ULL. ESIT. PROCESADORES DE LENGUAJES. Training Exam. 1 page

Full Name and Alu: _____

Introduction

In this exam you'll build a parser for the language of regular expressions (from now on **regexp**). The regexp language that we consider is defined by the following rules:

1. If `re1` and `re2` are two regexps then `re1 | re2` is a regexp
2. The concatenation of regexps is a regexp: If `re1` and `re2` are regexps then `re1 re2` is a regexp
3. The closures of regexps `re` as `re?`, `re*` and `re+` are regexps
4. If `re` is a regexp then `(re)` between parenthesis is a regexp
5. (Metasymbols) The dot `'.'`, the `'^'` and the `'$'` are regexps and have special meanings
6. Characters that are not metasymbols (that is, different from `'*'`, `'+'`, `'|'`, etc.) are regular expressions
7. Metasymbols (as `'*'`, `'+'`, `'?'`, `'|'`, `'^'`, `'$'`, etc.) when they are escaped `'\*'` are considered characters and are regular expressions
8. White Spaces are allowed. Comments can start with `#` and end with the end of the line or can be C style `/* ... */`

Regarding priority and associativity, other that the usual conventions, let us point that we want the meaning of `a|bc` to be `a|(bc)` and the meaning of `cd*` to be `c(d*)`

For all the following questions your goal will be to write a parser that receives as input a regular expression like `(ab)+|b` and outputs an Abstract Syntax Tree like this one:

```
{ "type": "OR", "children": [
    { "type": "PLUS", "children": [
        { "type": "PAREN", "children": [
            { "type": "CAT",
              "children": [ { "type": "CHAR", "value": "a" }, { "type": "CHAR", "value": "b" } ]
            }
          ]
        }
      ]
    },
    { "type": "CHAR", "value": "b" }
  ]
}
```

1. Write the lexical analyzer using your own lexical analyzer generator (as it was specified in the class lab *Lexer-Generator*) or `moo-ignore` or `moo`
2. Write a Nearley.JS non ambiguous grammar that generates a parser that recognizes this regexp language
3. Write a method `toJSRegexp(ast)` that receives the AST built by the parser and returns a JS string `str` whithout the spaces, without the comments and with the special characters properly escaped so that a JS regexp can be easily built using `new Regexp(str)`. For instance, for the former AST the output should be `(?:((?:ab))+|b)`. For the input

   ```
   a
   \s
   \* /* a literal star */
   ```

   the output should be `(?:(?:a\\s)\\*)`.