CSC 208 (23sp)—Lab 1: Introduction to LaTeX

(Minh Nguyen / Havin Lim)

Labs in this course will normally consist of `.tex` files for you to fill in with your partner. If you use Overleaf, you can import this file (along with `csc208.sty`) into a new project to get started. When you are done, please turn in a completed PDF of your work by the appropriate deadline. Please make sure that your PDF successfully compiles on overleaf without errors. Otherwise, your PDF may not be malformed or otherwise missing portions of your work.

Problem 1 (Introduction to LaTeX) At this point, you should have created an Overleaf account and worked through the "Learn LaTeXin 30 minutes" tutorial found on the site:

1. `https://www.overleaf.com/learn/latex/Learn_LaTeX_in_30_minutes`

Overleaf's documentation (`https://www.overleaf.com/learn`) also serves as an excellent resource for learning how to do specific things in LaTeXas the need arises. I recommend going to it as a first resource when you want to do something in a document.

Use what you learned from the tutorial in addition with information you gather from Overleaf's documentation to accomplish the following tasks:

1. Typeset the quadratic formula. You can remind yourself what this is at the following Wikipedia page:

   - `https://en.wikipedia.org/wiki/Quadratic_equation`.

   Make sure to use display math mode (rather than inline mode) and ensure that your document uses the same symbols as the Wikipedia article.

2. Typeset your recursive Python `sum` function from last class using the `verbatim` environment. (*Hint*: The LaTeXterm for a block of a code is a "code listing").

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

```
def sum(x):
    if x >= 1:
        return x + sum(x-1)
    else:
        return 0
```

Problem 2 (More Python Practice) In our first class period, we began introducing the Python programming language by mapping some of the basic constructs from Scheme into Python. Let's practice writing some more Python code to get a better feel for the language.

Hop into VSCode and implement and verify the following functions in a new Python file. Rather than turning in the Python file, though, typeset your functions when you are done as your answer to this problem. Make sure to use the `verbatim` environment to do so and that your code stays within the margins of the page, potentially using line breaks and/or multiple `verbatim` environments as necessary.

(a) `cToF(deg)` takes the temperature `deg` in Celsius and returns the temperature in `fahrenheit`.

(b) `gradeToLetter(gpa)` takes a `gpa`, a number assumed to be in the range 0 through 4 and returns a letter grade as a string for this GPA according to the following scale:

- Greater than or equal to 3.5: `A`
- Between 2.5 (inclusive) and 3.5 (exclusive): `B`
- Between 1.5 (inclusive) and 2.5 (exclusive): `C`
- Between 0.5 (inclusive) and 1.5 (exclusive): `D`
- Less than 0.5: `F`

```
def cToF(deg):
    return deg * 9/5 + 32

def gradeToLetter(gpa):
    if gpa >= 3.5:
        print('A')
    elif gpa < 3.5 and gpa >= 2.5:
        print('B')
    elif gpa < 2.5 and gpa >= 1.5:
        print('C')
    elif gpa < 1.5 and gpa >= 0.5:
        print('D')
    elif gpa < 0.5:
        print('F')
    else:
        print('Wrong Input')
```

Problem 3 (Basic Python Tracing) In CSC 151, you learned about a *substitutive model of computation* for Racket/Scheme programs where we model how a program evaluates as an extension of basic arithmetic. as our first foray into the world of program correctness.

Definition 1. ***(Substitutive evaluation)****. Expressions take* repeated steps of evaluation *until they evaluate to a final value. A step of evaluation consists of:*

1. Finding *the next subexpression to evaluate, respecting precedence and associativity of relevant operators.*

2. Evaluating *that subexpression to a value.*

3. Substituting *the value for the subexpression that produced it in the overall expression.*

As we talked discussed today, we will employ a similar model for Python programs as our first foray into the world of program verification. Let's review this model now and next week we will further adapt it to the particulars of Python in our future classes.

For each Python expression, give its step-by-step evaluation to a final value. To typeset your steps, use the `verbatim` environment to format your derivation as follows.

```
    1 + (2 - 3) * 4
--> 1 + -1 * 4
--> 1 + -4
--> 3
```

Note how each line of the derivation is indented with a long arrow (`-->`) denoting that the previous expression *steps* to the next expression.

1. `(3 * 5 - 20 / 2) ** ((2 + 8) - 6)`
2. `(8 + 3 * 2 - 1 / 5)`
3. `s[0] + str(len(s)) + s[2]` where `s = "abc"`.

- The `**` operator is the exponentiation operator, *i.e.*, `x ** y` is equivalent to x^y.
- Regarding strings, the indexing operation `s[0]` returns the character at the given position of the string, 0-indexed. In this case `s[0]` returns the first character of the string `s`.
- `len(s)` returns the length of the string `s` as a number.
- `str(v)` returns the string representation of the value `v`.

Make sure to check your work in Python by creating a source file and using `print` to see the results or use the Python REPL directly.

```
1. (3 * 5 - 20 / 2) ** ((2 + 8) - 6)
--> (15 - 20 / 2) ** ((2 + 8) - 6)
--> (15 - 10) ** ((2 + 8) - 6)
--> 5 ** ((2 + 8) - 6)
--> 5 ** (10 - 6)
--> 5 ** 4
--> 625
```

```
 2. (8 + 3 * 2 - 1 / 5)
--> (8 + 6 - 1 / 5)
--> (8 + 6 - 0.2)
--> (14 - 0.2)
--> 13.8

 3. s[0] + str(len(s)) + s[2] where s = "abc"
--> 'a' + str(len(s)) + s[2]
--> 'a' + str(3) + s[2]
--> 'a' + '3' + s[2]
--> 'a' + '3' + 'c'
--> 'a3' + 'c'
--> 'a3c'
```