ID1020 – Project 2
Tiny Search Engine

1 Organisation

The project is a programming task that is somewhat more challenging than the labs. You are expected to present your solution/code orally. We will assign grades to you based on the quality of your solution and your understanding of the involved concepts.

This is a description of *Project 2 (P2)* which expands on *Project 1 (P1)*.

1.1 Goals

The project has the following goals:

- Work with the algorithms and data structures presented in the course
- Reason about usage patterns in a software system and leverage this to make implementation decisions
- Work on a real problem within the context of the course
- Get an idea of CFGs and their "real" applications

2 Background

The goal of the project is to build a simple search engine for natural language text documents.
When completed the search engine will support the following operations:

- Index documents based on their content.
- Find and list documents which contain a single provided key word.
- Order search results by different properties (e.g. relevance, popularity).
- Use a *query language* to support more involved searches.

The documents themselves are provided in a specific format as part of the project framework and you don't have to worry about processing the raw files. However, it is important to notice that the given documents are english language texts of various types (e.g. short stories, newspaper articles, etc) and natural language texts differ in their treatment from more formal sources like programming languages, for example. The specific documents we provide are from the so called Brown Corpus[1]. To give you some quick context, we provide a short summary of the field of natural language processing in section 2.1. Furthermore, the query language in the last task will require you to understand the concepts of parsing a limited syntax and to that end section 2.2 introduces the concept of context-free grammars and their relationship with parsing syntax.

2.1 Natural Language Processing

Natural language processing (NLP) is a field of computer science, artificial intelligence, and linguistics concerned with the interactions between computers and human (natural) languages. To a large extent the purpose of NLP is to allow computers to derive and subsequently leverage information from documents written for and by humans. This information could be anything from simply observing word usage pattern in certain kinds of documents, to understanding and classifying the content of said documents on a semantic basis. NLP has a number of sub-fields of which the following might be of interest to our search engine:

Part-of-speech tagging Given a sentence, determine the part of speech (POS) for each word. Many words, especially common ones, can serve as multiple parts of speech. For example, "book" can be a noun ("the book on the table") or verb ("to book a flight"). Some languages have more such ambiguity than others. Languages with little inflectional morphology, such as English are particularly prone to such ambiguity. POS tagging is a necessary pre-requisite in most cases for any kind of syntactical analysis.

Parsing Determine the parse tree (grammatical analysis) of a given sentence. The grammar for natural languages is ambiguous and typical sentences have multiple possible analyses. In fact, perhaps surprisingly, for a typical sentence there may be thousands of potential parses (most of which will seem completely nonsensical to a human).

Sentence breaking (also known as sentence boundary disambiguation)
Given a chunk of text, find the sentence boundaries. Sentence boundaries are often marked by periods or other punctuation marks, but these same characters can serve other purposes (e.g. marking abbreviations).

Word sense disambiguation Many words have more than one meaning; we have to select the meaning which makes the most sense in context. For this problem, we

[1] See http://en.wikipedia.org/wiki/Brown_Corpus for further reading, if you are interested.

are typically given a list of words and associated word senses, e.g. from a dictionary or from an online resource such as WordNet.

If you are interested, http://en.wikipedia.org/wiki/Natural_language_processing give a good overview of the field.

2.2 Context-free Grammars

Context-free Grammars (CFGs) have their roots in formal language theory and were originally invented in an attempt to formalise the grammatical structure of natural languages. Since it has been shown that not all natural languages can be described by CFGs they have somewhat lost their importance in the NLP field. However, they retain immense importance in computer science as they form the basis of all programming languages (and also query languages, which is why we introduce them here).

Definition Formally a CFG is defined by a set of *production rules* of the form:

$$S \to s_1 s_2 \dots s_n \text{ for } n \in \mathbb{N}_0$$

where S is a *nonterminal* symbol and the s_i are either nonterminal or *terminal*, meaning that s_i is a *token* of the alphabet of the language. Terminal symbols may never appear on the left-hand side of a production rule and each nonterminal symbol has to appear on at least one left-hand side of some production rule.

Example Consider the following grammar for simple arithmetical expressions:

$$\begin{aligned}
&1 : E \to num \text{ where } num \in \mathbb{R} \text{ for example} \\
&2 : E \to E \cdot E \\
&3 : E \to \frac{E}{E} \\
&4 : E \to E + E \\
&5 : E \to E - E \\
&6 : E \to (E)
\end{aligned}$$

Also consider the following example sentence s in the language of arithmetical expressions $s = 2 \cdot (3 + 4)$.

Definition In order to show that a sentence is part of the language of a CFG we have to find a *derivation*, that starting with the start symbol and applying production rules to the right hand hand side, we arrive at the sentence.

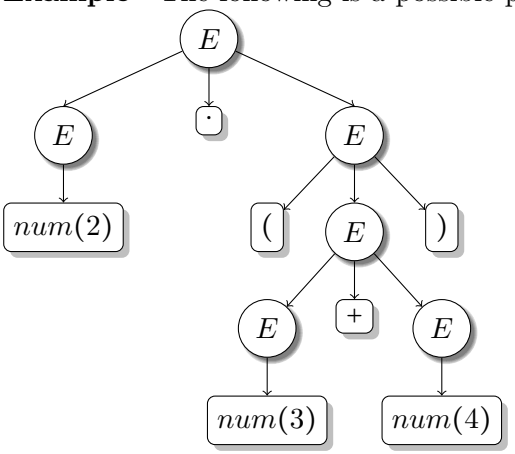
Example For the grammar above and sentence s the following would be a possible derivation:

E	start symbol
$E \cdot E$	rule 2
$num(2) \cdot E$	rule 1
$num(2) \cdot (E)$	rule 6
$num(2) \cdot (E + E)$	rule 4
$num(2) \cdot (num(3) + E)$	rule 1
$num(2) \cdot (num(3) + num(4))$	rule 1

Since we have derived s from the start symbol we have shown that s is part of the given grammar.

Definition While it is not difficult for a human to find a derivation that matches a sentence, a program would have try a (possibly large number) of different derivation until it by chance finds one that matches the input sentence. Since this is not feasible, programs usually apply the opposite process to derivation, which is called *parsing*. The result of the parsing process is a *parse tree* with the start symbol on top and the syntactical structure of the sentence represented in the structure of the tree. If parsing consumed all the symbols of a sentence that sentence is part of the language of the grammar.

Example The following is a possible parse tree for s:



Note The quick introduction to CFGs presented here is certainly far from exhaustive. There are important issues of ambiguity and associativity of operations we have not considered at all so far. However, it should be sufficient to perform the required tasks. If you require more information or are simply interested feel free to look at http://en.wikipedia.org/wiki/Context-free_grammar and follow references from there.

2.3 Notations

There are different types of notations used in logical/arithmetic formulas. In the next paragraphs we will discuss two of the notations which are relevant to our project.

Infix Notation This is the most common notation where you have an operator in-between its operands for example: $3 + 5$ you have an operator $+$ with two operands $3, 5$.

Prefix Notation In this notation you'll have the operator precedes the operands for example: $+3\ 5$ which is equivalent to $3 + 5$ in *infix notation*. A bit more complex example with nested expression :

```
+  -  3  5  +  4  9    is equivalent to    (3-5) + (4+9)
```

3 Helper Code

In order for you to focus on implementing the tasks, we are supplying a Java code skeleton which contains the basic functionality for you to start coding. It encapsulates the reading and parsing of the pre-tagged words from the Brown Corpus (see section 2) and supplies sentence by sentence using a *SentenceHandler* interface.

There are four main classes that you should know:

Word encapsulates the word itself and the POS (verb, noun, adverb, etc)

```
1 public class Word {
2     public final PartOfSpeech pos;
3     public final String word;
4 }
```

Sentence consists of a list of words

```
1 public class Sentence {
2     List<Word> words;
3 }
```

Attributes encapsulates the attributes that a word could have: a reference to the document in which the word was found and the occurrence (in number of preceding words) of the word in that document.

```
1 public class Attributes {
2     public final Document document;
3     public final int occurrence;
4 }
```

Document contains the name of the document and the popularity (think "number of views").

```
1  public class Document {
2      public final String name;
3      public final int popularity;
4  }
```

You are required to create your own *TinySeachEngine* which implements the *TinySearchEngineBase* interface.

```
1  public interface TinySearchEngineBase {
2      //Prepare to build the index
3      public void preInserts();
4      //Build the index
5      public void insert(Sentence sentence, Attributes attr);
6      //Finish up building the index
7      public void postInserts();
8      //Searching
9      public List<Document> search(String query);
10     //Convert Prefix into Infix
11     public String infix(String query);
12 }
```

The two most important methods are:

insert: adds the sentence with the given attribute to your index.

search: returns the list of documents that matches the query.

Additionally you have, **preInserts** and **postInserts** which you can use for setup purposes if you need to. Otherwise just leave the body of your implementation empty. Lastly a new method **infix** is used to print an infix notation version of your search query. You should use this to double check that your parser interpreted the query correctly. So please use the same code as your parser does here. No copy&paste; reuse your code! If you do not run exactly the same code, the method will be useless.

3.1 Setup

First you need to create a maven project and add our helper project as a dependency in your *pom.xml* as follows:

```
<dependencies>
    <dependency>
        <groupId>se.kth.id1020</groupId>
        <artifactId>tinySearchEngine</artifactId>
        <version>2.0</version>
    </dependency>
</dependencies>

<repositories>
    <repository>
        <id>sics-release</id>
        <name>SICS Release Repository</name>
        <url>http://kompics.sics.se/maven/repository</url>
    </repository>
</repositories>
```

In your main method you need to call *Driver.run* which will starts reading the documents and will call *insert* in turns to build the index, also it will start a search REPL "read-eval-print-loop". To exit the REPL you can just write *?exit* in front of search.

```
1  public static void main(String[] args) throws Exception{
2      TinySearchEngineBase searchEngine = new TinySearchEngine();
3      Driver.run(searchEngine);
4  }
```

Examples:

```
Building the index done in 1 seconds
Search:
```

As you can see first it prints that the indexing process is done in X seconds, and now on the next line you can enter your search query infront of search and hit enter for example, let's try to search for *nightmare* and sort the result by popularity in an ascending order.

```
Search: nightmare orderby popularity asc
Infix: nightmare orderby popularity asc
got 7 results in 157 microseconds
Document{cr04, pop=23661855}
Document{cp16, pop=314138678}
Document{ca04, pop=421696227}
Document{cl13, pop=728880313}
```

```
Document{cl23, pop=899544007}
Document{cc05, pop=1192381167}
Document{cf09, pop=1909872348}
Search:
```

4 Query Language

In this project you are going to extend the query language from project 1. We define a query language that should support prefix notation 2.3 which is more convenient and simpler to parse.

4.1 Operators

The query language should support the following operators:

Intersection (+) : returns all the documents containing both terms or more generally the intersection of both sub-queries results.

Union (|): returns all the documents that contain either one of the terms or more generally the union of both sub-queries results.

Difference (-): returns all the documents that contain the first term but not the second one, or more generally the set difference between the two sub-query results.

Ordering (orderby): order the final result by the popularity of the document or by the relevance of the document to the query. We will discuss the relevance computation in detail in 4.2.

Formal Definition A more formal definition of the query language is described by the following CFG:

$E \rightarrow T$	simple query
$E \rightarrow T$ *orderby Property Direction*	ordering
$T \rightarrow$ *word*	a search term
$T \rightarrow + \ T \ T$	intersection operation
$T \rightarrow \mid T \ T$	union operation
$T \rightarrow - \ T \ T$	set difference operation
Property $\rightarrow$ *relevance*	how relevant is the document?
Property $\rightarrow$ *popularity*	how popular is the document?
Direction $\rightarrow$ *asc*	increasing order
Direction $\rightarrow$ *desc*	decreasing order

According to our description of CFG in 2.2, In the above CFG, E is a *non-terminal* expression and T could be *terminal* or *non-terminal*. E is an expression consists of a set of tokens (words) that might end up by *orderby Property Direction* or not which is described in the first two production rules above. Similarly, T consists of a a single token or a set of tokens prefixed with an operator as described in the production rules above.

Note: We replaced the ($T \rightarrow T\ T$) rule from the first project with ($T \rightarrow \mid T\ T$) which means that we don't support using the space as an operator. In other words, T only consists of a single word also you can ignore the punctuation.

Examples: Let's have a more complex query that follows the query language described above.

```
+ + | nightmare stone | metaphysical stuck - dark night orderby  popularity  desc
```

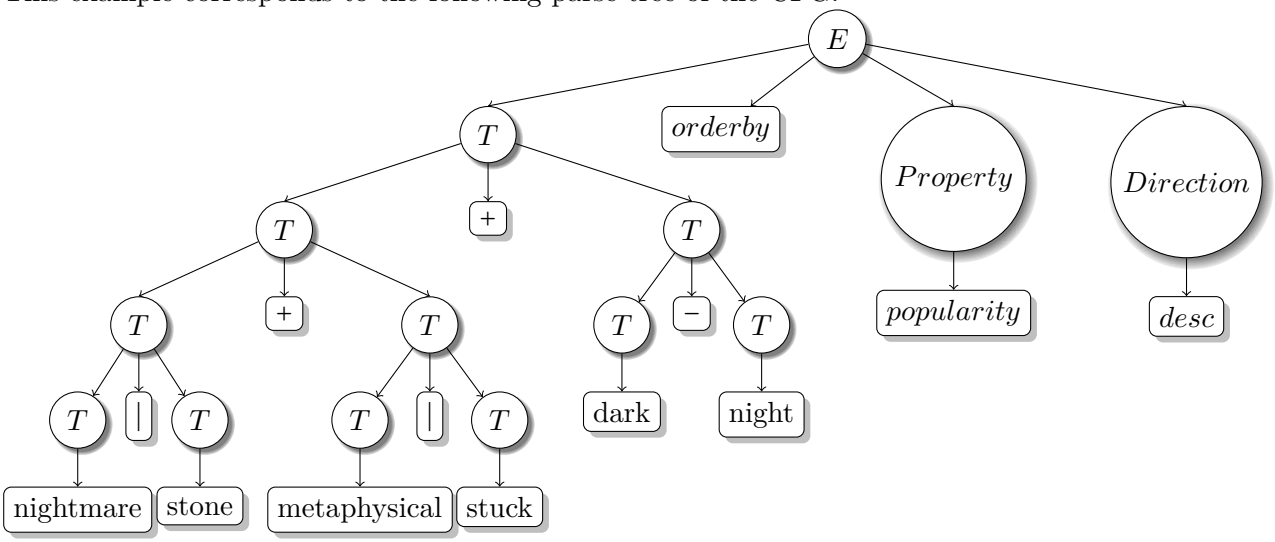
Now let's try to convert this query from *prefix notation* to *infix notation* for clarity, first let's add parentheses for each operation we see in the query.

```
(+ (+ (| nightmare stone) (| metaphysical stuck)) (- dark night)) orderby  popularity  desc
```

After adding the parentheses it became easy to convert it to *infix notation* as shown below:

```
(((nightmare | stone) + (metaphysical |  stuck)) + (dark - night)) orderby  popularity  desc
```

This example corresponds to the following parse-tree of the CFG:



Let's try to analyze the query above by substituting sub queries with a named variable for simplicity and reference:

```
A = (nightmare | stone)
B = (metaphysical |  stuck)
C = (dark - night)

(( A + B ) + C orderby  popularity  desc
```

A search for all documents that either contain `nightmare` or `stone`

B search for all the documents that either contain `metaphysical` or `stuck`

C search for all the documents that contain `dark` but not `night`

A+B+C combine the results from `A` and `B` and `C`, that is all documents that contain either `nightmare` or `stone` and either `metaphysical` or `stuck` and `dark` but not `night`

By the running this query in your search engine, it should gives the following results:

```
Search: + + | nightmare stone | metaphysical stuck - dark night orderby popularity desc
Infix: Query((((nightmare | stone) + (metaphysical | stuck)) + (dark - night)) ORDERBY POPULARITY DESC)
got 1 results in 0m 0s 36ms 955µs 160ns
Document{cn08, pop=1765493201}
```

Another example by replacing the only `-` with $+$ in the query, we got the following results:

```
Search: + + | nightmare stone | metaphysical stuck + dark night orderby popularity desc
Infix: Query((((nightmare | stone) + (metaphysical | stuck)) + (dark + night)) ORDERBY POPULARITY DESC)
got 1 results in 0m 0s 1ms 425µs 62ns
Document{ck02, pop=453330457}
```

4.2 Relevance

Previously in the first project, we used term frequencies to calculate relevance of documents to the query. Now, let's consider a more meaningful metric ***term frequency-inverse document frequency(tf-idf)***.

term frequency measures how frequent the search term appears in the document. The more often, the more relevant.
It is given by $\text{tf}(q,d) = \frac{n(q,d)}{T(d)}$ where $n(q,d)$ is how many times q appeared in document d, and $T(d)$ is the total number of terms (words) in d.

inverse document frequency measures how often the search term appears in the index; whether the term is common or rare across all documents. The more often, the less relevant.
It is given by $\text{idf}(q) = \log_{10}(\frac{N_D}{n_D(q)})$ where N_D is the total number of documents, and $n_D(q)$ is the number of documents that contains q.

Combing both equations, we get the tf-idf equation as follows:

$$\text{tfidf}(q,d) = \text{tf}(q,d) \cdot \text{idf}(q) = \frac{n(q,d)}{T(d)} \cdot \log_{10}(\frac{N_D}{n_D(q)}) \tag{1}$$

calculating the document relevance ρ for a query Q is just the sum of all underlying tfidf for each search term in the query

$$\rho(Q,d) = \sum_{q \in \tau(Q)} \text{tfidf}(q,d) \text{ where } \tau(Q) \text{ is the set of search terms in } Q \tag{2}$$

For example, if our query is `(+ nightmare stone)` then the relevance ρ for a document d would be

$$\begin{aligned}
&\rho((+ \text{ nightmare stone}), d) = \text{tfidf}(\text{nightmare}, d) + \text{tfidf}(\text{stone}, d) \\
&= \frac{n(\text{nightmare}, d)}{T(d)} \cdot \log_{10}\left(\frac{N_D}{n_D(\text{nightmare})}\right) + \frac{n(\text{stone}, d)}{T(d)} \cdot \log_{10}\left(\frac{N_D}{n_D(\text{stone})}\right)
\end{aligned}$$

Operators and Relevance:

Intersection(+) you should sum up the documents relevance of search terms, or more generally of sub queries.

Union(|) you should sum up the documents relevance of search terms, or more generally of sub queries.

Difference(-) you should consider only the relevance of the first operand.

5 Tasks

5.1 Indexing – 20P

Use *HashMap*s to implement an optimized version of your indexing solution from the first project. Additionally, you could use any other data structures to supplement your implementation.

5.2 Queries – 60P

1. Implement the query language described in section 4. **(20 P)**
2. Add support for nested queries in your query parsing as described in section 4.1 **(10 P)**
3. Implement the relevance ordering described in section 4.2. **(15 P)**
4. Implement the *infix* method in *TinySearchEngineBase* to convert from *prefix notation* to *infix notation*. **(15 P)**

Note: You are allowed to use Collections.sort for sorting your result query.

5.3 Caching Sub-Queries - 20P

In complex queries you'll end up by having a lot of nested sub-queries. In this task you are required to implement a simple mechanism to cache sub-queries results. For example, if a user searched first for (| nightmare stone) and (| metaphysical stuck), then for (+ | nightmare stone | metaphysical stuck), the later query would be faster if we have cached the results of the first two queries.

Note: For full points, you should consider the commutativity of (+) and (|) operations while designing your cache, for example (| metaphysical stuck) is the same as (| stuck metaphysical), otherwise **10P** deduction.