CS 61A
Spring 2022

Regular Expressions, BNF
Discussion 13: April 20, 2022 Solutions

Regular Expressions

Regular Expressions

Regular expressions are a way to describe sets of strings that meet certain criteria, and are incredibly useful for pattern matching.

The simplest regular expression is one that matches a sequence of characters, like `aardvark` to match any “aardvark” substrings in a string.

However, you typically want to look for more interesting patterns. We recommend using an online tool like regexr.com or regex101.com for trying out patterns, since you’ll get instant feedback on the match results.

Character classes

A character class makes it possible to search for any one of a set of characters. You can specify the set or use pre-defined sets.

Class	Description
`[abc]`	Matches a, b, or c
`[a-z]`	Matches any character between a and z
`[^A-Z]`	Matches any character that is not between A and Z.
`\w`	Matches any “word” character. Equivalent to `[A-Za-z0-9_]`.
`\d`	Matches any digit. Equivalent to `[0-9]`.
`[0-9]`	Matches a single digit in the range 0 - 9. Equivalent to `\d`.
`\s`	Matches any whitespace character (spaces, tabs, line breaks).
`.`	Matches any character besides new line.

Character classes can be combined, like in `[a-zA-Z0-9]`.

Combining patterns

There are multiple ways to combine patterns together in regular expressions.

Combo	Description
`AB`	A match for A followed immediately by one for B. Example: `x[.,]y` matches “x.y” or “x,y”.
A|B	Matches either A or B. Example: \d+|Inf matches either a sequence containing 1 or more digits **or** “Inf”.

A pattern can be followed by one of these quantifiers to specify how many instances of the pattern can occur.

Symbol	Description
`*`	0 or more occurrences of the preceding pattern. Example: `[a-z]*` matches any sequence of lower-case letters or the empty string.
`+`	1 or more occurrences of the preceding pattern. Example: `\d+` matches any non-empty sequence of digits.
`?`	0 or 1 occurrences of the preceding pattern. Example: `[-+]?` matches an optional sign.
`{1,3}`	Matches the specified quantity of the preceding pattern. `{1,3}` will match from 1 to 3 instances. `{3}` will match exactly 3 instances. `{3,}` will match 3 or more instances. Example: `\d{5,6}` matches either 5 or 6 digit numbers.

Groups

Parentheses are used similarly as in arithmetic expressions, to create groups. For example, `(Mahna)+` matches strings with 1 or more “Mahna”, like “MahnaMahna”. Without the parentheses, `Mahna+` would match strings with “Mahn” followed by 1 or more “a” characters, like “Mahnaaaa”.

Anchors

- `^`: Matches the beginning of a string. Example: `^(I|You)` matches I or You at the start of a string.
- `$`: Normally matches the empty string at the end of a string or just before a newline at the end of a string. Example: `(\.edu|\.org|\.com)$` matches .edu, .org, or .com at the end of a string.
- `\b`: Matches a “word boundary”, the beginning or end of a word. Example: `s\b` matches s characters at the end of words.

Special characters

The following special characters are used above to denote types of patterns:

```
\ ( ) [ ] { } + * ? | $ ^ .
```

That means if you actually want to match one of those characters, you have to *escape* it using a backslash. For example, `\(1\+3\)` matches “(1 + 3)”.

Using regular expressions in Python

Many programming languages have built-in functions for matching strings to regular expressions. We’ll use the Python re module in 61A, but you can also use similar functionality in SQL, JavaScript, Excel, shell scripting, etc.

The search method searches for a pattern anywhere in a string:

```
re.search(r"(Mahna)+", "Mahna Mahna Ba Dee Bedebe")
```

That method returns back a match object, which is considered truth-y in Python and can be inspected to find the matching strings. If no match is found, returns `None`.

For more details, please consult the re module documentation or the re tutorial.

Q1: Greetings

Let's say hello to our fellow bears! We've received messages from our new friends at Berkeley, and we want to determine whether or not these messages are *greetings*. In this problem, there are two types of greetings - salutations and valedictions. The first are messages that start with "hi", "hello", or "hey", where the first letter of these words can be either capitalized or lowercase. The second are messages that end with the word "bye" (capitalized or lowercase), followed by either an exclamation point, a period, or no punctuation. Write a regular expression that determines whether a given message is a greeting.

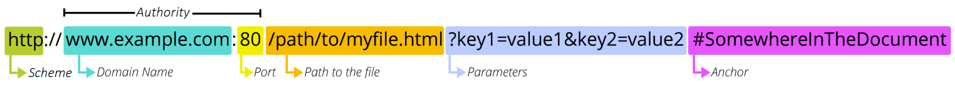
```
import re

def greetings(message):
    """
    Returns whether a string is a greeting. Greetings begin with
    either Hi, Hello, or
    Hey (first letter either capitalized or lowercase), and/or end
    with Bye (first letter
    either capitalized or lowercase) optionally followed by an
    exclamation point or period.

    >>> greetings("Hi! Let's talk about our favorite submissions to
    the Scheme Art Contest")
    True
    >>> greetings("Hey I love Taco Bell")
    True
    >>> greetings("I'm going to watch the sun set from the top of
    the Campanile! Bye!")
    True
    >>> greetings("Bye Bye Birdie is one of my favorite musicals.")
    False
    >>> greetings("High in the hills of Berkeley lived a legendary
    creature. His name was Oski")
    False
    >>> greetings('Hi!')
    True
    >>> greetings("bye")
    True
    """
    return bool(re.search(r"(^([Hh](ey|i|ello)\b))|(\b[bB]ye[!\.]?$)
    ", message))
```

Q2: Basic URL Validation

In this problem, we will write a regular expression which matches a URL. URLs look like the following:



URL

For example, in the link `https://cs61a.org/resources/#regular-expressions`, we would have:

- Scheme: `https`

- Domain Name: `cs61a.org`
- Path to the file: `/resources/`
- Anchor: `#regular-expressions`

The port and parameters are not present in this example and you will not be required to match them for this problem.

You can reference this documentation from MDN if you're curious about the various parts of a URL.

For this problem, a valid domain name consists of any sequence of letters, numbers, dashes, and periods. For a URL to be "valid," it must contain a valid domain name and will optionally have a scheme, path, and anchor.

A valid scheme will either be `http` or `https`.

Valid paths start with a slash and then must be a valid path to a file or directory. This means they should match something like `/composingprograms.html` or `path /to/file` but not `/composing.programs.html/`.

A valid anchor starts with `#`. While they are more complicated, for this problem assume that valid anchors will then be followed by letters, numbers, hyphens, or underscores.

> **Hint**: You can use \ to escape special characters in regex.

```
import re
def match_url(text):
    """
    >>> match_url("https://cs61a.org/resources/#regular-expressions
    ")
    True
    >>> match_url("https://pythontutor.com/composingprograms.html")
    True
    >>> match_url("https://pythontutor.com/should/not.match.this")
    False
    >>> match_url("https://link.com/nor.this/")
    False
    >>> match_url("http://insecure.net")
    True
    >>> match_url("htp://domain.org")
    False
    """
    scheme = r"(https?:\/\/)?"
    domain = r"\w+\.\w+"
    path = r"(\/\w+)*(\.\w+)?"
    anchor = r"(\/#[\w-]+)?$"
    full_string = scheme + domain + path + anchor
    return bool(re.match(full_string, text))
```

BNF

Backus-Naur Form (BNF) is a syntax for describing a context-free grammar. It was invented for describing the syntax of programming languages, and is still commonly used in documentation and language parsers. EBNF is a dialect of BNF which contains some convenient shorthands.

An EBNF grammar contains symbols and a set of recursive production rules. In 61A, we are using the Python Lark library to write EBNF grammars, which has a few specific rules for grammar writing.

There are two types of symbols: Non-terminal symbols can expand into non-terminals (including themselves) or terminals. In the Python Lark library, non-terminal symbols are always lowercase. Terminal symbols can be strings or regular expressions. In Lark, terminals are always uppercase.

Consider these two production rules:

```
numbers: INTEGER | numbers "," INTEGER
INTEGER: /-?\d+/
```

The symbol `numbers` is a non-terminal with a recursive production rule. It corresponds to either an `INTEGER` terminal or to the `numbers` symbol (itself) plus a comma plus an `INTEGER` terminal. The `INTEGER` terminal is defined using a regular expression which matches any number of digits with an optional - sign in front.

This grammar can describe strings like:

```
10
10,-11
10,-11,12
```

And so on, with any number of integers in front.

A grammar should also specify a start symbol, which corresponds to the whole expression being parsed (or the whole sentence, for a spoken language).

For the simple example of comma-separated numbers, the start symbol could just be the numbers terminal itself:

```
?start: numbers
numbers: numbers "," INTEGER | INTEGER
INTEGER: /-?\d+/
```

EBNF grammars can use these shorthand notations for specifying how many symbols to match:

EBNF Notation	Meaning	Pure BNF Equivalent
item*	Zero or more items	items: | items item

Note: This worksheet is a problem bank—most TAs will not cover all the problems in discussion section.

EBNF Notation	Meaning	Pure BNF Equivalent
item+	One or more items	items: item | items item
[item] item?	Optional item	optitem: | item

Lark also includes a few handy features:

- You can specify tokens to complete ignore by using the ignore directive at the bottom of a grammar. For example, `%ignore /\s+/` ignores all whitespace (tabs/spaces/new lines).
- You can import pre-defined terminals for common types of data to match. For example, `%import common.NUMBER` imports a terminal that matches any integer or decimal number.

Q3: Calculator BNF

Consider this BNF grammar for the Calculator language:

```
?start: calc_expr

?calc_expr: NUMBER | calc_op

calc_op: "(" OPERATOR calc_expr* ")"

OPERATOR: "+" | "-" | "*" | "/"

%ignore /\s+/
%import common.NUMBER
```

Let's understand and modify the functionality of this BNF with a few questions.

Will the following expressions be parsable according to this grammar?

```
(+ 1 2)
```

true

```
(+)
```

true

```
(1)
```

false

```
(+ 1 2 3)
```

true

```
(+ 1)
```

true

```
(1 + 2)
```

false

```
(+ 1 (+ 2 3))
```

true

```
(+ 1 - 2 3)
```

false

Which line of the BNF should we modify to add support for calculations using a modulus operator, like (% 15 5)?

```
OPERATOR: "+" | "-" | "*" | "/"
```

Does the BNF grammar provide enough information to create a working interpreter for this version of the Calculator language?

No, this grammar gives enough information for parsing a Calculator expression, but it does not provide enough information to evaluate it.

Q4: lambda BNF

We've written a simple BNF grammar to handle lambda expressions. The body of our lambda has to consist of a single expression, which can be a number, word, or another lambda expression.

```
?start: lambda_expression
lambda_expression:  "lambda " arguments ":" body
arguments: WORD ("," WORD)*
body: expression
?expression: value | lambda_expression
?value: WORD | NUMBER

%import common.WORD
%import common.NUMBER
%ignore /\s+/
```

For each of the given examples, draw the resulting tree created by this BNF.

```
lark> lambda x: 5
```

```
lambda_expression
  arguments  x
  body  5
```

```
lark> lambda x, y: x
```

```
lambda_expression
  arguments
    x
    y
  body  x
```

```
lark> lambda x: lambda y: x
```

```
lambda_expression
  arguments  x
  body
    lambda_expression
      arguments  y
      body  x
```

Q5: Simple CSV

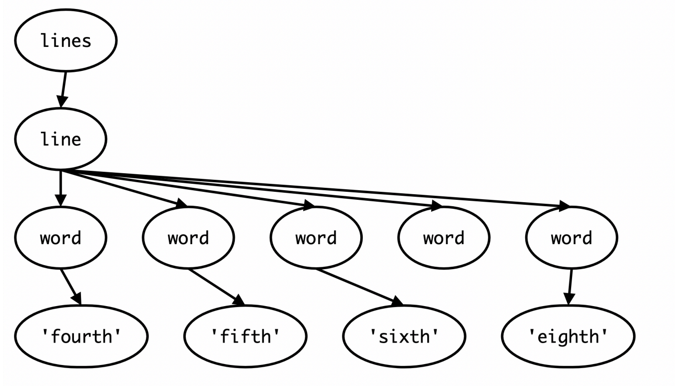
CSV, which stands for "Comma Separated Values," is a file format to store columnar information. We will write a BNF grammar for a small subset of CSV, which we will call SimpleCSV.

Create a grammar that reads SimpleCSV, where a file contains rows of words separated by commas. Words are characters `[a-zA-Z]` (and may be blank!) Spaces are not allowed in the file.

Here is an example of a 2-line SimpleCSV file:

```
first,second,third
fourth,fifth,sixth,,eighth
```

We should parse out the following as a result:



Parse Result

Note: If you want to test a multiline input in 61A Code, you can use the following format:

```
lark> .begin
....> Pressing enter after that first prompt lets you write more
   lines.
....> Keep typing and pressing enter to get the input you want.
....> When you're done, on the last line, you should type:
....> .end
(The output of your multiline input will show up here.)
```

```
?start: lines
lines: (line "\n")* line "\n"?
line: (word ",")* word
word: WORD?

%import common.WORD

%doctest
lark> first,second,third
....> fourth,fifth,sixth,,eighth
lines
  line
    word  first
    word  second
    word  third
  line
    word  fourth
    word  fifth
    word  sixth
    word
    word  eighth
lark> one,,,,three
lines
  line
    word  one
    word
    word
    word
    word  three
lark> ,,,word
lines
  line
    word
    word
    word
    word  word
%end
```