introduction

Regular expressions (abbreviated to regex, the pronunciation of which gives rise to endless flame wars...) are used in various programming languages and utilities to match entire classes of strings. This assignment will give you experience modelling a regular expression as a tree, and detecting which strings match a given regular expression.

We won’t favour any particular regex language or utility, such as Python, Java, or Linux’s grep, but use a stripped-down simplified regular expression form that contains all the essential principles. In particular, you are not permitted to import Python’s regular expression module re into any file you submit.

We’ll only be dealing with a ternary alphabet, i.e., \{0,1,2\}. Generalizing to an arbitrary alphabet is straightforward, but doesn’t buy us much for the purpose of this assignment. The elementary regex symbols we’ll use can be easily typed in Python source code. A regex (over ternary alphabet \{0,1,2\}) is a nonempty string made up of the following symbols.

1. '0' called zero
2. '1' one
3. '2' two
4. 'e' pronounced “ee” or “epsilon”
5. '|' bar
6. '.' dot
7. '*' star
8. '(' left parenthesis, or left
9. ')' right parenthesis, or right

There are several rules determining that a string made up of these symbols is a valid regular expression:

1. There are 4 regexes of length one. They are:
   - '0'
   - '1'

2. If \( r \) is a regular expression, then so is \( r^+ * \), where the plus symbol \( ^+ \) means string concatenation, as in Python.

3. If \( r_1 \) and \( r_2 \) are regexes, then so are:
   - \( (r_1 + r_2 + ) \)
   - \( (r_1 + r_2 + ) \)

Here are some examples of regexes:

- \('0'\)
- \('1'\)
- \('2'\)
- \('e'\)
- \('0*'\)
- \('1*'\)
- \('2*'\)
- \('e*'\)
- \('(0|1)'\)
- \('(1.2)'\)
- \('(e|0)'\)
- \'(2.e)'\)
- \'(0*|2*)'\)
- \'(0.1).2'\)
- \'(1.(0|2)*.0)'\)

**tree representation of regexes**

Every regex can be (uniquely) represented as a tree. Each leaf node contains exactly one of \('0'\), \('1'\), \('2'\), or \('e'\). Each internal node contains exactly one of \('.'\), \('|'\), or \('*').

A regex of length one is represented by a tree of one node containing the symbol in the regex. For example, the regex '\('0'\)' is represented by the tree whose root is the leaf node containing '\('0'\)'.

A regex of the form \( r^+ * \) is represented by a tree whose root node contains \('*'\), and that node has one child which is the tree that represents the regex \( r \). E.g., the regex '\('1*'\)' is represented by the following tree:
A regex of the form \((r_1 + r_2 + |)\) is represented by a tree whose root node contains \(|\), and that node has left and right children which are the trees that represent the regexes \(r_1\) and \(r_2\) respectively. For example, the regex ‘(0|1)’ is represented by the following tree:

\[
\begin{array}{c}
(\text{0}) \\
| \\
(\text{1})
\end{array}
\]

A regex of the form \((r_1 + r_2 + . + r_2 + .)\) is represented just as a regex of the form \((r_1 + r_2 + . + r_2 + .)\), except the root now contains \(\cdot\) rather than \(|\). For example, the regex ‘((0.1).0)’ is represented by the following tree:

\[
\begin{array}{c}
(\text{0}) \\
| \\
(\text{1}) \\
| \\
(\text{0}) \quad (\text{0})
\end{array}
\]

Here’s an example that combines all concepts from above. The regex ‘((1.(0|1)*).0)’ is represented by the following tree:

\[
\begin{array}{c}
(\text{1}) \\
| \\
(\text{0}) \\
| \\
(\text{1}) \\
| \\
(\text{1}) \\
| \\
(\text{0}) \quad (\text{1})
\end{array}
\]

**matching strings with regexes**

A ternary string is a string (possibly empty) that contains only the symbols ‘0’, ‘1’ and ‘2’. For a regex \(r\) and a ternary string \(s\), we define below what it means for \(r\) to match \(s\). (Equivalently we may also say that \(s\) matches \(r\), or that \(r\) and \(s\) match).

1. A regex of length one matches exactly one string. Specifically:
   - the regex ‘0’ matches the string ‘0’
   - the regex ‘1’ matches the string ‘1’
   - the regex ‘2’ matches the string ‘2’
   - the regex ‘e’ matches the string ‘’ (i.e., ‘e’ matches the empty string)
2. A regex of the form \( r^+ \) matches string \( s \) if and only if either
   
   (a) \( s \) equals \\
   \"\" (empty string), or
   \( b) \ s \) has the form \( s_1 + s_2 + \cdots + s_k \) where \( k > 0 \) and \( r \) matches every \( s_i \)
   
   For example, the regex \( '0*' \) matches any string (possibly empty) that contains no other symbols than
   the symbol \( '0' \).

3. A regex of the form \( '(r_1 + r_2 + \cdots)' \) matches string \( s \) if and only if:
   
   (a) \( r_1 \) matches \( s \), or
   (b) \( r_2 \) matches \( s \), or
   (c) both of the above
   
   For example, the regex \( '(2|0*)' \) matches the string \( '2' \) as well as any string that contains only the
   symbol \( '0' \).

4. A regex of the form \( '((r_1 + .) + .)' \) matches a string \( s \) if and only if there are two
   strings \( s_1 \) and \( s_2 \) (each possibly empty) such that
   
   (a) \( s \) is the concatenation of \( s_1 \) and \( s_2 \) (i.e., \( s \) equals \( s_1 + s_2 \)), and
   (b) \( r_1 \) matches \( s_1 \), and
   (c) \( r_2 \) matches \( s_2 \).
   
   For example, the regex \( '(1*.2)' \) matches any string that contains zero or more \( '1' \)'s, followed by
   exactly one \( '2' \).

Here's an example that combines all concepts from above. The regex \( '((1.(0|1)*).2)' \) matches any string
that starts with \( '1' \) and ends with \( '2' \), and has any number of \( '0' \)'s and \( '1' \)'s in between (including nothing
in between).

**your job**

You may work with up to two other students currently in CSC148 — in other words, a group of 1 to 3
people. Your tasks come in two stages

**Stage 1, due March 6th at 10:00 p.m.:** Design a collection of classes to represent the various sorts of regular
expression trees. Each class should implement or inherit an \_eq\_ method so that it can be compared
to other objects; and a \_repr\_ method so that it can be represented in a meaningful way as a string,
and so that an equivalent tree will be produced if you cut-and-paste the representation into a Python
shell. You should carefully consider how to use inheritance to reduce the amount of duplicated code.
You should also ensure that public attributes for a tree's symbol or children can only be set once,
during initialization. After that, they should be read-only.

Your class(es) should be declared in a file called regex\_design.py, which should be submitted on MarkUs
by 10 p.m. on March 6th. Of course, all your classes and methods should have good documentation.

Stage 1 is worth 40% of Assignment 2.

**Stage 2, due March 20th at 10:00 p.m.:** You should:
1. Download `regextree.py` and `regex_functions.py` (these will be available shortly after March 6th). Add your name(s) to the license at the top of `regex_functions.py`, indicating that you have added intellectual value to it. However, do not add any import statements, or change the import statement, in `regex_functions.py`. You may only distribute the files, or modified versions of them, with the same license, and along with the file `COPYING`.

2. In `regex_functions.py` implement the following module-level functions, as usual with good documentation:

   `is_regex(s)`: which takes a string `s` and produces `True` if it is a valid regular expression (according to the characterization in the introduction of this handout), but `False` otherwise.

   `all_regex_permutations(s)`: which takes a string `s` and produces the set of permutations of `s` that are also valid regular expressions (according to the characterization in the introduction of this handout).

   `regex_match(r, s)`: which returns `True` if and only if string `s` matches the regular expression tree rooted at `r`.

   `build_regex_tree(regex)`: which takes a valid regular expression `regex`, builds the corresponding regular expression tree, and returns its root.

3. Submit `regex_functions.py` on MarkUs by 10 p.m. on March 20th.

   Stage 2 is worth 60% of Assignment 2.