04.07.2020 Mikołów

SENTENCE GENERATION

CLP PROJECT REPORT

Krzysztof Kramarz

Topic of the project

For my project i've chosen "Generation of random grammatically good sentences"

Short description

Problem of sentence generation is being widely researched by many NLP enthusiasts.
Let's say you are writing an article with many paragraphs in it. And you do not want to waste time to name every one of them. With sentence generation you can create headlines for those paragraphs automatically, based on a "dictionary" of words that were used in them. I've based my idea on NLTK context-free grammar. However, as this idea was not ideal for me I've made some twists around their idea.
Context-Free grammar is a pretty easy and straight-forward idea. We provide the generator all possible sentence variations, ex. sentence = Noun + "is walking" or sentence = Noun + "walk". Additionally we provide the code with some dictionary of words that can be used in a given group, ex. NOUNS = {"man","dog", "tree" }.

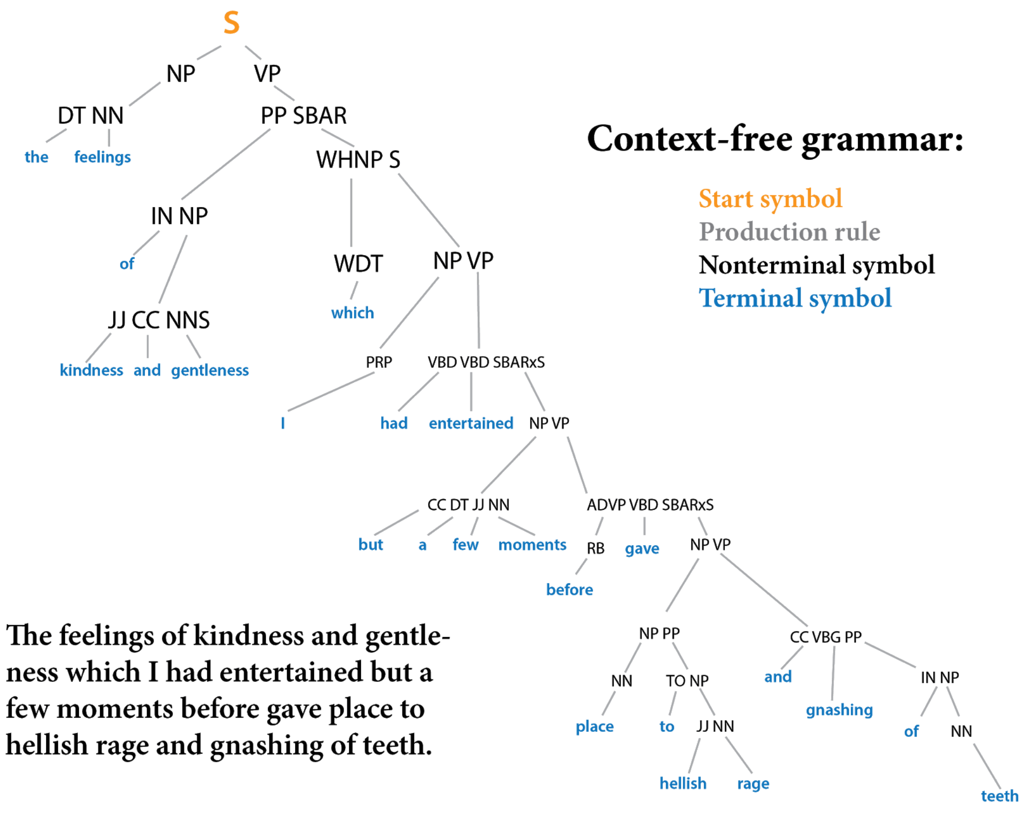


Fig 1. Graphical explanation of context-free grammar.

My Idea was to use CLP for generation of such possible variations. Also it would be nice if the solver would choose proper tenses (past-simple, past-continuous) and use proper rulers and verbs.

Implementation

First I've managed to create a solver that creates all possible grammatical structures of sentences based on provided grammar and maximal sentence length. Right now grammar is hardcoded, but it will be possible to pass it as an argument for the solver in .json format. Grammar has a structure of so-called recursive trees. Sentence starts with random word type. Each type has some other types that can follow it. I have also added some original constraints to be sure that each sentence has judgment and entity.

```
    //creating our grammar
            afterNoun = new IntVar(store, "after noun", 2, 2);
            //afterNoun.addDom(6,6);
            afterNoun.addDom(7,7);

            afterVerb = new IntVar(store, "after verb", 1, 1);
            afterVerb.addDom(3,3);
            afterVerb.addDom(4,4);
            afterVerb.addDom(5,5);
            //afterVerb.addDom(6,6);
            afterVerb.addDom(7,7);

            afterAdjective = new IntVar(store, "after adjective", 1, 1);
            afterAdjective.addDom(6,6);
            //afterAdjective.addDom(7,7);

            afterPrefix = new IntVar(store, "after Prefix", 1, 1);
            afterPrefix.addDom(3,3);
            afterPrefix.addDom(5,5);

            afterAdverb = new IntVar(store, "after Adverb", 3, 3);
            afterAdverb.addDom(5,5);
            //afterAdverb.addDom(7,7);


            afterConjunction = new IntVar(store, "after Conjunction", 3, 3);
            //afterConjunction.addDom(4,4);
            afterConjunction.addDom(5,5);
            endOfTheSentence = new IntVar(store, "end of the sentence", 7, 7);

            grammar = new IntVar[7];
            grammar[0] = afterNoun;
            grammar[1] = afterVerb;
            grammar[2] = afterAdjective;
            grammar[3] = afterPrefix;
            grammar[4] = afterAdverb;
            grammar[5] = afterConjunction;
            grammar[6] = endOfTheSentence;

```

```
         //creating sentence
         sentence = new IntVar[sentenceLenght];
         for(int i = 0; i < sentenceLenght; i++){
             sentence[i] = new IntVar(store, "word " + i, 1,grammar.length);
         }


         //creating logic for sentences
         for(int i = 0; i < sentenceLenght - 1; i++){
             store.impose(new Element(sentence[i], grammar, sentence[i+1]));
         }
        store.impose(new XeqY(sentence[sentenceLenght - 1], endOfTheSentence));

        store.impose(new XneqY(sentence[0], new IntVar(store, "wrong start", 2, 2)));
        store.impose(new XneqY(sentence[0], new IntVar(store, "wrong start", 6, 6)));
        store.impose(new XneqY(sentence[0], new IntVar(store, "wrong start", 7, 7)));
        // constrain for checking if at least one verb and noune are in the sentence
        // (verb, noun) in S
        PrimitiveConstraint[] c1 = new PrimitiveConstraint[sentenceLenght];
        PrimitiveConstraint[] c2 = new PrimitiveConstraint[sentenceLenght];
        for (int i = 0; i < sentenceLenght; i++){
            c1[i] = new XeqY(sentence[i], new IntVar(store, "wrong start", 1, 1));
            c2[i] = new XeqY(sentence[i], new IntVar(store, "wrong start", 2, 2));
        }
        store.impose(new Or(c1));
        store.impose(new Or(c2));
```

Code 1. Main part of declaration of constraints in CLP solver. One can see that it can be easily changed into a For loop that is reading grammar rules from a file, or IO stream. It is done like this to show the mathematical idea behind the solution.

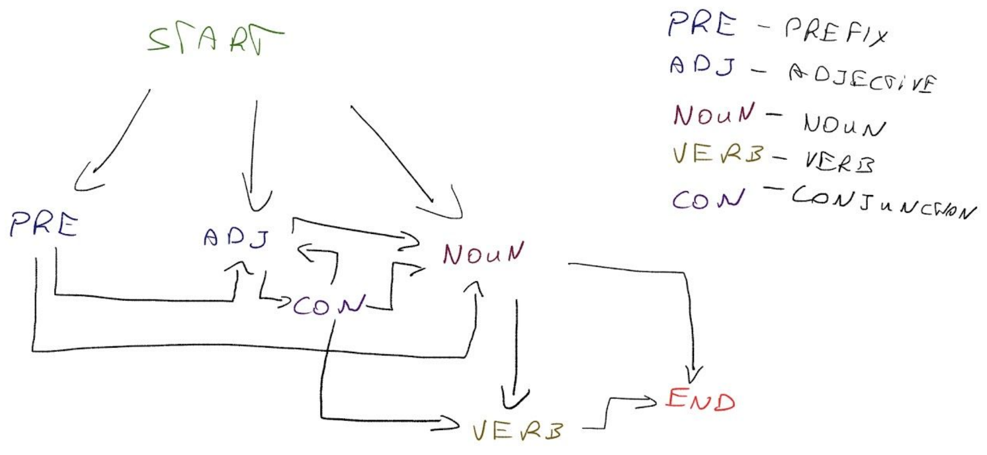


Fig 2. Example of recursive tree grammar.

CLP solver returns all grammatically possible structures of sentences. But these are just numbers, not something readable for us, casual humans. Thus I have implemented a simple algorithm for changing numbers into correct words. Each word is represented by class Word, which stores its string representation, integer representing its type and integer for person (ex. First person “I”, Second person “You” etc.).

```java
public class Word {
    public int type;
    public int person;
    public String wordString;

    public Word(String Word, int Type, int Person){
        person = Person;
        type = Type;
        wordString = Word;
    }
}
```

Code 2. Word class

Such words are stored in a list. To make my code as close to logic programming as possible, choosing proper random words from the list is done by lambda functions.

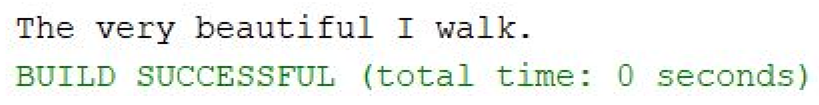
```java
public class Dictionary {

    private List<Word> dictionary = List.of(new Word("I", 1, 3),
                                  new Word("dog",1, 3),
                                  new Word("man",1, 3),
                                  new Word("you",1, 2),
                                  new Word("I",1, 1),
                                  new Word("the",4, 1),
                                  new Word("the",4, 3),
                                  new Word("walks",2, 3),
                                  new Word("walk",2, 1),
                                  new Word("walk",2, 2),
                                  new Word("and",6, 1),
                                  new Word("and",6, 2),
                                  new Word("and",6, 3),
                                  new Word("very",5, 1),
                                  new Word("very",5, 2),
                                  new Word("very",5, 3),
                                  new Word("beautiful",3, 1),
                                  new Word("beautiful",3, 2),
                                  new Word("beautiful",3, 3),
                                  new Word("love",1, 3),
                                  new Word("sun",1, 3),
                                 new Word("shines",2, 2));
    public Dictionary(){}

    public List filterByType(int type, int person){
        Predicate<Word> byType = word -> word.type == type && word.person == person;
        List<Word> result =
dictionary.stream().filter(byType).collect(Collectors.toList());
        return result;
    }
```

As one can see, some words are duplicated. It is because some of them belong to more than one group. I've decided to solve this problem like the M:N database relation.

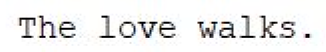
MyApp.main() can be completely refactorized to another class, which could be a part of a bigger application. But in this project I wanted to make research, thus it is written in such a way.

Conclusions and possible future innovations

I am pretty satisfied with current progress and I am impressed how fast my generator is. Even if I have not used any especially hard-to-compute constraints (mostly Element), the solver still needs to check thousands of possibilities. By constraining each sentence with the fact that it needs to have at least one verb and noun I've got really satisfying results.



```
The very beautiful I walk.
BUILD SUCCESSFUL (total time: 0 seconds)
```



```
The love walks.
```

Fig 3. and 4. Examples of program output.

Thanks to this project I have a small private library for NLP and logic programming concerning sentence generation. Big merit for it is that it can be easily modified. One of many possible modifications will be tenses control, which right now is extremely easy to add. All that needs to be done is to create variable "tense" in class "Word" and to add more words into the dictionary.

During the following free time I will definitely experiment around creating more strict grammar concerning tenses, also I will make it more usable for casual users. As this is the final report from the CLP subject, my future work can be seen on my github: https://github.com/Fakser, as well on my Linkedin: https://www.linkedin.com/in/krzysztof-kramarz-6013b81a2. Repository for this project will be created In a couple of days. If there is any interest in some future cooperation I encourage contacting me via email krzysztof.kraamarz@gmail.com, or any other source like already mentioned Linkedin/Github.