# Problem 1: Interference

Team-ID: 00082

Team-Name: Eww Java

Author(s) of this solution:
Ahmed Sami

20.11.2022

## Contents



The regex engine `https://gitlab.com/add003/rex` forms the core of my solution. I implemented the engine independently in the Go programming language version `1.19.3` without using any software libraries other than the Go standard library.

# 1 Solution Idea

## 1.1 Finite Automata and Regular Expressions

The solution idea can be divided into the following sub-steps:

1. A gap sentence is read in and transferred into a corresponding regular expression.
2. A finite automaton is constructed from the regular expression.
3. The finite automaton traverses the book and outputs all matching positions.

This approach is advantageous because regular expressions can, for example, find all words in a text that start with ‘A‘ and end with ‘e‘ without having to explicitly specify the intervening letters.

### 1.2 Character Intervals and Character Classes

The basis of this implementation, which will be explained in the next part, is Thompson's construction [Tho68]. In addition, we want to introduce **character intervals** and **character classes**. A regular expression describing all digits $0-9$ would only be possible as a tenfold disjunction $(0|1|2|3|4|5|6|7|8|9)$ without character intervals, with each disjunction corresponding to a state with two $\epsilon$-transitions in our finite automaton. For character intervals with ten elements, this does not seem to be a problem, but for our solution approach, it is necessary to describe the gaps of the gap sentence with character intervals consisting of more than 200 elements. A **character interval** $\psi = (lo|hi)$ represents a range of Unicode code points, where $lo$ and $hi$ are integers ($0 \leq lo \leq hi \leq 1114111$). Thus, a regular expression describing all digits $0-9$ can be realized with a single state in the automaton that does not match a character $\sigma$ with a literal character but checks if $\sigma$ lies in the character interval $\psi_d = (30|39)$. The character $\sigma$ lies in the character interval $\psi_d$ if and only if $30 \leq \sigma \leq 39$ holds. The concept of character classes builds on that of character intervals and should enable us to represent disjoint character intervals in a single state. A **character class** $\Psi$ is a set of one or more **disjoint** character intervals $\psi$. A regular expression describing all digits $0-9$ and all letters 'A' to 'Z' can be represented by the character class $\Psi_{d,AZ} = \{(30|39), (65|90)\}$. A character $\sigma$ lies in the character class $\Psi_{d,AZ}$ if and only if it lies in the character interval (30|39) or (65|90).

## 2 Implementation

### 2.1 Gap Sentence to Regular Expression

A regular expression is formed from the characters of a gap sentence as follows:

1. Space in the gap sentence: `[\s\p{P}]+`
   One or more whitespace or punctuation marks.

2. Underscore in the gap sentence: `\w+`
   One or more word characters.

3. All other characters: `string(char)`
   The literal character itself.

### 2.2 Regular Expression to Syntax Tree

To construct a finite automaton from a regular expression, we first set up a syntax tree using a formal grammar, which consists of lexicographic units that we will henceforth call **tokens**. This intermediate step makes sense because we can ensure compliance with the rules of our formal grammar before constructing the automaton and possibly optimize the expression. The **tokenizer** implements, through *recursive descent parsing*, the following grammar:

1/2

```
<Pattern> ::=
  <Disjunction>

<Disjunction> ::=
  <Term>
  <Term> | <Disjunction>

<Term> ::=
  <Factor>
  <Factor> <Term>

<Factor> ::=
  <Assertion>
  <Atom>
  <Atom> <Quantifier>

<Assertion> ::=
  ^
  $
  \ b
  \ B

<Quantifier> ::=
  *
  +
  ?
  { <DecimalDigits> }
  { <DecimalDigits> , }
  { <DecimalDigits> , <DecimalDigits> }
```

2/2

```
<Atom> ::=
  .
  \ <AtomEscape>
  <Class>
  ( <Disjunction> )
  any character but not one of
      ^ $ \ . * + ? ( ) [ ] { } |

<AtomEscape> ::=
  <Control>
  <Perl>
  <HexSeq>
  <UniSeq>

<Control> ::= one of
  f n r t v

<Perl> ::= one of
  d D s S w W

<Class> ::=
  [ <ClassRange> ]
  [ ^ <ClassRange> ]

<ClassRange> ::=
  [empty]
  <ClassAtom>
  <ClassAtom> - <ClassAtom> <ClassRange>

<ClassAtom> ::=
  \ <AtomEscape>
  any character but not one of
      \ ] -
```

## 2.3 Syntax Tree to Finite Automaton

The tokens of the syntax tree are recursively added by the parser to an array of `Regexp` objects, which represents the finite automaton. The different operations such as concatenation, disjunction, conjunction, and repetition are realized by the `Op` attribute of a `Regexp` object.

## 2.4 Running Characters through the Automaton

A character string accepted by the automaton, henceforth called a **match**, is a character string that reaches a `Regexp` state with `Regexp.Op = OpAccept`. A character that matches a state is *consumed* by it. When a character is consumed, the algorithm advances both to the next character of the string and to the next state of the automaton. In this way, the automaton goes through the character string until either the end of the string is reached, the end of the automaton is reached, or the end of the automaton is no longer reachable. The character string is a match if and only if the final state is a `Regexp` object with `Regexp.Op = OpAccept`.

# 3 Examples

## 3.1 BwInf Examples

stoerung0.txt

```
[In]:  ./main alice.txt stoerung0.txt

[Out]: Der Lückensatz 'das _ mir _ _ _ vor' wurde in das Regex Pattern
'das[\s\p{P}]+\w+[\s\p{P}]+mir[\s\p{P}]+\w+[\s\p{P}]+\w+[\s\p{P}]+\w+[\s\p{P}]+vor'
übernommen.

Passende Stellen aus dem Buch:
[1] [19550-19585]: "Das kommt mir gar nicht richtig vor"
```

stoerung1.txt

```
[In]:  ./main alice.txt stoerung1.txt

[Out]: Der Lückensatz 'ich mu _ clara _' wurde in das Regex Pattern
'ich[\s\p{P}]+mu[\s\p{P}]+\w+[\s\p{P}]+clara[\s\p{P}]+\w+'
übernommen.

Passende Stellen aus dem Buch:
[1] [19013-19041]: "Ich mu in Clara verwandelt"
[2] [19673-19697]: "Ich mu doch Clara sein"
```

stoerung2.txt

```
[In]:  ./main alice.txt stoerung2.txt

[Out]: Der Lückensatz 'fressen _ gern _' wurde in das Regex Pattern
'fressen[\s\p{P}]+\w+[\s\p{P}]+gern[\s\p{P}]+\w+'
übernommen.

Passende Stellen aus dem Buch:
[1] [7623-7650]: "Fressen Katzen\ngern Spatzen"
[2] [7652-7679]: "Fressen Katzen gern Spatzen"
[3] [7681-7708]: "Fressen Spatzen gern Katzen"
```

stoerung3.txt

```
[In]:  ./main alice.txt stoerung3.txt

[Out]: Der Lückensatz 'das _ fing _' wurde in das Regex Pattern
'das[\s\p{P}]+\w+[\s\p{P}]+fing[\s\p{P}]+\w+'
übernommen.

Passende Stellen aus dem Buch:
[1] [104905-104922]: "das Spiel fing an"
[2] [147472-147492]: "Das Publikum fing an"
```

stoerung4.txt

```
[In]:  ./main alice.txt stoerung4.txt

[Out]: Der Lückensatz 'ein _ _ tag' wurde in das Regex Pattern
'ein[\s\p{P}]+\w+[\s\p{P}]+\w+[\s\p{P}]+tag'
übernommen.

Passende Stellen aus dem Buch:
[1] [103801-103822]: "ein sehr schöner Tag"
```

## 3.2 Additional Examples

extra0.txt

```
[In]:  ./main alice.txt extra0.txt

[Out]: Der Lückensatz 'arme _' wurde in das Regex Pattern
'arme[\s\p{P}]+\w+'
übernommen.

Passende Stellen aus dem Buch:
[1] [10118-10128]: "arme Alice"
[2] [12869-12879]: "arme Alice"
[3] [13252-13264]: "arme, kleine"
[4] [13830-13840]: "arme Alice"
[5] [16348-16358]: "Arme Alice"
[6] [19599-19609]: "arme Alice"
[7] [21339-21348]: "arme Kind"
[8] [23974-23984]: "arme Thier"
[9] [36983-36993]: "arme Alice"
[10] [38995-39005]: "arme Alice"
[11] [43634-43644]: "arme Alice"
[12] [51212-51223]: "arme kleine"
[13] [60616-60626]: "arme Alice"
[14] [68699-68718]: "Arme hervorzuziehen"
[15] [75582-75593]: "arme kleine"
[16] [76255-76263]: "Arme und"
[17] [76351-76362]: "arme kleine"
[18] [77797-77808]: "arme kleine"
[19] [105386-105401]: "Arme festhalten"
[20] [123260-123270]: "arme Alice"
[21] [141447-141458]: "arme kleine"
[22] [149187-149200]: "Arme gekreuzt"
[23] [150066-150075]: "Arme sind"
[24] [151575-151586]: "arme kleine"
```

extra1.txt

```
[In]:  ./main alice.txt extra1.txt

[Out]: Der Lückensatz '_ Alice das _' wurde in das Regex Pattern
'\w+[\s\p{P}]+Alice[\s\p{P}]+das[\s\p{P}]+\w+'
übernommen.

Passende Stellen aus dem Buch:
[1] [16348-16372]: "Arme Alice! das Höchste"
[2] [71149-71169]: "sich Alice das nicht"
[3] [76333-76355]: "dachte Alice. Das arme"
[4] [77461-77486]: "gefiel Alice das Aussehen"
[5] [99954-99979]: "erkannte Alice das weie"
[6] [109361-109384]: "sagte\nAlice, żdas habe"
[7] [153578-153602]: "sagte Alice.\n\nżDas bist"
```

## References

[Tho68] Ken Thompson. "Programming Techniques: Regular Expression Search Algorithm". In: *Commun. ACM* (June 1968), pp. 419–422. URL: https://doi.org/10.1145/363347.363387.

[Cox07] Russ Cox. *Regular Expression Matching Can Be Simple And Fast*. 2007. URL: https://swtch.com/~rsc/regexp/regexp1.html.

[Cox10] Russ Cox. *Regular Expression Matching in the Wild*. 2010. URL: https://swtch.com/~rsc/regexp/regexp3.html.