# A declarative language for generating customized text processing and tagging applications

Paola Ortega Saborío*, Rodrigo Acuña*, Milton Lara*, Rodrigo A. Bartels* and Edgar Casasola*
*School of Computer Science and Informatics
University of Costa Rica, San Pedro, San José, Costa Rica

***Abstract*—This paper presents the preliminary results of the design and implementation of a declarative language that will allow linguists and researchers in the Natural Language Processing field to generate customized text processing and tagging applications without having to learn how to code or require a computer scientist. The language will allow the orchestration of different functions and transformations that are commonly used when dealing with text in the context of Natural Language Processing and its fields, such as Sentiment Analysis and Information Retrieval. The language compiler will generate a ready to use Java application implementing the user defined flow.**

## I. Introduction

Within the field of Natural Language Processing (NLP), text processing and tagging is a day to day task. The selection of algorithms, techniques and operations applied to the text can greatly affect the obtained results [1]. Because of this, most researchers working in NLP often use or develop computational tools for text processing, parsing and tagging. However this process can be cumbersome, as usually the help of a computer scientist is needed.

This paper presents the preliminary results of creating a declarative language that will allow research in NLP to easily generate custom programs for text processing and tagging. The language provides features for producing the tagged results in different formats. It allows the customization of the tokenization process, defining which dictionary should be used for word looking. Lastly, it allows the definition of any number of transformations to be applied to each of the tokens extracted from the text. The language allows a researcher to orchestrate the application of different transformations in order to easily process text in day to day NLP tasks.

The article is organized as follows. First, a description of the main concepts required for understanding this work is presented. Then, a small recollection of the main related work is shown. Next, the problem that motivates this research work is stated. Then the platform architecture, the expected user flow, is shown and some implementation details are discussed. Afterwards, the designed language is described and its main components are explained with some small examples. Finally, the conclusions and future work are presented.

## II. Context

The field of computational linguistics (CL), together with its engineering domain of natural language processing, has emerged as one of the main research areas due to the amount of text and content generated with the wide spread of the Internet [2]. No matter the subfield of NLP (Information Retrieval, Sentiment Analysis, Entity Recognition) there are two tasks that always need to be done: text processing and part-of-speech tagging [1].

Indurkhya et al. define text processing as: "the task of converting a raw text file, essentially a sequence of digital bits, into a well-defined sequence of linguistically meaningful units: at the lowest level characters representing the individual graphemes in a language's written system, words consisting of one or more characters, and sentences consisting of one or more words." This includes tasks like stop words, spacing and accents removal, tokenization and stemming [3].

Part-of-speech (POS) tagging tries to label each word with its correct part of speech [1]. This process is fundamental for more complex natural language processing tasks like automatic translation and sentiment analysis. Figure 1 shows an example of POS. The phrase: "I bought a present for her yesterday." is analyzed and split in tokens, and each is labeled with its grammatical function within the sentence. Due to the importance of these two tasks, lots of work has been done trying to make them as efficient, accurate and easy to use as possible. Some related efforts are described next.

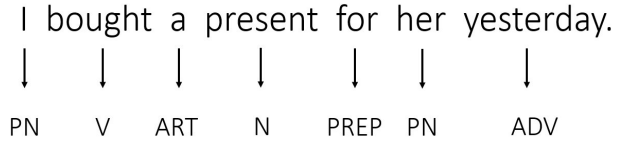


Fig. 1. Example of part-of-speech tagging.

## III. Related Work

Most of the work on the development of tools to be used by linguists has been focused on the creation of frameworks and tools that can be installed and used through user interfaces. Gate [4] is a well-known example of this. Other efforts focused on the creation of libraries that can be reused, but in order to use them programming experience is usually required; FreeLing [5] and Solr [6] are good examples.

To the best of the authors' efforts, a similar initiative where a declarative language is developed to generate programs as the output of the language compiler has not been widely published, as none of the queries posed on the most used libraries (ACM, IEEE, Google Scholar) showed significant results.

## IV. Problem and Motivation

Text processing and tagging are an unavoidable part of any Natural Language Processing task [3]. Several approaches are commonly used to address this. First, a linguist and a computer scientist might work together, the former providing all the linguistic analysis and the latter implementing tools for preprocessing, text enrichment, etc. Another approach consists on any of the two acquiring knowledge from the other field, so in that case there are linguists learning how to code and implementing tools themselves, or computer scientists studying and learning linguistics. Lastly, from the previous approaches some tools are usually developed, expanded and made available both for free or as commercial tools.

The problem with the first two approaches is that it is time consuming and not always possible. It essentially requires at least two resources for doing anything. The problem with the third approach is that available tools are seldom configurable and adaptable to the requirements and experiments that the linguist wants to do. So, in the end a choice has to be made between using a tool for performance reasons, having to modify the designed experiments, or keep the designed methodology but working with a custom developed tool or by hand. Even if the tool accommodates to the researcher needs, the implementation of different scenarios is usually complex and time consuming.

The motivation behind this work is to allow linguists to generate their own text processing and tagging programs without having to learn how to code in Java or any other imperative language. This usually requires several courses in programming and is not customary for linguists to achieve the level of knowledge necessary to develop efficient tools. We propose a declarative language, following the same idea behind the development of SQL in the 70s [7], that will easily allow a linguist to choose from a set of predefined functions, formats and if necessary to orchestrate the preprocessing and enrichment functions performed over the text from a set of transformation operations.

## V. Architecture & Implementation

The architecture of the proposed language is shown in Figure 2. The user will submit language statements to the compiler. The compiler will process each element of the statement and will generate its output as a completely independent and self-sufficient Java program (an executable jar package). This newly generated program receives a text to process and will follow the properties and flow stated by the user in the original language statement.

The compiler is implemented using JFlex and Cup [8]. The output of the compiler is a Java program composed of a main executable class, along with a set of packaged libraries for each of defined components: text reading (file, console, url), text preprocessing (stopwords, stemming, tokenization), enrichment (hashtag and emoticon expansion, duplicated words, idiomatic phrases) and the tagging output (xml, json, text). The idea is to developed a framework with auto discover components (probaby using OSGi[1]), so the end user can automatically expand the set of available features just by dropping a jar into a predefined folder. Components will be identified by a type and artifactId, which can be referenced later by the user in their sentences.

Once the user's statement has been validated and verified by the compiler, the program automatically generates a Java package. This executable jar is the user-defined preprocessor and tagger. The user is able to execute this jar and pass a source text. As a result, the outcome of the executable is a new file in the tag format specified by the user, where each token is preprocessed and tagged. The driving force behind the language design phase was to make it as simple as possible. Next, the main components of the language are presented.

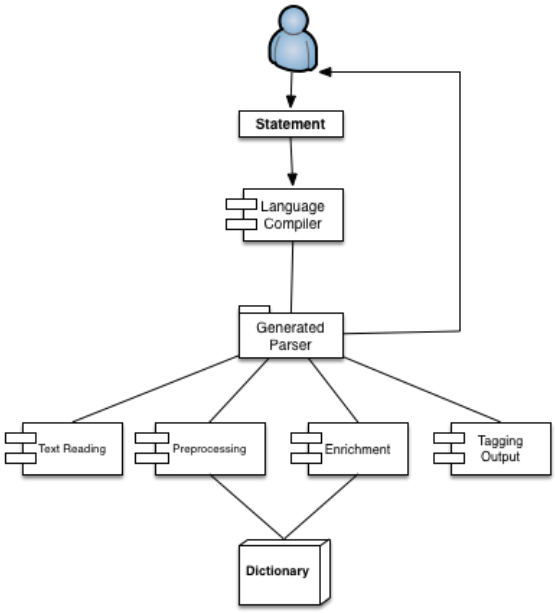


Fig. 2. Architecture of the proposed platform. The end user submits language statement and gets a Java parser back.

## VI. Language Specification

The designed language falls within the category of declarative languages, whose aim is to reduce the level of coding and technical expertise needed to interact and create language statements [7]. To make it easy to learn, the language's reserved words are similar to commonly used linguistic terms. Table I shows the main language keywords.

The language allows a user to specify the source for the tagged text and the format of the tagged output text. Out of the box, FILE, CONSOLE and URL sources are supported and JSON, XML and TEXT output formats are supported. The user is able to DEFINE the name of the generated jar file. The user

[1]The OSGi specification describes a modular system and a service platform for the Java programming language that implements a complete and dynamic component model. Management of Java packages/classes is specified in great detail.

is also able to define which type of TOKENIZER is going to be used by the generated preprocessor.

TABLE I
LANGUAGE RESERVED WORDS

| Keywords | |
|---|---|
| TAG FROM | USE |
| DEFINE | TRANSFORM() |
| FORMAT | IF() THEN |
| TOKENIZER | SWITCH() |

There are two types of functions which give the language its text processing capabilities. The first type are parsers which try to recognize a string of text received from the tokenizer against a predefined dictionary. For this type the user can specify which dictionary to USE. The other type of function are transformer functions, that operate on a token and might produce a new string of text as a result of applying the transformation. The language is designed in a modular fashion such that if any new transformer or parser function is implemented, it can be added easily to the language and be used with the TRANSFORM(function) command. Transformations also allow adding custom variables to be mapped to the output. These are added if the transformation was successfully applied.

There are two other conditional structures that might affect the execution flow of the tagging process. First, an if statement (without an else clause), which verifies if a predefined variable took an specific value in a previous transformation. The body contained inside the if clause should be executed only for tokens that have a "variable" with a specific desired "value". The other possible conditional is a switch statement. It receives as parameter a predefined variable. It follows the usual logic from a switch in most imperative languages. Depending on the value, different actions will be taken. We present now a couple of examples to show how the language would work.

## VII. EXAMPLES

In this section we present two small examples to show possible use cases of the language and its potential as a tool for creating custom taggers without requiring advanced programming skills.

### A. Example 1

The first example is shown in Figure 3. It features a statement that will generate a basic parser that will tokenize by removing white spaces. Line 1 states that the source for the text to be tagged comes from a file. For the generated program this means that it will expect a file path as a parameter of the program or it will prompt in the console for one. Line 2 defines the generated program name. Line 3 defines the format of the output of the program as XML, where the root node name will be in this case "experiment1" (name implies the program's name). The values in parenthesis are the nodes that will be added to each of the token nodes in the output. Line 4 defines that the text will be tokenized using white spaces as the delimiter character. Line 5 defines that the default dictionary will be used when looking for the tokens. Lastly, line 6 defines the only transformation within this statement. It defines that a transform of type stemmer will be applied to all the tokens. In this particular case the stemmer to be used will be the snowball-stemmer. It also shows the custom variables feature. Two additional properties will be added to those tokens to which the lemmatization transformation was successfully applied.

```
1. TAG FROM FILE
2. DEFINE experiment1
3. FORMAT XML(original, pos, startPosition, endPosition)
     <root=name>
4. TOKENIZER removeWhitespace
5. USE DICTIONARY(defaultDictionary)
6. TRANSFORM(stemmer:snowball-stemmer)
   <lemma=result, lemmatized=applied>
```

Fig. 3. Simple example of basic white space remover with stemming parser.

The expected output of a program generated with this statement after receiving a text is shown in Figure 4. Notice that all the tokens have the nodes stated in the XML format definition. Also notice that the lemmatization nodes are added only to the second token, which being a conjugated verbal form, its stem is the verb in infinitive form.

```
<experiment1>
    <tokens>
        <token>
            <original>He</original>
            <pos>personal-pronoun</pos>
            <startPosition>0</startPosition>
            <endPosition>1</endPosition>
        </token>
        <token>
            <original>runs</original>
            <pos>verb</pos>
            <startPosition>3</startPosition>
            <endPosition>6</endPosition>
            <lemma>run</lemma>
            <lemmatized>true</lemmatized>
        </token>
    </tokens>
</experiment1>
```

Fig. 4. Expected output of parsing the text "He runs." with a program generated with the statement shown in Figure 3.

### B. Example 2

The second example includes the conditional statements feature as shown in Figure 5. This time, line 1 expects the text to be parsed from the command line. Line 3 shows the other available format, JSON, adding the same attributes as before,

however in this case we can decide if the generated JSON string should have an array as its root element. Line 5 uses a different dictionary this time. Line 6 defines another type of transformation, text enrichment, in this case to recognize if a token is a hash-tag. Lastly line 7 conditionally applies another transformation to each token, expanding the hashtag into tokens, but it applies it only if the previous hashtag-recognizer transformation was successfully applied to the token.

```
1. TAG FROM CONSOLE
2. DEFINE experiment2
3. FORMAT JSON(original, pos, startPosition, endPosition)
     <rootArray=true>
4. TOKENIZER removeWhitespace
5. USE DICTIONARY(hunspell-dictionary)
6. TRANSFORM(enrich:hashtag-recognizer)
   <hashtag=result, isHashTag=applied>
7. IF(isHashTag) THEN TRANSFORM(enrich:hashtag-expander)
```

Fig. 5. An example of a statement with conditional statements.

The output generated by a program created from this statement is shown in Figure 6. Notice the array as the root element of the JSON string.


```
[{
        "original": "He",
        "pos": "personal-pronoun",
        "startPosition": 0,
        "endPosition": 1
    },
    {
        "original": "runs",
        "pos": "personal-pronoun",
        "startPosition": 3,
        "endPosition": 6
    },
    {
        "original": "#Healthy",
        "pos": "adjective",
        "startPosition": 8,
        "endPosition": 15,
        "hashtag": "Healthy",
        "isHashTag": true
    }
]
```


Fig. 6. Expected output of parsing the text “He runs #Healthy” with a program generated with the statement from Example 2.

## VIII. Conclusion

This paper presents the preliminary results of a declarative language created to allow linguists and researchers in general to generate customized text processing and tagging applications to be used in Natural Language Processing tasks.

The described platform compiles language statements, similar to an SQL statement, and produces a Java program that upon execution will consume text and will preprocess, enrich and tag it. The program will follow the order and generate the output as stated by the user in their statement, including conditional transformations, effectively allowing the user to orchestrate the text analysis flow at execution time.

The language will allow users without knowledge of programming to create their own analysis tools without needing a computer scientist or programmer by their side. This should increase the productivity and the flexibility of NLP research in groups without access to these type of resources.

## IX. Future Work

The current work in progress is the incorporation of the POS tagging definition within the scope of the language. We foresee a whole new set of keywords within the language for defining when and how POS should be done with the provided text. Next, we want to expand the platform to include a command line application that researchers can use to employ the language. Accessibility can also be expanded by developing a web service based platform that will provide the program generation service online for free. Lastly, performance analysis and completeness evaluations need to be done in order to improve the language specification and implement feedback obtained from early users.

## Acknowledgments


This work was done as part of the Natural Language Processing Research Group from the Computer Science Graduate Studies Program, thanks to all the members for their continuous support and feedback. The authors would also like to thank professor Luis Quesada, from the Computer Science department, for allowing his students to work on this project as part of their Compilers course. Lastly, our gratitude to the Research Center on Information and Communication Technology for their support and guidance.


## References


[1] N. I. F. J. D. (eds.), *Handbook of natural language processing*, 2nd ed., ser. Chapman Hall/CRC machine learning pattern recognition series. Chapman Hall/CRC, 2010.

[2] S. L. E. Alexander Clark, Chris Fox, *The Handbook of Computational Linguistics and Natural Language Processing (Blackwell Handbooks in Linguistics)*, 1st ed., ser. Blackwell Handbooks in Linguistics. Wiley-Blackwell, 2010.

[3] B. R.-N. Ricardo Baeza-Yates, *Modern Information Retrieval: The Concepts and Technology Behind Search*, 2nd ed., ser. ACM Press Books. Addison Wesley, 2010.

[4] [Online]. Available: https://gate.ac.uk

[5] P. L. P. M. Carreras X, Chao I, “An open-source suite of language analyzers,” *Fourth International Conference on Language Resources and Evaluation*, 2004.

[6] [Online]. Available: http://lucene.apache.org/solr/

[7] D. D. Chamberlin and R. F. Boyce, “Sequel: A structured english query language,” in *Proceedings of the 1974 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, ser. SIGFIDET ’74. New York, NY, USA: ACM, 1974, pp. 249–264. [Online]. Available: http://doi.acm.org/10.1145/800296.811515

[8] S. Hudson. Cup: Lalr parser generator for java. [Online]. Available: http://www2.cs.tum.edu/projects/cup/