A whole bunch of questions about the little programming languages in Lecture 10. 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!

Point values: 15 points for any of 1-6, 20 points for 3-6 if you do the while-less version as described in Problem 7. 15 points each for 8-12. 20 points for 13. 25 points each for 14 and 15, 10 points each for 16 and 17.

1 Tiny Python (aka Rubber Boa)

1. (P) Show that we can make tiny Python even tinier. That is, show that we do not lose any computing power if we get rid of assignment statements whose right-hand side contains a composition of two or more functions, such as

   \[ y = f(x, y, g(u, v, w)) \]

   and variable-copying statements such as

   \[ y = x \]
(That is, show how each of these operations can be implemented in terms of the remaining ones.)

2. \(P\) Show how to implement the standard Python \texttt{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. That is, write the equivalent of the standard Python

\[
\text{def } H(x, y):
    \text{if } x==0:
        z=f(y)
    \text{elif } x==1:
        z=g(y)
    \text{else:}
        z=h(y)
    \text{return } z
\]

3. \(C\) Write a function \(\text{power}(x, y)\) that returns \(x^y\). (If \(x = y = 0\), the function should return 1.)

4. \(C\) Write a function \(\text{divides}(x, y)\) that returns 1 if \(x\) is a divisor of \(y\), and 0 otherwise. (This should return 0 if \(x=0\).) You may find it useful along the way to write another function \(\text{minus}(x, y)\) that returns \(\max(x-y, 0)\).

5. \(C\) Write a function \(\text{prime}(x)\) that returns 1 if \(x\) is prime, and 0 otherwise.

6. \(C\) Write a function \(\text{log2}(x)\) that returns the largest integer \(y\) such that \(2^y \leq x\).

7. \(C\) Did your solutions to any of the last four problems use \texttt{while}? Solve them without \texttt{while}.

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 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.)

14. (P) *In the notes there is a sketch of a proof that Turing machines can be simulated by counter machines. Show in fact that a Turing machine can be simulated by a counter machine that uses only three counters. (It follows that anything can be computed using only three counters. As with most such statements, this needs to be qualified by adding that it depends on how the inputs and outputs are encoded—if you are using four counters to hold the input and five to hold the result, then of course you need more than three.)

15. (P) *Describe an algorithm for converting a counter machine program into an equivalent tiny Python function. Show that this can be done so that the resulting tiny Python function uses only one while statement. It follows that we never need more than one while in any program.
3 FRACTRAN

16. The FRACTRAN program

\[ \frac{2}{\text{}[2]} \]

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

\(1, 0, 1, 0, \ldots\)

17. What does the FRACTRAN program below do? You should think of this as computing a function of two variables, with the initial values \(x, y\) of the variables encoded as \(2^x 3^y\), and the result \(z\) encoded as \(5^z\). (Run this with a few different values of \(x\) and \(y\) in the simulator and you should see what’s happening.)

\([13/33, 17/11, 385/13, 19/119, 1/17, 51/19, 11/2, 1/3]\)