Grammar Checker

Markus Malmsten
Department of Computer Science
Lund Institute of Technology
Sweden,
e99mm@efd.lth.se

Simon Klasén
Department of Computer Science
Lund Institute of Technology
Sweden,
e99sk@efd.lth.se

Abstract

The goal with our project was to implement a grammar checker prototype. The work was influenced by the paper intelligent writing assistance (Heidorn, 2000), which describes the Microsoft word grammar checking technique. Our implementation uses a Perl script for text formatting and the Charniak parser for part of speech and syntactic tagging. The analyzing part was implemented in java.

1 Introduction

The goal with our project was to implement a grammar checker prototype. The purpose with a grammar checker is to check a text for grammatical errors that a grammar book would discuss. A grammar Checker can also include support for style checking (good writing style), but this is not part of our system.

One of the first widely used grammar checker was Writers Workbench (Macdonald et al., 1982) which was developed for Unix systems about 25 years ago. Today the built-in grammar checker in Microsoft Word probably is the most widely used one. It is based on the work that was started by the natural language processing (NLP) group at Microsoft Research in 1992. Our work was influenced by the paper intelligent writing assistance (Heidorn, 2000) which describes the Microsoft Word grammar checking technique.

The biggest difference between the Microsoft Word and our solution is that Microsoft Word is a total solution where all the necessary parts for grammar checking are all built in, while in contrast our solution is divided into three main steps. The steps are text formatting, parsing and analyze, which is taken care of by different tools.

To start with we must tag the input data(the text that should be analyzed); this is done by a simple Perl script that just delimits the sentences with a tag which makes the text ready for parsing by the Charniak Parser.

The Charniak parser takes the tagged input data and performs part of speech and syntactic tagging based on the Penn Treebank (Marcus et al., 1993) tagset. The result is a parse-tree delimited by parenthesis.

Finally is the rule-based analyzing part implemented in Java. We also developed a simple GUI which simplifies the usage of the system. Basically there is one input area for a Charniak parse-tree and an output area that displays the original text including suggested corrections.

2 Implementation

2.1 Overview

We have used three programs in our project; one Perl-script for formatting the original text, one parser that produces a tagged tree and our own Java program which reads the output from the parser, builds a tree and applies the implemented rules and presents the result in a GUI.

2.2 Perl-script

The Perl-script delimits each sentence by adding the <s> ... </s> tags. This format is required by the Charniak parser.

2.3 Charniak parser

We chose the Charniak parser to do the part-of-speech and syntactic tagging. Collins was the other suggested parser but that was neglected due to its much longer running time and the fact that it was 10 times larger. The Charniak parser takes the delimited sentences supplied by the Perl-script and produces a tree in text form where the branches and nodes are enclosed by parentheses.

2.3.1 Penn Treebank style

The tagset used by the Charniak parser is the one constructed by the Penn Treebank project (Marcus et al., 1993), which is a large annotated English corpus. The Penn Treebank tagset is based on the pioneering Brown Corpus which consists of 87 tags. Other tagsets uses up to around 200 tags. The Brown tagset was however pared down considerably. A key strategy in reducing the tagset was to eliminate redundancy by taking into account both lexical and syntactic information. The resulting tagset consists of 48 part-of-speech (see Appendix A, table 1) tags and 14 syntactic tags (see Appendix A, table 2).

2.4 Grammar Checker

2.4.1 System

Our program consists of 3 classes and one main-method. The main-method creates an instance of a GUI-object which includes two event handlers. The event handlers are bound to buttons, one for choosing a file and the other for executing the implemented rules.

It is also in this event handler that the tree is constructed through the method buildTree() in the class CorpusHandler. This function returns a PennNode-object which is the root of the tree. By invoking methods on this root node different rules can be applied and the modified content can be requested which is then displayed.

2.4.2 Building the tree

The program starts with storing the text produced by the Charniak parser in a string. The parentheses structure of the string is then used to decide when new nodes should be created and what they should contain. When a left parenthesis encountered, a new child is created and it becomes the current node. The tag type for the new node is the following word. There are now three possibilities; if the next character is a left parenthesis then a new node is created as above, if its a right parenthesis then this closes the node and the parent node is set to be current node and if its neither of these then the word is the content of the node i.e. a word in the input sentence. In each node we also store which depth in the tree it is in. This can then be used in the search algorithms.

2.4.3 Rules

The rules are applied on each sentence and are recursive. A finite state machine is used to keep track of what to search for and when a correction should be suggested. A special self-constructed node type, FLAG, is inserted if an error is found and it contains a text explaining to the user what can be corrected. Our rules are applied sequentially but they do not affect each other. However the inserted FLAG-nodes must be taken into consideration during the implementation of further rules.

2.4.4 GUI

There are two events that can be triggered in our program. First a file can be chosen by clicking the File-button and secondly the Submit-button which creates tree, runs the algorithms on it and finally prints the resulting tree and text to the lower window. You can chose which rules you want to apply by using the checkboxes at the top. There is also an option to hide the tree structure. See Apendix B for a screenshot of the GUI when the system analyzes a simple sentence.

3 Evaluation

3.1 Testing

Testing of the rules was done in parallel with program construction, one rule at a time. Our initial test samples were just single sentences which had the errors that a specific rule should trig on. After some modifications of the rules, the system was behaving as expected for the single sentences. Everything seemed to work fine.

The real problems started when we tried a larger test corpus. It turned out our rules were to general and was trigged not only when there was an error, but also when the sentences were grammatically correct. The main reason for the behavior was that we were analyzing too small parts of the sentences; we focused at the part of speech tags. This was solved by adding a depth value to all nodes in our parse tree, which enabled us to adjust the rules in respect to bigger text blocks, e.g. subordinate clauses.

The size of the test data was not large enough to make any statistic analyses of the system accuracy. A randomly chosen corpus downloaded from internet indicated the system to work correct, finding the errors and in the same time not trigging on any correct sentence.

4 Conclusions

Even though our grammar checker was inspired by the one found in Microsoft Word, the methods are not the same. The one found in Microsoft Word utilizes a recursive algorithm applied to top nodes. Depending on the type of the node, it applies the subset of rules that are applicable for that type of top node. Our approach on the other hand uses a finite state machine that steps through the text word by word, in order. To determine whether we still are inside the same clause we use the depth attribute. We check one rule at a time, sequentially, which means we keep one object containing the current state of our search. This object differ depending on the rule, and the type of node we search for depend on the current state. When the final state is reached and the requirements are met, a "FLAG"-node is inserted and the state machine is reset, or set to a specific state, depending on the rule. Our approach is of course more expensive, but for our purpose if was sufficient and resulted in code that is easier to understand and maintain.

A problem we encountered that would have made it even more difficult to utilize the Microsoft Word approach was that for incorrect sentences the whole structure of the tree produced by the Charniak parser was altered. This means that it is not sufficient to look at a correct sentence for patterns to which to look for. We got around some of those issues by looking at the text in order.

References

George E Heidorn. 2000. Intelligent writing assistance.

NH Macdonald, LH Frase, P Gingrich, and SA Keenan. 1982. The writer's workbench: Computer aids for text analysis.

Mitchell P. Marcus, Mary Ann Marcinkiewicz, and Beatrice Santorini. 1993. Building a large annotated corpus of english: The penn treebank.

Appendix A

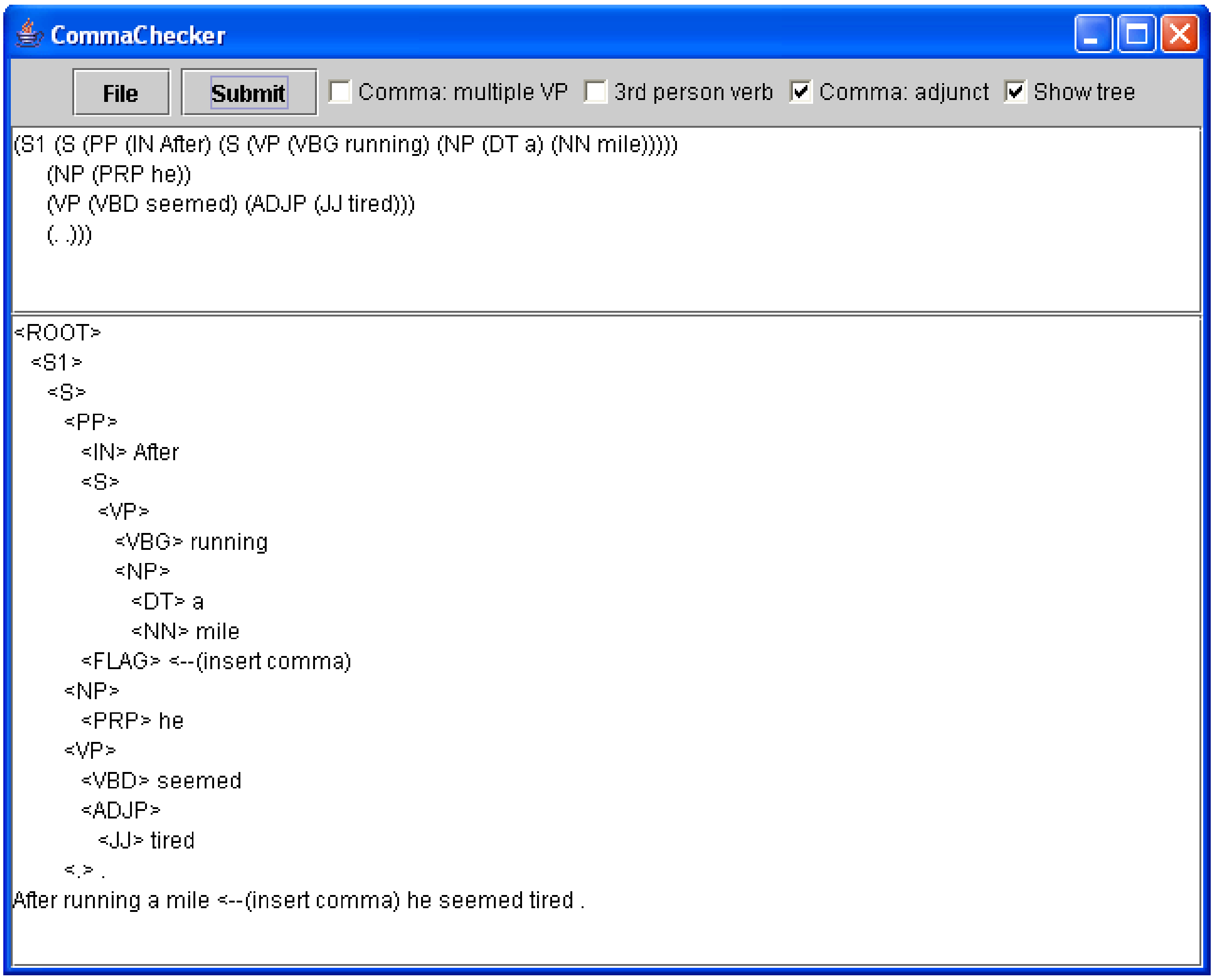
1.	CC	Coordinating conjunction	25.	TO	*to*
2.	CD	Cardinal number	26.	UH	Interjection
3.	DT	Determiner	27.	VB	Verb, base form
4.	EX	Existential *there*	28.	VBD	Verb, past tense
5.	FW	Foreign word	29.	VBG	Verb, gerund/present perticiple
6.	IN	Preposition/subord. conjunction	30.	VBN	Ver, past participle
7.	JJ	Adjective	31.	VBP	Verb, non-3rd ps. sing. present
8.	JJR	Adjective, comparative	32.	VBZ	Verb, 3rd ps. sing. present
9.	JJS	Adjective, superlative	33.	WDT	*wh*-determiner
10.	LS	List item marker	34.	WP	*wh*-pronoun
11.	MD	Modal	35.	WP$	Possessive *wh*-pronoun
12.	NN	Noun, singular or mass	36.	WRB	*wh*-adverb
13.	NNS	Noun, plural	37.	#	Pound sign
14.	NNP	Proper noun, singular	38.	$	Dollar sign
15.	NNPS	Proper noun, plural	39.	.	Sentence-final punctuation
16.	PDT	Predeterminer	40.	,	Comma
17.	POS	Possessive ending	41.	:	Colon, semi-colon
18.	PRP	Personal pronoun	42.	(	Left bracket character
19.	PP$	Possessive pronoun	43.	)	Right bracket character
20.	RB	Adverb	44.	”	Straight double quote
21.	RBR	Adverb, comparative	45.	‘	Left open single quote
22.	RBS	Adverb, superlative	46.	“	Left open double quote
23.	RP	Particle	47.	’	Right close single quote
24.	SYM	Symbol (mathematical or scientific)	48.	”	Right close double quote

Table 1: The Penn Treebank POS tagset

	Tags	
1.	ADJP	Adjective phrase
2.	ADVP	Adverb phrase
3.	NP	Noun phrase
4.	PP	Prepositional phrase
5.	S	Simple declarative clause
6.	SBAR	Clause introduced by subordinating conjunction or *0* (see below)
7.	SBARQ	Direct question introduced by *wh*-phrase
8.	SINV	Declarative sentence with subject-aux inversion
9.	SQ	Subconstituent of SBARQ excluding *wh*-word or *wh*-phrase
10.	VP	Verb phrase
11.	WHADVP	*Wh*-adverb phrase
12.	WHNP	*Wh*-noun phrase
13.	WHPP	*Wh*-prepositional phrase
14.	X	Constituent of unknown or uncertain category
	Null elements	
1.	*	"Understood" subject of infinitive or imperative
2.	0	Zero variant of *that* in subordinate clauses
3.	T	Trace-marks position where moved *wh*constituent is interpreted
4.	NIL	Marks position where preposition is interpreted in pied-piping context

Table 2: The Penn Treebank syntactic tagset

Appendix B



[ ] Comma: multiple VP [ ] 3rd person verb [✓] Comma: adjunct [✓] Show tree

```
(S1 (S (PP (IN After) (S (VP (VBG running) (NP (DT a) (NN mile)))))
    (NP (PRP he))
    (VP (VBD seemed) (ADJP (JJ tired)))
    (. .)))
```

```
<ROOT>
  <S1>
    <S>
      <PP>
        <IN> After
        <S>
          <VP>
            <VBG> running
            <NP>
              <DT> a
              <NN> mile
        <FLAG> <--(insert comma)
      <NP>
        <PRP> he
      <VP>
        <VBD> seemed
        <ADJP>
          <JJ> tired
      <.> .
After running a mile <--(insert comma) he seemed tired .
```

Image 1: Screenshot from the Java GUI.