

16. Write a FRACTRAN program that computes the minimum of its two arguments.

4 λ-calculus

For the first three problems below, there is a paper-and-pencil part, and a programming part. For the paper-and-pencil part, write the required function using standard λ-calculus notation (you can use the abbreviation $\lambda xy.E$ for $\lambda x.\lambda y.E$). Then demonstrate a complete calculation. For instance, with the OR problem below, you should demonstrate that

$$OR \ \ \mathbf{T} \ \ \mathbf{F} = \mathbf{T}$$

and with the multiplication problem, you should show why

$$MUL \quad \bar{m} \ \ \bar{n} = \overline{mn}.$$

In addition, you should implement the function in Python, imitating the examples in lambdaland.py. There are examples to show you how to test your answer when the function returns a boolean, and when it returns an integer.

17. Implement and test the OR function.

18. Implement and test the multiplication function.

19. We can define the function ISZERO by

$$\lambda z.z(\lambda x.\mathbf{F})T.$$

Demonstrate that

$$ISZERO\ \ \bar{0} = \mathbf{T}$$

and that

$$ISZERO\ \ \bar{n} = \mathbf{F}$$

whenever $n > 0$.

20. This is harder, but there is a road map below. We would like to implement the predecessor function PRED. If n is a nonnegative integer, we should have

$$PRED\ \ \bar{0} = \bar{0}$$

$$PRED\ \ \bar{n} = \overline{n-1} \text{ if } n > 0$$

Here are steps you can follow to do this: We will use pairing iteratively to create this sequence of pairs:

$$\mathbf{0} = ((\bar{0}, \bar{0}), \bar{0}),$$

$$\mathbf{1} = (\mathbf{0}, \bar{1})$$

$$\mathbf{2} = (\mathbf{1}, \bar{2}),$$

etc. Write a function BUILD such that

$$BUILD\ \ \bar{n} = \mathbf{n}.$$

We then can obtain PRED as

$$PRED\ \ \bar{n} = \text{ second(first}(BUILD\ \ \bar{n})).$$

21. The preceding problem used pairing and Church numerals to compute the predecessor function. You can also use these to implement primitive recursion in general. In the problem, you will use this approach to implement the factorial function. The idea is to create the function

$$F(n) = (n!, n + 1)$$

and project onto the first component. F is obtained by applying a function g n times to the pair $(1, 1)$, where

$$g(m, k) = (mk, k + 1).$$